Generating Optimized and Secure Binary Code

RODOTHEA MYRSINI TSOUPIDI

Doctoral Thesis in Information and Communication Technology
Stockholm, Sweden, 2023

KTH Royal Institute of Technology
School of Electrical Engineering and Computer Science
Division of Software and Computer Systems
TRITA-EECS-AVL-2023:44
978-91-8040-591-1
SE-10044 Stockholm
Sweden

Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framlägges till offentlig granskning för avläggande av Teknologie doktorexamen i elektroteknik Onsdag den 7 juni 2023 klockan 13.00 i Sal F3, Lindstedtsvägen 26/28, Kungliga Tekniska Högskolan, Stockholm.

Tryck: Universitetsservice US AB

Abstract

The increased digitalization of modern societies has resulted in a proliferation of a broad spectrum of embedded devices, ranging from personal smartphones and heart pacemakers to large-scale industrial IoT systems. Since they often handle various sensitive data, these devices increasingly become the targets of cyberattacks that threaten the integrity of personal data, financial security, and sometimes even people's safety.

A common source of security vulnerabilities in computing systems is software. Nowadays, the vast majority of embedded software is written in high-level programming languages and compiled to low-level assembly code using general-purpose compilers. However, general-purpose compilers typically ignore security aspects and mainly focus on improving performance and reducing the code size. Meanwhile, the security-targeting compilers often produce code that is suboptimal with respect to performance. This security-performance gap is particularly detrimental for embedded devices that are usually battery-operated and hence, have stringent restrictions on memory size and power consumption.

Among the most frequently carried out cyberattacks are code-reuse attacks. They insert data into the victim system via memory-corruption vulnerabilities to redirect the control flow and hijack the system. Automatic software diversification is an efficient mitigation approach against code-reuse attacks, however, it typically does not allow us to explicitly control of the introduced performance overhead.

Another large class of attacks is side-channel attacks. Such attacks often target cryptographic implementations and aim at extracting the information about the processed data by recording side-channel information, such as the execution time or the power consumption of the victim system. Typically, protection against side-channel attacks relies on software-based mitigations, which may lead to high performance overhead. An attacker that attempts to hijack the victim system may use either or both of these attacks and hence, often multiple mitigations have to be combined together to protect a system.

This dissertation proposes Secure-by-Construction Optimization (SecOpt), a constraint-based approach that combines performance goals with security mitigations. More specifically, SecOpt achieves performance-aware automatic code diversification against code-reuse attacks, while it generates highly-optimized code that preserves software mitigations against side-channel attacks. A key advantage of SecOpt is composability, namely the ability to combine conflicting mitigations and generate code that preserves these mitigations. In particular, SecOpt generates diverse code variants that are secure against side-channel attacks, therefore protecting against both code-reuse and side-channel attacks.

SecOpt features unique characteristics compared to conventional compiler-based approaches, including performance-awareness and mitigation composability in a formal framework. Since the combined security and performance goals are especially important for resource-constrained systems, SecOpt constitutes a practical approach for optimizing performance- and security-critical code for embedded devices.

Sammanfattning

Den ökande digitaliseringen av det moderna samhället har orsakat snabb spridning av ett brett utbud av inbyggda system, allt från smarta mobiltelefoner och hjärtstimulatorer, till storskaliga industriella IoT system. Dessa datorenheter blir allt oftare mål för cyberangrepp som hotar den personliga integriteten, den ekonomiska säkerheten och ibland även människors säkerhet.

En vanlig källa till säkerhetssårbarheter i datasystem är mjukvara. Nu för tiden är majoriteten av mjukvaran för inbyggda system skriven i högnivåprogrammeringsspråk som kompileras till maskinkod med hjälp av konventionella kompilatorer. Dessa kompilatorer tar ofta inte hänsyn till säkerhetsaspekter i programmets källkod och fokuserar istället på att förbättra prestanda och reducera kodstorlek. Samtidigt producerar säkerhetsinriktade kompilatorer ofta kod som är suboptimal med avseende på prestanda. Denna diskrepans mellan säkerhet och prestanda är problematisk för inbyggda system med stränga restriktioner vad gäller minnesanvändning och energiförbrukning.

Kodåteranvändningsattacker är en av de vanligaste typer av cyberangrepp. Dessa cyberangrepp injicerar data i det angripna systemet, via en minneskorruptionssårbarhet, som ger möjlighet att dirigera om mjukvarans kontrollflöde och kapa systemet. Automatiserad mjukvarudiversifiering är en effektiv skyddsåtgärd mot kodåteranvändningsattacker men nuvarande metoder tillåter inte explicit styrning av prestandaförsämringen. En annan stor cyberangreppsklass är sidokanalsattacker. Dessa cyberangrepp riktas ofta mot kryptografiska implementeringar och syftar till att utvinna säkerhetsviktig information som berör den behandlade datan. Angriparen läser av sidokanalsinformation under programmets exekvering, såsom exekveringstid eller energiförbrukning. Vanliga skyddsåtgärder mot sidokanalsattacker är mjukvaruåtgärder, som dessvärre kan leda till stor prestandaförsämring. En angripare som försöker kapa ett system kan använda en eller flera metoder för att utföra dessa cyberangrepp. Därför måste ofta olika skyddsåtgärder kombineras för att skydda ett system.

Denna avhandling introducerar Säker-vid-Konstruktion Kodoptimering (SecOpt), en villkorsbaserad kompileringsmetod som kombinerar prestandamål med skyddsåtgärder. Närmare bestämt utför SecOpt prestandamedveten automatisk diversifiering mot kodåteranvändningsattacker och genererar optimerad kod som bibehåller mjukvaruåtgärder mot sidokanalsattacker. SecOpts nykelegenskap är dess möjlighet att kombinera motstridiga skyddsåtgärder på ett sätt som bevarar dessa skyddsåtgärders egenskaper. Mer specifikt skapar SecOpt mångfaldiga kodvarianter som uppfyller säkerhetskrav mot sidokanalsattacker, vilket skyddar både mot kodåteranvändningsattacker och sidokanalsattacker.

SecOpt har unika egenskaper jämfört med konventionella kompileringsmetoder, såsom prestandamedvetenhet och komponering av olika skyddsåtgärder i ett formellt ramverk. Kombinationen av säkerhets- och prestandamål är särskilt viktig för resursbegränsade inbyggda system. Sammanfattningsvis är SecOpt en praktisk metod för att optimera säkerhetskritisk kod.

To my grandparents Myrsini and Stratis

Acknowledgments

First, I would like to thank my main supervisor Elena Troubitsyna, who trusted my work and supported me during my PhD. This dissertation would not have been possible without her trust, support, research advice, and supervision. I would also like to thank my co-advisor Panos Papadimitratos, who helped me focus on the security angle of my work and the presentation of my research results. I want to thank my former co-advisor Roberto Castañeda Lozano for teaching me a lot about presenting and writing research, supporting me during my studies, and continuing to advise and work with me until the end of my studies. A special thanks to Christian Schulte, who taught me all I know about Constraint Programming and gave me the opportunity to work on an exciting project that is the basis of this dissertation. You were an inspiration for my work, and you are dearly missed. In addition, I am especially grateful to Thomas Sjöland for his significant and continuous support. I want to thank Fernando Magno Quintão Pereira for serving as the opponent of this dissertation, Elisavet Kozyri, Christoph Kessler, and Marjan Sirjani for serving on the grading committee, Vladimir Vlassov for serving as a chair at my defense, and Roberto Guanciale for acting as the advanced reviewer for my thesis.

Big thanks to my friends and colleagues Amir M. Ahmadian, Nicolas Harrand, and Javier Cabrera Arteaga for improving the quality of my work and the quality of my life with many discussions, fikas, and activities inside and outsite KTH. I also want to thank my friends and colleagues, Saranya Natarajan, Alexandros Milolidakis, Orestis Floros, Nadia Campo Woytuk, Negar Safinianaini, Andreas Lindner, Deepika Tiwari, Anoud Alshnakat, Daniel Lundén, Gizem Çaylak, Tianze Wang, Han Fu, Linnea Stjerna, Javier Ron Arteaga, and Viktor Palmkvist, for the lunch and fika conversations, board-game nights, and taking care of my cats! I want to especially thank my parents, my grandmother Myrsini and my grandfather Stratis, my sister Sofia, my two brothers Panagiotis and Stratis, and recently my niece Marielsa, for their endless support and trust in me. Last but not least, I want to thank Oscar for all his continuous support, patience, and for sharing good and bad moments.

Contents

Thesis

Chapter 1

Introduction

The increasing digitalization of modern societies has created the need to protect sensitive data and safety-critical systems from malicious actors. Embedded devices, such as medical implants, traffic controllers, and car microcontrollers, are safety-critical systems that require preserving strict safety and security requirements. In addition, embedded devices are often battery driven, and thus, resource constrained [1].

Despite long-term efforts to identify and remove security vulnerabilities in software systems, computer systems are still vulnerable to security threats. These vulnerabilities are the result of design decisions, human errors, connectivity via public network, public exposure of the hardware, and/or insufficient control to ensure information security [2]. Furthermore, embedded software is typically written in unsafe languages, such as C and C++, which supports a wide range of target architectures. Mitigating these security vulnerabilities requires vulnerable software and/or hardware changes. Changes in the software are easier to implement and deliver than hardware due to the long development process of hardware. Software security mitigation approaches often apply changes to the source code of software implementations, which is typically compiled to machine code in binary format.

General-purpose compilers focus on the generated codes' performance efficiency size; however, they typically do not preserve security properties [3]. In recent years, secure compilers have aimed to fill this gap and automatically generate secure code. Unfortunately, these approaches often introduce significant overhead to the performance or code size of the generated code [4]. Resource-constrained Internet of Things (IoT) devices and embedded systems may not afford software mitigations that introduce high performance and code-size overhead. This performance-security gap in compiler approaches creates the need for approaches that mitigate cyber-attacks while generating highly optimized code. Constraint-based modeling and solving is a naturally versatile framework that allows the expression of diverse properties. Constraint-based compiler methods trade high-optimality guarantees, composability, and formal guarantees for compilation time [5].

Among the most powerful software-induced cyberattacks are code-reuse attacks [6]. These attacks insert data into the victim system via memory-corruption vulnerabilities to redirect the control flow and hijack the system. Automatic software diversification investigates the automatic generation of diverse program variants and is an efficient mitigation approach against code-reuse attacks [7]. Although the reported performance overhead is typically low [8], few approaches allow control over the introduced performance overhead [9] to generate diverse variants with predictable performance overhead. Furthermore, most automatic software diversification approaches do not provide any guarantees on the diversity of the generated program variants.

Cryptographic algorithms aim at securing sensitive information and communication in the presence of adversarial behavior. The design of popular cryptographic algorithms, such as RSA, is based on the assumption that breaking these algorithms is computationally too hard to be practical. However, with the advent of side-channel attacks, adversaries are able to break cryptographic algorithms using knowledge about the algorithm's implementation and side effects during the execution of the algorithm. In particular, these attacks exploit side-channel information, such as the execution time or the power consumption, during the execution of the victim algorithm to extract information about the processed data. These powerful attacks have challenged the security of popular cryptographic algorithms, such as AES, DES, and RSA [10, 11, 12]. Typical mitigations against side-channel attacks include software mitigations that aim at hiding secret information from side-channel traces. However, these mitigations may lead to high performance overhead. Hence, reducing this overhead is essential, not least for resource-constrained devices.

An attacker attempting to exploit a victim system may use either or both of these attacks; therefore, protecting a system often requires applying multiple mitigations. However, these mitigations may affect or invalidate each other; thus, combining such mitigations while preserving their security properties is highly important. At the same time, the sequential application of software mitigations may introduce prohibively high performance overhead; hence, controlling this overhead is essential, especially for resource-constrained devices.

This dissertation investigates the generation of highly optimized and secure binary code targeting code-reuse and side-channel attacks in embedded systems. Compiler approaches allow high control over the program structure and the generated binary code, which enables effective vulnerability mitigation. In addition, typically, compilers generate optimized code; thus, embedding security properties in a compiler-base approach allows the generation of highly optimized code [4]. This dissertation presents Secure-by-Construction Optimization (SecOpt), a performance-aware, secure compilation method, which uses Constraint Programming (CP), a combinatorial optimization method, to generate highly optimized and secure code. Compiler optimization has effectively used CP to describe the program properties, code transformations, and the target processor cost model [5]. SecOpt extends a constraint-based compiler approach to generate code that hinders cyberattacks while it generates highly efficient code at the cost of compilation-time overhead.

Code verification allows verifying security properties in the generated binary code to increase trust in the compiler result. The ultimate goal of SecOpt is to design a versatile compiler-based toolbox that protects binary code against code-reuse and side-channel attacks while reducing the introduced resource overhead.

1.1 Thesis Statement

This dissertation proposes a constraint programming approach to integrate compiler transformations and security constraints to generate optimized and secure code. The thesis statement of this dissertation is the following:

> Combinatorial binary-code hardening is effective, composable, and highly optimizing.

The proposed approach is *effective* as it achieves satisfactory mitigation effectiveness against different attacks, including code-reuse attacks, power side-channel attacks, and timing side-channel attacks. When the attacker has multiple methods to hijack a system, this approach may *compose* one solution that satisfies mitigations against all these threats. This property is valuable when two different mitigation approaches conflict, i.e. the transformations of one mitigation may invalidate the other mitigation(s). SecOpt is *highly optimizing* because it achieves software diversification with zero performance overhead and generates optimized code against side-channel attacks with reduced overhead compared to related approaches. These properties of SecOpt take advantage of the characteristics of CP, which allows control over both the modeling and solving.

1.2 Research Questions

This dissertation poses four research questions that investigate the feasibility and effectiveness of SecOpt at generating highly optimized secure programs, providing a formal and composable framework.

RQ1: How feasible and effective is performance-aware constrained-based software diversification against code-reuse attacks?

Automatic software diversity has been effective against code-reuse attacks. Fine-grained diversification approaches perform transformations at the binary or the compiler level to generate functionally equivalent program variants. However, most of these approaches 1) focus on x86 systems, 2) do not control how different the generated variants are, and/or 3) do not control the performance overhead of the generated program variants. With this question, we want to investigate the feasibility of a constraint-based diversification approach and its effectiveness against

code-reuse attacks. In addition, we investigate how to generate highly optimized and diverse solutions efficiently.

RQ2: How feasible is secure constraint-based optimization of cryptographic implementations?

Software transformations of cryptographic implementations that mitigate timing and power side-channel attacks may introduce significant performance overhead [13, 14, 15]. This dissertation considers two software mitigation approaches against timing and power side channels, respectively and investigates the feasibility of a constraint-based approach to generate optimized and secure code against these attacks. In addition, this question investigates the adequacy of a constraint-based approach to provide guarantees about the program security.

RQ3: How feasible and effective is a combined mitigation against code-reuse attacks and side-channel attacks?

Protecting a system against cyberattacks often requires combining multiple mitigations against different attack classes. In some cases, diverse mitigations may conflict with each other. In particular, the sequential application of different security mitigations may invalidate one another. This dissertation investigates the feasibility of a constraint-based approach to combine multiple mitigations and the security effect of combining fine-grained software diversification against code-reuse attacks with mitigations against side-channel attacks.

RQ4: How feasible is code verification of binary code against timing side channels?

The code that SecOpt generates against timing attacks needs to preserve timing properties. To improve our trust in SecOpt, we verify the intended timing properties in the generated code using external tools. Furthermore, this dissertation investigates the feasibility of a symbolic execution approach for verifying timing mitigations in WebAssembly. WebAssembly is a recent low-level language with multiple advantages, including security features, portability, and efficiency [16]. However, WebAssembly is vulnerable to timing side-channel attacks. This question aims to investigate the feasibility of code verification to preserve timing properties in real-world binary code.

1.3 Contributions

The contributions of this thesis are as follows:

C1: design and evaluate a local search algorithm for generating diverse solutions in CP;

C2: propose a software diversification method that allows explicit control over the execution-time overhead of the generated program variants and evaluate its effectiveness against code-reuse attacks;

C3: design and evaluate a structural decomposition algorithm to diversify medium-sized functions;

C4: model the automatic generation of optimized code that is secure against power side-channel attacks;

C5: provide a proof that the constraint model against power side-channel attacks protects against the leakage model;

C6: model the automatic generation of optimized code that is secure against timing side-channel attacks;

C7: model and evaluate a composable approach to code optimization that is secure against code-reuse attacks and side-channel attacks;

C8: design and evaluate a method to verify the constant-time property in WebAssembly programs.

1.4 Sustainability and Ethics

Sustainability goals and ethics considerations are an essential part of this thesis that aims at extending the state-of-the-art in secure code generation.

Sustainability: Cybersecurity is essential for preventing disinformation, fraud, and breach of sensitive data, while it promotes economic growth and political independence. This thesis concerns the development of software mitigations against cyberattacks that may result in the leakage of sensitive data or hijacking a possibly critical system, such as medical equipment, energy production, medical records, and more. In addition, the energy consumption of data centers accounts for around 1% to 1.5% of global energy use. Often, this data requires security measures to protect against variable attacker models, which typically increase the performance overhead, and thus, the total energy consumption for achieving the same results. We believe that our approach is a step towards improved performance consumption for software and, hence, reduced energy consumption.

Ethics: This thesis deals with data that consists of programs that are not subject to ethical considerations. The reproducibility of the research conducted during this thesis has been an important goal that we deal with by providing all artifacts for this work online.

1.5 Publications

This thesis includes the following publications:

P1: R. M. Tsoupidi, R. Castañeda Lozano, and B. Baudry, "Constraint-Based Software Diversification for Efficient Mitigation of Code-Reuse Attacks," in *International Conference on Principles and Practice of Constraint Programming*, 2020, pp. 791–808

Contributions: The author of this thesis contributed with design discussions, design decisions, implementation and evaluation of the method, paper writing, and presentation of the paper.

P2: R. M. Tsoupidi, R. Castañeda Lozano, and B. Baudry, "Constraint-based diversification of JOP gadgets," *Journal of Artificial Intelligence Research*, vol. 72, pp. 1471–1505, 2021

Contributions: The author of this thesis contributed with design discussions, algorithm design decisions, implementation and evaluation of the method, and paper writing.

P3: R. M. Tsoupidi, M. Balliu, and B. Baudry, "Vivienne: Relational Verification of Cryptographic Implementations in WebAssembly," in *2021 IEEE Secure Development Conference (SecDev)*, 2021, pp. 94–102

Contributions: The author of this thesis contributed with design discussions, algorithm design decisions, solver optimization decisions, invariant design, implementation and evaluation of all methods, paper writing, and presentation of the paper.

P4: R. M. Tsoupidi, R. C. Lozano, E. Troubitsyna, and P. Papadimitratos, "Securing optimized code against power side channels," *arXiv preprint arXiv:2207.02614*, 2022, to appear in CSF'23

Contributions: The author of this thesis contributed with design discussions, algorithm design decisions, search algorithms, proof design and implementation, implementation and evaluation of all methods, paper writing, and future presentation of the paper.

P5: R. M. Tsoupidi, E. Troubitsyna, and P. Papadimitratos, "Thwarting code-reuse and side-channel attacks in embedded systems," *arXiv preprint arXiv:2304.13458*, 2023, under submission

Contributions: The author of this thesis contributed with timing side-channel modeling, search algorithms, implementation and evaluation of all methods, paper writing, and potential presentation of the paper.

Table 1.1 shows the research questions for each paper, and Table 1.2 shows the contributions of each paper.

publication	R1	R2	R3	R4
A	✓			
B	✓			
C				✓
D		✓		
E	✓	✓	✓	✓

Table 1.1: Research questions addressed per publication

publication	C1	C2	C3	C4	C5	C6	C7	C8
A	✓	✓						
B		✓	✓					
C								✓
D				✓	✓			
E		✓				✓	✓	

Table 1.2: Contributions per publication

1.6 Outline

Chapter 2 discusses the background of this dissertation, including the cyberattacks that this dissertation considers, the combinatorial approach of this work, and the underlying constraint-based compiler backend. Chapter 3 describes the approach and methodology of this dissertation, while Chapter 4 discusses the related work. Chapter 5 presents the summary of the publications that this dissertation includes, and finally, Chapter 6 concludes this dissertation and discusses potential future-work directions.

Chapter 2

Background

This chapter presents the background of this dissertation. The background includes a description of the cyberattacks and mitigations of these cyberattacks that we consider in this dissertation (Section 2.1), a summary of CP, the primary solving method in this dissertation (Section 2.2), and finally, a description of the modeling of the combinatorial compiler that significant part of this dissertation lies upon (Section 2.3).

2.1 Cybersecurity Threats and Mitigations

Cybersecurity is an increasingly important scientific field due to the emergence of IoT devices and the digitalization of services, including transmission and storage sensitive medical information, safety-critical infrastructure, and financial transactions. Cyberattacks constitute a severe threat in modern societies because these attacks lead to economic loss, negative reputation effects, and negative social impact [22]. Many of these attacks are due to vulnerabilities in software or unintended behavior in the hardware [23].

An important stage in the modern software development chain is compilation. Typically, compilers translate code from a high-level language to a low-level language, such as the binary code that the processor executes. Compilers aim at generating code that is semantically equivalent to the source code, however, there is no requirement to preserve security properties. On the contrary, compilers have in some cases been found responsible for removing security software countermeasures or violating source-code security properties [3]. The reason for these security violations is that compiler research has focused on improving performance and/or code size, whereas security is a concept that has become relevant in recent years due to the increasing use of electronic transactions and IoT devices.

Code-reuse and side-channel attacks are two types of powerful attacks where compilers play an essential role. Code-reuse attacks depend on code snippets that appear in the compiler-generated code and, thus, code generation is a key

```
1  move  $a0, $zero          # Move zero to $a0
2  lw    $ra, 0x24($sp)      # Load address for next gadget
3  jr    $ra                 # Jump to address at $sp + 4*0x24
4  addiu $sp, $sp, 0x28      # Delay slot: increment $sp
```

Figure 2.1: Code-reuse gadget in Mips libc

software-development stage to affect these attacks. Similarly, compilers may affect mitigations against side-channel attacks [24, 25, 26], and thus, compilation is the appropriate stage for security property preservation.

Code-Reuse Attacks

Code-reuse attacks depend on memory-corruption vulnerabilities in the program memory space, such as a buffer overflow. These vulnerabilities allow a malicious actor to insert data into the target program memory. To prevent direct attacker-introduced payload execution, executable-space protection prohibits the execution of code in writable memory, e.g., the stack.

Code-reuse attacks, such as return-to-libc, and advanced attacks, such as Return-Oriented Programming (ROP) [6, 27] and Jump-Oriented Programming (JOP) [28, 29, 30], may bypass executable-space protection defenses. The attackers use code snippets in known locations in the program memory of the target system to design the attack. These code snippets, so-called gadgets, typically end with a control-flow instruction, such as a return, a jump, or a call instruction. The last control-flow instruction allows the attacker to build a gadget chain by transferring the control from one gadget to the next. The gadgets that are available in the victim program memory, such as in dynamically-loaded libraries, may provide high expressibility to allow the attacker to hijack the system [6].

Figure 2.1 shows a code-reuse gadget found in the libc library of a Mips32 Debian Linux system. At line 1, the gadget moves value zero to register `$a0`. Then, it loads the address of the next gadget to the return-address register `$ra`. The attacker has ensured that this address resides to address `0x24($sp)`, where `$sp` is the stack pointer. At line 3, the code moves to the new gadget, *jr* `$ra`, and the last instruction is a delay slot[1]. The delay slot increases and stack pointer `$sp` by `0x28`. This last step is important for the attacker to move the attack payload forward to enable moving to the data of the next gadget. Such code sequence appear at the return points of functions and are very common in compiler-generated code.

Classic code-reuse attacks assume that the attacker has access to binary code identical to the victim code and designs a payload offline before attacking the victim system. More advanced attacks allow reading the memory during the attack. In

[1]Delay slots in Mips follow a branch instruction but execute before them and their purpose is to reduce the branch target estimation delay.

```
0x8014: move  $a0, $zero
0x8018: lw    $ra, 0x24($sp)
0x801c: jr    $ra
0x8020: addiu $sp, $sp, 0x28
```

(a) Variant 1

```
0x8010: lw    $t9, 0x24($sp)
0x8014: move  $a0, $zero
0x8018: nop
0x801c: jr    $t9
0x8020: addiu $sp, $sp, 0x28
```

(b) Variant 2

Figure 2.2: Two diverse implementations of the gadget in Figure 2.1 in Mips.

particular, JIT-ROP [31] dynamically reads the program memory, selects appropriate gadgets, and performs the attack. Similarly, Blind ROP (BROP) [32] achieves reading the memory using repeated steps, however, BROP may lead to system crashes. A different approach by Seibert et al. [33] achieves identifying the location of code snippets in the code using timing side-channel information. In particular, when the victim binary is not identical with the original binary, an attacker may use timing side-channel information to decipher the binary's diversification. The attacker records the execution time of different parts of the code to recognize how the code has been diversified. The advantage of this approach is that it does not lead to system crashes, however, it may take a long time, up to a week, to achieve the attack.

Code-Reuse Attack Mitigations

There are two main mitigation approaches against code-reuse attacks, Control-Flow Integrity (CFI) [34] and automatic software diversification [7]. The main idea of CFI is to ensure that the program executes legitimate control flow [35]. In this way, CFI restricts code-reuse attacks to only transfer control to legitimate control-flow targets, which reduces the power of these attacks[2]. The main disadvantage of CFI is that it often leads to high execution-time overhead [34]. An alternative mitigation against code-reuse attacks is automatic software diversification, which introduces uncertainty to the implementation of the code and, in this way, hinders a code-reuse attack that depends on the addresses of known gadgets. More generally, software diversification is a method to improve the fault tolerance of software systems [36, 37] and security [38, 39] in computing systems. Software diversification investigates code diversity at the level of algorithm implementation, library implementation [40, 41], memory layout (Adress Space Layout Randomization (ASLR)), and binary-level implementation [8, 9]. This dissertation concerns automatic fine-grained software diversification that generates program variants derived from the same source code but with different binary implementations.

Many code-reuse attacks depend on gadgets, which are code sequences that exist in the program memory. Typically, classic code-reuse attacks depend on the

[2]Other approaches, such as stack canaries, have similar effect on ROP attacks.

```
1 uint8 check_bit(uint8 pub, uint8 key) {
2   uint8 t = 0;
3   if (pub == key)
4     t = foo();
5   return t;
6 }
```

Figure 2.3: Program with secret-dependent branching in C

exact addresses of the gadgets in memory. Software diversification as a mitigation against code-reuse attacks changes the address of these gadgets and/or their implementation, aiming to reduce the probability of success of an attack. Figure 2.2 shows two versions of the gadget in Figure 2.1, where Figure 2.2a corresponds to the original gadget in Figure 2.1. The gadget in Figure 2.2b differs from the gadget in Figure 2.2a in three points, 1) there is an additional No-Operation (NOP) instruction at address `0x8018`, 2) instructions *move* and *lw* are swapped, and 3) the return address is loaded at register `$t9` instead of register `$ra`. Assuming that the attacker has used the gadget in Figure 2.2a to generate their payload, this payload may fail against a user using the code in Figure 2.2b. More specifically, an attacker that uses the gadget in Figure 2.2a will instruct the previous gadget to jump to address `0x8014`, namely the beginning of the gadget. However, at address `0x8014`, the diversified gadget (Figure 2.2b) moves the value of zero to register `a0`, but it does not move the attacker-controlled address to `$t9`. Hence, the processor will not transfer the control flow to the next gadget to finalize the attack. This scenario leads, with a high probability, to attack failure. Additional diversification approaches like function shuffling or basic-block shuffling may increase the entropy of this diversified example.

Side-Channel Attacks and Mitigations

Side-channel attacks use side-channel information, such as the execution time or the power consumption of a program, to extract information about valuable program values. Side-channel attacks constitute a severe threat to cryptographic algorithms. The security of cryptographic algorithms often depends on values that should remain secret, such as symmetric or asymmetric keys. Using side-channel information, an attacker may extract information about these keys to break cryptographic security. Side-channel attacks have been successful against popular cryptographic algorithms, such as DES, AES, and RSA [10, 11, 12].

Timing Side-Channel Attacks

Timing side-channel attacks [12] measure the execution time of the program to extract information about program values. The attacker may perform the attack

```
1 uint8 sbox_get(uint8 *pub, uint8 key) {
2   return pub[key];
3 }
```

Figure 2.4: Program with secret-dependent memory operation

remotely via the network [42] or locally when the victim and the attacker share the same hardware [43]. Timing attacks may extract information about a secret value when this value affects the program execution.

In many implementations of cryptographic algorithms, the execution time may depend on the value of the encryption/decryption key. For example, consider function `check_bit` in Figure 2.3. Assume that the value of `pub` is known to the attacker, whereas `key` contains secret information. If these two values are equal, the program executes function `foo()`, whereas otherwise, the `check_bit` function returns immediately. Hence, if the two values are equal, then the execution of function `check_bit` takes longer time than otherwise. Therefore, an attacker that measures the execution of this function may distinguish the difference between the execution time, e.g. by controlling the value of `pub` and, subsequently, extract one bit of information about the value of `key`.

Another timing vulnerability that appears in cryptographic implementations is secret-dependent memory operations. Figure 2.4 shows function `sbox_get`, which takes two inputs, a public array, `pub`, and a secret value, `key`. The function returns the element of `pub` at index `key`. Here, the source of the leakage is the cache hierarchy, which aims at providing recently-accessed address regions faster. In particular, cache memories have low data access latency and store recently accessed memory blocks for faster access in future memory requests by the processor. The cache stores these blocks based on their address in memory, which depends on the array index, `key`, in our example. Upon a memory request in the cache, if the relevant cache line is full, the cache replaces old data with the new data. An attacker may take advantage of the cache hierarchy to extract information about secret values. For example, an attacker may fill a shared cache with their own data before the victim runs their code. Subsequently, the attacker measures the access time of their data to infer the memory access patterns of the victim (Prime+Probe) [44].

Other sources of timing vulnerabilities include variable-latency instructions with secret operands, such as division and multiplication instructions in some architectures. In general, when the execution time depends on secret values that should remain unrevealed, execution time may leak information about these secret values.

Timing Side-Channel Mitigations

In the research literature, there are diverse mitigation approaches against timing side-channel attacks. This dissertation concerns two mitigation approaches: constant-time programming and constant-resource programming.

```
1 uint8 check_bit(uint8 pub, uint8 key) {
2   uint8 t; sint8 m;
3   t = foo();
4   m = -(pub == key);
5   return (t&m | 0&~m);
6 }
```

Figure 2.5: Constant-Time Program from Listing 2.3 in C

Constant-Time Programming: The constant-time programming discipline is a set of programming guidelines that aim at removing secret-dependent timing variations. The constant-time discipline converts secret-dependent branch instructions and memory accesses to constant-time equivalent. In addition, some approaches consider secret-dependent variable-latency instructions, such as division and multiplication, when their operands are secret values.

Figure 2.5 shows a constant-time version of function `check_bit` in Figure 2.3. This implementation starts by executing function `foo()` (line 3), regardless of the result of the comparison between the two input values. Then, the code stores the negation of the result of the comparison between the two input values in the signed variable `m` (line 4). That is, if the result of the comparison is true or `000000001` in binary encoding, then variable `m` is minus one or `11111111` in binary encoding. Similarly, if the result of the comparison is false or equal to zero, then minus zero is zero, which leads to `00000000` in binary encoding. The return result (line 5) is either equal to `t`, when the value of `m` is `11111111`, or `0`, when `m` is `00000000`. Hence, in the constant-time reimplementation of Figure 2.3, the execution time does not depend on the secret value, instead, it is constant.

Constant-time implementations may contain complex logic-operation code that is often difficult to write, read, and debug. Hence, verification approaches are important for guaranteeing the correctness of these implementations. Furthermore, compilers may break such implementations, for example, by converting logical operations back to a secret-dependent branch [24].

Constant-Resource Policy: The constant-time programming discipline often leads to complex code. Another alternative mitigation approach against timing side-channel attacks is the constant-resource policy [45]. In contrast to the constant-time policy, the constant-resource policy does not require the absence of secret-dependent branches and memory operations, instead it requires that the program uses the same resources for different secret values [46]. In particular, the constant-resource policy allows secret-dependent branches when both branch directions lead to the same execution time.

Figure 2.6 shows a constant-resource version of function `check_bit` in Figure 2.3. At line 3, the implementation compares the two input values. If the input values are equal, the implementation calls function `foo()` and stores the result in variable

```
1 uint8 check_bit(uint8 pub, uint8 key) {
2   uint8 t = 0, _t;
3   if (pub == key)
4     t = foo();
5   else
6     _t = foo();
7   return t;
8 }
```

Figure 2.6: Balanced Constant-Resource Program from Listing 2.3 in C

`t` (line 4). If the two input variables are not equal, the code calls function `foo()` and stores its result in an unused variable `_t` (line 6). The return result of this function is `t`, which is either `0` or the return value of function `foo()`. The idea of this transformation is that both branch directions take the same execution time[3].

This implementation is easier to read and more similar to the original implementation than the constant-time equivalent in Figure 2.5. However, there are two main compilation challenges of such implementations. First, dead-code elimination passes may remove the functionally unused call to function `foo()` in the `else` branch. Second, timing variations between the two branches may depend on different microarchitectural features, such as instruction latencies, branch prediction, and memory accesses. The latter may lead to a longer execution time for either of the branches. These challenges make compiler-based approaches that consider an accurate cost model valuable to guarantee that different execution paths use the same resources.

Power Side-Channel Attacks

Power side-channel attacks measure the power consumption during the execution of a program to exploit the victim system or extract security-critical information. For power side-channel attacks, any value transitions in hardware, such as hardware registers, the memory, or the memory bus, may reveal information about these values. When a program manipulates secret information, the attacker may disclose this information by recording the power consumption during the program execution. Figure 2.7 shows function `xor`, which takes two values as input, `pub`, which is known and/or controlled by the attacker and `key`, which is a secret value unknown to the attacker. The function returns the exclusive-OR operation of the two input values. These operations may affect the device's power consumption because the processor manipulates the secret value with a public value in the hardware.

A typical leakage model for power attacks is the Hamming model. A data word consists of m bits, each of which takes a value from $[0, 1]$. We can write a data

[3]In general, the actual timing of each of the branches depends also on microarchitectural features including, branch decisions, instruction latencies, and memory accesses

```
1  uint32 xor(uint32 pub, uint32 key) {
2    uint32 t = pub ^ key;
3    return t;
4  }
```

Figure 2.7: Exclusive-OR implementation in C

word in form $D = \sum_{i=0}^{m-1} d_i 2^i$, where d_i is the value of the binary encoding at the i_{th} position. The Hamming weight of a data word corresponds to the number of bits that are one, i.e. $H(D) = \sum_{i=0}^{m-1} d_i$. Many works assume that the data leakage through a power side channel depends on the number of bits switching from one to zero or vice versa at a given time. Hence, the data leakage at the transition of one hardware variable from D_1 to D_2 equals $H(D_1 \oplus D_2)$, where $\oplus$ is the exclusive-OR operator. The transition between these values happens at distinct time points, for example, at the positive or negative clock edge in an electronic device [11]. The data leakage may happen at different parts of the processor, for example, the memory bus, the hardware registers, the memory cells, and more [11, 13, 47]. According to Papagiannopoulos and Veshchikov [13], the easiest to exploit transitional leakages are hardware-register and memory-bus reuse in an AVR processor.

Power side-channel attacks require the attacker to have local access to the victim device and have equipment such as an oscilloscope to record the power consumption of the target algorithm at the victim device. Simple Power Analysis (SPA) is a technique to make direct observations on the power trace of an algorithm in time. SPA allows the attacker to extract information when indirect branches, value comparisons, multiplication operations, and exponentiation operations depend on secret values. SPA allows secret information extraction from multiple DES procedures [10]. Differential Power Analysis (DPA) may distinguish smaller variations in the power traces that may be too small to distinguish using SPA. DPA records the power traces of multiple executions of the algorithm. By observing the algorithm's output, e.g., the ciphertext, the attacker determines the values of secret data, such as cryptographic keys [10]. Correlation Power Analysis (CPA) [11] uses the correlation factor between the hamming distance of data and the measured power to determine the relationship between a guessed value and the actual measurement. This analysis depends on the Hamming distance model and is as powerful as DPA. In recent years, power attacks based on deep learning has enabled more powerful attacks [48].

Power Side-Channel Mitigations

Power side-channel attacks do not affect the program execution and are, therefore, difficult to detect. One mitigation approach against power side-channel attacks is software masking. This mitigation aims at randomizing the secret values using

```
1  uint32 sec_xor(uint32 pub, uint32 key, uint32 mask) {
2    uint32 mk = mask ^ key;
3    uint32 t  = mk ^ pub;
4    return (t,mask)
5  }
```

Figure 2.8: Exclusive-OR with software masking

```
1  sec_xor(r0 ← pub, r1 ← key, r2 ← mask) {
2    r2 ← r2 ^ r1;
3    r0 ← r2 ^ r0;
4  }
```

Figure 2.9: Exclusive-OR with software masking in machine code

randomly generated values at every program execution. Software masking aims at statistically hiding the secret information.

Software masking depends on finite field theory and uses the exclusive-OR operation as the addition operation in $GF(2^n)$. Figure 2.8 shows function `sec_xor`, which is an implementation of Figure 2.7 using software masking. First, this function randomizes the secret value `key` using a newly introduced randomly generated value `mask` (line 2). Then, the function uses the randomized key value, `mk`, to perform an exclusive-OR operation with `mask` (line 3). Finally, function `sec_xor` returns the final value and variable `mask`, which is necessary for retrieving the final result, namely `(mask ^ key ^ pub) ^ mask` is equal to `key ^ pub`.

Although the function implementation in Figure 2.8 randomizes the secret before interacting with the public value, hardware interactions of values may introduce secret-dependent power dependencies that appear in the power traces. These interactions occur when hardware registers, the memory bus, or memory cells transition from one value to another. For example, when a hardware register takes a new value or when a new value is transferred via the memory bus to the main memory, may result in a transition. Assuming that transitions from one to zero and from zero to one result in a similar power consumption change, the hardware's total power change depends on the hamming distance between the old value and the new value at the transition. Figure 2.9 shows an implementation of the masked algorithm in Figure 2.8 using hardware registers. At line 2, register `r2`, which holds value `mask`, transitions to the value of `r2 ^ r1`, which holds value `key ^ mask`. The leakage from this transition is equal to `mask ^ (key ^ mask)` or `key`, which is a secret value. Hence, this hardware implementation of the masked code leads to transitional leaks due to register reuse. Similarly, other hardware value transmissions may lead to transitional leaks of secret values.

Many transitional leaks depend on compiler-generated machine code, which determines hardware register assignment and instruction order. Furthermore, some mitigations against these transitional leaks result in high performance overhead [13]. Therefore, compiler-based approaches may provide opportunities to mitigate these leakages at a reduced performance overhead.

2.2 Constraint Programming

Constraint Programming (CP) is a method to solve or optimize combinatorial problems. Compared to other combinatorial solving or optimizing approaches, such as Boolean Satisfiability (SAT) and linear programming, CP enables larger flexibility with regard to the domain of variables and the type of constraints. The main strength of CP is its ability to exploit substructures in combinatorial problems and has been particularly successful in solving scheduling, resource allocation, and rectangle packing problems [49].

CP solving usually consists of two parts, 1) modeling, where the user models the problem as a finite set of variables and constraints, and 2) solving, where the solver attempts to find solutions, i.e. variable assignments that satisfy all constraints, to the problem. A Constraint Satisfaction Problem (CSP) models a problem for which we need to find one or multiple solutions to the problem. Constraint Optimization Problems (COPs) include an *objective* function, and the goal is to find the solution(s) that optimize(s) this objective function.

Modeling

In CP, a problem is modeled as a finite set of variables, V, that takes values from a finite set, U, and a set of constraints, C, among the variables in V. Typical variable domains include integer and Boolean sets. CP solvers provide different constraint implementations among different variable types.

Problem Modeling: The first step in CP is problem modeling. Modeling is important because modeling decisions affect the solving time of the problem.

A typical example of a combinatorial problem is the eight-queen problem, namely the problem of placing eight (chess) queens on a chessboard, so that they do not threaten each other. There are (at least) two ways to model the eight-queens problem. One way to model this problem is to use eight variables, q_i, one for each queen. Each variable q_i corresponds to the i_{th} column (or row) of the chessboard, and its value corresponds to the position in the row (or column). The variable domain, in this case, will be set $\{1, 2, ..., 8\}$. A different way to model the eight queens problem is using one variable, c_i^j for each chessboard position. In this way, we have 64 variables with domain, $\{0, 1\}$. A variable takes value one when a queen is present on the corresponding cells and zero otherwise. These two different modeling approaches may lead to different solving times due to different properties

of the solver and individual constraint implementations. Therefore, constraint modeling is an essential step in constraint solving.

Constraints: Constraints are important for modeling a constraint problem. Different constraints affect the efficiency and expressiveness of the solving procedure. Constraints that involve three or more variables are also called *global* constraints [49] and often provide improved efficiency. Global constraints constitute one of the key strengths of CP.

The first modeling approach of the eight-queens example may use a global constraint, `all-different`$(q_1, ..., q_8)$ [50], to make sure that all variables differ from each other; namely, they do not share the same row/column. The `all-different` constraint has improved efficiency over a set of disjunctive constraints $\forall i, j \in \{1, ..., 8\}.q_i \neq q_j$.

The second modeling approach may use a global linear constraint $\forall i \in \{1, ..., 8\}. \sum_{j \in \{1,...,8\}} c_i^j = 1$ to ensure that every row accommodates only one queen. Note that this constraint is an efficient way to model that one and only one queen appears in a row or $\forall i \in \{1, ..., 8\}.\exists j \in \{1, ..., 8\}. \left(c_i^j = 1 \wedge \forall k \in \{1, ..., 8\} - \{j\}.c_i^k = 0\right)$.

Solutions: The solutions to the problem are the variable assignments that satisfy all constraints.

Example 1 *Give a CSP $P = \langle V, U, C\rangle$, where $V = \{x, y, z\}$, $U = \{1, 2, 3, 4\}$, and $C = \{x > 1, x + y = z\}$, the solutions to the problem are: $sol(P) = \{\langle 2, 1, 3\rangle, \langle 2, 2, 4\rangle, \langle 3, 1, 4\rangle\}$*

In some problems, all solutions are not equivalent, and the application requires a solution that, e.g., maximizes or minimizes an *objective* function, O. In this case, the goal of the solver is to find the *optimal* solution to the problem.

Example 2 *CSP P may be extended to a COP, $P' = \langle V, U, C, O\rangle$, with an optimization function $O =$ `maximize` x. Then, the solution that we are looking for is $sol(P') = \{\langle 3, 1, 4\rangle\}$.*

For many problems, it is sufficient to find one (optimal or not) solution, whereas, for other problems, it is essential to find a set of diverse solutions [51]. Example applications include automatic test generation [52], finding alternative optimal solutions in process plant layout optimization [51], and solving complex constraints by generating multiple solutions and then verifying the suitability using more exact methods [53]. Defining the meaning of the difference between solutions is essential in this context. To achieve this, we define function δ that takes two solutions and returns the difference between these solutions.

Example 3 *For problem P, we may define the function $\delta(s, s') = |s_x - s'_x| + |s_y - s'_y| + |s_z - s'_z|$. Then, the distance between all three pairs of solutions is $\delta(s_i, s_j) = 2$, $i, j \in \{1, 2, 3\}$.*

Solving

Given a CSP or a COP, the solving process aims at finding feasible solutions. Typically, solving in CP is an iterative procedure and consists of two main steps, 1) propagation, which reduces the variable domains based on the constraints, and 2) search, which sets the value of one variable from its domain in each step.

Propagation: Constraint propagation is the procedure that reduces the variable domains based on the problem constraints. Often the propagation process considers one constraint at a time and is repeated until reaching a fixpoint.

Example 4 *CSP* $P = \langle V, U, C \rangle$*, where* $V = \{x, y, z\}$*,* $U = \{1, 2, 3, 4\}$*, and* $C = \{x > 1, x+y = z\}$*, has two constraints* $C_1 = x > 1$ *and* $C_2 = x+y = z$*. Propagation of* C_1 *results in* $x \in \{2, 3, 4\}, y \in \{1, 2, 3, 4\}, z \in \{1, 2, 3, 4\}$*. Then, propagation of* C_2 *results in* $x \in \{2, 3\}, y \in \{1, 2\}, z \in \{3, 4\}$*, which is a fixpoint.*

Search: Propagation is usually not sufficient to solve a problem. After propagation has reached a fixpoint, the solver applies search to decompose the problem into simpler subproblems. In particular, the solver selects one variable and splits its domain, often in two parts.

Example 5 *In our problem, P, search may split the previous fixpoint (*$x \in \{2, 3\}, y \in \{1, 2\}, z \in \{3, 4\}$*) into two subproblems, one for* $x = 2$ *and one for* $x = 3$*.*

Subsequently, the solver applies propagation to each subproblem to find solutions and repeats the search step until it finds one solution, all solutions, or a specific solution.

Example 6 *For the first branch,* $x = 2$*, propagation returns fixpoint* $x \in \{2\}, y \in \{1, 2\}, z \in \{3, 4\}$*, which is not a single solution. Thus, we need to apply search again for* $y = 1$ *and* $y = 2$*, which gives two solutions* $x \in \{2\}, y \in \{1\}, z \in \{3\}$ *and* $x \in \{2\}, y \in \{2\}, z \in \{4\}$*, respectively.*

For the second branch, $x = 3$*, we get directly one solution after propagation:* $x \in \{3\}, y \in \{1\}, z \in \{4\}$*.*

Deciding which variable to branch on and how to split the value space of the selected variable at an invocation of search may affect the efficiency of the solving procedure and is an important design decision. These branching schemes are called search heuristics. Typical search heuristics include *smaller value first*, *variable with the smallest value set first*, and more.

Branch-and-bound search is an approach to solving COP problems. First, the algorithm finds one solution. Then, the algorithm adds a new constraint to the problem that requires the following solution to be better than the current one. The solver continues until there are no better solutions.

Example 7 *For P' that we used before, with $O =$* `maximize` *x, we have the first propagation fixpoint as: $x \in \{2,3\}, y \in \{1,2\}, z \in \{3,4\}$. Assume we first find solution $\langle 2,1,3\rangle$. Then, branch-and-bound adds constraint $x > 2$, which leads to the optimal solution, $\langle 3,1,4\rangle$.*

Apart from the classic search heuristics, there are other search procedures called metaheuristics [54]. In this dissertation, we use Large Neighborhood Search (LNS) [55], a form of local search that is consistent with CP. LNS is often used for solving optimization problems. After finding the first solution, LNS uses part of this solution to find a better solution. To do that, LNS *destroys* parts of the solution (assignments to variables) and then tries to find other solutions that improve the objective function.

Example 8 *In our example, P', with $O =$* `maximize` *x, we have the first propagation fixpoint as: $x \in \{2,3\}, y \in \{1,2\}, z \in \{3,4\}$. Assume we first find solution $\langle 2,1,3\rangle$. Then, LNS destroys variables x and z, and adds an optimization constraint $x > 2$ and after propagation we have: $x \in \{3\}, y \in \{1\}, z \in \{4\}$, which is the optimal solution.*

2.3 Compiler Backend

Compilers are essential components in the software development chain. Typical general-purpose compilers take as input a program written in a high-level language and translate it to a low-level language or machine code. Conventional compilers, such as LLVM [56] and GCC [57], consist of a series of analysis and transformation passes that aim to improve the code performance, reduce the code's size, or minimize the energy consumption.

Compiler front- and middle-end passes perform high-level transformations such as loop unrolling, dead-code elimination, and expression rewriting, whereas compiler back-end passes are responsible for target-processor-related transformations. At the compiler backend, there are three main transformations, 1) instruction selection, where machine instructions replace abstract instructions, 2) instruction scheduling, which decides the order of the instructions in the final code, and 3) register allocation, which assigns virtual registers to hardware registers and memory. These transformations are very important for the quality of the generated code in the hardware and are increasingly important in architectures that require significant effort from the compiler, such as static multiple-issue architectures. Such architectures typically require the compiler to schedule multiple instructions to different processing units statically. However, the compiler backend transformations are known to be combinatorial problems, where finding the optimal hardware code implementation may take exponential time. Instead, many compilers use heuristics that find efficient but not optimal solutions.

Combinatorial Compiler Backend

For compiler-demanding architectures or performance-critical functions, there are combinatorial compiler-backend approaches to find optimal low-level implementations [58, 59]. These approaches use an abstract processor model and represent the quality of each solution in the form of a cost function. Subsequently, combinatorial compiler approaches generate code that optimizes this objective function. In combinatorial optimization, an optimal solution corresponds to a solution that maximizes or minimizes the objective function. Combinatorial models do not exclude the presence of multiple optimal solutions.

Unison, a recent work, has shown the benefits of unifying multiple compiler passes to generate highly optimized code. In particular, instruction scheduling and register allocation are strongly interdependent problems, and thus, modeling both problems together improves the quality of the generated code [5]. Unison is the first practical combinatorial compiler-backend approach, and SecOpt is based on Unison. In particular, SecOpt extends Unison to consider security properties.

Modeling: Typically, compiler-backend passes process the program in Static Single Assignment (SSA) form, where every variable is assigned a value only once. A combinatorial compiler models a program as a set of basic blocks B, i.e. pieces of code with no branches apart from the exit of the block. Each basic block contains a number of optional operations, $o \in \mathit{Operations}$, that may be *active* or not. An active operation appears in the generated code, while an inactive operation is omitted. These optional operations enable transformations that are necessary for register allocation and instruction scheduling. Ins_o denotes the set of hardware instructions that implement operation o. Each operation includes a number of operands $p \in \mathit{Operands}$, each of which may be implemented by different, equally-valued temporaries, $t \in \mathit{Temps}$. Temporaries represent an infinite number of virtual registers, which the solver assigns to a hardware register or a memory location in the stack. Alternatively, a temporary may not be alive.

Fig. 2.10 shows a simplified version of the constraint-based compiler backend model for Fig. 2.7. Temporaries `t0` and `t1` contain the input arguments `pub` and `key`, respectively. Copy operations (`o2, o3, o5`) enable copying program values from one register to another (or to the stack) and are critical for flexibility in register allocation. Operation `o2` allows the copy of value `pub` from `t0` to `t3`. In the final solution, a copy operation may not be active (shown by the dash in the set of instructions: `[-, copy]`). The `xor` operation (`o4`) takes two operands, and each of these operands may use equally valued temporary variables, e.g., `t1` and `t4`.

Objective Function: Combinatorial compiler backends use an objective function that is based on the target processor characteristics to generate optimized code. Unison's objective function optimizes metrics such as *code size* and *execution time*. Unison captures these goals in a generic objective function that sums up the

```
o1: in [t0 ← pub, t1 ← key]
o2: t2 ← [-, copy] t0
o3: t3 ← [-, copy] t1
o4: t4 ← xor [t1,t3] [t0,t2]
o5: t5 ← [-, copy] t5
o6: out [t6 ← [t4,t5]]
```

Figure 2.10: Exclusive-OR operation

weighted cost of each basic block:

$$\sum_{b \in B} \mathit{weight}(b) \cdot \mathbf{cost}(b),$$

where $\mathbf{cost}(b)$ for basic block b is a variable, which estimates the cost of a specific implementation of the basic block, and *weight* is a constant value that represents the contribution of the particular basic block to the total cost. For execution-time optimization, Unison uses statically extracted basic-block frequencies to estimate the contribution of each basic block. This cost model is accurate for predictable hardware architectures. The accuracy of the cost model reduces in the presence of advance microarchitectural features, such as cache hierarchy, dynamic branch prediction, and/or out-of-order execution.

Solving: After modeling the problem, the constraint solver attempts to optimize the problem using scheduling constraints. Unison uses structural decomposition and advanced search strategies to find the optimal or a good solution efficiently.

Figure 2.11 shows a solution to the program in Figure 2.10. Temporaries `t0` and `t1` are assigned to hardware registers `r0` and `r1`, respectively, due to the calling conventions of the target architecture. Temporary `t2` is assigned to `r2`, and the operation copies the value of `r0` to `r2`. Operation `o3` is not active, and thus, temporary `t3` is not live. Temporary `t4` is assigned to register `r0`. Operation `o5` is also not active, and temporary `t5` is not live. Finally, temporary `t6` is assigned to the return register `r0`. This solution is suboptimal, instead, the optimal solution (see Figure 2.12) deactivates all `copy` operations.

```
o1: in [t0:r0 ← pub, t1:r1 ← key]
o2: t2:r2 ← copy t0:r0
o4: t4:r0 ← xor t1:r1 t2:r2
o6: out [t6:r0]
```

Figure 2.11: Solution of exclusive-OR operation

```
o1: in [t0:r0 ← pub, t1:r1 ← key]
o4: t4:r0 ← xor t1:r1 t0:r0
o6: out [t6:r0]
```

Figure 2.12: Optimal solution of exclusive-OR operation

Transformations: Unison enables the following transformations: 1) hardware register assignment, 2) register copying, 3) memory spilling, 4) constant rematerialization, 5) instruction order, and 6) NOP insertion. Hardware register assignment maps a hardware register to an operand, register copying copies an operand from one hardware register to another, and memory spilling allocates a memory slot in the stack to store an operand. Constant rematerialization allows re-running an operation, like value loading, instead of copying its result. SecOpt may also alter the instruction order as long as there are no data dependencies or insert NOP instructions by delaying the issue cycle of an operation.

Chapter 3

Approach and Methodology

3.1 Secure-by-Design Optimization (SecOpt)

This section presents our approach, SecOpt, to generate secure and optimized code by design. SecOpt implements the following mitigation approaches: 1) fine-grained software diversification, 2) software masking, and 3) preservation of the constant-resource property.

Overall, there are three main advantages of SecOpt compared to other approaches, 1) performance awareness, 2) composability, and 3) a formal definition of mitigations in the form of constraints. SecOpt inherits an accurate cost model from Unison [5], which allows control over the performance overhead of the generated code. In particular, SecOpt may generate optimal or near-optimal code with a known performance overhead that is based on the cost model. An important advantage of SecOpt is combining different security mitigations. More specifically, SecOpt allows combining fine-grained software diversification and software masking or the constant-resource property to ensure the preservation of the combination of these properties. Moreover, many conventional compiler-based mitigation approaches do not provide formal guarantees that the intended properties hold in the generated code. SecOpt defines the target mitigations in the form of constraints,

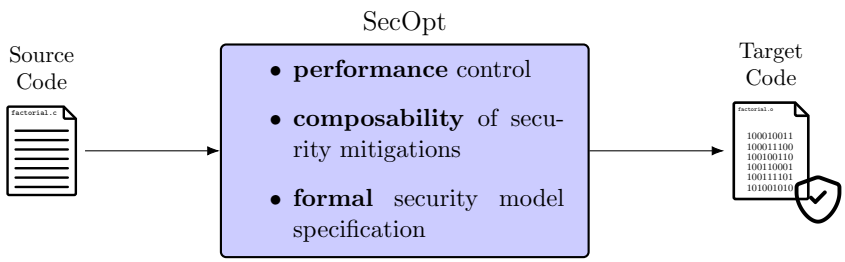


Figure 3.1: High-level view of SecOpt

providing guarantees about preserving the intended properties. However, SecOpt's security analysis, transformations, and the underlying constraint solvers are not verified. Verifying these compilation stages is part of future work for Unison [5] and SecOpt. Code verification uses formal methods to prove software properties or identify violations of these properties in the code. To ensure that the generated code against timing side channels satisfies the constant-resource property, we use external static-analysis tools that output the over- and under-approximation of the execution time. This verification stage increases our trust in SecOpt's code generation. Furthermore, we investigate the constant-time property in WebAssembly programs using relational symbolic execution, which generalizes symbolic execution to prove relational properties [60].

Figure 3.1 shows a high-level view of SecOpt. SecOpt takes as input a program in a high-level language, such as C and C++, and outputs a secure binary-code implementation. The following sections describe the approach of this dissertation to protect binary code against code-reuse and side-channel attacks.

Fine-Grained Software Diversification

To enable code diversification, SecOpt uses the transformation search space that the constraint model allows. In particular, there are multiple solutions to the constraint model of SecOpt that satisfy the model of the program semantics, the target processor model, and the low-level transformations. These solutions may be optimal according to the cost model or suboptimal. To generate diverse solutions, we need to define the a *distance measure*, δ, which is a constraint between two alternative solutions to the problem and measures how *different* these solutions are. The problem definition is the following:

Definition 1 *Diverse Code Generation: Consider a program p and a set, S, of program implementations of p, $p_i \in S$, which are functionally equivalent ($\sim$) with the original program, $\forall p_i \in S.\ p \sim p_i$, and each other, $\forall p_i, p_j \in S.\ p_i \sim p_j$. Furthermore, these program implementations differ from each other based on a distance function δ, namely: $\forall p_i, p_j \in S.\ \delta(p_i, p_j)$.*

In this definition and later in this section, we define δ as a predicate that is true when the compared solutions are different and false otherwise. This simplification implies that our model assumes that two program implementations are different when they differ by one model variable, which decides the register assignment or the program schedule. Note that this constraint enforces that the program implementations are different but does not restrict how different these implementations are. The actual implementation of our approach allows control over the value of the distance function to enforce more diversity among the solutions.

To generate highly diverse and optimized code, SecOpt uses a local-search method, LNS, to navigate in the program's search space around the optimal solution. More specifically, SecOpt finds first the optimal solution, y_{opt}, according to

the cost model and then uses LNS to find alternative solutions around this optimal solution. Initiating the search starting from the optimal solution allows the solver to locate highly optimized solutions quickly. To control the performance overhead in the generated variants, we introduce a constraint C_{opt} that restricts the cost function to have at most $g\%$ overhead. At the same time, SecOpt introduces a distance measure that forces the solutions to differ from each other. In particular, SecOpt uses an iterative algorithm as follows:

```
1  y ← y_opt;                      // Start with optimal solution
2  S ← {y_opt};                    // Add optimal solution to S
3  C' ← C ∪ {Δ(y_opt), C_opt};     // Add constraints
4  while cont_cond()               // Iterate until limit
5    y ← solve_LNS(y, C');         // Find next solution
6    S ← S ∪ {y};                  // Add new solution to S
7    C' ← C' ∪ {Δ(y)};             // Add diversity constraint
8  return S                        // Return set of diverse solutions
```

At line 1, the algorithm copies the optimal solution to the current solution to proceed in the iteration. At line 2, the algorithm adds the optimal solution to the set of solutions, S. Then, at line 3, the algorithm updates the set of constraints with two constraints. First, distance constraint $\Delta(y_{opt})$ ensures that all future solutions to the problem y' will differ from y_{opt}, namely $\delta(y', y_{opt})$. The second constraint C_{opt} restricts the solutions to have at most $g\%$ performance overhead. Lines 4 to 7 implement the iterative algorithm that proceeds until it reaches a condition, such as a time limit or the maximum number of variants (line 4). At line 5, the algorithm takes the previous solution and finds a new solution using LNS. Subsequently, the algorithm inserts the new solution to set S (line 6), and finally, the algorithm updates the set of constraints C' so that the future solutions are different from the newly found solution.

Automatic fine-grained software diversification is effective against code-reuse attacks, however, software diversification approaches against side-channel attacks lead to a large overhead [14]. Instead, to secure the code against side-channel attacks, SecOpt preserves software mitigations against side-channel attacks.

Optimizing Side-Channel Mitigations

The underlying constraint-based compiler backend of SecOpt generates highly-optimized binary code. However, these optimal solutions do not necessarily satisfy security constraints. Enforcing the generation of secure solutions requires extending the constraint model to include security constraints. The security properties that SecOpt implements are software masking against power side channels and the constant-resource property against timing side channels. The selected mitigations depend on the underlying compiler-backend's transformation space and our goal to generate highly optimized code for resource-constrained devices. In particular,

both software masking and constant-resource programming introduce performance overhead that a combinatorial approach may reduce [13, 61].

To investigate the feasibility and adequacy of a secure optimizing approach, we express each mitigation as part of the constraint model of the underlying constraint-based compiler backend. Definition 2 defines the problem statement for secure code generation.

Definition 2 *Secure Constraint-Based Optimization: Given a constraint problem* $P = \langle V, U, C, O \rangle$ *that describes a constraint-based compiler backend in CP, we define constraints* C_{sec}*, such that problem* $P_{sec} = \langle V, U, C \cup C_{sec}, O \rangle$ *satisfies solutions that mitigate the relevant vulnerabilities.*

An important initial step for generating the input data for the security constraints, C_{sec}, is security analysis of the code. This analysis takes as input a security policy that defines which program variables are secret, public, or random to identify possible vulnerabilities in the program.

Software Masking

SecOpt generates code that protects against transitional leakages due to hardware-register reuse and memory-bus reuse. Before generating the constraints, SecOpt performs static analysis to identify possible sources of leaks in the code. In particular, SecOpt adapts the type-inference algorithm from Wang et al. [25] to extract a set of program variables and operations that may lead to transitional leakages. This analysis returns a set of pairs of program variables, $RPairs$, and a set of pairs of memory operations, $MPairs$. Each pair in $RPairs$ leaks secret information when a hardware register transitions from one value to another. Analogously, a pair of operations in $MPairs$ leaks secret information if the two operations write to the memory bus subsequently. The constraint model that preserves software masking consists of constraints that use $RPairs$ and $MPairs$ to restrict the register allocation and instruction scheduling of the code generation. In particular, SecOpt extends the compiler-backend constraint model with the following set of constraints to protect against register-reuse leakages:

`conflict_rassign(`$RPairs$`):`
$\forall \mathtt{t}_1, \mathtt{t}_2 \in RPairs.\ r(\mathtt{t}_1) = r(\mathtt{t}_2) \implies \neg \mathtt{subseq}(\mathtt{t}_1, \mathtt{t}_2)$

Constraint `conflict_rassign` implies that if the temporaries in a pair of program variables that appears in $RPairs$, $\mathtt{t}_1, \mathtt{t}_2$, are assigned to the same register, $r(\mathtt{t}_1) = r(\mathtt{t}_2)$, then they should not be assigned subsequently (`subseq`). Here, constraint (`subseq`) considers both directions, namely $\mathtt{t}_1$ is assigned to $r(\mathtt{t}_1)$ immediately before $\mathtt{t}_2$ is assigned to the same register and vice versa.

Figure 3.2 shows one vulnerable (Figure 3.2a) and one secure (Figure 3.2b) implementation of the code in Figure 2.7. In both figures, the input variables, `pub`, `key`, `mask`, are stored in registers `r0`, `r1`, and `r2`, respectively. In these figures, we denote public values as (`p`), secret values as (`s`), and random values as (`r`). The

```
1  @ r0: pub (p), r1: key (s), r2: mask (r)
2  eors r2, r1 @ r2:r->r^s
3  eors r0, r2 @ r0:p->p^s^r
4  bx   lr
```

(a) Insecure

```
2  eors r1, r2 @ r1:s->s^r
3  eors r0, r1 @ r0:p->s^r^p
4  bx   lr
```

(b) Secure

Figure 3.2: Two program implementations of Figure 2.7 for ARM Cortex M0

```
1  @ r0: [pub] (p), r1: [key] (s), r2: [mask] (r), r3: [res]
2  ...
3  ldr  r1, [r1] @ M:p->s,r1:p->s
4  ldr  r2, [r2] @ M:s->r,r2:p->r
5  eors r2, r1   @ r2:r->r^s
6  ldr  r0, [r0] @ M:r->p,r0:p->p
7  eors r2, r0   @ r2:r^s->r^s^p
8  str  r2, [r3] @ M:p->r^s^p
9  ...
10 ...
11 ...
```

(a) Insecure (LLVM)

```
2  ...
3  ldr  r2, [r2] @ M:p->m,r2:p->m
4  str  r2, [sp] @ M:m->m
5  ldr  r2, [r1] @ M:m->k,r2:m->k
6  ldr  r1, [sp] @ M:k->m,r1:p->m
7  eors r2, r1   @ r2:k->k^m
8  ldr  r0, [r0] @ M:m->p,r0:p->p
9  eors r0, r2   @ r0:k^m->k^m^p
10 str  r0, [r3] @ M:p->m^k^p
11 ...
```

(b) Secure (SecOpt)

Figure 3.3: Two program implementations of the equivalent of Figure 2.7 using pointers for ARM Cortex M0

comments next to the code denote the value transitions (v_{old}->v_{new}) in a register, e.g. *r0*. The first instruction at line 2, *eors*, takes two operands r2 and r1, performs the exclusive-OR, and writes the result in register r2 (two-address instruction). This operation implies that there is a value transmission in register r2, from value mask to value mask ^ key, which leads to a hamming-distance leakage (mask ^ key) ^ mask, which is equal to key(circled in red). This leakage implies that the implementation leaks information about value key to a power side-channel attacker. The rest of the code does not lead to any leaks. To generate a secure implementation for the code in Figure 2.7, SecOpt changes the operand order of the *eors* instruction, as shown in Figure 3.2b (line 2). The new implementation leads to a leak of value mask, which is random.

Similarly, we add the following constraints to protect against memory-bus sharing leakages:

$$\texttt{conflict\_order}(MPairs): \quad \forall \texttt{o}_1, \texttt{o}_2 \in MPairs.\ \neg \texttt{msubseq}(\texttt{o}_1, \texttt{o}_2)$$

Constraint conflict_order implies that two operations $\texttt{o}_1, \texttt{o}_2$ should not be scheduled subsequently (msubseq). Constraint (msubseq) considers both directions, namely, $\texttt{o}_1$ before $\texttt{o}_2$ and vice versa.

Figure 3.3 shows one vulnerable implementation (Figure 3.3a) generated by

Unison and one secure implementation (Figure 3.2b) generated by SecOpt. This implementation is a variant of the code in Figure 2.7, where the inputs and the output are passed as references. In both figures, the addresses of the input variables, `pub`, `key`, `mask`, are stored in registers `r0`, `r1`, and `r2`, respectively. To mark the value transitions in memory M and registers, e.g. *r0*, we denote all public values with `p`, including the addresses to the program input variables and the initial value in the memory bus. At line 3, the implementation loads the secret value from the address in register `r1` to register `r1`. These instructions lead to two leaks, one register-reuse transitional leakage in register `r1` and one memory-bus transition leakage (circled). That is, the initial value of `r1` is public (address to the value `key`), and the initial value in the memory bus is public, which we assume in this work. At line 4, the code loads the mask, which leads to no leaks. However, at line 5, we have a register-reuse leakage as in the previous example (Figure 3.2a) because the result of the exclusive-OR operation is stored at the same register as value `mask`, which leads to a leak related to the value of `key` (circled). To generate secure code, SecOpt needs to schedule the memory operations in a specific order ensure that there are no register-reuse and memory-bus leaks. To do that, SecOpt uses a stack slot to store the random value `mask` (line 4), then load the secret value (line 5), and finally load the random value again (line 6). One of these load operations may be optimized away, however, Unison does not allow the allocation of unused variables. Forcing the constraint model to consider dead copies is part of future work.

Constant-Resource Code

SecOpt aims at generating constant-resource code, where secret-dependent branches are balanced. Although in some cases it is possible to generate constant-resource code, in other cases, this is not possible because the source code is not balanced or because front- and middle-end compiler transformations have removed the balancing (dead) code. To enable mitigation of such code, we perform two program transformations, 1) add an empty basic block that the solver will fill with NOP operations and 2) add a basic block that consists of instructions from the basic block to balance that are deactivated (only in the case of one basic block).

Figure 3.4 demonstrates our transformations for a simple program that returns one if the input variables `key` and `pub` are equal and zero, otherwise. The first version of the transformed code (Figure 3.4a) adds an empty `else` block in the code, while the second version (Figure 3.4b) copies the assignment of `mask` to one in the `if` block to a newly defined `else` block.

To preserve the constant-resource property, the constraint model enforces all paths starting from a secret-dependent control-flow operation to have equal latency. Identifying these paths requires a prior analysis, which first identifies all secret-dependent control-flow instructions and, subsequently, finds all program paths that begin from these instructions. This analysis generates a set of lists of paths, where each list, $paths_{sec}$, consists of a set of paths that start from the same secret-

```
1 u32 check_bit(u32 pub, u32 key) {
2   u32 t = 0;
3   if (pub == key)
4     t = 1;
5   else
6     // nop;
7   return t;
8 }
```

(a) Add Empty Block

```
2   u32 _t, t = 0;
3   if (pub == key)
4     t = 1;
5   else
6     _t = 1;
7   return t;
8 }
```

(b) Copy Unbalanced Block

Figure 3.4: Balancing transformations

dependent control-flow instruction. The following constraint enforces the same execution time for each path in $paths_{sec}$.

`balance_blocks(`$paths_{sec}$`):`
$\forall p_1, p_2 \in paths_{sec}. \sum_{b \in p_1} \texttt{cost}(b) = \sum_{b \in p_2} \texttt{cost}(b)$

Figure 3.5a shows an implementation of the code snippet in Figure 3.4b in assembly code for processor ARM Cortex M0. First, the code (line 3) copies value zero to register `r2` (variable `t` in Figure 3.4b). At line 4, the implementation compares the two input variables, and if they are not equal (taken branch), the control flow goes to line 9; otherwise, it continues to line 7. These branches depend on the secret value in register `r1`, and hence, the attacker should not distinguish the execution time regardless of the branch destination. The not taken branch starting at line 7 copies value `#1` to the return register `r0` (1 cycle) and then branches to the exit block `.LBB0_3` (3 cycles). If the branch is taken, then there is an additional overhead of two cycles because the processor needs to calculate the target address and/or the comparison result. The taken branch starting at line 9 copies the content of variable `r2` to the return register (1 cycle) and assigns value `#1` to register `r1` (1 cycle), which corresponds to the unused temporary `_t` in Figure 3.4b. In total, each branch takes four cycles.

Composability of Security Mitigations

One of the major advantages of constraint-based approaches is composability of multiple properties in the form of constraints. The constraint-based approach of SecOpt allows the combination of multiple mitigations against different attacks. As we show in Publication 4, fine-grained software diversification may conflict with both constant-resource programming and software masking. SecOpt allows combining multiple mitigations while preserving the properties of these mitigations at the same time. This work focuses on fine-grained software diversification and software masking or constant-resource programming. The problem statement in this problem is the following:

```
1  @ r0: pub, r1: key
2  @ BB#0:
3    movs  r2, #0
4    cmp   r1, r0
5    bne   .LBB0_2
6  @ BB#1:
7    movs  r0, #1
8    b     .LBB0_3
9  .LBB0_2:
10   mov   r0, r2
11   movs  r1, #1
12 .LBB0_3:
13   bx    lr
14 ...
15 ...
```

(a) Secure variant 1

```
2  @ BB#0:
3    movs  r2, #0
4    cmp   r1, r0
5    bne   .LBB0_2
6  @ BB#1:
7    movs  r3, #1
8    mov   r0, r3
9    b     .LBB0_3
10 .LBB0_2:
11   mov   r3, r2
12   mov   r0, r3
13   movs  r3, #1
14 .LBB0_3:
15   bx    lr
```

(b) Secure variant 2

Figure 3.5: Two program implementations that preserve constant-resource property for ARM Cortex M0

Definition 3 *Secure Constraint-Based Code Diversification: Given problem* $P = \langle V, U, C, O \rangle$ *that describes a constraint-based compiler backend, we add constraints* C_{sec}*, that protect against side-channel vulnerabilities,* $P_{sec} = \langle V, U, C \cup C_{sec}, O \rangle$*. We aim at generating a set,* S*, of solutions to the constraint problem,* P_{sec} *that are different with each other,* $\forall p_i, p_j \in S.\delta(p_i, p_j)$ *and are functionally equivalent* $\forall p_i, p_j \in S.p_i \sim p_j$*.*

To achieve these goals, we combine the constraints and methods we discussed in the previous sections. In particular, the combined approach extends the set of constraints with constraints `balance_blocks` or `conflict_rassign` and `conflict_order` and then, uses LNS to find diverse program solutions. Figure 3.5 shows two program variants that preserve the constant-resource property based on the copy transformation in Figure 3.4b. The two variants balance the execution time of the two paths that split at the branch instruction `bne`. In both implementations, the first path consists of basic block one, `BB#1`, and the second consists of basic block two, `.LBB0_2`. The two variants differ with regard to the register assignment, including copying a result from one register to another (lines 7-8 and 11-12). The two variants differ in the code size, which may result in different addresses in the binary code. This relocation, together with different register assignment, e.g. `movs r1, #1` (line 11) and `movs r3, #1` (line 13), respectively, alter the semantics of possible gadgets and may break code-reuse gadgets that consider either of the variants. Performing code re-randomization may further harden the implementation against both attacks.

Verification of Security Properties

Code verification on binary code tests security properties in the binary code to verify the preservation of these security properties or identify security vulnerabilities. Typically, code verification approaches rely on formal methods to provide guarantees that the requested security properties hold. There are different code-verification methods, including symbolic execution that uses constraint solving to prove the requested properties and abstract interpretation that depends on a formal abstraction of the program values and semantics.

SecOpt generates secure code against timing side channels. After code generation, we verify the constant-resource property in the generated code to increase the trust in SecOpt. In particular, to ensure that the constant-resource property holds in the generated code, we verify this property using external static-analysis tools. More specifically, for each processor we target, we use a Worst Case Execution Time (WCET) tool to verify that secret-dependent paths take the same time. WCET analysis generates an overapproximation (and optionally an underapproximation, Best Case Execution Time (BCET)) of the execution time of a program. WCET analysis is an important topic in embedded systems because it is an input to schedulability analysis, which tests whether the system under analysis meets its deadlines. To verify the constant-resource property, we derive the WCET and the BCET using symbolic values for the secret input variables of the function under analysis and compare these values. If the two values are equal, we have an indication that the execution time does not depend on these secret values. If the two values are not equal, then we need to investigate further if the execution time depends on secret values, or if the difference between WCET and BCET is due to overapproximation of the analysis.

In some cases, secure compilation uses verification approaches to accept compiler-generated code if the verification test succeeds or reject the code if the process identifies security vulnerabilities [62]. In this work, we verify the constant-time property in WebAssembly. Our approach uses relational symbolic execution, a method that performs symbolic execution on a program with two input states or two programs with the same input state [60]. For testing the constant-time property, we execute symbolically two executions of the same program with two input states. The two states differ with regard to the secret input values, which take different initial symbolic values. Then, the analysis executes the program symbolically using one state that considers both executions. The analysis is possible by using paired variables, each consisting of two versions that correspond to the different executions. Given that the two states differ only with regard to the secret values, a potential execution path divergence signifies a timing leakage. Similarly, the analysis needs to ensure that there are no secret-dependent memory operations, which allow cache timing attacks. More formally, we consider $\{\Phi\}\langle(M_1, c)|(M_2, c)\rangle\{\Psi\}$, where Φ is the precondition, namely the security policy that defines which input variables are secret or public. Ψ denotes the verification properties, namely the absence of secret-dependent control-flow instructions and

secret-dependent memory operations. M_1 and M_2 are the initial memory instances, and c is the program instructions. When discovering a vulnerability, the approach returns the vulnerability as a solution to the requirement.

Loop analysis is a challenge in symbolic execution because it may lead to path explosion. Among different methods to deal with loops in symbolic execution are using bounded loop unrolling or loop invariants. The former may lead to increased analysis overhead and unsound analysis, whereas the latter is challenging to perform automatically. Our work considers both unbounded loop unrolling and the generation of a relational invariant.

3.2 Methodology

The topic of this dissertation combines cybersecurity and computer science methods, which include theoretical research and applied experimental research [63].

In the bibliography, there are several compiler-based approaches to tackle security properties. The process of conducting research for each research question starts with a survey in the area to identify challenges in state-of-the-art research. We identify the need for highly optimizing security approaches and evaluating the effect of the sequential composition of multiple mitigations. After defining the problem, we follow an iterative process that consists of three steps, 1) define a hypothesis, 2) implement/extend a research prototype to evaluate the hypothesis, 3) refine the hypothesis based on the results. This process repeats until the final research project is complete. The work towards this dissertation consists of the following incremental steps:

- Design and evaluate an LNS-based algorithm as an extension to Unison to generate highly diverse solutions in Mips. This step is the first part of the development of SecOpt.
- Design and evaluate a new LNS-based algorithm based on the structural decomposition of a function. Evaluate this approach in a whole-program diversification scheme for Mips32.
- Extend SecOpt to optimize code that preserves software masking and supports the ARM Cortex M0 processor. As part of this step, we prove that the proposed constraint model leads to code that does not leak secret information through transitional leakages due to register reuse and memory-bus reuse.
- Extend SecOpt to optimize code that preserves the constant-resource property and combine with diversification.

Apart from these incremental steps towards SecOpt, we investigate a different approach to perform whole-program verification of the constant-time property in WebAssembly.

We evaluate Publications 1 and 2 using benchmark functions from two benchmark suites, MediaBench [64] and SPEC CPU2006 [65] that are popular for evaluating embedded-system and compiler-based approaches. Publication 1 uses 17 small, randomly selected functions from both benchmark suites. The evaluation shows that the proposed LNS-based approach trades scalability for diversity in CP. In addition, the evaluation shows our approach allows the generation of multiple optimal diverse solutions for the majority of the benchmarks. Relaxing the optimality constraint enables more diverse solutions. Publication 2 uses 20 medium-size functions from MediaBench. The evaluation of Publication 2 confirms the results of Publication 1 for larger programs. In addition, Publication 2 presents a whole-program diversification evaluation using a case study from MediaBench, G.721. This application is an implementation of a set of voice compression algorithms. We use a GCC-based tool to link the generated variants for the MIPS32-based Pic32MX microcontroller. This case study shows up to 95% code-reuse gadget diversification or relocation.

Publications 3, 4, and 5 use cryptographic implementations from the real world and previous work that evaluates similar approaches. The evaluation of Publication 3 uses real-world constant-time implementations from diverse libraries, including libsodium [66], HACL* [67], BearSSL [68]. In addition, the evaluation uses known timing vulnerabilities from the literature. This publication presents two approaches to verify constant-time programs. The first approach uses unbound loop unrolling and is able to verify 55 out of 57 implementations. The second approach uses a lightweight relational invariant generation and is able to verify the rest two implementations but fails to analyze many implementations due to the limited precision of the generated invariant.

Publication 4 applies theoretical and experimental research methodology in the evaluation. To ensure security, we prove that the security constraint model implies secure code generation based on our leakage model. In addition, the evaluation in Publication 4 uses twelve benchmark functions that we derive from previous work [25] to perform an empirical evaluation. We evaluate the performance overhead of the generated code and the compilation-time overhead compared with Unison. To evaluate the speedup of our approach compared to other security-aware approaches, we compare with the approach by Wang et al. [25] and LLVM with no optimizations, -O0. This evaluation shows a high speedup of up to three times faster than LLVM -O0 and the work by Wang et al. [25] at the expense of a compilation time increase.

The evaluation of Publication 5 uses the same benchmarks as Publication 4 and an additional set of five benchmarks to evaluate the constant-resource property. These benchmark functions comprise diverse algorithms that may take secret values as inputs and contain secret-dependent control-flow instructions [69]. To verify the security of the generated solutions, we use two WCET tools for ARM Cortex M0 [70] and Mips [71, 72]. These tools calculate the worst- and best-case execution time, and we use them to show that the generated variants do not depend on secret values. The evaluation indicates that property-preserving diversification introduces

diversification-time overhead, however, this does not reduce the effectiveness against code-reuse attacks.

Chapter 4

Related Work

This chapter presents state-of-the-art research in defenses against code-reuse attacks (Section 4.1), and side-channel attacks (Section 4.2). The latter consists of three parts and includes related work on mitigations against timing side-channel attacks, mitigations against power side-channel attacks, and verification approaches for timing side-channel attacks.

4.1 Code-Reuse Attacks Mitigations

Table 4.1: Mitigation approaches against code-reuse attacks

Pub.	Mitigation	InL	OutL	ML	Av.
Abadi et al. [35]	CFI	x86	x86	Bin	✗
Pappas et al. [8]	Div	x86	x86	Bin	✓
Homescu et al. [9]	Div	C, C++	x86	llvm	✓
AVRAND [73]	Div, RR	AVR	AVR	Bin	✓
C-Flat [74]	CFI	ARM	ARM	Bin	✓
CFI CaRE [75]	CFI	ARM	ARM	Bin	✗
Koo et al. [76]	Div, RR	C, C++	x86	llvm	✓
MicroGuard [1]	Div, CFI	C, C++	ARM	llvm/Bin	✗
HARM [77]	Div, RR	ARM	ARM	Bin	✓
FH-CFI [78]	HWCFI	ARM	ARM	Bin	✗
SecOpt [17, 18, 21]	Div	C, C++	Mips, ARM	llvm	✓

Code-reuse attacks constitute a serious threat to computer software in both high-end computers [29] and embedded systems [27, 28]. There are two main mechanisms to mitigate code-reuse attacks, CFI [34] and automatic software diversification [7]. CFI includes hardware and software mechanisms to prevent illegitimate

control-flow violations during program execution. On the other end, automatic software diversification hinders code-reuse attacks by introducing uncertainty to the program implementation. This uncertainty affects the location and exact implementation of code-reuse gadgets, which are the building blocks of code-reuse attacks.

Table 4.1 shows a number of representative approaches against code-reuse attacks. The table includes the publication citation (Pub.), the mitigation each approach uses (Mitigation), the input language (InL), the output language (OutL), the mitigation level (ML), which is either at binary code (Bin) or a compiler (e.g. llvm), and finally, the availability of the respective artifact (Av.). For the availability field (Av.), ✓indicates that the artifact is available, whereas ✗indicates that the main author of this dissertation was not able to find the artifact.

Table 4.1 shows a set of approaches that use CFI against code-reuse attacks [35, 74, 75, 1, 78]. The majority of these approaches target embedded systems, including C-Flat [74], CFI CaRE [75], and FH-CFI [78], and MicroGuard [1] that target ARM systems. One of these approaches, MicroGuard [1] combines CFI with software diversification against code-reuse attacks. In general, CFI approaches lead to higher execution-time overhead than software diversification approaches [34, 7].

Automatic software diversification (Div in Table 4.1) is another approach against code-reuse attacks. ASLR is a coarse-grained software diversification method that randomly selects the address space of key data areas, such as the address of dynamic libraries. ASLR is the most widely-used diversification method, and its effect on performance is insignificant. However, ASLR leads to low entropy, which enables brute-force code-reuse attacks [79]. Fine-grained software diversification, which includes diversification at the function- or instruction-level of the program, provides improved protection against code-reuse attacks.

Pappas et al. [8] perform fine-grained software diversification at the binary level and apply zero-cost transformations, namely register randomization, instruction schedule randomization, and function shuffling. However, they do not evaluate the actual performance overhead of their approach. In contrast, SecOpt uses a cost model to calculate the performance overhead and allows diversification with no performance degradation. Also, SecOpt enables more transformations including NOP insertion, register copying, spilling, and constant rematerialization. SecOpt may be combined with function shuffling to achieve whole-program diversification [18].

Homescu et al. [9] present a fine-grained diversification approach that inserts NOP instructions to the code. To reduce the introduced overhead, they use profiling information to prioritize NOP insertion in pieces of code that have low execution frequency. Seibert et al. [33] show that static frequency NOP insertion is possible to bypass using side-channel information. SecOpt is also able to control the introduced overhead by using a static cost model, while it allows targeted diversification in code without introducing performance overhead. The latter is possible because SecOpt uses a larger variety of transformations than NOP insertion.

The introduction of advanced code-reuse attacks allows for deciphering the diversification scheme by using a memory vulnerability to read the program mem-

ory [31, 32], or using timing information [33]. These attacks give rise to re-randomization (RR in Table 4.1) approaches. Re-randomization typically switches between different program variants at specific time intervals [76] or at different events, such as at reboot time [73]. Re-randomization may introduce additional runtime performance overhead that may be low, such as HARM [77] that introduces 5% overhead. SecOpt may be used in a re-randomization scheme at boot time against attacks, such as BROP [32]. We leave the evaluation of such an approach as future work.

4.2 Defending Side-Channel Attacks

Side-Channel attacks constitute a serious threat to cryptographic implementations. There are different side-channel attacks, including timing, power, electromagnetic, and sound side-channel attacks. This dissertation focuses on timing and power side-channel mitigations. The next sections present a set of representative mitigation approaches against these attacks.

Defending Timing Side-Channel Attacks

In this section, we discuss methods to mitigate and verify timing side-channel attacks.

Timing Side-Channel Mitigations

Table 4.2: Mitigation approaches against timing side-channel attacks

Pub.	Attack	Mitig.	InL	OutL	ML	Av.
Crane et al. [14]	TSC	Div	C, C++	x86[a]	llvm	✗
Raccoon [15]	TSC	Obf	C, C++	x86	llvm	✗
Fact [80]	TSC	CT	DSL	C	Custom	✓
HACL* [81]	TSC, MC	CT	DSL	C	F_{low}	✓
Jasmin [82]	TSC, MC	CT	DSL	x86	Custom	✓
Winderix et al. [61]	TSC, IL	BB	C, C++	MSP430	llvm	✓
Constantine [83]	TSC	CT	C, C++	x86	llvm	✓
Crow [84]	TSC	Div	C, C++	Wasm	llvm	✓
Vu et al. [4]	VBL, TSC	SM,CT	C, C++	ASM	llvm	✗
SecOpt [21]	TSC, TBL, CRA	Div, SM/BB	C, C++	Mips, ARM	llvm	✓

[a]The evaluation targets x86, however the method applies to other architectures

There are different approaches to mitigate timing side-channel attacks. These approaches either eliminate the secret-dependent timing differences in the program execution [85] or obfuscate the timing profile of the program to reduce the ability of the attacker to identify secret values [15, 14] Table 4.2 shows a set of representative mitigation approaches against timing side-channel attacks (TSC). The table includes the publication (Pub.), the attack, the approach mitigates (Attack), the mitigation the approach applies (Mitig.), the input language (InL), the output language (OutL), the mitigation level (ML), which is a custom compiler (custom), a specific compiler (F_{low}), or the LLVM compiler (llvm). The last field (Av.) indicates that the artifact is available (✓) or not available (✗).

There are two main approaches that eliminate secret-dependent time differences, 1) cryptographic constant-time (CT) programming discipline [85], which eliminates secret-dependent timing differences by rewriting the code, and 2) constant-resource (BB) programming [45], which instead balances the different execution paths to take the same time.

The constant-time programming discipline replaces secret-dependent branch and memory operations with constant-time equivalent that make use of logic operations. HACL* [81] is an approach to generate code that is constant time and memory safe against memory corruption (MC). The output code is in C, and thus, there is another compilation step from C to assembly code that may use CompCert [86], a verified compiler. Jasmin [82] is low-level optimizing cryptographic Domain Specific Languages (DSLs) that generate efficient constant-time code for cryptographic implementations. The main drawback of Jasmin is that the input code is written in a low-level language, which requires re-implementing legacy cryptographic algorithm implementations and acquiring a deep understanding of low-level code.

Constant-resource programming is a more relaxed mitigation approach compared to the constant-time programming discipline. In particular, constant-resource programming does not require the absence of secret-dependent branches. Instead, it allows balancing secret-dependent branches with NOP instructions to hinder the attacker from identifying the selected execution path. Winderix et al. [61] implement an approach to balance secret-dependent branches on MSP430. Their approach protects against both timing attacks and interrupt-latency (IL) side-channel attacks. SecOpt focuses on timing attacks and achieves balancing secret-dependent branches with up to 70% overhead.

Constantine [83] is a different approach that achieves constant time by automatically linearizing code. Their approach introduces large overhead of up to five times. Raccoon [15] uses obfuscation (Obf) to hide secret-dependent leaks. The main disadvantage of this approach is high performance overhead of up to 16 times. Crane et al. [14] mitigate timing side channels using fine-grained code diversification (Div) by inserting memory NOP operations. However, their approach may lead to up to 8 times performance overhead. The main disadvantage of these approaches is high execution-time overhead compared to SecOpt that introduces up to 70% overhead.

Verification

Code verification is a way to identify timing vulnerabilities in programs. Almeida et al. [85] use product programs to verify constant-time programs in C. Vale [87] verifies the correctness, safety, and security of binary code in ARM and x86. Among other security properties, Vale preserves the constant-time property using taint analysis. Binsec/Rel [24] performs relational symbolic execution [60] to verify constant-time program in binary code.

The verification approach of this thesis, Vivienne [19] uses also relational symbolic execution to verify the constant-time property in WebAssembly code. In addition, Vivienne implements a lightweight invariant inference approach. Bastys et al. [88] is another approach that uses concolic execution to verify the constant-time property in WebAssembly.

Power Side-Channel Mitigations

Table 4.3: Mitigation approaches against power side-channel attacks

Pub.	Attack	Mitig.	InL	OutL	ML	Av.
Eldib and Wang [89]	VBL	SM	DSL	-	Custom	✗
Papagiannopoulos and Veshchikov [13]	TBL	SM	AVR	AVR	Binary	✓
Besson et al. [62]	IFL	-	C	ASM	CompCert	✗
Wang et al. [25]	TBL	SM	C, C++	ASM	llvm	✓
Athanasiou et al. [26]	TBL	SM	ARM	ARM	Binary	✓✗
Vu et al. [4]	VBL, TSC	SM,CT	C, C++	ASM	llvm	✗
Rosita [47]	TBL	SM	ARM	ARM	Binary	✓
SecOpt [20]	TBL	SM	C, C++	Mips, ARM	llvm	✓

Power side-channel attacks record the power traces of a computer to extract secret values, such as cryptographic keys. There are different mitigation approaches to hinder power side channel attacks during the program execution. An approach to mitigate power side channels is to randomize the secret data, so that the power traces do not reveal secret information to the attacker. Software masking (SM), uses the exclusive-OR operation, $\oplus$, to mix the secret value with a randomly generated value. This randomly generated value, or mask, allows the randomization of the secret value and requires the same mask to decipher.

We consider different types of power leakage, Value-Based Leakage (VBL) and Transition-Based Leakage (TBL). VBLs appear when secret values are not masked,

i.e. public values known to the attacker interact with secret values [89, 4], whereas, TBL appear when fundamental hardware structures, such as hardware registers, memory cells, and memory bus, leak information by transitioning from one value to another. The absence of VBLs does not guarantee the absence of TBLs, whereas the opposite is true.

Table 4.3 shows a set of representative mitigation approaches against power side-channel attacks. For each of the approaches, Table 4.3 shows the publication (Pub.), the attack the approach mitigates (Attack), the mitigation the approach applies (Mitig.), the input language (InL), the output language (OutL), the mitigation level (ML), which is either binary or a compiler, like CompCert or LLVM (llvm). The last field (Av.) indicates that the artifact is available (✓), not available (✗), parts of the artifact are missing (✓✗).

The approaches by Papagiannopoulos and Veshchikov [13] and Rosita [47] are processor specific, focusing on AVR and ARM Cortex M0, respectively. They mitigate different types of TBLs, including register-reuse leakage, memory-reuse leakage, and memory-bus-reuse leakage, which are detected in a specific processor implementations.

Wang et al. [25] take as input masked code and use a more generic type-inference-based approach [90] to identify possible register-reuse leaks and subsequently mitigate them. The main disadvantage of this approach is that it does not generate highly optimized code. SecOpt follows a similar type-inference-based approach to optimize masked code with no register-reuse leaks and memory-bus reuse leaks.

Athanasiou et al. [26] use the same type-inference approach to find and mitigate possible register-reuse leakages. SecOpt generates code that is free from register-reuse leaks and memory-bus reuse leaks.

4.3 Secure Compilation and Optimization

Popular languages like C enable security vulnerabilities, such as memory corruption and many undefined behaviors [3]. Compiler development and research focus mostly on functional correctness in accordance with the language specification. In addition, general-purpose compilers focus on optimizing the performance efficiency or the size of the code, however, they rarely consider security properties [3]. Therefore, important compiler algorithms and heuristics are designed with performance and code size in mind and not security. With the advent and popularity of the Internet and recently the IoT devices, where multiple computers connect to each other, security has become a major concern. Secure compilation is a field that aims at generating secure code, where performance is a secondary aspect.

Table 4.3 presents approaches that combine security features with highly optimized code. Jasmin [82] is an approach that generates secure and optimized code. The main disadvantage of Jasmin is that it uses a low-level DSL, that requires writing cryptographic code in a new assembly-like language.

Vu et al. [91] present an approach that prevents compiler-introduced vulnerabilities in LLVM. Vu et al. [91] generate highly optimized code that preserves security properties, such as software masking (SM) against VBLs and the constant-time property for timing side channels (TSC). Unfortunately, the artifact for their approach is not available.

A different approach by Besson et al. [62] proves that the compiler preserves security properties. In particular, they show that two optimization passes in CompCert [86] preserve information-flow properties at function entries and exits.

Chapter 5

Summary of Publications

The following sections (Sections 5.1 to 5.5), provide a summary of each paper that is included in this dissertation

5.1 Publication 1: Constraint-Based Software Diversification for Efficient Mitigation of Code-Reuse Attacks

Fine-grained software diversification is an approach that is effective against code-reuse attacks. Related work focuses on high-end computer architectures, with little focus on embedded devices, although, code-reuse attacks target embedded devices [28, 29, 30]. An additional advantage of fine-grained software diversification is its low introduced overhead, which makes it suitable for resource-constrained devices. Unfortunately, many related approaches do not control the introduced performance overhead.

This publication presents a fine-grained software diversification approach that uses CP to enable control over the introduced performance overhead. A wide range of low-level program transformations in the underlying compiler backend enables fine-grained diversification that targets Mips32, an embedded-system architecture. The underlying constraint-based compiler backend generates optimized code with regard to performance and code size. To achieve high diversity and scalability, this publication proposes LNS, a local-search-based heuristic, which attempts to find solutions to the constraint model that correspond to machine-code implementations. The evaluation of the algorithm on 17 small functions from MediaBench and SPEC CPU 2006 shows that the presented algorithm enables the generation of program variants that are different from each other with an acceptable diversification time. The available diversification transformations allow the generation of zero-overhead variants, which corresponds to highly optimized function variants.

5.2 Publication 2: Constraint-Based Diversification of JOP Gadgets

Publication 1 presents an algorithm that is efficient and effective for small functions that represent around 24% of the total set of functions in MediaBench. This publication presents a different algorithm that allows the diversification of larger functions using structural decomposition. This algorithm first solves a subproblem that consists of the inter-block program-variable assignments and then, for each basic block, it generates multiple solutions that can be combined to generate diversified program variants. This publication also presents a distance measure that is adjusted to the code-reuse attack properties. The evaluation uses 20 functions from MediaBench that cover 96% of the function size in the bench suite. The evaluation shows that the global LNS-based algorithm is more effective against code-reuse attacks but scales up to around 60% of the functions in MediaBench, whereas the decomposition-based algorithm scales to up to 93% of the functions. This publication evaluates also the effect of whole-program diversification on a case study. This case study shows that the proposed diversification algorithms have high effectiveness against code-reuse attacks while performing additional randomization steps, such as function shuffling, improves the effectiveness of diversification measured by gadget relocation.

5.3 Publication 3: Vivienne: Relational Verification of Cryptographic Implementations in WebAssembly

Timing side-channel attacks constitute a serious threat to the security of cryptographic implementations. Most languages and many algorithm implementations are vulnerable to side-channel attacks, including WebAssembly, a new language for the web, which is portable, efficient, and features security properties. The characteristics of WebAssembly make it a suitable choice for implementing cryptographic libraries with multiple cryptographic implementations already available.

Relational verification is an efficient approach for verifying programs that follow the constant-time policy and/or locate constant-time violations. Vivienne presents an approach that uses relational symbolic execution to identify timing vulnerabilities in WebAssembly. Loop analysis is the main bottleneck of symbolic execution. To deal with this, this publication proposes an approach to automatic relational invariant generation.

The evaluation uses 57 cryptographic library implementations and shows that relational symbolic execution with loop unrolling is efficient for the verification of the constant-time property, when the loop bounds of the analyzed program are not very high. For the benchmarks that contain large loop bounds, relational invariant generation is more effective than unbound loop unrolling. However, sometimes the automatic relational invariant does not capture the loop bounds, which results in low effectiveness in analyzing compiler-generated code.

5.4 Publication 4: Securing Optimized Code Against Power Side Channels

Power side channels are a serious threat to cryptographic libraries. Power attacks typically require access to the victim's physical location and record the power consumption of the target machine in time using devices, such as an oscilloscope. These attacks are very powerful because they can identify small variations in the power consumption of the executing program.

Software masking is a software mitigation against power side-channel attacks, which hides secret values from the power traces using randomly generated variables. These random values statistically remove the dependencies of the secret values from the power traces.

Secret-dependent transitional effects, for example, when hardware registers or the memory bus transition from one value to another, may leak secret information through power side channels. Although the source code of a program may be masked correctly, the compiler may invalidate these transformations by, for example, reusing hardware registers. This publication presents a compiler-based approach that generates highly optimized code that mitigates power side-channel attacks. The paper presents a formal proof that the proposed constraint model is correct with regard to the leakage model. The evaluation of the approach on twelve masked programs targeting two embedded architectures, ARM Cortex M0 and Mips, shows that our approach leads up to 13% performance overhead compared to optimal non-secure compilation and a geometric mean speedup of approximately three compared to other secure-compilation approaches.

5.5 Publication 5: Thwarting Code-Reuse and Side-Channel Attacks in Embedded Systems

Security protection of a computing system typically requires application of multiple mitigations against different attacks that may affect the system. Applying these mitigations sequentially may lead to mitigation conflicts, when the latest applied mitigation reverts or invalidates the changes of the previously applied mitigations. Embedded systems have additional constraints that derive from resource limitations, such as battery life, and thus, overall resource estimation is of major importance.

Publication 5 concerns thwarting code-reuse attacks and side-channel attacks in embedded systems. An efficient mitigation against code-reuse attacks is fine-grained code diversification, while typical mitigations against power and timing side-channel attacks include low-level code transformations. Both fine-grained code diversification and side-channel mitigations operate at the low-level implementation of the input code. The evaluation runs on 15 benchmark programs derived from previous work against side-channel attacks. This publication shows that 1) fine-grained software diversification may break side-channel mitigations, 2) a combined

mitigation against both code-reuse attacks and side-channel attack is feasible but with increased compilation overhead, and 3) there is no clear negative effect of the effectiveness of diversification against code-reuse attacks, when protecting also against side-channel attacks.

Chapter 6

Conclusion and Future Work

This chapter presents the conclusion of this dissertation (Section 6.1) and discusses future research directions (Section 6.2).

6.1 Summary of Contributions

This dissertation presents a combinatorial approach to secure code generation. This work features three properties: performance awareness, composability, and formalisation. Below we summarise the answers to the research questions of this dissertation and discuss the obtained research results.

RQ1: How feasible and effective is performance-aware constrained-based software diversification against code-reuse attacks?

Our experiments show that performance-aware constraint-based software diversification can effectively diversify code-reuse gadgets. The proposed algorithm allows for the efficient generation of highly diverse solutions, while it controls the generated performance overhead. Our approach scales to up to medium-sized programs of approximately 500 Machine Intermediate Representation (MIR) instructions. Hence, we believe that our approach constitutes an important building block for whole-program diversification or re-randomization to provide high effectiveness against code-reuse attacks.

RQ2: How feasible is secure constraint-based optimization of cryptographic implementations?

Our approach to constraint-based optimization of cryptographic implementation is highly optimizing for small cryptographic functions of up to 100 MIR instructions. The results that generate programs free of transitional leaks show a high speedup

compared to competing approaches and non-optimized code. This is achieved at the expense of compilation time. Constant-resource preservation is effective against timing side channels and generates secure low-level code.

RQ3: How feasible and effective is a combined mitigation against code-reuse attacks and side-channel attacks?

Our experiments show that combining software diversification with software mitigations against side-channel attacks enables the generation of multiple program variants that are secure against side-channel attacks. There is an overhead on the diversification time, however, there is no clear effect on the effectiveness of diversifying code-reuse gadgets and thus hindering code-reuse attacks.

RQ4: How feasible is code verification of binary code against timing side channels?

Our constant-resource verification approach verifies the constant-resource property in all generated programs by SecOpt. Similarly, our approach to relation verification to test the constant-time property in WebAssembly is able to analyze successfully 55 out of 57 real-world implementations consisting of large code bases. In addition, the relational invariant approach is able to analyze the remaining two implementations, while a more precise invariant generation approach may successfully analyze the total set of implementations.

Conclusion

This dissertation developed a constraint-based compiler backend approach that presents a concrete step towards secure-by-design optimized compilation. The main features of this approach are composability of security measures, performance awareness that allows the generation of highly optimized code, and a constraint-based framework that exhibits properties that have been formalised. Code verification is an additional step toward highly trusted code generation. To summarize, this work proposed a novel compiler-based approach to generate highly optimized and secure code against major vulnerabilities that affect security-critical software.

6.2 Future Work

There are several directions for future work that can focus on improvements and extensions to proposed approaches. The extension of the current work could include 1) evaluating our approach against full-fledged attacks, 2) extending the software-masking optimization approach to larger programs by decomposing linearized programs into smaller pieces, and 3) extending SecOpt to support more cybersecurity mitigations. Below we discuss the directions of the future work in detail.

Evaluation against Complete Attacks

This dissertation concerns the automatic and correct-by-design generation of code that satisfies security mitigations. Our work evaluates these approaches using formal methods (see Publication 4), empirical evaluation (see Publication 3 and 5), or statistical properties (see Publication 1 and 2). However, an interesting research direction would be to investigate the effect on full-fledged attacks. Two examples of attacks include code-reuse attacks and power side-channel attacks that have recently received increased interest.

Software diversification provides statistical properties that hinder code-reuse attacks. However, an interesting direction is to design different types of code-reuse attacks, such as ROP and JOP attacks, in Mips and ARM. This research direction may give further insights into how to improve and target diversification towards full-fledged attacks.

Power side-channel attacks have advanced significantly in the recent years, including statistical analysis of the power traces using deep learning. These attacks are powerful and evaluating our approach against such attacks may provide additional insights on extending the mitigation of our approach to further transitional leaks.

Scalability Enhancement

Scalability, namely the ability to analyze large problems is an active and demanding research topic in combinatorial optimization. This dissertation has made clear steps toward increased scalability in Publication 2. In addition, Publication 3 investigates additional solving methods to enhance the scalability of the optimization approach. The introduction of security constraints increases the complexity of the problem and, hence, its compilation time. However, Publication 3 uses linearized input programs that consist of a single basic block. One step towards improving the scalability of the approach in Publication 3 is extending the security analysis to support if statements and loops. This direction may reduce the accuracy of the security analysis and lead to reduced code efficiency.

Cybersecurity Countermeasures

This dissertation concerns mitigations against code-reuse attacks and side-channel attacks. However, there are additional attacks that depend on compiler-generated code.

Memory-Probing Attacks

Memory-probing attacks allow an adversary to read the content of the main memory and/or the register file [92]. One idea is to use SecOpt to reduce the presence of secret values in memory by reducing the live range of registers and stack operations. Overwriting the secret data in memory, including the stack and register after the end

of the live range is an additional transformation towards reducing the capabilities of memory-probing attacks.

Speculation Attacks

Modern processors use speculation to improve the performance of the program execution when the result of a branch is not known. In particular, the processor uses statistical information from previous branches to take a branching decision before the processor calculates the actual branch decision. Speculative execution improves the processor performance when the branch prediction is correct. In case of misprediction, the processor discards the speculatively executed instruction results. However, the processor does not reverse any side effects of the speculative execution, such as cache updates. A compiler-based approach to mitigate these attacks may include padding vulnerable branches with NOPs that delay the speculative execution of instructions that leak secret information. This approach may introduce high performance overhead. Instead, verification of the absence of speculation leaks using relational verification is an effective method to ensure the security of the generated software [93].

References

[1] M. Salehi, D. Hughes, and B. Crispo, "MicroGuard: Securing Bare-Metal Microcontrollers against Code-Reuse Attacks," in *2019 IEEE Conference on Dependable and Secure Computing (DSC)*, Nov. 2019, pp. 1–8.

[2] A. Bendovschi, "Cyber-Attacks – Trends, Patterns and Security Countermeasures," *Procedia Economics and Finance*, vol. 28, pp. 24–31, Jan. 2015.

[3] V. D'Silva, M. Payer, and D. Song, "The Correctness-Security Gap in Compiler Optimization," in *2015 IEEE Secur. Priv. Workshop*, 2015, pp. 73–87.

[4] S. T. Vu, A. Cohen, A. De Grandmaison, C. Guillon, and K. Heydemann, "Reconciling optimization with secure compilation," *Proceedings of the ACM on Programming Languages*, vol. 5, no. OOPSLA, pp. 1–30, 2021.

[5] R. C. Lozano, M. Carlsson, G. H. Blindell, and C. Schulte, "Combinatorial Register Allocation and Instruction Scheduling," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 41, no. 3, pp. 17:1–17:53, 2019.

[6] H. Shacham, "The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86)," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, ser. CCS '07. New York, NY, USA: ACM, 2007, pp. 552–561.

[7] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "SoK: Automated Software Diversity," in *2014 IEEE Symposium on Security and Privacy*, May 2014, pp. 276–291.

[8] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Smashing the Gadgets: Hindering Return-Oriented Programming Using In-place Code Randomization," in *2012 IEEE Symposium on Security and Privacy*, May 2012, pp. 601–615, iSSN: 1081-6011.

[9] A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz, "Profile-guided Automated Software Diversity," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, ser. CGO '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 1–11.

[10] P. Kocher, J. Jaffe, and B. Jun, "Differential Power Analysis," in *Advances in Cryptology — CRYPTO' 99*, ser. Lecture Notes in Computer Science. Springer, 1999, pp. 388–397.

[11] E. Brier, C. Clavier, and F. Olivier, "Correlation Power Analysis with a Leakage Model," in *Cryptographic Hardware and Embedded Systems - CHES 2004*, ser. Lecture Notes in Computer Science. Springer, 2004, pp. 16–29.

[12] P. C. Kocher, "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems," in *Annual International Cryptology Conference*. Springer, 1996, pp. 104–113.

[13] K. Papagiannopoulos and N. Veshchikov, "Mind the Gap: Towards Secure 1st-Order Masking in Software," in *International Workshop on Constructive Side-Channel Analysis and Secure Design*. Springer, 2017, pp. 282–297.

[14] S. Crane, A. Homescu, S. Brunthaler, P. Larsen, and M. Franz, "Thwarting Cache Side-Channel Attacks Through Dynamic Software Diversity," in *Proceedings 2015 Network and Distributed System Security Symposium*. Internet Society, 2015.

[15] A. Rane, C. Lin, and M. Tiwari, "Raccoon: Closing Digital {Side-Channels} through Obfuscated Execution," in *26th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 431–446.

[16] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the web up to speed with WebAssembly," in *Proc. of the Conf. on Programming Language Design and Implementation (PLDI)*, 2017, pp. 185–200.

[17] R. M. Tsoupidi, R. Castañeda Lozano, and B. Baudry, "Constraint-Based Software Diversification for Efficient Mitigation of Code-Reuse Attacks," in *International Conference on Principles and Practice of Constraint Programming*, 2020, pp. 791–808.

[18] R. M. Tsoupidi, R. Castañeda Lozano, and B. Baudry, "Constraint-based diversification of JOP gadgets," *Journal of Artificial Intelligence Research*, vol. 72, pp. 1471–1505, 2021.

[19] R. M. Tsoupidi, M. Balliu, and B. Baudry, "Vivienne: Relational Verification of Cryptographic Implementations in WebAssembly," in *2021 IEEE Secure Development Conference (SecDev)*, 2021, pp. 94–102.

[20] R. M. Tsoupidi, R. C. Lozano, E. Troubitsyna, and P. Papadimitratos, "Securing optimized code against power side channels," *arXiv preprint arXiv:2207.02614*, 2022, to appear in CSF'23.

[21] R. M. Tsoupidi, E. Troubitsyna, and P. Papadimitratos, "Thwarting code-reuse and side-channel attacks in embedded systems," *arXiv preprint arXiv:2304.13458*, 2023, under submission.

[22] "ENISA Threat Landscape 2022." [Online]. Available: https://www.enisa.europa.eu/publications/enisa-threat-landscape-2022

[23] S. Liu and B. Cheng, "Cyberattacks: Why, what, who, and how," *IT Professional*, vol. 11, no. 3, pp. 14–21, 2009.

[24] L.-A. Daniel, S. Bardin, and T. Rezk, "Binsec/Rel: Efficient Relational Symbolic Execution for Constant-Time at Binary-Level," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 1021–1038.

[25] J. Wang, C. Sung, and C. Wang, "Mitigating power side channels during compilation," in *Proc. 2019 27th ACM Jt. Meet. Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, ser. ESEC/FSE 2019. Association for Computing Machinery, 2019, pp. 590–601.

[26] K. Athanasiou, T. Wahl, A. A. Ding, and Y. Fei, "Automatic Detection and Repair of Transition- Based Leakage in Software Binaries," in *Softw. Verification*, ser. Lecture Notes in Computer Science. Springer International Publishing, 2020, pp. 50–67.

[27] G.-A. Jaloyan, K. Markantonakis, R. N. Akram, D. Robin, K. Mayes, and D. Naccache, "Return-Oriented Programming on RISC-V," in *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, ser. ASIA CCS '20, Oct. 2020, pp. 471–480.

[28] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented Programming: A New Class of Code-reuse Attack," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ser. ASIACCS '11. New York, NY, USA: ACM, 2011, pp. 30–40.

[29] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented Programming Without Returns," in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, ser. CCS '10. New York, NY, USA: ACM, 2010, pp. 559–572.

[30] O. Gilles, F. Viguier, N. Kosmatov, and D. G. Pérez, "Control-flow integrity at risc: Attacking risc-v by jump-oriented programming," 2022.

[31] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A. Sadeghi, "Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization," in *2013 IEEE Symposium on Security and Privacy*, May 2013, pp. 574–588.

[32] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh, "Hacking Blind," in *2014 IEEE Symposium on Security and Privacy*, May 2014, pp. 227–242, iSSN: 2375-1207.

[33] J. Seibert, H. Okhravi, and E. Söderström, "Information Leaks Without Memory Disclosures: Remote Side Channel Attacks on Diversified Code," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14, Nov. 2014, pp. 54–65.

[34] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer, "Control-Flow Integrity: Precision, Security, and Performance," *ACM Computing Surveys*, vol. 50, no. 1, pp. 16:1–16:33, Apr. 2017.

[35] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM conference on Computer and communications security*, ser. CCS '05, Nov. 2005, pp. 340–353.

[36] B. Randell, "System structure for software fault tolerance," in *Proceedings of the international conference on Reliable software*. New York, NY, USA: Association for Computing Machinery, Apr. 1975, pp. 437–449.

[37] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser, "N-variant systems: A secretless framework for security through diversity." in *USENIX Security Symposium*, 2006, pp. 105–120.

[38] F. B. Cohen, "Operating system protection through program evolution." *Comput. Secur.*, vol. 12, no. 6, pp. 565–584, 1993.

[39] S. Forrest, A. Somayaji, and D. Ackley, "Building diverse computer systems," in *Proceedings. The Sixth Workshop on Hot Topics in Operating Systems (Cat. No.97TB100133)*, May 1997, pp. 67–72.

[40] B. Persaud, B. Obada-Obieh, N. Mansourzadeh, A. Moni, and A. Somayaji, "Frankenssl: Recombining cryptographic libraries for software diversity," in *Proceedings of the 11th Annual Symposium On Information Assurance. NYS Cyber Security Conference*, 2016, pp. 19–25.

[41] N. Harrand, T. Durieux, D. Broman, and B. Baudry, "Automatic diversity in the software supply chain," *arXiv preprint arXiv:2111.03154*, 2021.

[42] D. Brumley and D. Boneh, "Remote timing attacks are practical," *Computer Networks*, vol. 48, no. 5, pp. 701–716, 2005.

[43] D. J. Bernstein, "Cache-timing attacks on aes," 2005.

[44] D. A. Osvik, A. Shamir, and E. Tromer, "Cache Attacks and Countermeasures: The Case of AES," in *Topics in Cryptology – CT-RSA 2006*, ser. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2006, pp. 1–20.

[45] G. Barthe, S. Blazy, R. Hutin, and D. Pichardie, "Secure Compilation of Constant-Resource Programs," in *CSF 2021 - 34th IEEE Computer Security Foundations Symposium*. IEEE, Jun. 2021, pp. 1–12.

[46] V. C. Ngo, M. Dehesa-Azuara, M. Fredrikson, and J. Hoffmann, "Verifying and Synthesizing Constant-Resource Implementations with Types," in *2017 IEEE Symposium on Security and Privacy (SP)*, May 2017, pp. 710–728, iSSN: 2375-1207.

[47] M. A. Shelton, N. Samwel, L. Batina, F. Regazzoni, M. Wagner, and Y. Yarom, "Rosita: Towards Automatic Elimination of Power-Analysis Leakage in Ciphers," *Proceedings 2021 Network and Distributed System Security Symposium*, 2021, appears in NDSS 2022.

[48] K. Ngo, E. Dubrova, and T. Johansson, "Breaking Masked and Shuffled CCA Secure Saber KEM by Power Analysis," in *Proceedings of the 5th Workshop on Attacks and Solutions in Hardware Security*, Nov. 2021, pp. 51–61.

[49] W.-J. van Hoeve and I. Katriel, "Global constraints," in *Foundations of Artificial Intelligence*. Elsevier, 2006, vol. 2, pp. 169–208.

[50] J.-C. Régin, "A filtering algorithm for constraints of difference in CSPs," in *AAAI*, vol. 94, 1994, pp. 362–367.

[51] L. Ingmar, M. Garcia de la Banda, P. J. Stuckey, and G. Tack, "Modelling diversity of solutions," in *Proceedings of the thirty-fourth AAAI conference on artificial intelligence*, 2020.

[52] E. Hebrard, B. Hnich, B. O'Sullivan, and T. Walsh, "Finding Diverse and Similar Solutions in Constraint Programming," in *National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference*, 2005, p. 6.

[53] L. Popovic, A. Côté, M. Gaha, F. Nguewouo, and Q. Cappart, "Scheduling the equipment maintenance of an electric power transmission network using constraint programming," in *28th International Conference on Principles and Practice of Constraint Programming (CP 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

[54] M. Gendreau, J.-Y. Potvin *et al.*, *Handbook of metaheuristics*. Springer, 2010, vol. 2.

[55] P. Shaw, "Using constraint programming and local search methods to solve vehicle routing problems," in *Principles and Practice of Constraint Programming*, ser. Lecture Notes in Computer Science, vol. 1520. Springer, 1998, pp. 417–431.

[56] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *CGO*. IEEE, 2004.

[57] R. M. Stallman, *Using the GNU Compiler Collection: a GNU manual for GCC version 4.3.3*. CreateSpace, 2009.

[58] G. Hjort Blindell, *Instruction Selection*. Springer International Publishing, 2016.

[59] R. C. Lozano and C. Schulte, "Survey on Combinatorial Register Allocation and Instruction Scheduling," *ACM Computing Surveys (CSUR)*, vol. 52, no. 3, pp. 62:1–62:50, 2019.

[60] G. P. Farina, S. Chong, and M. Gaboardi, "Relational Symbolic Execution," in *Proceedings of the 21st International Symposium on Principles and Practice of Declarative Programming*, ser. PPDP '19. Association for Computing Machinery, Oct. 2019, pp. 1–14.

[61] H. Winderix, J. T. Mühlberg, and F. Piessens, "Compiler-Assisted Hardening of Embedded Software Against Interrupt Latency Side-Channel Attacks," in *2021 IEEE European Symposium on Security and Privacy (EuroS P)*, Sep. 2021, pp. 667–682.

[62] F. Besson, A. Dang, and T. Jensen, "Information-Flow Preservation in Compiler Optimisations," in *2019 IEEE 32nd Comput. Secur. Found. Symp. CSF*, 2019, pp. 230–23 012.

[63] T. Edgar and D. Manz, *Research methods for cyber security*. Syngress, 2017.

[64] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: A tool for evaluating and synthesizing multimedia and communicatons systems," in *MICRO*. IEEE, 1997, pp. 330–335.

[65] *CPU 2006 Benchmarks*, SPEC, 2020, https://www.spec.org/cpu2006, accessed on 2020-03-20.

[66] Libsodium Community, "The sodium cryptography library (Libsodium)," 2018. [Online]. Available: https://libsodium.gitbook.io/doc

[67] J. Protzenko, B. Beurdouche, D. Merigoux, and K. Bhargavan, "Formally Verified Cryptographic Web Applications in WebAssembly," in *2019 IEEE Symposium on Security and Privacy (SP)*, May 2019, pp. 1256–1274.

[68] T. Pornin, "Bearssl, a smaller SSL/TLS library," last accessed May 14, 2021. [Online]. Available: https://bearssl.org/

[69] H. Mantel and A. Starostin, "Transforming Out Timing Leaks, More or Less," in *Computer Security – ESORICS 2015*, ser. Lecture Notes in Computer Science. Springer International Publishing, 2015, pp. 447–467.

[70] A. Lindner, R. Guanciale, and M. Dam, "Proof-producing symbolic execution for binary code verification," 2023.

[71] D. Broman, "A Brief Overview of the KTA WCET Tool," Dec. 2017, number: arXiv:1712.05264 arXiv:1712.05264 [cs].

[72] R. M. Tsoupidi, "Two-phase WCET analysis for cache-based symmetric multiprocessor systems," Master's thesis, Royal Institute of Technology KTH, 2017.

[73] S. Pastrana, J. Tapiador, G. Suarez-Tangil, and P. Peris-López, "AVRAND: A Software-Based Defense Against Code Reuse Attacks for AVR Embedded Devices," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, ser. Lecture Notes in Computer Science, 2016, pp. 58–77.

[74] T. Abera, N. Asokan, L. Davi, J.-E. Ekberg, T. Nyman, A. Paverd, A.-R. Sadeghi, and G. Tsudik, "C-FLAT: Control-Flow Attestation for Embedded Systems Software," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16, Oct. 2016, pp. 743–754.

[75] T. Nyman, J.-E. Ekberg, L. Davi, and N. Asokan, "CFI CaRE: Hardware-Supported Call and Return Enforcement for Commercial Microcontrollers," in *Research in Attacks, Intrusions, and Defenses*, ser. Lecture Notes in Computer Science. Springer International Publishing, 2017, pp. 259–284.

[76] H. Koo, Y. Chen, L. Lu, V. P. Kemerlis, and M. Polychronakis, "Compiler-Assisted Code Randomization," in *2018 IEEE Symposium on Security and Privacy (SP)*, May 2018, pp. 461–477.

[77] J. Shi, L. Guan, W. Li, D. Zhang, P. Chen, and P. Chen, "HARM: Hardware-assisted continuous re-randomization for microcontrollers," in *2022 IEEE european symposium on security and privacy (EuroS P)*, 2022.

[78] A. Fu, W. Ding, B. Kuang, Q. Li, W. Susilo, and Y. Zhang, "FH-CFI: Fine-grained hardware-assisted control flow integrity for ARM-based IoT devices," *Computers & Security*, vol. 116, p. 102666, May 2022.

[79] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proceedings of the 11th ACM conference on Computer and communications security*, ser. CCS '04. Association for Computing Machinery, Oct. 2004, pp. 298–307.

[80] S. Cauligi, G. Soeller, F. Brown, B. Johannesmeyer, Y. Huang, R. Jhala, and D. Stefan, "FaCT: A Flexible, Constant-Time Programming Language," in *2017 IEEE Cybersecurity Dev. SecDev*, 2017, pp. 69–76.

[81] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche, "HACL*: A Verified Modern Cryptographic Library," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1789–1806.

[82] J. B. Almeida, M. Barbosa, G. Barthe, A. Blot, B. Grégoire, V. Laporte, T. Oliveira, H. Pacheco, B. Schmidt, and P.-Y. Strub, "Jasmin: High-Assurance and High-Speed Cryptography," in *Proc. 2017 ACM SIGSAC Conf. Comput. Commun. Secur.*, ser. CCS '17. Association for Computing Machinery, 2017, pp. 1807–1823.

[83] P. Borrello, D. C. D'Elia, L. Querzoni, and C. Giuffrida, "Constantine: Automatic Side-Channel Resistance Using Efficient Control and Data Flow Linearization," *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pp. 715–733, Nov. 2021.

[84] J. Cabrera Arteaga, O. Floros, O. Vera Perez, B. Baudry, and M. Monperrus, "Crow: Code diversification for webassembly," in *MADWeb, NDSS 2021*, 2021.

[85] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, "Verifying constant-time implementations," in *25th USENIX security symposium (USENIX security 16)*. Austin, TX: USENIX Association, Aug. 2016, pp. 53–70.

[86] X. Leroy, "Formal verification of a realistic compiler," *Communications of the ACM*, vol. 52, no. 7, pp. 107–115, Jul. 2009.

[87] B. Bond, C. Hawblitzel, M. Kapritsos, K. R. M. Leino, J. R. Lorch, B. Parno, A. Rane, S. Setty, and L. Thompson, "Vale: Verifying {High-Performance} Cryptographic Assembly Code," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 917–934.

[88] I. Bastys, M. Algehed, A. Sjösten, and A. Sabelfeld, "Secwasm: Information flow control for webassembly," in *Static Analysis: 29th International Symposium, SAS 2022, Auckland, New Zealand, December 5–7, 2022, Proceedings*. Springer, 2022, pp. 74–103.

[89] H. Eldib and C. Wang, "Synthesis of Masking Countermeasures against Side Channel Attacks," in *Comput. Aided Verification*, ser. Lecture Notes in Computer Science. Springer International Publishing, 2014, pp. 114–130.

[90] P. Gao, J. Zhang, F. Song, and C. Wang, "Verifying and Quantifying Side-channel Resistance of Masked Software Implementations," *ACM Transactions on Software Engineering and Methodology*, vol. 28, no. 3, pp. 16:1–16:32, 2019.

[91] S. T. Vu, A. Cohen, K. Heydemann, A. de Grandmaison, and C. Guillon, "Secure optimization through opaque observations," *arXiv preprint arXiv:2101.06039*, 2021.

[92] F. Besson, A. Dang, and T. Jensen, "Securing Compilation Against Memory Probing," in *Proc. 13th Workshop Program. Lang. Anal. Secur.*, ser. PLAS '18. Association for Computing Machinery, 2018, pp. 29–40.

[93] L.-A. Daniel, S. Bardin, and T. Rezk, "Hunting the haunter-efficient relational symbolic execution for spectre with haunted relse," in *NDSS 2021-Network and Distributed Systems Security*, 2021.

Part I

Included Publications

Appendix A

Publication 1

Constraint-Based Software Diversification for Efficient Mitigation of Code-Reuse Attacks

Rodothea Myrsini Tsoupidi[1], Roberto Castañeda Lozano[2], and Benoit Baudry[1]

[1] KTH Royal Institute of Technology, Sweden
{tsoupidi,baudry}@kth.se
[2] University of Edinburgh, United Kingdom
roberto.castaneda@ed.ac.uk

Abstract. Modern software deployment process produces software that is uniform, and hence vulnerable to large-scale code-reuse attacks. *Compiler-based diversification* improves the resilience and security of software systems by automatically generating different assembly code versions of a given program. Existing techniques are efficient but do not have a precise control over the quality of the generated code variants.
This paper introduces *Diversity by Construction (DivCon)*, a constraint-based compiler approach to software diversification. Unlike previous approaches, DivCon allows users to control and adjust the conflicting goals of diversity and code quality. A key enabler is the use of Large Neighborhood Search (LNS) to generate highly diverse assembly code efficiently. Experiments using two popular compiler benchmark suites confirm that there is a trade-off between quality of each assembly code version and diversity of the entire pool of versions. Our results show that DivCon allows users to trade between these two properties by generating diverse assembly code for a range of quality bounds. In particular, the experiments show that DivCon is able to mitigate code-reuse attacks effectively while delivering near-optimal code ($< 10\%$ optimality gap).
For constraint programming researchers and practitioners, this paper demonstrates that LNS is a valuable technique for finding diverse solutions. For security researchers and software engineers, DivCon extends the scope of compiler-based diversification to performance-critical and resource-constrained applications.

Keywords: compiler-based software diversification · code-reuse attacks · constraint programming · embedded systems

1 Introduction

Good software development practices, such as code reuse [19], continuous deployment, and automatic updates contribute to the emergence of software monocultures [3]. While such monocultures facilitate software distribution, bug reporting, and software authentication, they also introduce serious risks related to the wide spreading of attacks against all users that run identical software.

Software diversification is a method to mitigate the problems caused by uniformity. Similarly to biodiversity, software diversification improves the resilience and security of a software system [2] by introducing diversity in it. Software diversification can be applied in different phases of the software development cycle, i.e. during implementation, compilation, loading, execution, and more [20]. This paper is concerned with *compiler-based* diversification, which automatically generates different assembly code versions from a single source program.

Modern compilers do not merely aim to generate correct code, but also code that is of high quality. Existing compiler-based diversification techniques are efficient and effective at diversifying assembly code [20] but do not have a precise control over its quality and may produce unsatisfactory results. These techniques (discussed in Section 5) are either based on randomizing heuristics or in high-level superoptimization methods that do not capture accurately the quality of the generated code.

This paper introduces Diversity by Construction (DivCon), a compiler-based diversification approach that allows users to control and adjust the conflicting goals of quality of each code version and diversity among all versions. DivCon uses a Constraint Programming (CP)-based compiler backend to generate multiple solutions corresponding to functionally equivalent program variants according to an accurate code quality model. The backend models the input program, the hardware architecture, and the compiler transformations as a constraint problem, whose solution corresponds to assembly code for the input program.

The use of CP makes it possible to 1) control the quality of the generated solutions by constraining the objective function, 2) introduce application-specific constraints that restrict the diversified solutions, and 3) apply sophisticated search procedures that are particularly suitable for diversification. In particular, DivCon uses Large Neighborhood Search (LNS) [29], a popular metaheuristic in multiple application domains, to generate highly diverse solutions efficiently.

Our experiments compiling 17 functions from two popular compiler benchmark suites to the MIPS32 architecture confirm that there is a trade-off between code quality and diversity, and demonstrate that DivCon allows users to navigate this conflict by generating diverse assembly code for a range of quality bounds. In particular, the experiments show that DivCon is able to mitigate code-reuse attacks effectively while guaranteeing a code quality of 10% within optimality.

For constraint programming researchers and practitioners, this paper demonstrates that LNS is a valuable technique for finding diverse solutions. For security researchers and software engineers, DivCon extends the scope of compiler-based diversification to performance-critical and resource-constrained applications, and provides a solid step towards secure-by-construction software.

Contributions. To summarize, this paper:

- proposes a CP-based technique for compiler-based, quality-aware software diversification (Section 3);
- shows that LNS is a promising technique for generating highly diverse solutions efficiently (Section 4.3);

```
1  0x9d001408: ...
2  0x9d00140c: lw    $s2, 4($sp)
3  0x9d001410: lw    $s4, 0($sp)
4  0x9d001414: jr    $t9
5  0x9d001418: addiu $sp, $sp, 16
```

(a) Original gadget.

```
1  0x9d001408: lw    $s2, 4($sp)
2  0x9d00140c: nop
3  0x9d001410: lw    $s4, 0($sp)
4  0x9d001414: jr    $t9
5  0x9d001418: addiu $sp, $sp, 16
```

(b) Diversified gadget.

Fig. 1: Example gadget diversification in MIPS32 assembly code

– quantifies the trade-off between code quality and diversity (Section 4.4); and
– demonstrates that DivCon mitigates code-reuse attacks effectively while preserving high code quality (Section 4.5).

2 Background

This section describes code-reuse attacks (Section 2.1), diversification approaches in CP (Section 2.2), and combinatorial compiler backends (Section 2.3).

2.1 Code-reuse Attacks

Code-reuse attacks take advantage of memory vulnerabilities, such as buffer overflows, to reuse program code for malicious purposes. More specifically, code-reuse attacks insert data into the program memory to affect the control flow of the program and execute code that is valid but unintended.

Jump-Oriented Programming (JOP)[3] is a code-reuse attack [7,4] that combines different code snippets from the original program code to form a Turing complete language for attackers. These code snippets terminate with a branch instruction. The building blocks of a JOP attack are *gadgets*: meta-instructions that consist of one or multiple code snippets with specific semantics. Figure 1a shows a JOP gadget found by the *ROPgadget* tool [27] in a MIPS32 binary. Assuming that the attacker controls the stack, lines 2 and 3 load attacker data in registers $s2 and $s4, respectively. Then, line 4 jumps to the address of register $t9. The last instruction (line 5) is placed in a delay slot and hence it is executed before the jump [31]. The semantics of this gadget depends on the attack payload and might be to load a value to register $s2 or $s4. Then, the program jumps to the next gadget that resides at the stack address of $t9.

Statically designed JOP attacks use the absolute binary addresses for installing the attack payload. Hence, a simple change in the instruction schedule of the program as in Figure 1b prevents a JOP attack designed for Figure 1a. An attacker that designs an attack based on the binary of the original program assumes the presence of a gadget (Figure 1a) at position 0x9d00140c. However, in the diversified version, address 0x9d00140c does not start with the initial `lw`

[3] This paper focuses on JOP due to the characteristics of MIPS32, but could be generalized to other code-reuse attacks such as Return-Oriented Programming (ROP) [28].

instruction of Figure 1a, and by the end of the execution of the gadget, register $s2 does not contain the attacker data. In this way, diversification can break the semantics of the gadget and mitigate an attack against the diversified code.

2.2 Diversity in Constraint Programming

While typical CP applications aim to discover either some solution or the optimal solution, some applications require finding *diverse* solutions for various purposes.

Hebrard *et al.* [13] introduce the MaxDiverse*k*Set problem, which consists in finding the most diverse set of k solutions, and propose an exact and an incremental algorithm for solving it. The exact algorithm does not scale to a large number of solutions [32,16]. The incremental algorithm selects solutions iteratively by solving a distance maximization problem.

Automatic Generation of Architectural Tests (ATGP) is an application of CP that requires generating many diverse solutions. Van Hentenryck *et al.* [32] model ATGP as a MaxDiverse*k*Set problem and solve it using the incremental algorithm of Hebrard et. al. Due to the large number of diverse solutions required (50-100), Van Hentenryck *et al.* replace the maximization step with local search.

In software diversity, solution quality is of paramount importance. In general, earlier CP approaches to diversity are concerned with satisfiability only. An exception is the approach of Petit *et al.* [26]. This approach modifies the objective function for assessing both solution quality and solution diversity, but does not scale to the large number of solutions required by software diversity. Ingmar *et al.* [16] propose a generic framework for modeling diversity in CP. For tackling the quality-diversity trade-off, they propose constraining the objective function with the optimal (or best known) cost o. DivCon applies this approach by allowing solutions $p\%$ worse than o, where p is configurable.

2.3 Compiler Optimization as a Combinatorial Problem

A Constraint Satisfaction Problem (CSP) is a problem specification $P = \langle V, U, C \rangle$, where V are the problem variables, U is the domain of the variables, and C the constraints among the variables. A Constraint Optimization Problem (COP), $P = \langle V, U, C, O \rangle$, consists of a CSP and an objective function O. The goal of a COP is to find a solution that optimizes O.

Compilers are programs that generate low-level assembly code, typically optimized for *speed* or *size*, from higher-level source code. A compilation process can be modeled as a COP by letting V be the decisions taken during the translation, C be the constraints imposed by the program semantics and the hardware resources, and O be the cost of the generated code.

Compiler backends generate low-level assembly code from an Intermediate Representation (IR), a program representation that is independent of both the source and the target language. Figure 2 shows the high-level view of a *combinatorial* compiler backend. A combinatorial compiler backend takes as input the IR of a program, generates and solves a COP, and outputs the optimized low-level assembly code described by the solution to the COP.

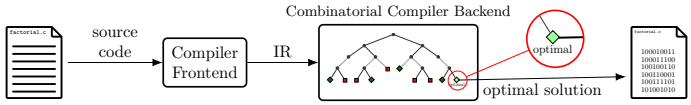


Fig. 2: High-level view of a combinatorial compiler backend

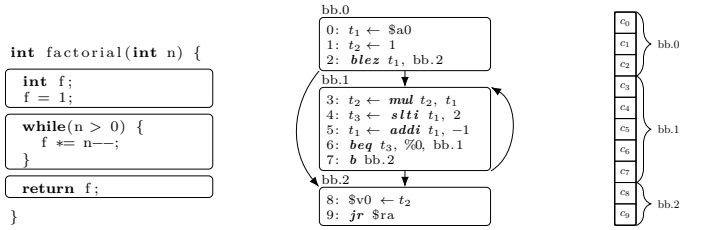
This paper assumes that programs at the IR level are represented by their Control-Flow Graph (CFG). A CFG is a representation of the possible execution paths of a program, where each node corresponds to a *basic block* and edges correspond to intra-block jumps. A *basic block*, in its turn, is a set of abstract instructions (hereafter just *instructions*) with no branches besides the end of the block. Each instruction is associated with a set of operands characterizing its input and output data. Typical decision variables V of a combinatorial compiler backend are the issue cycle $c_i \in \mathbb{N}_0$ of each instruction i, the processor instruction $m_i \in \mathbb{N}_0$ that implements each instruction i, and the processor register $r_o \in \mathbb{N}_0$ assigned to each operand o.

DivCon aims at mitigating code-reuse attacks. Therefore, DivCon considers the order of the instructions in the final binary, which directly affects the feasibility of code-reuse attacks (see Figures 1a and 1b). For this reason, the diversification model uses the issue cycle sequence of instructions, $c = \{c_0, c_1, ..., c_n\}$, to characterize the diversity among different solutions.

Figure 3a shows an implementation of the factorial function in C where each basic block is highlighted. Figure 3b shows the IR of the program. The example IR contains 10 instructions in three basic blocks: bb.0, bb.1, and bb.2. bb.0 corresponds to initializations, where `$a0` holds the function argument `n` and t_1 corresponds to variable `f`. bb.1 computes the factorial in a loop by accumulating the result in t_1. bb.2 stores the result to `$v0` and returns. Some instructions in the example are interdependent, which leads to serialization of the instruction schedule. For example, ***beq*** (6) consumes data (t_3) defined by ***slti*** (4) and hence needs to be scheduled later. Instruction dependencies limit the amount of possible assembly code versions and can restrict diversity significantly, as seen in Section 4.3. Finally, Figure 3c shows the arrangement of the issue cycle variables in the constraint model used by the combinatorial compiler backend.

3 DivCon

This section introduces DivCon, a software diversification method that uses a combinatorial compiler backend to generate program variants. Figure 4 shows a high-level view of the diversification process. DivCon uses 1) the optimal solution to start the *search* for diversification and 2) the cost of the optimal solution to restrict the variants within a maximum gap from the optimal. Subsequently, DivCon generates a number of solutions to the CSP that correspond to diverse program variants.



(a) C code (b) IR (c) Issue cycles

Fig. 3: Factorial function example

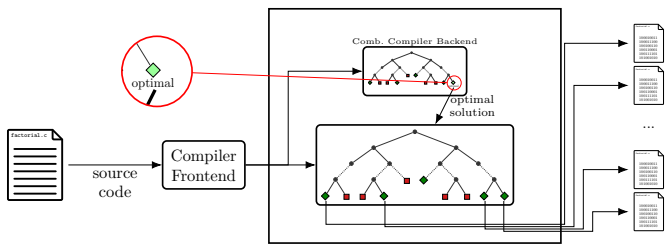


Fig. 4: High-level view of DivCon

The rest of this section describes the diversification approach of DivCon. Section 3.1 formulates the diversification problem in terms of the constraint model of a combinatorial compiler backend, Section 3.2 defines the distance measures, and finally, Section 3.3 describes the search strategy for generating program variants.

3.1 Problem Description

Let $P = \langle V, U, C \rangle$ be the compiler backend CSP for the program under compilation, O be the objective function, and o be the cost of the optimal or best known solution to the COP, $\langle V, U, C, O \rangle$. Let δ be a function that measures the distance between two solutions of P (two such functions are defined in Section 3.2). Let $h \in \mathbb{N}$ be the minimum pairwise distance and $p \in \mathbb{R}_{\geq 0}$ be the maximum optimality gap specified by the user. Our problem is to find a subset of the solutions to the CSP, $S \subseteq sol(P)$, such that $\forall s_1, s_2 \in S \,.\, s_1 \neq s_2 \implies \delta(s_1, s_2) \geq h$ and $\forall s \in S \,.\, O(s) \leq (1 + p) \cdot o$.

To solve the above problem, DivCon employs the incremental algorithm listed in Algorithm 1. Starting with the optimal solution y_{opt}, the algorithm adds the distance constraint for y_{opt} and the optimality constraint with $o = y_{opt}(O)$ (line

2). Notation $\delta(y)$ is used instead of $\delta(y, s) \,|\, \forall s \in sol(\langle V, U, C' \rangle)$ for readability. While the termination condition is not fulfilled (line 3), the algorithm uses LNS as described in Section 3.3 to find the next solution y (line 4), adds the next solution to the solution set S (line 5), and updates the distance constraints based on the latest solution (line 6). When the termination condition is satisfied, the algorithm returns the set of solutions S corresponding to diversified assembly code variants.

Algorithm 1: Incremental algorithm for generating diverse solutions

```
1  S ← {y_opt}, y ← y_opt,
2  C' ← C ∪ {δ(y_opt) ≥ h, O(V) ≤ (1 + p) · o}
3  while not term_cond() // e.g. |S| > k ∨ time_limit()
4      y ← solve_LNS(relax(y), ⟨V, U, C'⟩)
5      S ← S ∪ {y}
6      C' ← C' ∪ {δ(y) ≥ h}
```

Figure 5 shows two MIPS32 variants of the factorial example (Figure 3), which correspond to two solutions of DivCon. The variants differ in two aspects: first, the ***beqz*** instruction is issued one cycle later in Figure 5b than in Figure 5a, and second, the temporary variable t_3 (see Figure 3) is assigned to different MIPS32 registers ($t0 and $t1).

3.2 Distance Measures

This section defines two alternative distance measures: Hamming Distance (HD) and Levenshtein Distance (LD). Both distances operate on the schedule of the instructions, i.e. the order in which the instructions are issued in the CPU.

Hamming Distance (HD). HD is the Hamming distance [12] between the issue cycle variables of two solutions. Given two solutions $s, s' \in sol(P)$:

$$\delta_{HD}(s, s') = \sum_{i=0}^{n} (s(c_i) \neq s'(c_i)), \tag{1}$$

```
1  bb.0: blez  $a0, bb.2
2        addiu $v0, $zero, 1
3  bb.1: mul   $v0, $v0, $a0
4        slti  $t0, $a0, 2
5        beqz  $t0, bb.1
6        addi  $a0, $a0, -1
7  bb.2: jr    $ra
8        nop
```

(a) Variant 1.

```
1  bb.0: blez  $a0, bb.2
2        addiu $v0, $zero, 1
3  bb.1: mul   $v0, $v0, $a0
4        slti  $t1, $a0, 2
5        nop
6        beqz  $t1, bb.1
7        addi  $a0, $a0, -1
8  bb.2: jr    $ra
9        nop
```

(b) Variant 2.

Fig. 5: Two MIPS32 variants of the factorial example in Figure 3

where n is the maximum number of instructions.

Consider Figure 1b, a diversified version of the gadget in Figure 1a. The only instruction that differs from Figure 1a is the instruction at line 1 that is issued one cycle before. The two examples have a HD of one, which in this case is enough for breaking the functionality of the original gadget (see Section 2.1).

Levenshtein Distance (LD). LD (or edit distance) measures the minimum number of edits, i.e. insertions, deletions, and replacements, that are necessary for transforming one instruction schedule to another. Compared to HD, which considers only *replacements*, LD also considers *insertions* and *deletions*. To understand this effect, consider Figure 5. The two gadgets differ only by one `nop` operation but HD gives a distance of three, whereas LD gives one, which is more accurate. LD takes ordered vectors as input, and thus requires an ordered representation (as opposed to a detailed schedule) of the instructions. Therefore, LD uses vector $c^{-1} = channel(c)$, a sequence of instructions ordered by their issue cycle. Given two solutions $s, s' \in sol(P)$:

$$\delta_{LD}(s, s') = \texttt{levenshtein\_distance}(s(c^{-1}), s'(c^{-1})), \tag{2}$$

where `levenshtein_distance` is the Wagner–Fischer algorithm [33] with time complexity $O(nm)$, where n and m are the lengths of the two sequences.

3.3 Search

Unlike previous CP approaches to diversity, DivCon employs Large Neighborhood Search (LNS) for diversification. LNS is a metaheuristic that defines a neighborhood, in which *search* looks for better solutions, or, in our case, different solutions. The definition of the neighborhood is through a *destroy* and a *repair* function. The *destroy* function unassigns a subset of the variables in a given solution and the *repair* function finds a new solution by assigning new values to the *destroyed* variables.

In DivCon, the algorithm starts with the optimal solution of the combinatorial compiler backend. Subsequently, it destroys a part of the variables and continues with the model's branching strategy to find the next solution, applying a restart after a given number of failures. LNS uses the concept of *neighborhood*, i.e. the variables that LNS may destroy at every restart. To improve diversity, the neighborhood for DivCon consists of all decision variables, i.e. the issue cycles c, the instruction implementations m, and the registers r. Furthermore, LNS depends on a *branching strategy* to guide the *repair* search. To improve security and allow LNS to select diverse paths after every restart, DivCon employs a random variable-value selection branching strategy as described in Table 1b.

4 Evaluation

The evaluation of DivCon addresses four main questions:

– RQ1. What is the scalability of the distance measures in generating multiple program variants? Here, we evaluate which of the distance measures is the most appropriate for software diversification.
– RQ2. How effective and how scalable is LNS for code diversification? Here, we investigate LNS as an alternative approach to diversity in CP.
– RQ3. How does code quality relate to code diversity and what are the involved trade-offs?
– RQ4. How effective is DivCon at mitigating code-reuse attacks? This question is the main application of CP-based diversification in this work.

4.1 Experimental Setup

Implementation. DivCon is implemented as an extension of Unison [6], and is available at `https://github.com/romits800/divcon`. Unison implements two backend transformations: instruction scheduling and register allocation. DivCon employs Unison's solver portfolio that includes Gecode v6.2 [11] and Chuffed v0.10.3 [8] to find optimal solutions, and Gecode v6.2 only for diversification. The LLVM compiler [21] is used as a front-end and IR-level optimizer.

Benchmark functions and platform. The evaluation uses 17 functions sampled randomly from MediaBench [22] and SPEC CPU2006 [30], two benchmark suites widely employed in embedded and general-purpose compiler research. The size of the functions is limited to between 10 and 30 instructions (with a median of 20 instructions) to keep the evaluation of all methods and distance measures feasible regardless of their computational cost. Table 2 lists the ID, application, name, basic blocks (b), and instructions (i) of each sampled function. The functions are compiled to MIPS32 assembly code. MIPS32 is a popular architecture within embedded systems and the security-critical Internet of Things [1].

Host platform. All experiments run on an Intel®Core™i9-9920X processor at 3.50GHz with 64GB of RAM running Debian GNU/Linux 10 (buster). Each of the experiments runs for 20 random seeds. The results show the mean value and the standard deviation from these experiments. The available virtual memory for each of the executions is 10GB. The experiments for different random seeds run in parallel (5 seeds at a time), with two unique cores available for every seed for overheating reasons. DivCon runs as a sequential program.

Table 1: ORIGINAL and RANDOM branching strategies

(a) ORIGINAL branching strategy.

Variable	Var. Selection	Value Selection
c_i	in order	min. val first
m_i	in order	min. val first
r_o	in order	randomly

(b) RANDOM branching strategy.

Variable	Var. Selection	Value Selection
c_i	randomly	randomly
m_i	randomly	randomly
r_o	randomly	randomly

Table 2: Benchmark functions

ID	app	function name	b	i
b1	sphinx3	ptmr_init	1	10
b2	gcc	ceil_log2	1	14
b3	mesa	glIndexd	1	14
b4	h264ref	symbol2uvlc	1	15
b5	gobmk	autohelperowl_defen..	1	23
b6	mesa	glVertex2i	1	23
b7	hmmer	AllocFancyAli	1	25
b8	gobmk	autohelperowl_vital..	1	27
b9	gobmk	autohelperpat1088	1	29
b10	gobmk	autohelperowl_attac..	1	30
b11	gobmk	get_last_player	3	13
b12	h264ref	UpdateRandomAccess	3	16
b13	gcc	xexit	3	17
b14	gcc	unsigned_condition	3	24
b15	sphinx3	glist_tail	4	10
b16	gcc	get_frame_alias_set	5	20
b17	gcc	parms_set	5	25

Table 3: Scalability of δ_{HD}, δ_{LD}

ID	δ_{HD}		δ_{LD}	
	$t(s)$	num	$t(s)$	num
b1	0.1±0.2	26	131.2±131.4	26
b2	1.0±0.1	200	-	68
b3	1.1±0.1	200	-	58
b4	0.7±0.0	200	-	73
b5	2.3±0.3	200	-	38
b6	2.5±0.2	200	-	35
b7	2.0±0.3	200	-	37
b8	3.8±0.8	200	-	35
b9	4.0±0.6	200	-	28
b10	4.5±0.7	200	-	27
b11	1.3±0.1	200	-	56
b12	1.1±0.2	200	-	47
b13	0.8±0.1	200	-	91
b14	1.8±0.3	200	-	27
b15	1.7±0.2	200	-	60
b16	2.7±0.4	200	-	31
b17	1.6±0.2	200	-	35

Algorithm Configuration. The experiments focus on speed optimization and aim to generate 200 variants within a timeout. Parameter h in Algorithm 1 is set to one because even small distance between variants is able to break gadgets (see Figure 1). LNS uses restart-based search with a limit of 500 failures, and a relax rate of 70%. The *relax rate* is the probability that LNS destroys a variable at every restart, which affects the distance between two subsequent solutions. A higher relax rate increases diversity but requires more solving effort. We have found experimentally that 70% is an adequate balance between the two. All experiments are available at `https://github.com/romits800/divcon_experiments`.

4.2 RQ1. Scalability of the Distance Measures

The ability to generate a large number of variants is paramount for software diversification. This section compares the distance measures introduced in Section 3.2 with regards to scalability.

Table 3 presents the results of the distance evaluation, where a time limit of 10 minutes and optimality gap of $p = 10\%$ are used. For each distance measure (δ_{HD} and δ_{LD}) the table shows the diversification time t, in seconds (or "-" if the algorithm is not able to generate 200 variants) and the number of generated variants *num* within the time limit.

The results show that for δ_{HD}, DivCon is able to generate 200 variants for all benchmarks except *b1*, which has exactly 26 variants. The diversification time for δ_{HD} is less than 5 seconds for all benchmarks. Distance δ_{LD}, on the other hand, is not able to generate 200 variants for any of the benchmarks within the time limit. This poor scalability of δ_{LD} is due to the quadratic complexity of its implementation [33], whereas HD can be implemented linearly. Consequently, the rest of the evaluation uses δ_{HD}.

4.3 RQ2. Scalability and Diversification Effectiveness of LNS

This section evaluates the diversification effectiveness and scalability of LNS compared to incremental MaxDiverse*k*Set (where the first solution is found randomly and the maximization step uses the branching strategy from Table 1a) and Random Search (RS) (which uses the branching strategy from Table 1b).

To measure the diversification effectiveness of these methods, the evaluation uses the relative pairwise distance of the solutions. Given a set of solutions S and a distance measure δ, the pairwise distance d of the variants in S is $d(\delta, S) = \sum_{i=0}^{|S|} \sum_{j>i}^{|S|} \delta(s_i, s_j) \,/\, \binom{|S|}{2}$. The *larger* this distance, the more diverse the solutions are, and thus, diversification is more effective. Table 4 shows the pairwise distance d and diversification time t for each benchmark and method, where the experiment uses a time limit of 30 minutes and optimality gap of $p = 10\%$. The best values of d (larger) and t (lower) are marked in **bold** for the completed experiments, whereas incomplete experiments are highlighted in *italic* and their number of variants in parenthesis.

Table 4: Distance and Scalability of LNS with RS and MaxDiverse*k*Set

ID	MaxDiverse*k*Set		RS		LNS (0.7)	
	d	$t(s)$	d	$t(s)$	d	$t(s)$
b1	*4.1±0.0*	0.2±0.0 (26)	*4.1±0.0*	**0.0±0.0 (26)**	*4.1±0.0*	0.1±0.2 (26)
b2	**10.8±0.0**	761.8±10.1	6.4±0.2	**0.6±0.1**	8.6±0.6	1.0±0.1
b3	*14.6±0.0*	- (21)	5.8±0.1	**0.6±0.1**	**10.8±0.8**	1.0±0.1
b4	*14.4±0.0*	- (19)	4.3±0.1	**0.2±0.0**	**12.1±0.3**	0.6±0.0
b5	*22.0±0.0*	- (2)	4.3±0.3	**0.5±0.0**	**16.1±1.1**	2.2±0.3
b6	*22.9±0.4*	- (2)	5.3±0.0	**1.0±0.1**	**16.4±0.6**	2.4±0.2
b7	*24.9±0.1*	- (6)	4.5±0.2	**0.4±0.0**	**18.1±1.2**	1.9±0.3
b8	*24.8±0.4*	- (2)	6.5±0.2	**3.5±0.5**	**17.2±0.9**	3.8±0.8
b9	*26.0±0.0*	- (2)	4.2±0.3	**0.4±0.0**	**19.8±0.7**	3.9±0.6
b10	*28.0±0.0*	- (2)	6.0±0.0	5.3±1.0	**20.1±1.1**	**4.5±0.7**
b11	**13.8±0.0**	356.9±8.2	5.3±0.1	**0.2±0.0**	10.1±1.0	1.2±0.1
b12	*21.5±0.1*	- (5)	6.4±0.9	**0.2±0.0**	**14.9±1.0**	1.0±0.2
b13	*17.4±0.0*	- (122)	6.7±0.0	0.9±0.1	**12.0±0.9**	**0.7±0.1**
b14	*30.1±0.0*	- (20)	7.5±0.2	**0.2±0.0**	**24.9±0.7**	1.8±0.3
b15	-	-	2.6±0.3	**0.1±0.0**	**20.2±0.5**	1.6±0.2
b16	-	-	5.6±0.4	**0.3±0.0**	**21.3±0.8**	2.6±0.4
b17	-	-	*2.9±0.1*	- (91)	**28.1±1.5**	**1.6±0.2**

The scalability results ($t(s)$) show that RS and LNS are scalable (generate the maximum 200 variants for almost all benchmarks), whereas MaxDiverse*k*Set scales poorly (cannot generate 200 variants for any benchmark but *b2* and *b11*). Both *b2* and *b11* have a small search space (few, highly interdependent instructions), which leads to restricted diversity but facilitates solving. For *b1*, all instructions are interdependent on each other, which forces a linear schedule and results in only 26 possible variants (given $p = 10\%$). On the other end, MaxDiverse*k*Set is not able to find any variants for *b15*, *b16*, and *b17*. These benchmarks have many basic blocks resulting in a more complex objective function. For the largest benchmark (*b17*), only LNS is able to scale up to 200 solutions. LNS is generally slower than RS, but for both LNS and RS all benchmarks have a diversification time less than six seconds.

The diversity results (d) show that LNS is more effective at diversifying than RS. The improvement of LNS over RS ranges from 35% (for *b2*) to 675% (for

b15). In the two cases where MaxDiverse*k*Set terminates (benchmarks *b2* and *b11*), it generates the most diverse code, as can be expected.

In summary, LNS offers an attractive balance between scalability and diversification effectiveness: it is close in scalability to, and sometimes improves, the overly fastest method (RS), but it is significantly and consistently more effective at diversifying code.

4.4 RQ3. Trade-off Between Code Quality and Diversity

A key advantage of using a CP-based compiler approach for software diversity is the ability to control the quality of the generated solutions. This ability enables control over the relation between the quality of each individual solution and the diversity of the entire pool of solutions. Insisting in optimality limits the number of possible diversified variants and their pairwise distance, whereas relaxing optimality allows higher diversity.

Table 5 shows the pairwise distance d (defined in Section 4.3), and the number of generated variants *num*, for all benchmarks and different values of the optimality gap $p \in \{0\%, 5\%, 10\%, 20\%\}$. LNS is used with a time limit of 10 minutes. The best values of d are marked in **bold**.

Table 5: Solution diversity for different optimality gap values

ID	0%		5%		10%		20%	
	d	num	d	num	d	num	d	num
b1	-	-	-	-	*4.1±0.0*	26	**6.5±0.1**	200
b2	*3.5±0.0*	9	6.7±0.4	200	8.6±0.6	200	**10.0±0.8**	200
b3	7.0±0.1	200	9.4±0.5	200	10.8±0.8	200	**14.8±1.0**	200
b4	7.8±0.2	200	10.1±0.3	200	12.1±0.3	200	**14.0±0.2**	200
b5	8.4±0.1	200	11.9±0.7	200	16.1±1.1	200	**19.7±0.6**	200
b6	10.8±0.1	200	14.7±0.4	200	16.4±0.6	200	**20.9±0.8**	200
b7	11.3±0.3	200	13.8±0.7	200	18.1±1.2	200	**22.8±1.1**	200
b8	11.0±0.1	200	13.6±0.6	200	17.2±0.9	200	**22.4±1.1**	200
b9	12.7±0.1	200	17.7±0.8	200	19.8±0.7	200	**24.4±0.6**	200
b10	13.7±0.1	200	18.1±0.9	200	20.1±1.1	200	**26.3±0.6**	200
b11	*2.0±0.0*	4	6.6±0.1	200	10.1±1.0	200	**14.2±0.9**	200
b12	*3.8±0.0*	10	10.3±1.2	200	14.9±1.0	200	**19.8±1.0**	200
b13	*2.1±1.3*	4	10.1±0.9	200	12.0±0.9	200	**15.7±1.2**	200
b14	*3.6±0.0*	24	21.0±0.6	200	24.9±0.7	200	**29.0±0.5**	200
b15	*2.4±0.0*	8	15.6±0.6	200	20.2±0.5	200	**23.5±1.4**	200
b16	*4.1±0.0*	44	15.1±1.1	200	21.3±0.8	200	**30.7±0.9**	200
b17	7.5±0.2	200	20.3±1.4	200	28.1±1.5	200	**38.4±0.9**	200

The first interesting observation is that even with no degradation of quality ($p = 0\%$), DivCon is able to generate a large number of variants for a significant fraction of the benchmarks. These include functions with a relatively large solution space, typically with a few large basic blocks where instructions are relatively independent of each other (*b3*-*b10* and *b17*). On the other hand, benchmarks with small basic blocks and many instruction dependencies (*b1*, *b2*, and *b11*-*b16*) provide fewer options for diversification, which results in a limited number of optimal variants.

Second, we observe that as soon as we slightly relax the constraint over optimality ($p = 5\%$), diversity radiates and DivCon generates 200 variants for all

benchmarks except *b1*. Then, the more we increase the optimality gap, the larger the diversification space grows and the distance between the variants increases. Table 5 illustrates one of the key contributions of DivCon: the ability to explore the trade-off between optimal solutions and highly diverse solutions.

In summary, depending on the characteristics of the compiled code, it is possible to generate a large number of variants without sacrificing optimality, and the code quality can be adjusted to further improve diversity if required by the targeted application.

4.5 RQ4. Code-Reuse Mitigation Effectiveness

Software Diversity has various applications in security, including mitigating code-reuse attacks. To measure the level of mitigation that DivCon achieves, we assess the gadget survival rate $srate(s_i, s_j)$ between two variants $s_i, s_j \in S$, where S is the set of generated variants. This metric determines how many of the gadgets of variant s_i appear at the same position on the other variant s_j, that is $srate(s_i, s_j) = |gad(s_i) - gad(s_j)| \,/\, |gad(s_i)|$, where $gad(s_i)$ are the gadgets in solution s_i. The procedure for computing $srate(s_i, s_j)$ is as follows: 1) run ROPgadget [27] to find the set of gadgets $gad(s_i)$ in solution s_i, and 2) for every $g \in gad(s_i)$, check whether there exists a gadget identical to g at the same address of s_j. This comparison is syntactic after removing all `nop` instructions.

This section compares the *srate* for all permutations of pairs in S, for all benchmarks, and for different values of the optimality gap using a time limit of 10 minutes. Low *srate* corresponds to higher mitigation effectiveness because code-reuse attacks based on gadgets in one variant have lower chances of locating the same gadgets in the other variants (see Figure 1).

Table 6 summarizes the gadget survival distribution for all benchmarks and different values of the optimality gap (0%, 5%, 10%, and 20%). Due to its skewness, the distribution of *srate* is represented as a histogram with four buckets (0%, (0%, 10%], (10%,40%], and (40%, 100%]) rather than summarized using common statistical measures. Here the best is a $srate(s_i, s_j)$ of 0%, which means that s_j does not contain any gadgets that exist in s_i, whereas a $srate(s_i, s_j)$ in range (40%,100%] means that s_j shares more than 40% of the gadgets of s_i. The values in **bold** correspond to the mode(s) of the histogram.

First, we notice that DivCon can generate some pairs of variants that share no gadget, even without relaxing the constraint of optimality ($p = 0\%$). This indicates that the pareto front of optimal code naturally includes software diversity that is good for security. Second, the results show that this effectiveness can be further increased by relaxing the constraint on code quality, with diminishing returns beyond $p = 10\%$. For $p = 0\%$, there are 10 benchmarks dominated by a 0% survival rate, whereas there are 7 benchmarks dominated by a weak $10\% - 40\%$-survival rate. The latter are still considered vulnerable to code-reuse attacks. However, increasing the optimality gap to just $p = 5\%$ makes 0% survival rate the dominating bucket for all benchmarks, and further increasing the gap to 10% and 20% increases significantly the number of pairs where no single

Table 6: Gadget survival rate for different optimality gap values

ID	0%					5%					10%					20%				
	=0	≤10	≤40	≤100	num	=0	≤10	≤40	≤100	num	=0	≤10	≤40	≤100	num	=0	≤10	≤40	≤100	num
b1	-	-	-	-	-	-	-	-	-	-	**84**	3	3	10	26	**94**	4	2	1	200
b2	-	-	**69**	31	9	**60**	12	23	4	200	**76**	11	12	1	200	**81**	9	10	-	200
b3	**66**	15	18	1	200	**71**	14	15	1	200	**73**	13	13	1	200	**77**	14	9	-	200
b4	**94**	6	-	-	200	**96**	4	-	-	200	**96**	4	-	-	200	**98**	2	-	-	200
b5	**90**	1	9	-	200	**93**	2	5	-	200	**95**	2	3	-	200	**95**	3	2	-	200
b6	**88**	5	7	1	200	**89**	5	6	-	200	**90**	4	6	-	200	**91**	4	5	-	200
b7	**48**	1	**48**	3	200	**74**	5	21	1	200	**83**	6	11	-	200	**89**	6	5	-	200
b8	**46**	-	**51**	3	200	**57**	4	36	2	200	**74**	3	21	1	200	**81**	4	14	1	200
b9	42	-	**56**	2	200	**66**	9	24	1	200	**73**	8	18	-	200	**83**	7	9	-	200
b10	**47**	-	**50**	3	200	**65**	2	30	2	200	**73**	4	22	1	200	**82**	5	13	1	200
b11	38	-	**61**	1	4	**66**	3	31	-	200	**68**	9	23	-	200	**83**	7	10	-	200
b12	**94**	-	5	1	10	**99**	1	-	-	200	**99**	-	-	-	200	**99**	1	-	-	200
b13	**43**	9	34	14	4	**69**	20	11	-	194	**69**	21	10	-	200	**71**	19	10	-	200
b14	-	-	**78**	22	24	**60**	23	17	-	200	**63**	22	15	-	200	**70**	19	11	-	200
b15	41	**53**	5	-	8	**97**	2	1	-	200	**98**	1	1	-	200	**98**	1	1	-	200
b16	**64**	28	6	-	44	**76**	21	2	-	200	**82**	17	1	-	200	**90**	9	1	-	200
b17	33	**66**	1	-	200	**61**	39	-	-	200	**75**	25	-	-	200	**87**	13	-	-	200

gadget is shared. For example, at $p = 10\%$ the rate of pairs that do not share any gadgets ranges from 63% (*b14*) to 99% (*b12*).

Related approaches (discussed in Section 5) report the *average srate* across all pairs for different benchmark sets. Pappas *et al.*'s zero-cost approach [25] achieves an average *srate* between $74\% - 83\%$ without code degradation, comparable to DivCon's $41\% - 99\%$ at $p = 0\%$. Homescu *et al.*'s statistical approach [15] reports an average *srate* between $82\% - 100\%$ with a code degradation of less than 5%, comparable to DivCon's $83\% - 100\%$ at $p = 5\%$. Both approaches report results on larger code bases that exhibit more opportunities for diversification. We expect that DivCon would achieve higher overall survival rates on these code bases compared to the benchmarks used in this paper.

5 Related Work

There are many approaches to software diversification against cyberattacks. The majority apply randomized transformations at different stages of the software development, while a few exceptions use search-based techniques [20]. This section focuses on quality-aware software diversification approaches.

Superdiversifier [17] is a search-based approach for software diversification against cyberattacks. Given an initial instruction sequence, the algorithm generates a random combination of the available instructions and performs a verification test to quickly reject non equivalent instruction sequences. For each non-rejected sequence, the algorithm checks semantic equivalence between the original and the generated instruction sequences using a SAT solver. Superdiversifier affects the code execution time and size by controlling the length of the generated sequence. Along the same lines, Lundquist *et al.* [24,23] use program synthesis for generating program variants against cyberattacks, but no results

are available yet. In comparison, DivCon uses a combinatorial compiler backend that measures the code quality using a more accurate cost model that also considers other aspects, such as execution frequencies.

Most diversification approaches use randomized transformations to generate multiple program variants [20]. Unlike DivCon, the majority of these approaches do not control the quality of the generated variants during diversification but rather evaluate it afterwards [10,34,18,14,5,9]. However, there are a few approaches that control the code quality during randomization.

Some compiler-based diversification approaches restrict the set of program transformations to control the quality of the generated code [9,25]. For example, Pappas *et al.* [25] perform software diversification at the binary level and apply three zero-cost transformations: register randomization, instruction schedule randomization, and function shuffling. In contrast, DivCon's combinatorial approach allows it to control the aggressiveness and potential cost of its transformations: a cost overhead limit of 0% forces DivCon to apply only zero-cost transformations; a larger limit allows DivCon to apply more aggressive transformations, potentially leading to higher diversity.

Homescu *et al.* [15] perform only garbage (`nop`) insertion, and use a profile-guided approach to reduce the overhead. To do this, they control the `nop` insertion probability based on the execution frequency of different code sections. In contrast, DivCon's cost model captures different execution frequencies, which allows it to perform more aggressive transformations in non-critical code sections.

6 Conclusion and Future Work

This paper introduces DivCon, a CP approach to compiler-based, quality-aware software diversification against code-reuse attacks. Our experiments show that LNS is a promising technique for a CP-based exploration of the space of diverse program, with a fine-grained control on the trade-off between code quality and diversity. In particular, we show that the set of optimal solutions naturally contains a set of diverse solutions, which increases significantly when relaxing the constraint of optimality. Our experiments demonstrate that the diverse solutions generated by DivCon are effective to mitigate code-reuse attacks.

Future work includes investigating different distance measures to further reduce the gadget survival rate, improving the overall scalability of DivCon in the face of larger programs and larger values of parameter k, and examining the effectiveness of DivCon against an actual code-reuse exploit.

Acknowledgments. We would like to give a special acknowledgment to Christian Schulte, for his critical contribution at the early stages of this work. Although no longer with us, Christian continues to inspire his students and colleagues with his lively character, enthusiasm, deep knowledge and understanding. We would also like to thank Linnea Ingmar and the anonymous reviewers for their useful feedback, and Oscar Eriksson for proof reading.

References

1. Alaba, F.A., Othman, M., Hashem, I.A.T., Alotaibi, F.: Internet of Things security: A survey. Journal of Network and Computer Applications **88**, 10–28 (Jun 2017). https://doi.org/10.1016/j.jnca.2017.04.002
2. Baudry, B., Monperrus, M.: The Multiple Facets of Software Diversity: Recent Developments in Year 2000 and Beyond. ACM Comput. Surv. **48**(1), 16:1–16:26 (Sep 2015). https://doi.org/10.1145/2807593
3. Birman, K.P., Schneider, F.B.: The monoculture risk put into context. IEEE Security & Privacy **7**(1), 14–17 (2009)
4. Bletsch, T., Jiang, X., Freeh, V.W., Liang, Z.: Jump-oriented Programming: A New Class of Code-reuse Attack. In: Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security. pp. 30–40. ASIACCS '11, ACM, New York, NY, USA (2011)
5. Braden, K., Crane, S., Davi, L., Franz, M., Larsen, P., Liebchen, C., Sadeghi, A.R.: Leakage-Resilient Layout Randomization for Mobile Devices. In: Proceedings 2016 Network and Distributed System Security Symposium. Internet Society, San Diego, CA (2016). https://doi.org/10.14722/ndss.2016.23364
6. Castañeda Lozano, R., Carlsson, M., Blindell, G.H., Schulte, C.: Combinatorial Register Allocation and Instruction Scheduling. ACM Trans. Program. Lang. Syst. **41**(3), 17:1–17:53 (Jul 2019)
7. Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.R., Shacham, H., Winandy, M.: Return-oriented Programming Without Returns. In: Proceedings of the 17th ACM Conference on Computer and Communications Security. pp. 559–572. CCS '10, ACM, New York, NY, USA (2010)
8. Chu, G.G.: Improving combinatorial optimization. Ph.D. thesis, The University of Melbourne, Australia (2011)
9. Crane, S., Liebchen, C., Homescu, A., Davi, L., Larsen, P., Sadeghi, A., Brunthaler, S., Franz, M.: Readactor: Practical Code Randomization Resilient to Memory Disclosure. In: 2015 IEEE Symposium on Security and Privacy. pp. 763–780 (May 2015). https://doi.org/10.1109/SP.2015.52
10. Davi, L.V., Dmitrienko, A., Nürnberger, S., Sadeghi, A.R.: Gadge me if you can: secure and efficient ad-hoc instruction-level randomization for x86 and ARM. In: Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security. pp. 299–310 (2013), tex.organization: ACM
11. Gecode Team: Gecode: Generic constraint development environment (2020), `https://www.gecode.org`
12. Hamming, R.W.: Error detecting and error correcting codes. The Bell system technical journal **29**(2), 147–160 (1950). https://doi.org/10.1002/j.1538-7305.1950.tb00463.x
13. Hebrard, E., Hnich, B., O'Sullivan, B., Walsh, T.: Finding Diverse and Similar Solutions in Constraint Programming. In: National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference. p. 6 (2005)
14. Homescu, A., Jackson, T., Crane, S., Brunthaler, S., Larsen, P., Franz, M.: Large-Scale Automated Software Diversity—Program Evolution Redux. IEEE Transactions on Dependable and Secure Computing **14**(2), 158–171 (Mar 2017). https://doi.org/10.1109/TDSC.2015.2433252
15. Homescu, A., Neisius, S., Larsen, P., Brunthaler, S., Franz, M.: Profile-guided Automated Software Diversity. In: Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). pp.

1–11. CGO '13, IEEE Computer Society, Washington, DC, USA (2013). https://doi.org/10.1109/CGO.2013.6494997
16. Ingmar, L., de la Banda, M.G., Stuckey, P.J., Tack, G.: Modelling diversity of solutions. In: Proceedings of the thirty-fourth AAAI conference on artificial intelligence (2020)
17. Jacob, M., Jakubowski, M.H., Naldurg, P., Saw, C.W.N., Venkatesan, R.: The Superdiversifier: Peephole Individualization for Software Protection. In: Matsuura, K., Fujisaki, E. (eds.) Advances in Information and Computer Security. pp. 100–120. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (2008). https://doi.org/10.1007/978-3-540-89598-5_7
18. Koo, H., Chen, Y., Lu, L., Kemerlis, V.P., Polychronakis, M.: Compiler-Assisted Code Randomization. In: 2018 IEEE Symposium on Security and Privacy (SP). pp. 461–477 (May 2018). https://doi.org/10.1109/SP.2018.00029
19. Krueger, C.W.: Software reuse. ACM Comput. Surv. **24**(2), 131?183 (Jun 1992). https://doi.org/10.1145/130844.130856
20. Larsen, P., Homescu, A., Brunthaler, S., Franz, M.: SoK: Automated Software Diversity. In: 2014 IEEE Symposium on Security and Privacy. pp. 276–291 (May 2014). https://doi.org/10.1109/SP.2014.25
21. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis & transformation. In: Code Generation and Optimization. IEEE (2004)
22. Lee, C., Potkonjak, M., Mangione-Smith, W.H.: MediaBench: A tool for evaluating and synthesizing multimedia and communicatons systems. In: International Symposium on Microarchitecture. pp. 330–335. IEEE (1997)
23. Lundquist, G.R., Bhatt, U., Hamlen, K.W.: Relational processing for fun and diversity. In: Proceedings of the 2019 miniKanren and relational programming workshop. p. 100 (2019)
24. Lundquist, G.R., Mohan, V., Hamlen, K.W.: Searching for Software Diversity: Attaining Artificial Diversity Through Program Synthesis. In: Proceedings of the 2016 New Security Paradigms Workshop. pp. 80–91. NSPW '16, ACM, New York, NY, USA (2016). https://doi.org/10.1145/3011883.3011891
25. Pappas, V., Polychronakis, M., Keromytis, A.D.: Smashing the Gadgets: Hindering Return-Oriented Programming Using In-place Code Randomization. In: 2012 IEEE Symposium on Security and Privacy. pp. 601–615 (May 2012). https://doi.org/10.1109/SP.2012.41
26. Petit, T., Trapp, A.C.: Finding Diverse Solutions of High Quality to Constraint Optimization Problems. In: Twenty-Fourth International Joint Conference on Artificial Intelligence (Jun 2015)
27. Salwan, J.: ROPgadget Tool (2020), `http://shell-storm.org/project/ROPgadget/`
28. Shacham, H.: The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86). In: Proceedings of the 14th ACM Conference on Computer and Communications Security. pp. 552–561. CCS '07, ACM, New York, NY, USA (2007)
29. Shaw, P.: Using constraint programming and local search methods to solve vehicle routing problems. In: Principles and Practice of Constraint Programming. Lecture Notes in Computer Science, vol. 1520, pp. 417–431. Springer (1998)
30. SPEC: CPU 2006 Benchmarks (2020), `https://www.spec.org/cpu2006`, accessed on 2020-03-20
31. Sweetman, D.: See MIPS Run, Second Edition. Morgan Kaufmann (2006)

32. Van Hentenryck, P., Coffrin, C., Gutkovich, B.: Constraint-Based Local Search for the Automatic Generation of Architectural Tests. In: Gent, I.P. (ed.) Principles and Practice of Constraint Programming - CP 2009. pp. 787–801. Lecture Notes in Computer Science, Springer Berlin Heidelberg (2009)
33. Wagner, R.A., Fischer, M.J.: The string-to-string correction problem. Journal of the ACM (JACM) **21**(1), 168–173 (1974)
34. Wang, S., Wang, P., Wu, D.: Composite Software Diversification. In: 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME). pp. 284–294 (Sep 2017). https://doi.org/10.1109/ICSME.2017.61

Appendix B

Publication 2

Journal of Artificial Intelligence Research 72 (2021) 1471-1505 Submitted 03/2021; published 12/2021

Constraint-based Diversification of JOP Gadgets

Rodothea Myrsini Tsoupidi TSOUPIDI@KTH.SE
Royal Institute of Technology, KTH,
Stockholm, Sweden

Roberto Castañeda Lozano ROBERTO.CASTANEDA@ED.AC.UK
University of Edinburgh,
Edinburgh, United Kingdom

Benoit Baudry BAUDRY@KTH.SE
Royal Institute of Technology, KTH,
Stockholm, Sweden

Abstract

Modern software deployment process produces software that is uniform, and hence vulnerable to large-scale code-reuse attacks, such as *Jump-Oriented Programming (JOP)* attacks. *Compiler-based diversification* improves the resilience and security of software systems by automatically generating different assembly code versions of a given program. Existing techniques are efficient but do not have a precise control over the quality, such as the code size or speed, of the generated code variants.

This paper introduces *Diversity by Construction (DivCon)*, a constraint-based compiler approach to software diversification. Unlike previous approaches, DivCon allows users to control and adjust the conflicting goals of diversity and code quality. A key enabler is the use of Large Neighborhood Search (LNS) to generate highly diverse assembly code efficiently. For larger problems, we propose a combination of LNS with a structural decomposition of the problem. To further improve the diversification efficiency of DivCon against JOP attacks, we propose an application-specific distance measure tailored to the characteristics of JOP attacks.

We evaluate DivCon with 20 functions from a popular benchmark suite for embedded systems. These experiments show that DivCon's combination of LNS and our application-specific distance measure generates binary programs that are highly resilient against JOP attacks (they share between 0.15% to 8% of JOP gadgets) with an optimality gap of $\leq$ 10%. Our results confirm that there is a trade-off between the quality of each assembly code version and the diversity of the entire pool of versions. In particular, the experiments show that DivCon is able to generate binary programs that share a very small number of gadgets, while delivering near-optimal code.

For constraint programming researchers and practitioners, this paper demonstrates that LNS is a valuable technique for finding diverse solutions. For security researchers and software engineers, DivCon extends the scope of compiler-based diversification to performance-critical and resource-constrained applications.

1. Introduction

Common software development practices, such as code reuse (Krueger, 1992) and automatic updates, contribute to the emergence of software monocultures (Birman & Schneider, 2009). While such monocultures facilitate software distribution, bug reporting, and software au-

thentication, they also introduce serious risks related to the wide spreading of attacks against all users that run identical software.

Embedded devices, such as controllers in cars or medical implants, which manage sensitive and safety-critical data, are particularly exposed to this class of attacks (Kornau et al., 2010; Bletsch et al., 2011). Yet, this type of software usually cannot afford expensive defense mechanisms (Salehi et al., 2019).

Software diversification is a method to mitigate the problems caused by software monocultures, initially explored in the seminal work of Cohen (1993) and Forrest, Somayaji, and Ackley (1997). Similarly to biodiversity, software diversification improves the resilience and security of a software system (Baudry & Monperrus, 2015) by introducing diverse variants of code in it. Software diversification can be applied in different phases of the software development cycle, i.e. during implementation, compilation, loading, or execution (Larsen et al., 2014). This paper is concerned with *compiler-based* diversification, which automatically generates different binary code versions from a single source program.

Modern compilers do not merely aim to generate correct code, but also code that is of high quality. There exists a variety of compilation techniques to optimize code for speed or size (Ashouri et al., 2018). However, there exist few compiler techniques that target code diversification. These techniques are effective at synthesizing diverse variants of assembly code for one source program (Larsen et al., 2014). However, they do not have a precise control over other binary code quality metrics, such as speed or size. These techniques (discussed in Section 5) are either based on randomizing heuristics or in high-level superoptimization methods that do not capture accurately the quality of the generated code.

This paper introduces Diversity by Construction (DivCon), a compiler-based diversification approach that allows users to control and adjust the conflicting goals of quality of each code version and diversity among all versions. DivCon uses a Constraint Programming (CP)-based compiler backend to generate diverse solutions corresponding to functionally equivalent program variants according to an accurate code quality model. The backend models the input program, the hardware architecture, and the compiler transformations as a constraint problem, whose solutions correspond to assembly code for the input program. The synthesis of code diversity is motivated by Jump-Oriented Programming (JOP) attacks (Checkoway et al., 2010; Bletsch et al., 2011) that exploit the presence of certain binary code snippets, called JOP gadgets, to craft an exploit. Our goal is to generate binary variants that are functionally equivalent, yet do not have the same gadgets and hence cannot be targeted by the exact same JOP attack.

The use of CP makes it possible to 1) control the quality of the generated solutions by constraining the objective function, 2) introduce constraints tailored towards JOP gadgets, and 3) apply search procedures that are particularly suitable for diversification. More specifically, we propose the use of Large Neighborhood Search (LNS) (Shaw, 1998), a popular metaheuristic in multiple application domains, to generate highly diverse binaries. For larger problems, we investigate a combination of LNS with a structural decomposition of the problem. Focusing on our application, DivCon provides different distance measures that trade diversity for scalability.

Our experiments compiling 14 functions from a popular embedded systems suite to the MIPS32 architecture confirm that there is a trade-off between code quality and diversity. We demonstrate that DivCon allows users to navigate this space of near-optimal, diverse

assembly code for a range of quality bounds. We show that the Paretto front of optimal solutions synthesized by DivCon with LNS and a distance measure tailored against JOP attacks, naturally includes code variants with few common gadgets. We show that DivCon is able to synthesize significantly diverse variants, while guaranteeing a code quality of 10% within optimality. We further evaluate an additional set of six functions, which belong to the set of the 30% largest functions of the benchmark suite, to investigate the scalability of DivCon.

For constraint programming researchers and practitioners, this paper demonstrates that LNS is a valuable technique for finding diverse solutions. For security researchers and software engineers, DivCon extends the scope of compiler-based diversification to performance-critical and resource-constrained applications, and provides a solid step towards secure-by-construction software.

To summarize, the main contributions of this paper are:

- the first CP-based technique for compiler-based, quality-aware software diversification;
- an experimental demonstration of the effectiveness of LNS at generating highly diverse solutions efficiently;
- the evaluation of DivCon on a wide set of benchmarks of different sizes, including large functions of up to 500 instructions;
- a quantitative assessment of the technique to mitigate code-reuse attacks effectively, while preserving high code quality; and
- a publicly available tool for constraint-based software diversification[1].

This paper extends our previous work (Tsoupidi, Castañeda Lozano, & Baudry, 2020). We extend our investigation of LNS for code diversification with Decomposition-based Large Neighborhood Search (DLNS) (Sections 3.2, 4.2, and 4.4), a specific LNS-based approach for generating diverse solutions for larger programs. We propose a new distance measure to explore the space of program variants, which specifically targets JOP gadgets: Gadget Distance (GD) (Sections 3.3, 4.3, and 4.5). We perform a new set of experiments to compare the diversification algorithms and the distance measures, with 19 new benchmark functions up to 16 times larger than our previous dataset, providing new insights on the scalability of our approach (Section 4.2). Finally, we add a case study on a voice compression application, which provides a more complete picture on whole-program, multi-function diversification using DivCon (Section 4.7).

2. Background

This section describes code-reuse attacks (Section 2.1), diversification approaches in CP (Section 2.3), and combinatorial compiler backends (Section 2.4).

2.1 JOP Attacks

Code-reuse attacks take advantage of memory vulnerabilities, such as buffer overflows, to reuse program legitimate code and repurpose it for malicious usages. More specifically, code-reuse attacks insert data into the program memory to affect the control flow of the

1. https://github.com/romits800/divcon

```
1  0x9d001408: ...
2  0x9d00140c: lw    $s2, 4($sp)
3  0x9d001410: lw    $s4, 0($sp)
4  0x9d001414: jr    $t9
5  0x9d001418: addiu $sp, $sp, 16
```

(a) Original gadget.

```
1  0x9d001408: lw    $s2, 4($sp)
2  0x9d00140c: nop
3  0x9d001410: lw    $s4, 0($sp)
4  0x9d001414: jr    $t8
5  0x9d001418: addiu $sp, $sp, 16
```

(b) Diversified gadget.

Figure 1: Example gadget diversification in MIPS32 assembly code

program. Consequently, the original, valid code is executed but the modified control flow triggers and executes code that is valid but unintended.

Return-Oriented Programming (ROP) (Shacham, 2007) is a code-reuse attack that combines different snippets from the original binary code to form a Turing complete language for attackers. The building blocks of a ROP attack are the *gadgets*: meta-instructions that consist of one or multiple code snippets with specific semantics. The original publication considers the x86 architecture and the gadgets terminate with a `ret` instruction. Later publications generalize ROP for different architectures and in the absence of `ret` instructions, such as JOP (Checkoway et al., 2010; Bletsch et al., 2011). This paper focuses on JOP due to the characteristics of MIPS32, but could be generalized to other code-reuse attacks. The code snippets for a JOP attack terminate with a branch instruction. Figure 1a shows a JOP gadget found by the *ROPgadget* tool (Salwan, 2020) in a MIPS32 binary. Assuming that the attacker controls the stack, lines 2 and 3 load attacker data in registers $s2 and $s4, respectively. Then, line 4 jumps to the address of register $t9. The last instruction (line 5) is placed in a delay slot and hence it is executed before the jump (Sweetman, 2006). The semantics of this gadget depends on the attack payload and might be to load a value to register $s2 or $s4. Then, the program jumps to the next gadget, which resides at the stack address of $t9.

Statically designed JOP attacks use the absolute binary addresses for installing the attack payload. Hence, a simple change in the instruction schedule of the program as in Figure 1b prevents a JOP attack designed for Figure 1a. An attacker that designs an attack based on the binary of the original program assumes the presence of a gadget (Figure 1a) at position 0x9d00140c. However, in the diversified version, address 0x9d00140c does not start with the initial `lw` instruction of Figure 1a, and by the end of the execution of the gadget, register $s2 does not contain the attacker data. Moreover, by assigning a different jump target register, $t8, the next target will not be the one expected by the attacker. In this way, diversification can break the semantics of the gadget and mitigate an attack against the diversified code.

2.2 Attack Model

We assume an attack model, where the attacker 1) knows the original C code of the application, but 2) does not know the exact variant that each user runs, i.e. we assume that each user runs a different diversified version of the program, as suggested by Larsen et al. (2014). Also, 3) we assume the existence of a memory corruption vulnerability that enables a buffer overflow. The defenses of the users include, Data Execution Prevention (DEP) (or $W \oplus X$),

which ensures that no writable memory (W) is executable (X) and vice versa. This ensures that the attacker is not able to execute code that is directly inserted into the executable code memory, for example the program stack.

For more advanced attacks, like JIT-ROP attacks (Snow, Monrose, Davi, Dmitrienko, Liebchen, & Sadeghi, 2013), we discuss later (Section 4.8) possible configurations using our approach.

2.3 Diversity in Constraint Programming

While typical CP applications aim to discover either some solution or the optimal solution, some applications require finding *diverse* solutions for various purposes.

Hebrard et al. (2005) introduce the MaxDiverse$k$Set problem, which is the problem of finding the most diverse set of k solutions, and propose an exact and an incremental algorithm for solving it. The exact algorithm does not scale to a large number of solutions (Van Hentenryck et al., 2009; Ingmar et al., 2020). The incremental algorithm selects solutions iteratively by solving a distance maximization problem.

Automatic Generation of Architectural Tests (ATGP) is an application of CP that requires generating many diverse solutions. Van Hentenryck et al. (2009) model ATGP as a MaxDiverse$k$Set problem and solve it using the incremental algorithm of Hebrard et al. (2005). Due to the large number of diverse solutions required (50-100), Van Hentenryck et al. (2009) replace the maximization step with local search.

In software diversity, solution quality is of paramount importance. In general, earlier CP approaches to diversity are concerned with satisfiability only. An exception is the approach of Petit and Trapp (2015). This approach modifies the objective function for assessing both solution quality and solution diversity, but does not scale to the large number of solutions required by software diversity. Ingmar et al. (2020) propose a generic framework for modeling diversity in CP. For tackling the quality-diversity trade-off, they propose constraining the objective function with the optimal (or best known) cost o. DivCon applies this approach by allowing solutions p% worse than o, where p is a user-defined parameter.

2.4 Compiler Optimization as a Combinatorial Problem

A Constraint Satisfaction Problem (CSP) is a problem specification $P = \langle V, U, C \rangle$, where V are the problem variables, U is the domain of the variables, and C the constraints among the variables. A Constraint Optimization Problem (COP), $P = \langle V, U, C, O \rangle$, consists of a CSP and an objective function O. The goal of a COP is to find a solution that optimizes O.

Compilers are programs that generate low-level assembly code, typically optimized for *speed* or *size*, from higher-level source code. A compilation process can be modeled as a COP by letting V be the decisions taken during the translation, C be the constraints that the program semantics and the hardware resources impose, and O be the cost of the generated code.

Compiler backends typically generate low-level assembly code from an Intermediate Representation (IR), a program representation that is independent of both the source and the target language. Figure 2 shows the high-level view of a *combinatorial* compiler backend. A combinatorial compiler backend takes as input the IR of a program, generates and solves

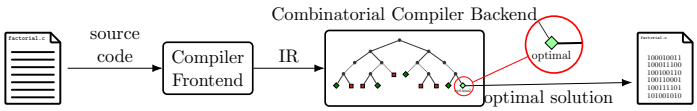


Figure 2: High-level view of a combinatorial compiler backend

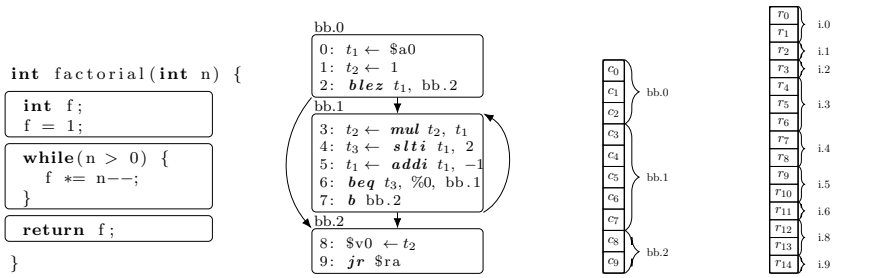
a COP, and outputs the optimized low-level assembly code described by the solution to the COP.

This paper assumes that programs at the IR level are represented by their Control-Flow Graph (CFG). A CFG is a representation of the possible execution paths of a program, where each node corresponds to a *basic block* and edges correspond to intra-block jumps. A *basic block* is a set of abstract instructions (hereafter just *instructions*) with no branches besides the end of the block. Each instruction is associated with a set of operands characterizing its input and output data. Typical decision variables V of a combinatorial compiler backend are the issue cycle $c_i \in \mathbb{N}_0$ of each instruction i, the processor instruction $m_i \in \mathbb{N}_0$ that implements each instruction i, and the processor register $r_o \in \mathbb{N}_0$ assigned to each operand o.

Figure 3a shows an implementation of the factorial function in C where each basic block is highlighted. Figure 3b shows the IR of the program. The example IR contains 10 instructions in three basic blocks: bb.0, bb.1, and bb.2. Basic block bb.0 corresponds to initializations, where `$a0` holds the function argument `n`, and t_1 corresponds to variable `f`. bb.1 computes the factorial in a loop by accumulating the result in t_2. bb.2 stores the result to `$v0` and returns. Some instructions in the example are interdependent, which leads to serialization of the instruction schedule. For example, ***beq*** (6) consumes data (t_3) defined by ***slti*** (4) and hence needs to be scheduled later. Instruction dependencies limit the amount of possible assembly code versions and may restrict diversity significantly. Finally, Figure 3c shows the arrangement of the issue-cycle variables in the constraint model used by the combinatorial compiler backend. Similarly, Figure 3d shows the arrangement of the register variables.

The CFG representation of a program offers a natural decomposition of the COP into subproblems, each consisting of a basic block. This partitioning requires first solving the *global* problem that assigns registers to the program variables that are live (active) through different basic blocks (Castañeda Lozano et al., 2012). For example, in Figure 3b, the global problem has to assign a register to t_1 because both bb.0 and bb.1 use it. Subsequently, it is possible to solve the COP by optimizing each of the *local* problems (for every basic block), independently.

DivCon aims at mitigating code-reuse attacks. Therefore, DivCon considers the order of the instructions and the assignment of registers to their operands in the final binary, which directly affects the feasibility of code-reuse attacks (see Figures 1a and 1b). For this reason, the diversification model uses the issue-cycle sequence of instructions, $c = \{c_0, c_1, ..., c_n\}$, and the register allocation, $r = \{r_0, r_1, ..., r_n\}$, to characterize the diversity among different solutions.



(a) C code (b) IR (c) Issue cycles (d) Register allocation

Figure 3: Factorial function example

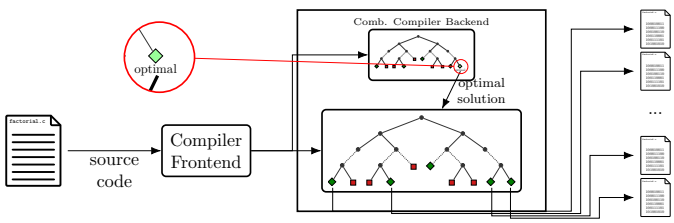


Figure 4: High-level view of DivCon

3. DivCon

This section introduces DivCon, a software diversification method that uses a combinatorial compiler backend to generate program variants. Figure 4 shows a high-level view of the diversification process. DivCon uses 1) the optimal solution (see Definition 1) to start the *search* for diversification and 2) the cost of the optimal solution to restrict the variants within a maximum gap from the optimal. Subsequently, DivCon generates a number of solutions to the CSP that correspond to diverse program variants.

The rest of this section describes the diversification approach of DivCon. Section 3.1 formulates the diversification problem in terms of the constraint model of a combinatorial compiler backend, Section 3.2 defines the proposed diversification algorithms, Section 3.3 defines the distance measures, and finally, Section 3.4 describes the search strategy for generating program variants.

3.1 Problem Description

In this section, we will define the program diversification problem and stress important concepts that we will use later in the evaluation part (Section 4). Let $P = \langle V, U, C \rangle$ be the compiler backend CSP for the program under compilation and O the objective function of the COP, $\langle V, U, C, O \rangle$.

Definition 1 ***Optimal solution*** *is the solution $y_{opt} \in sol(P)$ that the combinatorial compiler backend (see Section 2.4) returns and for which $O(y_{opt}) = o$.*

We then define the *optimality gap* as follows:

Definition 2 ***Optimality gap*** *is the ratio, $p \in \mathbb{R}_{\geq 0}$, that constrains the optimization function, such that $\forall s \in sol(P) \,.\, O(s) \leq (1 + p) \cdot o$.*

We define the *distance* function (three such functions are defined in Section 3.3) as follows:

Definition 3 ***Distance*** *$\delta(s_1, s_2)$ is a function that measures the distance between two solutions of P, $s_1, s_2 \in sol(P)$.*

Let parameter $h \in \mathbb{N}$ be the minimum allowed pairwise distance between two generated solutions. Our problem is to find a subset of the solutions to the CSP, $S \subseteq sol(P)$, such that:

$$\forall s_1, s_2 \in S \,.\, s_1 \neq s_2 \implies \delta(s_1, s_2) \geq h \text{ and } \forall s \in S \,.\, O(s) \leq (1 + p) \cdot o \tag{1}$$

To solve the above problem, this paper proposes two LNS-based incremental algorithms defined in Section 3.2. LNS is a metaheuristic that allows searching for solutions in large parts of the search tree. This property makes LNS a good candidate for generating a large number of diverse solutions. To guarantee that the new variants are sufficiently different from each other, we define three distance measures (Section 3.3) that quantify the concept of program difference for our application.

3.2 Diversification Algorithms

This section presents two LNS-based algorithms for the generation of a large number of solutions for software diversification. The first algorithm (Algorithm 1), referred to as simply LNS, solves the problem monolithically using an LNS-based approach. The second algorithm, DLNS (Algorithm 2), decomposes the problem into subproblems and uses LNS to diversify each of these subproblems independently and in parallel. The final solutions are then composed by randomly combining the solutions of the subproblems.

LNS Algorithm. Algorithm 1 presents a monolithic LNS-based diversification algorithm. It starts with the optimal solution y_{opt} (line 3). Subsequently, the algorithm adds a distance constraint for y_{opt} and the optimality constraint with $o = O(y_{opt})$ (line 4). While the termination condition is not fulfilled (line 5), the algorithm uses LNS as described in Section 3.4 to find the next solution y (line 6), adds the next solution to the solution set S (line 7), and updates the distance constraints based on the latest solution (line 8). When the termination condition is satisfied, the algorithm returns the set of solutions S corresponding to diversified assembly code variants (line 9).

In our experience, our application does not require large values of h because even small distance between variants breaks gadgets (see Figure 1). An alternative algorithm that may improve Algorithm 1 for larger values of h, is replacing $\texttt{solve}_{LNS}$ on line 6 and the constraint update on line 8 with an LNS maximization step that returns a solution by iteratively improving its pairwise distance with all current solutions in S until reaching the value of h.

Algorithm 1: Incremental algorithm for generating diverse solutions

```
1   function solve_lns(y_opt , ⟨V,U,C⟩)
2   begin
3     S ← {y_opt}, y ← y_opt ,
4     C′ ← C ∪ {δ(y_opt) ≥ h , O(V) ≤ (1 + p) · o}
5     while not term_cond() // e.g. |S| > k ∨ time_limit()
6       y ← solve_LNS(relax(y), ⟨V,U,C′⟩)
7       S ← S ∪ {y}
8       C′ ← C′ ∪ {δ(y,s) ≥ h | ∀s ∈ sol(⟨V,U,C′⟩) }
9     return S
10  end
```

Decomposition Algorithm. This section presents DLNS (Algorithm 2), an LNS-based algorithm that uses decomposition to enable diversification of large functions. To enable this, the algorithm divides the problem into a *global* problem and a set of *local* subproblems, one for each basic block of the function.

Algorithm 2 starts by adding the optimal solution to the set of solutions (line 3) and continues by adding the optimality constraints (line 4). While the termination condition is not satisfied, the algorithm solves the *global* problem (line 7). After finding a global solution, the algorithm solves the *local* problems for each basic block $b \in B$ in parallel and generates a number of *local* solutions for each basic block (lines 9 and 10). Then, the algorithm combines one randomly selected solution for each basic block (line 13). This combined solution may be invalid (line 14) due to, for example, exceeded *cost*. In case the solution is valid (line 14), the algorithm adds this solution to the set of solutions S (line 15) and, finally, adds a diversity constraint to the problem (line 16).

Algorithm 2: Decomposition-based incremental algorithm for generating diverse solutions

```
1   function solve_decomp_lns(y_opt , ⟨V,U,C⟩)
2   begin
3     S ← {y_opt}, y ← y_opt ,
4     C′ ← C ∪ {δ(y_opt) ≥ h , O(V) ≤ (1 + p) · o}
5     while not term_cond() // e.g. |S| > k ∨ time_limit()
6       // Find partial solution
7       y ← psolve_LNS(relax(y), ⟨V,U,C′⟩)
8       // Solve local problems
9       for b ∈ B
10          S_b ← spawn solve_lns(y_b , ⟨V_b,U_b,C′_b⟩)
11      // Select solutions
12      for |S_1 × S_2 × ... × S_b|
13          y ← combine(∀ b ∈ B.∃ y_b ∈ S_b . y_b , ⟨V,U,C′⟩)
14          if valid(y):
15              S ← S ∪ {y}
16              C′ ← C′ ∪ { δ(y,s) ≥ h | ∀s ∈ sol(⟨V,U,C′⟩) }
17  end
```

Example. Figure 5 shows two MIPS32 variants of the factorial example (Figure 3), which correspond to two solutions of DivCon. The variants differ in two aspects: first, the ***beqz*** instruction is issued one cycle later in Figure 5b than in Figure 5a, and second, the temporary variable t_3 (see Figure 3) is assigned to different MIPS32 registers (`$t0` and `$t1`).

LNS diversifies the function that consists of three basic blocks by finding different solutions that assign values to the registers and the instruction schedule simultaneously. DLNS solves first the global problem by assigning registers to the temporary variables that are live across multiple basic blocks (t_1 and t_2) and then assigns the issue schedule and the rest of the registers for each basic block, independently and possibly in parallel. The diversified variants in Figure 5 serve presentation purposes. Figure 7 in Appendix B presents a more elaborated example of two diversified functions.

3.3 Distance Measures

This section defines three alternative distance measures: Hamming Distance (HD), Levenshtein Distance (LD), and Gadget Distance (GD). HD and LD operate on the schedule of the instructions, i.e. the order in which the instructions are issued in the CPU, whereas GD operates on both the instruction schedule and the register allocation, i.e. the hardware register for each operand. Early experimental results that we have performed have shown that diversifying register allocation is less effective than diversifying the instruction schedule against code-reuse attacks. However, restricting register allocation diversity to the instructions very near a branch instruction (a key component of a JOP gadget), improves DivCon's gadget diversification effectiveness.

Hamming Distance (HD). HD is the Hamming distance (Hamming, 1950) between the issue-cycle variables of two solutions. Given two solutions $s, s' \in sol(P)$:

$$\delta_{HD}(s, s') = \sum_{i=0}^{n} (s(c_i) \neq s'(c_i)), \tag{2}$$

where n is the maximum number of instructions.

Consider Figure 1b, a diversified version of the gadget in Figure 1a. The only instruction that differs from Figure 1a is the instruction at line 1 that is issued one cycle before. The two examples have a HD of one, which in this case is enough for breaking the functionality of the original gadget (see Section 2.1).

Levenshtein Distance (LD). Levenshtein Distance (or edit distance) measures the minimum number of edits, i.e. insertions, deletions, and replacements, that are necessary for transforming one instruction schedule to another. Compared to HD, which considers only

```
1  bb.0: blez  $a0, bb.2
2        addiu $v0, $zero, 1
3  bb.1: mul   $v0, $v0, $a0
4        slti  $t0, $a0, 2
5        beqz  $t0, bb.1
6        addi  $a0, $a0, -1
7  bb.2: jr    $ra
8        nop
```

(a) Variant 1

```
1  bb.0: blez  $a0, bb.2
2        addiu $v0, $zero, 1
3  bb.1: mul   $v0, $v0, $a0
4        slti  $t1, $a0, 2
5        nop
6        beqz  $t1, bb.1
7        addi  $a0, $a0, -1
8  bb.2: jr    $ra
9        nop
```

(b) Variant 2

Figure 5: Two MIPS32 variants of the factorial example in Figure 3

replacements, LD also considers *insertions* and *deletions*. To understand this effect, consider Figure 5. The two gadgets differ only by one `nop` operation but HD gives a distance of three, whereas LD gives one, which is more accurate. LD takes ordered vectors as input, and thus requires an ordered representation (as opposed to a detailed schedule) of the instructions. Therefore, LD uses vector $c^{-1} = channel(c)$, a sequence of instructions ordered by their issue cycle. Given two solutions $s, s' \in sol(P)$:

$$\delta_{LD}(s, s') = \texttt{levenshtein\_distance}(s(c^{-1}), s'(c^{-1})), \tag{3}$$

where `levenshtein_distance` is the WagnerFischer algorithm (Wagner & Fischer, 1974) with time complexity $O(nm)$, where n and m are the lengths of the two sequences.

Gadget Distance (GD). GD is an application-specific distance measure targeting JOP gadgets that we propose in this paper. GD operates on both register allocation and instruction scheduling, focusing on the instructions preceding a branch instruction because JOP gadgets terminate with a branch instruction. In this way, GD enforces the program variants to differ with regards to the gadgets. Here, the set of branch instructions, B, consists of all indirect *jump* or *call* instructions (e.g. line 7 in Figure 5a). A gadget may also use a direct jump (e.g. line 5 in Figure 5a). However, the majority of gadgets require control over the jump target, which is not possible with direct jumps. GD uses two configuration parameters, n_c and n_r. Parameter n_c denotes the number of instructions before each branch, $br \in B$, that the issue cycle of two variants may differ. Similarly, parameter n_r denotes the number of instructions preceding a branch of two variants that the register assignment of the instruction operands may differ. Consider Figure 1b. The two gadgets differ by one `nop` instruction and a different register at instruction 4. Then, the GD distance is two, given $n_c = 3$ and $n_r = 0$.

Given two solutions $s, s' \in sol(P)$, the partial distance $\delta_{PGD}^{n_r,n_c}$ on branch $br \in B$ is :

$$\delta_{PGD}^{n_r,n_c}(s, s', br) = \sum_{i=0}^{N_i} \left(f(s, n_c, i, br)(s(c_i) \neq s'(c_i)) + \sum_{p \in ps(c_i)} f(s, n_r, i, br)(s(r_p) \neq s'(r_p)) \right), \tag{4}$$

where N_i is the number of instructions, $ps(c_i)$ is the set of operands in instruction i, and $f(s, n, i, br)$ is a function that takes four inputs, i) one solution $s \in S$, ii) a natural number that corresponds to the allowed distance of an instruction i from a branch instruction br, iii) instruction i, and iv) branch instruction br. The definition of f is as follows:

$$f(s, n, i, br) = \begin{cases} 1, & s(c_{br}) - s(c_i) \in [0, n] \\ 0, & \text{otherwise} \end{cases}. \tag{5}$$

Finally, given two solutions $s, s' \in sol(P)$, the Gadget Distance $\delta_{GD}^{n_r,n_c}$ is defined as:

$$\delta_{GD}^{n_r,n_c}(s, s') = \sum_{br \in B} \left(\delta_{PGD}^{n_r,n_c}(s, s', br) \right). \tag{6}$$

Note that in Algorithm 1 and Algorithm 2, GD will result in a number of constraints equal to the number of branches in B plus one.

Table 1: Original and Random branching strategies

(a) Original branching strategy

Variable	Var. Selection	Value Selection
c_i	in order	min. val first
m_i	in order	min. val first
r_o	in order	randomly

(b) Random branching strategy

Variable	Var. Selection	Value Selection
c_i	randomly	randomly
m_i	randomly	randomly
r_o	randomly	randomly

3.4 Search

Unlike previous CP approaches to diversity, DivCon employs Large Neighborhood Search (LNS) (Shaw, 1998) for diversification. LNS is a metaheuristic that defines a neighborhood, in which *search* looks for better solutions, or in our case, different solutions. The definition of the neighborhood is through a *destroy* and a *repair* function. The *destroy* function unassigns a subset of the variables in a given solution and the *repair* function finds a new solution by assigning new values to the *destroyed* variables.

In DivCon, the algorithm starts with the optimal solution (Definition 1) of the combinatorial compiler backend. Subsequently, it destroys a part of the variables and continues with the model's branching strategy to find the next solution, applying a restart after a given number of failures. LNS uses the concept of *neighborhood*, i.e. the variables that LNS may destroy at every restart. To improve diversity, the neighborhood for DivCon consists of all decision variables, i.e. the issue cycles c, the instruction implementations m, and the registers r. Furthermore, LNS depends on a *branching strategy* to guide the *repair* search. To improve security and allow LNS to select diverse paths after every restart, DivCon employs a random variable-value selection branching strategy as described in Table 1b.

4. Evaluation

This section evaluates DivCon experimentally. For simplicity, the section uses the acronyms LNS and DLNS to refer to the specific application of Algorithms 1 and 2 in DivCon. The diversification effectiveness and the scalability of DivCon depend on three main dimensions:

- **Optimality gap** (see Definition 2), which relaxes the optimization function. Here, we evaluate the diversification effectiveness and scalability for four different values of p, 0%, 5%, 10%, and 20%
- **Diversification algorithm**. We compare our two proposed algorithms, LNS (Algorithm 1) and DLNS (Algorithm 2) with Random Search (RS) and incremental MaxDiverse$k$Set (Hebrard et al., 2005). RS uses the branching strategy of Table 1b. For MaxDiverse$k$Set, the first solution corresponds to the optimal solution (see Definition 1) and the maximization step uses the branching strategy of Table 1a.
- **Distance measure**. We compare four distance measures (Section 3.3), HD, δ_{HD}, LD, δ_{LD}, and two configurations of GD for different values of parameters n_r and n_c (see Section 3.3), $\delta_{LD}^{0,2}$ and $\delta_{LD}^{0,8}$. The two parameters control the number of instructions preceding a branch that differ among different solutions. The smaller these parameters

are, the higher the chance of breaking a larger number of JOP gadgets, given that all gadgets end with a branch instruction.

The output of DivCon is a set of diverse binary variants. To evaluate the diversification effectiveness of each approach, we compare the generated binaries using the following three measures:

- **Code diversity**, which measures the pairwise distance of the final binaries using the same distance that was used for diversification. The definition is in Equation 7.
- **Gadget diversity**, which measures the rate of gadgets that DivCon diversifies successfully (see Section 4.4).
- **Scalability**, which is related to the number of variants generated within a fixed time budget or the total time required to generate the maximum number of variants.

The six research questions (RQs) below investigate the influence of the **optimality gap**, **diversification algorithm**, **distance measure**, and **program scope** with respect to our three diversity measures.

- RQ1. How effective are our two novel diversification algorithms? Here, we compare LNS and DLNS with state-of-the-art diversification algorithm, with respect to their ability to generate binary code that is as diverse as possible. To address this question, we evaluate the **code diversity** of DivCon for the different **diversification algorithms**.
- RQ2. What is the scalability of the distance measures when generating multiple program variants? Here, we evaluate which of the distance measures is the most appropriate for software diversification. To address this question, we evaluate the **scalability** of DivCon for the different **distance measures**.
- RQ3. How effective is DivCon using LNS and DLNS at mitigating JOP attacks? In this part, we evaluate which method is the most effective against JOP attacks by comparing the rate of shared gadgets among the generated solutions. To address this question, we evaluate the **gadget diversity** of DivCon for the different **diversification algorithms**.
- RQ4. How effective is DivCon using different distance measures against JOP attacks? Here, we evaluate the effectiveness of DivCon using four different distance measures against JOP attacks. To address this question, we evaluate the **gadget diversity** of DivCon for the different **distance measures**.
- RQ5. How does code quality affect the effectiveness of LNS against JOP attacks using an application-specific distance measure? Here, we evaluate the effect of code quality on the effectiveness of DivCon at mitigating JOP attacks. To address this question, we evaluate the **gadget diversity** of DivCon for the different **optimality gaps**.
- RQ6. What is the effect of function diversification with DivCon at the application level? Here, we evaluate the effect of diversification using DivCon with a voice compression case study. To address this question, we evaluate the **gadget diversity** of DivCon in a compiled whole-program binary consisting of multiple functions.

4.1 Experimental Setup

The following paragraphs describe the experimental setup for the evaluation of DivCon.

Implementation. DivCon is implemented as an extension of Unison (Castañeda Lozano, Carlsson, Blindell, & Schulte, 2019), and is available online[2]. Unison implements two back-end transformations: instruction scheduling and register allocation. As part of register allocation, Unison captures many interdependent transformations such as spilling, register assignment, coalescing, load-store optimization, register packing, live range splitting, re-materialization, and multi-allocation (Castañeda Lozano et al., 2019). Unison models two objective functions for code quality, *speed* and *code size*. This evaluation uses the *speed* objective function, which considers statically derived basic-block frequencies and the execution time of each basic block that depends on the shared resources, the instruction issue cycles, and the instruction latencies. These execution times and latencies were based on a generic MIPS32 model of LLVM (Castañeda Lozano et al., 2019). DivCon relies on Unison's solver portfolio that includes Gecode v6.2 (Gecode Team, 2020) and Chuffed v0.10.3 (Chu, 2011) to find optimal binary programs. We use Gecode v6.2 for automatic diversification because Gecode provides an interface for customizing *search*. The LLVM compiler (Lattner & Adve, 2004) is used as a front-end and IR-level optimizer, as well as for the final emission of assembly code. DivCon operates on the Machine Intermediate Representation (MIR)[3] level of LLVM.

Benchmark functions and platform. We evaluate the ability of DivCon to generate program variants with 20 functions sampled randomly from MediaBench[4] (Lee et al., 1997). This benchmark suite is widely employed in embedded systems research. We select two sets of benchmarks. The first set consists of 14 functions ranging from 10 to 100 MIR instructions with a median of 58 instructions. The second set consists of six functions ranging between 100 and 1000 lines of MIR instructions. Functions with size below 100 MIR instructions compose the 65% of the functions in MediaBench, whereas functions with size less than 500 MIR instruction compose the 93%, and those with size less than 1000 MIR instructions compose the 97% of the functions in MediaBench.

Smaller functions in the first set allow the evaluation of all algorithms and distance measures regardless of their computational cost, whereas larger functions challenge our diversification algorithms. Table 2 lists the ID, application, function name, the number of basic blocks, and the number of MIR instructions of each sampled function. For evaluating the scalability of DivCon, we perform an additional experiment consisting of the second set of functions. Table 3 describes these additional benchmarks. The evaluation for scalability of these benchmarks for all the random seeds takes more than one week due to memory limitations that force sequential execution. Therefore, we use these benchmarks only for evaluating the scalability of DivCon.

Furthermore, for evaluating the effectiveness of our approach at the application level, we perform a case study of one of application from MediaBench, G.721. This application consists of functions that we present in Table 11.

The functions are compiled to MIPS32 assembly code, a popular architecture within embedded systems and the security-critical Internet of Things (Alaba et al., 2017).

2. https://github.com/romits800/divcon
3. Machine Intermediate Representation: https://www.llvm.org/docs/MIRLangRef.html
4. A later version of MediaBench, MediabBench II was not complete by the time we are writing this paper.

Table 2: Benchmarks functions - 10 to 100 MIR instructions

ID	application	function name	# blocks	# instructions
b1	rasta	FR2TR	4	19
b2	mesa	glColor3ubv	1	20
b3	mesa	glTexCoord1dv	1	21
b4	g721	ulaw2alaw	4	22
b5	jpeg	start_pass_main	5	26
b6	mesa	glTexCoord4sv	1	27
b7	mesa	glEvalCoord2d	5	47
b8	mesa	glTexGendv	5	58
b9	rasta	open_out	8	58
b10	jpeg	quantize3_ord_dither	7	71
b11	mpeg2	pbm_getint	12	86
b12	mesa	gl_save_PolygonMode	11	89
b13	ghostscript	gx_concretize_DeviceCMYK	13	93
b14	mesa	gl_save_MapGrid1f	11	96

Host platform. All experiments run on an Intel®Core™i9-9920X processor at 3.50GHz with 64GB of RAM running Debian GNU/Linux 10 (buster). Each experiment runs for 15 random seeds. The aggregated results of the evaluation (RQ1) show the mean value and the standard deviation for the maximum number of generated variants, where at least five seeds are able to terminate within a time limit. For the smaller benchmarks (Table 2), we have 10GB of virtual memory for each of the executions. The experiments for different random seeds run in parallel (five seeds at a time), with two unique cores available for every seed for overheating reasons. To take advantage of the decomposition scheme, DLNS experiments use eight threads (four physical cores) with three experiments (three seeds at a time) running in parallel. The rest of the algorithms run as sequential programs. For the larger benchmarks (Table 3), the available virtual memory for each of the executions is 64GB. The experiments for different random seeds run sequentially and the DLNS experiments use eight threads.

Algorithm Configuration. The experiments focus on speed optimization and aim to generate 200 variants within a timeout. Parameter h in Algorithms 1 and 2 is set to one because even small distance between variants is able to break gadgets (see Figure 1). LNS uses restart-based search with a limit of 1000 failures and a relax rate of 60%. The *relax rate* is the probability that LNS destroys a variable at every restart, which affects the distance between two subsequent solutions. The relax rate is selected empirically based on preliminary experiments (Appendix A). Note that in our previous paper (Tsoupidi et al., 2020), the best relax rate on a different benchmark set was found to be 70%. This suggests that the optimal relax rate depends on the properties of the program under compilation, where the number of instructions appears to be a significant factor. DLNS uses the same parameters as LNS for the *local* problems, which consist of the individual basic blocks, and a different relax rate for the global problem (50% for b1 to b14 and 10% for the larger benchmarks).

Table 3: Benchmarks functions - 100 to 1000 MIR instructions

ID	application	function name	#blocks	#instructions
b15	mesa	gl_xform_normals_3fv	10	107
b16	jpeg	start_pass_1_quant	34	215
b17	mesa	apply_stencil_op_to_span	65	267
b18	mesa	antialiased_rgba_points	39	366
b19	mesa	gl_depth_test_span_generic	102	403
b20	mesa	general_textured_triangle	40	890

4.2 RQ1. Scalability and Diversification Effectiveness of LNS and DLNS

This section evaluates the diversification effectiveness and scalability of LNS and DLNS compared to incremental MaxDiverse$k$Set and RS. Here, effectiveness is the ability to maximize the difference between the different variants generated by a given algorithm. Scalability is related to the number of variants generated within a fixed time budget and the total time required to generate the maximum number of variants. This experiment uses HD as the distance measure because HD is a general-purpose distance that may be valuable for different applications.

We measure the diversification effectiveness of these methods based on the relative pairwise distance of the solutions. Given a set of solutions S and a distance measure δ, the pairwise distance d of the variants in S is:

$$d(\delta, S) = \sum_{i=0}^{|S|} \sum_{j>i}^{|S|} \delta(s_i, s_j) \,/\, \binom{|S|}{2}. \tag{7}$$

The *larger* this distance, the more diverse the solutions are, and thus, diversification is more effective. Tables 4 and 5 shows the pairwise distance d and diversification time t (in seconds) for each benchmark and method. Each experiment uses a time budget of 20 minutes and an optimality gap of $p = 10\%$. The best values of d (larger) and t (lower) are marked in **bold** for the completed experiments. Multiple values may be marked in **bold** when these values do not differ significanlty. Incomplete experiments are highlighted in *italic* and their number of variants in parenthesis. A complete experiment is an experiment, where the algorithm was able to generate the maximum number of 200 variants within the time limit for at least five of the random seeds. The values of d and t correspond to the results for these random seeds.

Scalability. The scalability results (t) show that only DLNS is scalable to large benchmarks, i.e. it is able to generate the maximum of 200 variants for all benchmarks except for *b20*. Benchmark *b20* contains a large number of MIR instructions and a small number of basic blocks (see Table 3) and thus, exceeds Unison's solving capability (Castañeda Lozano et al., 2019). RS and LNS are scalable for the majority of the benchmarks between 10 and 100 lines of MIR instructions (Table 4). In both benchmark sets, MaxDiverse$k$Set scales poorly, it cannot generate 200 variants for any benchmark. MaxDiverse$k$Set is able to find a small number of variants for *b1-b6*. However, it is not able to find any variant for the rest of the benchmarks. The first six benchmarks are small functions with less that 30

ID	MaxDiverse$k$Set		RS		LNS (0.6)		DLNS (0.6)	
	d	$t(s)$	d	$t(s)$	d	$t(s)$	d	$t(s)$
b1	*36.4±7.7*	- (2)	4.1±0.3	**0.1±0.0**	**26.6±6.6**	2.4±0.9	12.0±1.6	9.4±5.8
b2	*18.7±0.2*	- (4)	5.7±0.1	**0.2±0.0**	**13.2±0.6**	1.7±0.3	9.7±1.1	9.4±2.0
b3	*19.3±1.2*	- (3)	5.1±0.1	**0.2±0.0**	**14.8±1.1**	1.4±0.3	9.8±1.9	5.8±1.2
b4	*22.4±0.0*	- (27)	5.3±0.0	**0.2±0.0**	**15.4±1.4**	1.1±0.2	11.8±1.9	11.7±9.2
b5	*35.0±0.7*	- (2)	5.3±0.0	**0.2±0.0**	**22.8±2.3**	2.9±0.3	13.1±1.6	5.7±0.8
b6	*28.0±0.0*	- (2)	4.5±0.0	**0.4±0.0**	**23.5±0.8**	13.8±2.2	22.0±1.0	51.6±12.2
b7	-	-	4.9±0.1	**0.4±0.0**	**45.2±2.4**	7.3±1.1	19.9±5.0	4.3±0.8
b8	-	-	4.3±0.1	**0.5±0.0**	**57.4±3.0**	11.1±1.4	25.6±5.6	4.6±0.7
b9	-	-	3.0±0.0	**0.7±0.0**	**64.0±7.2**	15.6±5.2	28.1±6.7	6.1±2.1
b10	-	-	*1.0±0.0*	- (3)	**160.9±16.0**	332.1±88.8	30.4±14.3	**7.6±0.9**
b11	-	-	1.9±0.0	**7.6±0.1**	**155.9±4.4**	110.0±27.1	48.9±13.6	7.7±1.3
b12	-	-	1.7±0.0	**4.5±0.7**	**127.4±3.7**	361.3±77.3	32.2±15.1	6.0±0.4
b13	-	-	1.9±0.0	**3.0±0.0**	**103.7±5.4**	94.6±39.7	46.7±9.8	15.5±21.9
b14	-	-	1.2±0.1	**6.0±0.1**	**139.3±2.9**	865.4±99.4	39.3±20.1	7.0±0.5

Table 4: Distance and Scalability of LNS and DLNS against RS and MaxDiverse$k$Set - 10 to 100 MIR instructions

ID	MaxDiverse$k$Set		RS		LNS (0.6)		DLNS (0.6)	
	d	$t(s)$	d	$t(s)$	d	$t(s)$	d	$t(s)$
b15	-	-	*1.0±0.0*	- (7)	*278.5±4.2*	- (159)	**30.6±25.5**	**103.3±51.2**
b16	-	-	-	-	-	-	**73.1±41.0**	**57.3±14.7**
b17	-	-	2.7±0.2	318.8±0.2	*375.4±13.4*	- (27)	**147.9±37.1**	**92.0±33.7**
b18	-	-	-	-	-	-	**167.5±169.8**	**287.2±4.0**
b19	-	-	1.0±0.0	2902.8±1.6	-	-	**222.8±48.6**	**139.3±22.8**
b20	Unison and DivCon cannot handle this function.							

Table 5: Distance and Scalability of LNS and DLNS against RS and MaxDiverse$k$Set - 100 to 1000 MIR instructions

MIR instructions, whereas the rest of the benchmarks are larger and consist or more than 47 instructions (see Table 2).

LNS is slower than RS and DLNS, requiring up to 855 seconds or approximately 14.25 minutes for diversifying *b14*. Similar to MaxDiverse$k$Set, the number of instructions appears to be the main factor that determines the scalability of LNS. For the large benchmarks of Table 4, *b10*-*b12*, and *b14*, the diversification time is larger than one minute, whereas for smaller benchmarks *b1*-*b9*, which have less than 60 MIR instructions, the diversification time is less than one minute. For the largest benchmarks (Table 5), LNS is able to generate 159 variants for *b15* in around 4.63 minutes, but is not able to scale for larger benchmarks.

DLNS is generally slower than RS for the benchmarks of Table 4, but is able to scale to larger benchmarks, as seen in Table 5, where RS manages to generate 200 variants only for *b17* and *b19*. We can see that DLNS has similar performance regardless of the benchmark size, with a general increase in the diversification time for larger benchmarks (Table 5). This increase depends on the number of threads (eight) that is smaller than the number of basic blocks. For small benchmarks with basic blocks that contain few instructions, decomposition is not advantageous because it does not reduce the search space significantly. Instead, DLNS

introduces an overhead when some versions of the local solutions cannot be combined into a solution. Among the commonly scalable benchmarks, the advantage of DLNS compared to RS is clear in medium and large benchmarks, *b10*-*b14*, *b17*, and *b19*, where DLNS is able generate a large number of variants. At the same time, DLNS demonstrates a large variation in the solutions with the different seeds. This is due to the decomposition scheme of Algorithm 2. That is, depending on the random seed, the algorithm might need to restart the global problem just once or multiple times.

Overall, for small benchmarks, i.e. less than 60 MIR instructions, RS, LNS, and DLNS are all able to generate program variants efficiently (less than 16 seconds), whereas for larger benchmarks, only DLNS is able to generate a large number of variants efficiently.

Diversity. The diversity results (d) show that LNS is more effective at diversifying than RS and DLNS. The improvement of LNS over RS ranges from 1.3x (for *b2*) to 115x (for *b14*), whereas the improvement of LNS over DLNS is smaller and ranges from 7% (for *b6*) to 429% or 4x (for *b10*). DLNS is clearly less effective at generating highly diverse variants than LNS, but more effective than RS. In particular, the improvement of DLNS over RS ranges from 70% (for *b1*) to 222x (for *b19*). The difference between LNS and DLNS in generating diverse solutions is due to the ability of the former to consider the problem as a whole, enabling more fine-grained solutions.

MaxDiverse$k$Set is not able to generate 200 variants for any of the benchmarks but may give an indication of an upper bound for diversification of the smaller benchmarks. That is, although MaxDiverse$k$Set is not exact, i.e. it maximizes the pairwise distance iteratively, we expect that LNS, DLNS, and RS are not able to achieve higher pairwise diversity than MaxDiverse$k$Set. However, a direct comparison is not possible because MaxDiverse$k$Set is not able to generate 200 variants for any of the benchmarks.

Conclusion. In summary, LNS and DLNS provide two attractive solutions for diversifying code: LNS is significantly and consistently more effective at diversifying code than both RS and DLNS, but does not scale efficiently for large benchmarks, whereas DLNS is more effective than both LNS and RS at generating variants for large problems, and is still able to improve significantly the diversity over RS.

4.3 RQ2. Scalability of LNS with Different Distance Measures

In this section, we compare the distance measures introduced in Section 3.3 with regards to their ability to steer the search towards diverse program variants within a maximum time budget. Based on the results of RQ1, we focus on the LNS search algorithm, and run it with each distance metric. For the problem-specific distance measure, GD, we compare two configurations, i) $n_r = 0$ and $n_c = 2$ and ii) $n_r = 0$ and $n_c = 8$. The two parameters control the number of instructions preceding a branch that differ among different solutions. The smaller these parameters are, the higher the chance of breaking a larger number of gadgets, given that all gadgets end with a branch instruction.

Table 6 presents the results of the distance evaluation, where the time limit is 10 minutes and the optimality gap $p = 10\%$. For each distance measure (δ_{HD}, δ_{LD}, $\delta^{0,2}_{GD}$, and $\delta^{0,8}_{GD}$), the table shows the diversification time t, in seconds (or "-" if the algorithm is not able to generate 200 variants) and the number of generated variants num within the time limit.

Table 6: Scalability of δ_{HD}, δ_{LD}, $\delta_{GD}^{0,2}$, and $\delta_{GD}^{0,8}$

ID	δ_{HD}		δ_{LD}		$\delta_{GD}^{0,2}$		$\delta_{GD}^{0,8}$	
	$t(s)$	num	$t(s)$	num	$t(s)$	num	$t(s)$	num
b1	**2.7±0.9**	200	-	37	6.9±7.1	200	**2.9±1.0**	200
b2	**1.8±0.4**	200	-	41	-	75	5.8±2.6	200
b3	**1.6±0.2**	200	-	44	-	121	4.5±3.4	200
b4	**1.3±0.2**	200	-	38	2.5±0.9	200	**1.4±0.4**	200
b5	**3.6±0.3**	200	-	27	-	12	112.4±126.8	200
b6	**14.1±2.3**	200	-	15	172.1±179.4	200	17.6±3.3	200
b7	**7.9±1.3**	200	-	12	181.5±183.4	200	19.6±4.0	200
b8	**12.1±1.5**	200	-	8	73.1±22.2	200	32.1±6.6	200
b9	**17.0±4.6**	200	-	5	-	56	217.5±158.6	200
b10	348.6±90.7	200	-	-	359.8±59.4	200	**319.3±81.8**	200
b11	**121.1±29.0**	200	-	-	-	77	445.1±64.6	200
b12	**377.9±76.7**	200	-	-	-	105	-	60
b13	**107.6±44.1**	200	-	-	377.7±158.4	200	208.7±110.6	200
b14	-	152	-	-	-	55	-	36

The value of *num* is the maximum number of variants that at least five (out of 15) of the random seeds generate.

The results show that when DivCon uses LNS with Hamming Distance, δ_{HD}, it generates 200 variants for all benchmarks except *b12*, where it generates 157 variants. The diversification time with δ_{HD} ranges from one second for *b4* to approximately six minutes for *b12*. On the other hand, DivCon using Levenshtein Distance (LD), δ_{LD}, is not able to generate 200 variants for any of the benchmarks within the time limit. The scalability issues of δ_{LD} are due to the quadratic complexity of its implementation (Wagner & Fischer, 1974), whereas Hamming Distance can be implemented linearly. DivCon using the first configuration of Gadget Distance (GD), $\delta_{GD}^{0,2}$, generates the maximum number of variants for seven benchmarks, i.e. *b1*,*b4*, *b6*-*b8*, *b10*, and *b13*. Distance $\delta_{GD}^{0,2}$ uses small values for parameters $n_r = 0$ and $n_c = 2$, which leads to a reduced number of solutions (see Section 3.3). This has a negative effect on the scalability, resulting in low variant generation for the rest of the benchmarks. Using the second configuration of GD, distance $\delta_{GD}^{0,8}$, with $n_r = 0$ and $n_c = 8$, DivCon generates the maximum number of variants for all benchmarks except *b12* and *b14*. The time to generate the variants with $\delta_{GD}^{0,8}$ is larger than with δ_{HD}. With this gadget-targeting metric, DivCon takes up to seven minutes for generating 200 variants for *b11*.

Conclusion. DivCon using LNS and the $\delta_{GD}^{0,8}$ or δ_{HD} distance can generate a large number of diverse program variants for most of the benchmark functions. Scalability, given the maximum number of variants, comes with slightly longer diversification time for $\delta_{GD}^{0,8}$ than with δ_{HD}. In Section 4.5, we evaluate the distance measures with regards to security.

4.4 RQ3. JOP Attacks Mitigation: Effectiveness of LNS and DLNS

Software Diversity has various applications in security, including mitigating code-reuse attacks. To measure the level of mitigation that DivCon achieves, we assess the JOP gadget survival rate $srate(s_i, s_j)$ between two variants $s_i, s_j \in S$, where S is the set of generated variants. This metric determines how many of the gadgets of variant s_i appear at the same position on the other variant s_j, that is $srate(s_i, s_j) = |gad(s_i) - gad(s_j)| \,/\, |gad(s_i)|$, where $gad(s_i)$ are the gadgets in solution s_i. The procedure for computing $srate(s_i, s_j)$ is as follows: 1) find the set of gadgets $gad(s_i)$ in solution s_i, and 2) for every $g \in gad(s_i)$, check whether there exists a gadget identical to g at the same address of s_j. For step 1, we use the state-of-the-art tool, ROPgadget (Salwan, 2020), to automatically find the gadgets in the `.text` section of the compiled code. For step 2, the comparison is syntactic after removing all `nop` instructions. Syntactic comparison is scalable but may result in false negatives.

This and the following sections evaluate the effectiveness of DivCon against code-reuse attacks. To achieve this, all experiments compare the distribution of *srate* for all pairs of generated solutions. Due to its skewness, the distribution of *srate* is represented as a histogram with four buckets (0%, (0%, 10%], (10%,40%], and (40%, 100%]) rather than summarized using common statistical measures. Here, the best is an $srate(s_i, s_j)$ of 0%, which means that s_j does not contain any gadgets that exist in s_i, whereas an $srate(s_i, s_j)$ in range (40%,100%] means that s_j shares more than 40% of the gadgets of s_i.

To evaluate the gadget diversification efficiency, we compare the *srate* for all permutations of pairs in S for LNS and DLNS with RS as a baseline. Low *srate* corresponds to higher mitigation effectiveness because code-reuse attacks based on gadgets in one variant have lower chances of locating the same gadgets in the other variants (see Figure 1). Tables 7 and 8 summarize the gadget survival distribution for all benchmarks for algorithms RS, LNS, and DLNS. We use 10% as the optimality gap and HD because, as we saw in RQ2, DivCon using HD is the most scalable diversification configuration. The values in **bold** correspond to the most frequent value(s) of the histogram. The time limit for this experiment is 20 minutes. Column *num* shows the average of the generated number of variants for all random seeds.

First, we notice that for the smaller benchmarks, *b2* to *b3*, and *b6*, all algorithms are able to generate variant pairs that share no gadgets, i.e. the most frequent values are in the first bucket (column *=0*). RS generates diverse variants that share a small number of gadgets for *b2*-*b4*, *b6*, and *b10* (only three variants). For the other benchmarks, the most common values are in the second (*b11*), or the third (*b5*, *b7*-*b9*, *b12*-*b14*, *b17*, and *b19*) bucket, which provides poor mitigation effectiveness against JOP attacks. The poor effectiveness of RS against code-reuse attacks can be correlated with the poor diversity effectiveness of the method (see Section 4.2).

LNS generates diverse variants that do not share any gadgets (belong to the first bucket) for all benchmarks except *b5*. Benchmark *b5* has different behavior because it has a highly constrained register allocation due to specific constraints imposed by the calling conventions.

Finally, DLNS has similar performance to RS for medium size benchmarks (Table 7), but worse performance for large benchmarks (Table 8). In particular, only five benchmarks *b1*-*b4* and *b6* are mostly in the first bucket. Although DLNS has relatively high pairwise distance (see Table 4), its effectiveness against code-reuse attacks is low. This is because

Table 7: Gadget survival rate for 10% optimality gap with Hamming distance for RS, LNS, and DLNS - 10 to 100 MIR instructions

ID	RS					LNS					DLNS				
	=0	≤10	≤40	≤100	num	=0	≤10	≤40	≤100	num	=0	≤10	≤40	≤100	num
b1	**34**	13	19	**33**	200	**85**	12	2	-	200	**50**	24	25	1	200
b2	**86**	7	5	1	200	**88**	1	11	1	200	**83**	1	11	5	200
b3	**84**	11	5	-	200	**90**	3	6	-	200	**88**	4	7	1	200
b4	**92**	7	1	-	200	**95**	4	1	-	200	**52**	38	8	3	200
b5	2	5	**48**	**45**	200	14	14	**51**	21	200	-	13	**44**	**43**	200
b6	**74**	18	8	-	200	**92**	3	5	-	200	**92**	3	4	1	200
b7	-	26	**72**	2	200	**87**	11	2	-	200	7	23	**52**	18	200
b8	-	36	**63**	1	200	**88**	10	2	-	200	7	22	**48**	23	200
b9	-	10	**83**	8	200	**57**	24	18	1	200	3	11	**49**	36	200
b10	**68**	2	11	19	3	**98**	-	1	1	200	22	1	6	**71**	200
b11	-	**72**	28	-	200	**73**	23	3	-	200	4	5	41	**51**	200
b12	-	-	**99**	1	200	**80**	18	2	-	200	1	8	**59**	32	187
b13	26	9	**35**	**30**	200	**92**	4	3	-	200	31	11	19	**39**	149
b14	-	-	**98**	2	200	**77**	21	2	-	200	-	3	**61**	36	179

Table 8: Gadget survival rate for 10% optimality gap with Hamming distance for RS, LNS, and DLNS - 100 to 1000 MIR instructions

ID	RS					LNS					DLNS				
	=0	≤10	≤40	≤100	num	=0	≤10	≤40	≤100	num	=0	≤10	≤40	≤100	num
b15	**98**	-	2	-	7	**99**	1	-	-	118	**43**	2	7	**47**	188
b16	-	-	-	-	-	**98**	2	-	-	71	-	1	33	**66**	200
b17	30	13	**47**	10	200	**87**	6	6	1	42	15	10	35	**40**	173
b18	-	-	-	-	-	-	-	-	-	-	-	-	2	**98**	187
b19	18	27	**52**	3	200	-	-	-	-	-	1	-	40	**60**	200
b20	Unison and DivCon cannot handle this function.														

in many small programs with a large number of basic blocks, the number of registers that are shared among different basic blocks (and thus assigned by the *global* problem, see Algorithm 2) is high, resulting in low diversity of the register allocation among variants.

Conclusion. The LNS diversification algorithm is significantly more effective than both DLNS and RS at generating binary variants that share a minimal number of JOP gadgets.

4.5 RQ4. JOP Attacks Mitigation: Effectiveness of Different Distance Measures

Section 4.3 shows that Hamming Distance (HD), δ_{HD}, is the most scalable distance measure followed closely by the second configuration of Gadget Distance (GD), $\delta_{GD}^{0,8}$. This section investigates the impact of the distance measure on the effectiveness of DivCon against JOP attacks.

Table 9 shows the gadget-replacement effectiveness of DivCon using distances: δ_{HD}, δ_{LD}, $\delta_{GD}^{0,2}$, and $\delta_{GD}^{0,8}$. The time limit for this experiment is ten minutes and the optimality

gap is 10%. This experiment uses LNS as the diversification algorithm because, as we have seen in Section 4.4, LNS is more effective against JOP attacks than DLNS.

The results for the Hamming Distance (HD), δ_{HD}, are in the first column of the table. For all benchmarks, except *b5*, the highest values are under the first subcolumn. This means that a large proportion of the variant pairs do not share any gadgets, which is a strong indication of JOP attack mitigation. In particular, the most frequent values range from 57 to 98 percent. Benchmark *b5* has weak diversification capability due to hard constraints in register allocation (see Section 4.4).

The results for Levenshtein Distance, δ_{LD}, appear in the second column of the table. Similar to HD, almost all benchmarks, where DivCon generates at least two variants, have their most common value in the first subcolumn except for *b5*. These values range from 51% to 85%, which corresponds to poorer gadget diversification effectiveness than using δ_{HD}. As discussed in Section 4.3, DivCon using Levenshtein Distance is not able to generate the maximum requested number of variants (200) within the time limit of ten minutes for any of the benchmarks.

The third column of Table 9 shows the results for Gadget Distance (GD) with parameters $n_r = 0$ and $n_c = 2$. Parameter $n_r = 0$ enforces diversity of the register allocation for the instructions that are issued on the same cycle as the branch instruction. Similarly, parameter $n_c = 2$ enforces diversity for the instruction schedule of the instructions preceding the branch instruction by at most two cycles. Distance $\delta_{GD}^{0,2}$ measures the sum of these two constraints (and enforces it to be greater than $h = 1$) for all branch instructions of the benchmark in question. DivCon with this distance measure has very high effectiveness against JOP attacks, with the most frequent values ranging from 65 to 100 percent. However, using $\delta_{GD}^{0,2}$, DivCon is not able to generate a large number of variants for almost half of the benchmarks.

The last distance measure, $\delta_{GD}^{0,8}$, differs from $\delta_{GD}^{0,2}$ in that it allows diversifying the instruction schedule for a larger number of instructions preceding the branch instruction, i.e. $n_c = 8$. Here, the most common values range from 48 to 99 percent for different benchmarks and the scalability is satisfiable with DivCon being able to generate the total number of requested variants for almost all the benchmarks. Using $\delta_{GD}^{0,8}$, DivCon improves the gadget diversification efficiency of the overall fastest distance measure, δ_{HD}, for all benchmarks except *b3*, where the difference is very small. The largest improvement is for *b9* and *b5*. For *b9* the most frequent value is 57% with δ_{HD} and gets improved to 66% with $\delta_{GD}^{0,8}$. For *b5* the majority of the variant pairs are under the third bucket, which corresponds to the weak $(10\% - 40\%]$-survival rate with δ_{HD} and under the first bucket (column *=0*) with $\delta_{GD}^{0,8}$, which is a significant improvement.

Conclusion. Distances δ_{HD} and $\delta_{GD}^{0,8}$ are both appropriate distances for our application, trading scalability with security effectiveness. DivCon using δ_{HD} has better scalability than using $\delta_{GD}^{0,8}$ (see Section 4.3), whereas DivCon using $\delta_{GD}^{0,8}$ is more effective against code-reuse attacks compared to using δ_{HD}.

4.6 RQ5. JOP Attacks Mitigation: Effectiveness for Different Optimality Gaps

This section investigates the trade-off between code quality and diversity and evaluates the effectiveness of DivCon against code-reuse attacks. Table 10 summarizes the gadget survival distribution for all benchmarks and different values of the optimality gap (0%, 5%, 10%,

Table 9: Gadget survival rate for 10% optimality gap for the distances: δ_{HD}, δ_{LD}, $\delta_{GD}^{0,2}$, and $\delta_{GD}^{0,8}$

ID	δ_{HD}					δ_{LD}					$\delta_{GD}^{0,2}$					$\delta_{GD}^{0,8}$				
	=0	≤10	≤40	≤100	num	=0	≤10	≤40	≤100	num	=0	≤10	≤40	≤100	num	=0	≤10	≤40	≤100	num
b1	**85**	12	2	-	200	**52**	30	10	8	29	**99**	1	-	-	200	**92**	7	1	-	200
b2	**88**	1	11	1	200	**85**	-	12	2	37	**94**	4	2	-	67	**90**	4	6	-	200
b3	**90**	3	6	-	200	**85**	5	9	1	41	**95**	4	1	-	111	**89**	6	5	-	200
b4	**95**	4	1	-	200	**85**	12	2	1	36	**99**	1	-	-	200	**97**	3	-	-	200
b5	14	14	**51**	21	200	15	10	32	**42**	25	**65**	9	24	2	12	**48**	28	20	3	178
b6	**92**	3	5	-	200	**84**	4	10	2	14	**96**	3	1	-	187	**92**	4	4	-	200
b7	**87**	11	2	-	200	**54**	28	16	2	10	**83**	15	2	-	145	**87**	12	1	-	200
b8	**88**	10	2	-	200	**53**	23	20	4	7	**87**	12	1	-	188	**88**	11	1	-	200
b9	**57**	24	18	1	200	**51**	11	21	17	4	**74**	15	11	-	52	**66**	24	10	-	167
b10	**98**	-	1	1	200	-	-	-	-	-	**99**	-	-	-	200	**99**	-	1	1	200
b11	**73**	23	3	-	200	-	-	-	-	-	**91**	8	1	-	62	**79**	20	2	-	198
b12	**80**	18	2	-	200	-	-	-	-	-	**96**	4	-	-	83	**87**	12	1	-	48
b13	**92**	4	3	-	200	-	-	-	-	-	**100**	-	-	-	185	**97**	1	2	-	200
b14	**77**	21	2	-	141	-	-	-	-	-	**95**	5	-	-	44	**85**	14	1	-	31

and 20%). Based on the results of RQ3, we select LNS for this evaluation because we have observed that DivCon using LNS is the most effective at diversifying gadgets. Similarly, in RQ4, we were able to identify that the gadget-specific distance, $\delta_{GD}^{0,8}$, is the most effective among the two scalable distance measures at diversifying gadgets. The values in **bold** correspond to the mode(s) of the histogram and the time limit for this experiment is ten minutes.

First, we notice that DivCon with LNS and $\delta_{GD}^{0,8}$ can generate some pairs of variants that share no gadgets, even without relaxing the constraint of optimality ($p = 0\%$). In particular, for $p = 0\%$, all benchmarks except *b7* are dominated by a 0% survival rate and only *b7* is dominated by a weak $(0\% - 10\%]$-survival rate. This indicates that optimal code naturally includes software diversity that is good for security. For example, DivCon generates on average 110 solutions for benchmark *b6*. Comparing pairwise the gadgets for these solutions, we are able to determine that 91 percent of the solution pairs do not share any gadgets, whereas five percent of these pairs share up to 10% of the gadgets and four percent share between 10% and 40% of the gadgets. Furthermore, we can see that for only two of the benchmarks (*b5* and *b9*), DivCon with LNS and $\delta_{GD}^{0,8}$ is unable to generate any variants, whereas for three of the benchmarks (*b1*, *b3*, and *b13*) it generates a large number of variants without quality loss. Among the benchmarks that are dominated by the first bucket (0% gadget survival rate), the rates range from 52% up to 100%. These results indicate that it is possible to achieve high security-aware diversity without sacrificing code quality.

Second, the results show that the effectiveness of DivCon at diversifying gadgets can be further increased by relaxing the constraint on code quality, with diminishing returns beyond $p = 10\%$. Increasing the optimality gap to just $p = 5\%$ makes 0% survival rate (column *=0*) the dominating bucket for all benchmarks except *b5*. Benchmark *b5* is subjected to hard register allocation constraints, which reduces DivCon's gadget diversification ability.

Table 10: Gadget survival rate for different optimality gap values of the Gadget Distance ($\delta_{GD}^{0,8}$) using LNS

ID	$p = 0\%$					$p = 5\%$					$p = 10\%$					$p = 20\%$				
	=0	≤10	≤40	≤100	num	=0	≤10	≤40	≤100	num	=0	≤10	≤40	≤100	num	=0	≤10	≤40	≤100	num
b1	**93**	3	4	-	200	**89**	9	2	-	200	**92**	7	1	-	200	**98**	1	1	-	200
b2	**93**	-	7	-	20	**90**	4	6	-	200	**90**	4	6	-	200	**90**	4	5	-	200
b3	**80**	13	6	1	149	**90**	5	5	-	200	**89**	6	5	-	200	**93**	3	4	-	200
b4	**98**	1	-	-	24	**97**	3	-	-	200	**97**	3	-	-	200	**98**	2	-	-	200
b5	-	-	-	-	-	10	13	**42**	35	29	**48**	28	20	3	178	**66**	18	14	2	200
b6	**91**	5	4	-	110	**92**	4	4	-	200	**92**	4	4	-	200	**94**	3	3	-	200
b7	38	**48**	14	-	82	**85**	14	1	-	200	**87**	12	1	-	200	**89**	10	1	-	200
b8	**60**	30	10	-	40	**89**	10	1	-	200	**88**	11	1	-	200	**90**	9	1	-	200
b9	-	-	-	-	-	**59**	28	13	-	171	**66**	24	10	-	167	**66**	23	11	-	167
b10	**75**	3	3	19	4	**99**	-	1	1	200	**99**	-	1	1	200	**99**	-	-	-	193
b11	**84**	14	2	-	87	**82**	17	2	-	190	**79**	20	2	-	198	**84**	14	1	-	199
b12	**82**	15	3	-	12	**90**	9	1	-	36	**87**	12	1	-	48	**90**	9	1	-	57
b13	**100**	-	-	-	175	**96**	1	2	-	200	**97**	1	2	-	200	**97**	1	1	-	200
b14	**52**	41	7	-	3	**88**	11	1	-	25	**85**	14	1	-	31	**91**	8	1	-	44

The rate of the variant pairs that do not share any variants ranges from 59 percent for *b9* to 99 percent for *b10*. Further increasing the gap to 10% and 20% increases significantly the number of pairs that share no gadgets (column *=0*). For example, with an optimality gap of $p = 10\%$, the dominating bucket for all benchmarks corresponds to 0% survival rate (column *=0*) and ranges from 48% (*b5*) to 99% (*b10*) of the total solution pairs. An optimality gap of $p = 20\%$ improves further the effectiveness of DivCon. The improvement is substantial for benchmark *b5*, where the register allocation of this benchmark is highly constrained. Larger optimality gap allows the generation of more solutions that differ with regards to the instructions schedule. This leads to an improvement indicated by an increase in the rate of the first bucket (column *=0*) from 48% for $p = 10\%$ to 66% for $p = 20\%$.

Related approaches (discussed in Section 5) report the *average* gadget elimination rate across all pairs for different benchmark sets. The zero-cost approach of Pappas et al. (2012) achieves an average gadget elimination rate between $74\% - 83\%$ without code degradation, comparable to DivCon's $93\% - 100\%$ at $p = 0\%$ (including only benchmarks for which DivCon generates variants). Homescu et al. (2013) propose a statistical approach that reports an average *srate* between $82\% - 100\%$ with a code degradation of less than 5%, comparable to DivCon's $62\% - 100\%$ at $p = 5\%$. Both approaches report results on larger code bases that exhibit more opportunities for diversification. We expect that DivCon would achieve higher overall survival rates on these code bases compared to the benchmarks used in this paper as we can see in case study of RQ6 (Section 4.7).

Conclusion. Empirical evidence shows that DivCon with the LNS algorithm and distance measure $\delta_{GD}^{0,8}$ achieves high JOP gadget diversification rate without sacrificing code quality. Increasing the optimality gap to just 5% improves the effectiveness of DivCon significantly, while further increase in the optimality gap does not have a similarly large effect on gadget diversity.

Table 11: G.721 functions

ID	app	module	function name	#blocks	#instructions	LNS time (s)	DLNS time (s)
g1	g721	g711	ulaw2linear	1	14	0.4 ± 0.0	7.8 ±0.0
g2	g721	g711	alaw2ulaw	4	19	0.8 ± 0.0	52.6 ±0.0
g3	g721	g711	ulaw2alaw	4	22	1.3 ± 0.0	34.8 ±0.0
g4	g721	g711	alaw2linear	6	23	0.9 ± 0.0	22.5 ±0.0
g5	g721	g72x	reconstruct	4	24	0.8 ± 0.0	22.4 ±0.0
g6	g721	g72x	step_size	7	27	3.2 ± 0.0	7.1 ±0.0
g7	g721	g72x	predictor_pole	1	28	2.4 ± 0.0	15.8 ±0.0
g8	g721	g72x	g72x_init_state	1	29	1.1 ± 0.0	3.1 ±0.0
g9	g721	g711	linear2ulaw	11	54	6.0 ± 0.0	9.1 ±0.0
g10	g721	g711	linear2alaw	13	60	30.5 ± 0.0	6.7 ±0.0
g11	g721	g72x	tandem_adjust_ulaw	9	75	140.8 ± 0.8	6.8 ±0.0
g12	g721	g72x	predictor_zero	1	77	43.8 ± 0.1	5.3 ±0.0
g13	g721	g72x	tandem_adjust_alaw	13	89	182.1 ± 0.9	8.1 ±0.0
g14	g721	g72x	quantize	23	99	246.2 ± 0.2	17.9 ±0.0
g15	g721	g721	g721_encoder	7	135	214.7 ± 0.4	11.0 ±0.0
g16	g721	g721	g721_decoder	7	135	323.3 ± 6.3	10.7 ±0.0
g17	g721	g72x	update	105	523	-	128.0±1.1
g18	main	main	main	9	40	7.3 ± 0.0	7.8 ±0.0
g19	main	main	pack_output	3	23	0.8 ± 0.0	6.5 ±0.0
g20	stubs	stubs	_nmi_handler	2	1	- (1)	- (1)
g21	stubs	stubs	_on_bootstrap	1	1	- (1)	- (1)
g22	stubs	stubs	_on_reset	1	1	- (1)	- (1)

4.7 RQ6. Case Study: Effectiveness of DivCon at the Application Level

DivCon operates at the function level. In this section, we evaluate the effectiveness of DivCon against JOP attacks for programs that consist of multiple functions. To do that, we study an application from MediaBench I and evaluate it using the JOP gadget survival rate as in RQ3, RQ4, and RQ5. To diversify a program, we diversify the functions that comprise this program and then combine them randomly. This approach results in up to n^f different variants, where n is the number of variants per function and f the number of functions in the program. If we also perform function permutation, the number of possible program variants increases to $f! \cdot n^f$.

We apply these methods on G.721, an application of the MediaBench I benchmark suite (Lee et al., 1997). This application is an implementation of the International Telegraph and Telephone Consultative Committee (CCITT) G.711, G.721, and G.723 voice compression algorithms. We compile G.721 for the MIPS32-based Pic32MX microcontroller[5].

Table 11 shows 1) the functions that comprise the G.721 application, 2) a custom `main` function[6] that performs encoding, and 3) a number of required system functions, `stubs`. The columns show the number of basic blocks (#blocks), the number of MIR instructions (#instructions) and the diversification time in seconds for generating 200 variants using LNS (LNS time (s)) and DLNS (DLNS time (s)) after running the experiment five times with

5. PIC32MX Microprocessor Family: https://www.microchip.com/en-us/products/microcontrollers-and-microprocessors/32-bit-mcus/pic32-32-bit-mcus/pic32mx
6. The main function is a simplified version of the encoding example that g721 provides.

Table 12: Gadget survival rate for 10% optimality gap with the Hamming distance, δ_{HD}, for the G.721 application with function randomization at link level (FS) and without (NFS)

App	NFS									FS								
	=0	≤0.5	≤1	≤2	≤5	≤10	≤40	≤100	num	=0	≤0.5	≤1	≤2	≤5	≤10	≤40	≤100	num
G.721	**85**	12	1	1	-	-	-	-	200	**98**	2	-	-	-	-	-	-	200

the same random seed (seed = 42). The `stubs` functions consist of two empty functions (`on_reset` and `on_bootstrap`) and one function (`nmi_handler`) that contains one empty infinite loop. These functions contain only one MIR instruction each, and, therefore, there are no variants within a 10% optimality gap. We diversify the rest of the functions using DivCon with 0.5 relax rate, 10% optimality gap, and a time limit of 20 minutes. We run the experiment using the same random seed for DivCon and the function randomization. For the cases that LNS manages to generate all variants (all but *g17*), we use the LNS-generated variants and for the rest of the benchmarks (*g17*), we use DLNS. For compiling the application, we generate the textual assembly code of the function variants using DivCon and `llc`. To compile and link the application, we use a Pic32MX microcontroller toolchain[7] that uses `gcc`. To deactivate instruction reordering by `gcc`, `llc` sets the `noreorder` directive.

For combining the functions in the final binaries, we use two approaches, 1) No Function Shuffling (NFS), which generates the binary combining the different function variants in the same order and 2) Function Shuffling (FS), which randomizes the function order at the linking time.

Table 12 shows the results of the diversification of G.721 using the NFS and FS schemes after generating 200 variants of the G.721 application. The results show that combining the diversified variants without shuffling the functions at link time (NFS) results in most of the variants, 85% of the pairs, sharing no gadgets, while 12% share between 0% and 0.5% of the gadgets. We calculate the average of gadget survival rate over the variant pairs as $0.068 \pm 0.128\%$. Using function shuffling at link time (FS) results in $0.008 \pm 0.008\%$ average gadget survival rate, with 98% of all variant pairs not sharing any gadget (first bucket). This shows that the fine-grained diversification of DivCon using function shuffling improves further the result for NFS.

Conclusion. In this case study, we show that with our method, we are able to diversify whole programs and not just functions. Additionally, we show that randomly combining the diversified functions using DivCon achieves the diversification and/or relocation of JOP gadgets with an average of less than 0.1% survival rate in a multi-function program. Function shuffling reduces further the gadget survival rate to approximately 0.01% survival rate, indicating that hardly any variant pairs share gadgets.

4.8 Discussion

This section discusses two main topics, 1) the use of DivCon against more advanced attacker models, and 2) scalability limitations of our approach and how to address them.

7. https://github.com/is1200-example-projects/mcb32tools

Advanced code-reuse attacks. Our attack model considers basic-ROP/JOP attacks. However, in literature there exist more advanced attacks, like JIT-ROP (Snow et al., 2013), where the attacker is able to read the code from the memory and identify gadgets during the attack. Static diversification of a binary is not effective against these types of attacks. Instead, some approaches (Chen, Wang, Whalley, & Lu, 2016; Williams-King, Gobieski, Williams-King, Blake, Yuan, Colp, Zheng, Kemerlis, Yang, & Aiello, 2016) use re-randomization, a technique to re-randomize the binary by switching between variants of the code at run time. Using our approach, it is possible to perform re-randomization of an application by switching between different function variants that DivCon generates.

Large Functions. Unison is not scalable to large functions for MIPS (Castañeda Lozano et al., 2019) and in this paper we have evaluated DivCon for functions up to 523 lines of LLVM MIR instructions. However, there are functions that are larger than what Unison supports. In particular, in MediaBench I, approximately 7% of the functions contain more than 500 instructions. For these cases, one may use other diversification schemes for just these functions and DivCon for the rest of the functions. Another approach is to deactivate some of the transformations that Unison and DivCon perform for larger benchmarks or improve the scalability of Unison (Castañeda Lozano et al., 2019). We leave this as future work.

5. Related Work

State of the art software diversification techniques apply randomized transformations at different stages of the software development. Only a few exceptions use search-based techniques (Larsen et al., 2014). This section focuses on quality-aware software diversification approaches.

Superdiversifier (Jacob et al., 2008) is a search-based approach for software diversification against cyberattacks. Given an initial instruction sequence, the algorithm generates a random combination of the available instructions and performs a verification test to quickly reject non equivalent instruction sequences. For each non-rejected sequence, the algorithm checks semantic equivalence between the original and the generated instruction sequences using a SAT solver. Superdiversifier affects the code execution time and size by controlling the length of the generated sequence. A recent approach, Crow (Arteaga et al., 2021), presents a superdiversification approach as a security mitigation for the Web. Along the same lines, Lundquist et al. (2016, 2019) use program synthesis for generating program variants against cyberattacks, but no results are available, yet. In comparison, DivCon uses a combinatorial compiler backend that measures the code quality using a more accurate cost model that also considers other aspects, such as execution frequencies.

Most diversification approaches use randomized transformations in the stack (Lee, Kang, Jang, & Kang, 2021), on binary code (Wartell et al., 2012; Abrath et al., 2020), at the binary interface level (Kc, Keromytis, & Prevelakis, 2003), in the compiler (Homescu, Jackson, Crane, Brunthaler, Larsen, & Franz, 2017) or in the source code (Baudry, Allier, & Monperrus, 2014) to generate multiple program variants. Unlike DivCon, the majority of these approaches do not control the quality of the generated variants during diversification but rather evaluate it afterwards (Davi et al., 2013; Wang et al., 2017; Koo et al., 2018; Homescu

et al., 2017; Braden et al., 2016; Crane et al., 2015). However, there are a few approaches that control the code quality during randomization.

Some compiler-based diversification approaches restrict the set of program transformations to control the quality of the generated code (Crane et al., 2015; Pappas et al., 2012). For example, Pappas et al. (2012) perform software diversification at the binary level and apply three zero-cost transformations: register randomization, instruction schedule randomization, and function shuffling. In contrast, DivCon's combinatorial approach allows it to control the aggressiveness and potential cost of its transformations: a cost overhead limit of 0% forces DivCon to apply only zero-cost transformations; a larger limit allows DivCon to apply more aggressive transformations, potentially leading to higher diversity.

Homescu et al. (2013) perform only garbage (`nop`) insertion, and use a profile-guided approach to reduce the overhead. To do this, they control the `nop` insertion probability based on the execution frequency of different code sections. In contrast, DivCon's cost model captures different execution frequencies, which allows it to perform more aggressive transformations in non-critical code sections.

6. Conclusion

This paper introduces DivCon, a constraint-based code diversification technique against code-reuse attacks. The key novelty of this approach is that it supports a systematic exploration of the trade-off between code diversity and code size and speed. Our experiments show that Large Neighborhood Search (LNS) is an effective algorithm to explore the space of diverse binary programs, with a fine-grained control on the trade-off between code quality and JOP gadgets diversification. In particular, we show that the set of optimal solutions naturally contains a set of diverse solutions, which increases significantly when relaxing the constraint of optimality. For improving the effectiveness of our approach against JOP attacks, we propose a novel gadget-specific distance measure. Our experiments demonstrate that the diverse solutions generated by DivCon using this distance measure are highly effective to mitigate JOP attacks.

Acknowledgments

We would like to give a special acknowledgment to Christian Schulte, for his critical contribution at the early stages of this work. Although no longer with us, Christian continues to inspire his students and colleagues with his lively character, enthusiasm, deep knowledge, and understanding. We would also like to thank Linnea Ingmar and the anonymous reviewers of CP2020 and JAIR for their useful feedback, and Oscar Eriksson for proof reading. This work is partially supported by the Wallenberg AI, Autonomous Systems, and Software Program (WASP) funded by Knut and Alice Wallenberg Foundation and by the TrustFull project funded by the Swedish Foundation for Strategic Research.

Appendix A. Relax Rate Selection

The LNS configuration of DivCon requires selecting the *relax rate*. The relax rate is the probability that LNS destroys a variable at every restart, which affects the distance between

two subsequent solutions. A higher relax rate increases diversity but requires more solving effort.

In LNS, the relax rate, r, affects how many of the assigned variables of the last solution LNS destroys for finding the next solution. To evaluate that, we use two metrics and RS as a baseline. P_δ and P_t correspond to the rate of the LNS over RS with regards to the pairwise distance and the diversification time as follows:

$$P_\delta(\delta, S_1, S_2) = \begin{cases} \dfrac{d(\delta, S_1)}{d(\delta, S_2)}, & d(\delta, S_1) > d(\delta, S_2) \\ \dfrac{d(\delta, S_2)}{d(\delta, S_1)}, & otherwise \end{cases} \tag{8}$$

and

$$P_t(t_1, t_2) = \begin{cases} \dfrac{t_1}{t_2}, & t_1 > t_2 \\ \dfrac{t_2}{t_1}, & otherwise \end{cases}, \tag{9}$$

where t_1 is the diversification time for generating the solution set S_1 for RS and t_2 is the diversification time for generating the solution set S_2 for LNS.

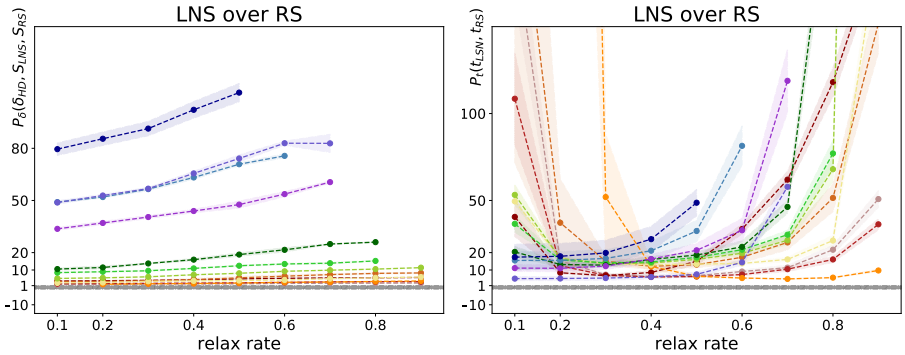
Figure 6 depicts the effect of different relax rates on the distance, P_δ, and the diversification time, P_t, when generating 200 variants for the 14 benchmarks of Table 2. The figure shows the results for each of the benchmarks as a separate colored line with the corresponding standard deviation shown in light color. The time limit is ten minutes and the distance measure is Hamming Distance (HD), δ_{HD}. Figure 6a shows that increasing the relax rate increases the pairwise distance improvement, P_δ, of the generated program variants. Figure 6b shows the diversification time overhead P_t. This figure shows that low values and large values of r have large time overhead compared to RS, whereas values $r = 0.3$, $r = 0.4$, $r = 0.5$, and $r = 0.6$ have acceptable time overhead. As we have seen in Figure 6a, the larger the relax rate, the higher the diversity improvement for LNS compared to RS. Improved diversity can be achieved by increasing the relax rate, whereas, moderate relax rate improves scalaility. Therefore, $r = 0.6$ is a good trade-off between diversity and scalability. Ultimately, we would like to automatically select the relax rate that fits a specific function. We leave this as a future work.

Appendix B. Diversification Example

This section shows a more elaborated example of diversified code using DivCon. Figure 7 shows two variants of function `ulaw2alaw` from application `g721`. This function converts u-law (μ-law) values to a-law (A-law) values. Algorithms μ-law and A-law are the two main companding algorithms of G.711 (ITU, 1993).

The two variants, Listing 7a and Listing 7b, are generated by DivCon with relax rate 0.6, optimality gap 10%, and the cycle hamming distance, δ_{HD}. Figure 7 highlights four different ways in which the two variants differ.

First, DivCon may add no-operation instructions that affect the memory layout but not the semantics of the program. Interestingly, DivCon added an empty stack frame to Variant 2. The prologue (line 13 in Variant 2) and epilogue (line 42 in of Variant 2) instructions



(a) Diversity improvement

(b) Diversification time overhead

Figure 6: Improvement in diversity and diversification time overhead of LNS over RS for different values of the relax rate and the Hamming Distance δ_{HD}

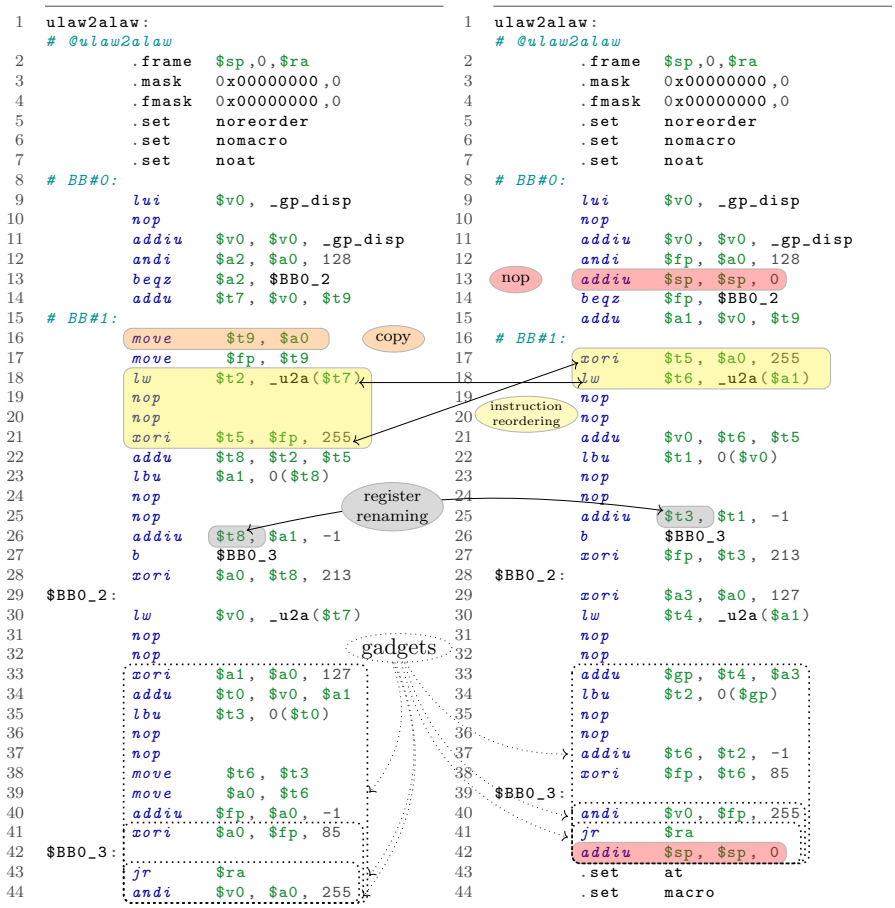
that build and destroy the empty stack frame are no-operations, however they contribute to the diversification of the function. Otherwise, DivCon adds MIPS `nop` instructions to fill the instruction schedule empty slots including the instruction delays due to their execution latency (see lines 19 and 20 of Variant 1). DivCon may add no-operations as long as the overhead they introduce does not exceed the allowed optimality gap.

Another transformation is the addition of `copy` operations to move data from one register to the other (highlighted at line 16 of Variant 1). This transformation assists register renaming, which improves diversification.

The third transformation that we have highlighted (lines 18-21 of Variant 1 and lines 17-18 of Variant 2) is instruction reordering. Here, whenever there is no data dependency between the instructions, the order of the instructions might change. Instruction reordering may break gadgets because the attacker expects a different instruction than the reordered instruction that is present at the same address.

Finally, the register assignment of different operations differs, with an example highlighted at line 26 of Variant 1 and line 25 of Variant 2. Register renaming breaks the attacker assumptions about the register that each gadget affects and uses. Other transformations, like spilling to the stack, are also possible. The function of Figure 7 is small and does not require spilling. However, DivCon may enable spilling if the overhead is not more than the allowed optimality gap (10% here).

Figure 7 shows some of the gadgets that are available in function `ulaw2alaw` surrounded in dotted rectangles. Interestingly, both variants contain a number of gadgets that all include the last gadget. This last gadget consists of a return jump, `jr`, and its delay slot, i.e. the instruction that follows the branch but is executed before it. No pair of gadgets in the two variants is identical with regards to either the content or the position in the code.



(a) g721.g711.ulaw2alaw - Variant 1

(b) g721.g711.ulaw2alaw - Variant 2

Figure 7: Example function diversification in MIPS32 assembly code

References

Abrath, B., Coppens, B., Mishra, M., den Broeck, J. V., & Sutter, B. D. (2020). Breakpad: Diversified binary crash reporting. *IEEE Transactions on Dependable Secure Computing*, *17*(4), 841–856.

Alaba, F. A., Othman, M., Hashem, I. A. T., & Alotaibi, F. (2017). Internet of Things security: A survey. *Journal of Network and Computer Applications*, *88*, 10–28.

Arteaga, J. C., Malivitsis, O. F., Pérez, O. L. V., Baudry, B., & Monperrus, M. (2021). Crow: Code diversification for webassembly. In *MADWeb'21-NDSS Workshop on Measurements, Attacks, and Defenses for the Web*.

Ashouri, A. H., Killian, W., Cavazos, J., Palermo, G., & Silvano, C. (2018). A survey on compiler autotuning using machine learning. *ACM Computing Surveys (CSUR)*, *51*(5), 1–42.

Baudry, B., Allier, S., & Monperrus, M. (2014). Tailored source code transformations to synthesize computationally diverse program variants. In *Proc. of ISSTA*, pp. 149–159.

Baudry, B., & Monperrus, M. (2015). The Multiple Facets of Software Diversity: Recent Developments in Year 2000 and Beyond. *ACM Comput. Surv.*, *48*(1), 16:1–16:26.

Birman, K. P., & Schneider, F. B. (2009). The monoculture risk put into context. *IEEE Security & Privacy*, *7*(1), 14–17.

Bletsch, T., Jiang, X., Freeh, V. W., & Liang, Z. (2011). Jump-oriented Programming: A New Class of Code-reuse Attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, pp. 30–40, New York, NY, USA. ACM.

Braden, K., Crane, S., Davi, L., Franz, M., Larsen, P., Liebchen, C., & Sadeghi, A.-R. (2016). Leakage-Resilient Layout Randomization for Mobile Devices. In *Proceedings 2016 Network and Distributed System Security Symposium*, San Diego, CA. Internet Society.

Castañeda Lozano, R., Carlsson, M., Blindell, G. H., & Schulte, C. (2019). Combinatorial Register Allocation and Instruction Scheduling. *ACM Trans. Program. Lang. Syst.*, *41*(3), 17:1–17:53.

Castañeda Lozano, R., Carlsson, M., Drejhammar, F., & Schulte, C. (2012). Constraint-Based Register Allocation and Instruction Scheduling. In Milano, M. (Ed.), *Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science, pp. 750–766, Berlin, Heidelberg. Springer.

Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.-R., Shacham, H., & Winandy, M. (2010). Return-oriented Programming Without Returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pp. 559–572, New York, NY, USA. ACM.

Chen, Y., Wang, Z., Whalley, D., & Lu, L. (2016). Remix: On-demand Live Randomization. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, CODASPY '16, pp. 50–61, New York, NY, USA. Association for Computing Machinery.

Chu, G. G. (2011). *Improving combinatorial optimization.* Ph.D. thesis, The University of Melbourne, Australia.

Cohen, F. B. (1993). Operating system protection through program evolution.. *Comput. Secur.*, *12*(6), 565–584.

Crane, S., Liebchen, C., Homescu, A., Davi, L., Larsen, P., Sadeghi, A., Brunthaler, S., & Franz, M. (2015). Readactor: Practical Code Randomization Resilient to Memory Disclosure. In *2015 IEEE Symposium on Security and Privacy*, pp. 763–780.

Davi, L. V., Dmitrienko, A., Nrnberger, S., & Sadeghi, A.-R. (2013). Gadge me if you can: secure and efficient ad-hoc instruction-level randomization for x86 and ARM. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pp. 299–310. tex.organization: ACM.

Forrest, S., Somayaji, A., & Ackley, D. H. (1997). Building diverse computer systems. In *Proceedings. The Sixth Workshop on Hot Topics in Operating Systems (Cat. No. 97TB100133)*, pp. 67–72. IEEE.

Gecode Team (2020). Gecode: Generic constraint development environment. Online: https://www.gecode.org.

Hamming, R. W. (1950). Error detecting and error correcting codes. *The Bell system technical journal*, *29*(2), 147–160.

Hebrard, E., Hnich, B., O'Sullivan, B., & Walsh, T. (2005). Finding Diverse and Similar Solutions in Constraint Programming. In *National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference*, p. 6.

Homescu, A., Jackson, T., Crane, S., Brunthaler, S., Larsen, P., & Franz, M. (2017). Large-Scale Automated Software Diversity—Program Evolution Redux. *IEEE Transactions on Dependable and Secure Computing*, *14*(2), 158–171.

Homescu, A., Neisius, S., Larsen, P., Brunthaler, S., & Franz, M. (2013). Profile-guided Automated Software Diversity. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, CGO '13, pp. 1–11, Washington, DC, USA. IEEE Computer Society.

Ingmar, L., de la Banda, M. G., Stuckey, P. J., & Tack, G. (2020). Modelling diversity of solutions. In *Proceedings of the thirty-fourth AAAI conference on artificial intelligence*.

ITU, T. (1993). General aspects of digital transmission systems. *ITU-T Recommendation G*, *729*.

Jacob, M., Jakubowski, M. H., Naldurg, P., Saw, C. W. N., & Venkatesan, R. (2008). The Superdiversifier: Peephole Individualization for Software Protection. In Matsuura, K., & Fujisaki, E. (Eds.), *Advances in Information and Computer Security*, Lecture Notes in Computer Science, pp. 100–120, Berlin, Heidelberg. Springer.

Kc, G. S., Keromytis, A. D., & Prevelakis, V. (2003). Countering code-injection attacks with instruction-set randomization. In *Proc. of CCS*, pp. 272–280.

Koo, H., Chen, Y., Lu, L., Kemerlis, V. P., & Polychronakis, M. (2018). Compiler-Assisted Code Randomization. In *2018 IEEE Symposium on Security and Privacy (SP)*, pp. 461–477.

Kornau, T., et al. (2010). Return oriented programming for the ARM architecture. Master's thesis, Ruhr-Universität Bochum.

Krueger, C. W. (1992). Software reuse. *ACM Comput. Surv.*, *24*(2), 131–183.

Larsen, P., Homescu, A., Brunthaler, S., & Franz, M. (2014). SoK: Automated Software Diversity. In *2014 IEEE Symposium on Security and Privacy*, pp. 276–291.

Lattner, C., & Adve, V. (2004). LLVM: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization*. IEEE.

Lee, C., Potkonjak, M., & Mangione-Smith, W. H. (1997). MediaBench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *International Symposium on Microarchitecture*, pp. 330–335. IEEE.

Lee, S., Kang, H., Jang, J., & Kang, B. B. (2021). Savior: Thwarting stack-based memory safety violations by randomizing stack layout..

Lundquist, G. R., Bhatt, U., & Hamlen, K. W. (2019). Relational processing for fun and diversity. In *Proceedings of the 2019 miniKanren and relational programming workshop*, p. 100.

Lundquist, G. R., Mohan, V., & Hamlen, K. W. (2016). Searching for Software Diversity: Attaining Artificial Diversity Through Program Synthesis. In *Proceedings of the 2016 New Security Paradigms Workshop*, NSPW '16, pp. 80–91, New York, NY, USA. ACM.

Pappas, V., Polychronakis, M., & Keromytis, A. D. (2012). Smashing the Gadgets: Hindering Return-Oriented Programming Using In-place Code Randomization. In *2012 IEEE Symposium on Security and Privacy*, pp. 601–615.

Petit, T., & Trapp, A. C. (2015). Finding Diverse Solutions of High Quality to Constraint Optimization Problems. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*.

Salehi, M., Hughes, D., & Crispo, B. (2019). Microguard: Securing bare-metal microcontrollers against code-reuse attacks. In *2019 IEEE Conference on Dependable and Secure Computing (DSC)*, pp. 1–8. IEEE.

Salwan, J. (2020). ROPgadget Tool. Online: http://shell-storm.org/project/ROPgadget/.

Shacham, H. (2007). The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pp. 552–561, New York, NY, USA. ACM.

Shaw, P. (1998). Using Constraint Programming and Local Search Methods to Solve Vehicle Routing Problems. In Maher, M., & Puget, J.-F. (Eds.), *Principles and Practice of Constraint Programming — CP98*, Lecture Notes in Computer Science, pp. 417–431, Berlin, Heidelberg. Springer.

Snow, K. Z., Monrose, F., Davi, L., Dmitrienko, A., Liebchen, C., & Sadeghi, A. (2013). Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *2013 IEEE Symposium on Security and Privacy*, pp. 574–588.

Sweetman, D. (2006). *See MIPS Run, Second Edition*. Morgan Kaufmann.

Tsoupidi, R. M., Castañeda Lozano, R., & Baudry, B. (2020). Constraint-based software diversification for efficient mitigation of code-reuse attacks. In *International Conference on Principles and Practice of Constraint Programming*, pp. 791–808. Springer.

Van Hentenryck, P., Coffrin, C., & Gutkovich, B. (2009). Constraint-Based Local Search for the Automatic Generation of Architectural Tests. In Gent, I. P. (Ed.), *Principles and Practice of Constraint Programming - CP 2009*, Lecture Notes in Computer Science, pp. 787–801. Springer Berlin Heidelberg.

Wagner, R. A., & Fischer, M. J. (1974). The string-to-string correction problem. *Journal of the ACM (JACM)*, *21*(1), 168–173.

Wang, S., Wang, P., & Wu, D. (2017). Composite Software Diversification. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 284–294.

Wartell, R., Mohan, V., Hamlen, K. W., & Lin, Z. (2012). Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pp. 157–168, New York, NY, USA. ACM.

Williams-King, D., Gobieski, G., Williams-King, K., Blake, J. P., Yuan, X., Colp, P., Zheng, M., Kemerlis, V. P., Yang, J., & Aiello, W. (2016). Shuffler: Fast and Deployable Continuous Code Re-Randomization.. pp. 367–382.

Appendix C

Publication 3

Vivienne: Relational Verification of Cryptographic Implementations in WebAssembly

Rodothea Myrsini Tsoupidi
KTH Royal Institute of Technology
Stockholm, Sweden
tsoupidi@kth.se

Musard Balliu
KTH Royal Institute of Technology
Stockholm, Sweden
musard@kth.se

Benoit Baudry
KTH Royal Institute of Technology
Stockholm, Sweden
baudry@kth.se

***Abstract*—We investigate the use of relational symbolic execution to counter timing side channels in WebAssembly programs. We design and implement Vivienne, an open-source tool to automatically analyze WebAssembly cryptographic libraries for constant-time violations. Our approach features various optimizations that leverage the structure of WebAssembly and automated theorem provers, including support for loops via relational invariants. We evaluate Vivienne on 57 real-world cryptographic implementations, including a previously unverified implementation of the HACL* library in WebAssembly. The results indicate that Vivienne is a practical solution for constant-time analysis of cryptographic libraries in WebAssembly.**

I. Introduction

The introduction of WebAssembly [1], a portable low-level language with focus on security and efficiency, has led to an array of security-sensitive applications. Cryptography libraries such as libsodium [2] and HACL* [3] are a prime example of such applications. Unfortunately, WebAssembly programs can be vulnerable to different types of attacks [4], including timing side channels.

The constant-time programming discipline is a well-known practice to defend against timing attacks [5], [6]. The main idea is to disallow the program's control flow and the memory access patterns that depend on program secrets. This is surprisingly challenging because many cryptographic routines are human-written [2], [7], [8] and thus, prone to errors, while compilers that preserve constant time are yet to emerge [2], [7]. This motivates the need for verification of constant-time implementations in WebAssembly.

Drawing on the verification-friendly structure of WebAssembly, existing solutions such as CT-wasm [9] enrich the WebAssembly type system with security annotations to enforce constant time. The efficiency of CT-wasm comes at the expense of a conservative analysis, e.g., by considering the whole memory as secret, thus leading to false positives or refactoring of constant-time programs. This paper explores the use of Relational Symbolic Execution (RelSE) to verify constant-time implementations in WebAssembly. The approach relies on an accurate modelling of the memory and other program optimizations, enabling a precise analysis that scales to real-world cryptographic implementations. In summary, this paper offers the following contributions:

- An RelSE-based approach for verifying constant-time implementations in WebAssembly programs.
- An automated invariant generation technique for analyzing implementations with loops.
- A thorough evaluation on 45 secure implementations and 12 insecure implementations in WebAssembly, including the previously non-verified WebAssembly implementation of HACL* (WHACL*).
- Vivienne, an open-source implementation of the approach.

II. Problem Setting

This section presents the problem setting, including the constant-time policy, and background on WebAssembly and related works.

A. Constant-time Policy

Constant-time programming discipline is a software-based defense against timing side-channel attacks. This discipline relies on the constant-time policy [10], which classifies values as secret (`high`) and public (`low`). The policy constrains the control-flow instructions and the memory operations to solely depend on public values, thus disallowing any secret-dependent control-flow instructions and memory accesses. Intuitively, the policy requires that any program executions with the same `low` values execute the same instructions and yield the same memory access patterns, independently of `high` values. This indicates that execution time of the program is not affected by secret data.

Listing 1
C function `tls1_cbc_remove_padding`

```
1  int tls1_cbc_remove_padding(const SSL *s,
2             SSL3_RECORD *rec, unsigned bs,
3             unsigned mac_size) {
4   int ii, i, j;
5   int l = rec->length;
6   ii = i = rec->data[l-1]; /* padding_length */
7   i++;
8   ...
9   for (j=(int)(l-i); j<(int)l; j++)
10   if (rec->data[j] != ii) /* Incorrect padding */
11    return -1;
```

```
12    ...
13  }
```

Listing 1 reports a code snippet of the OpenSSL's Lucky 13 timing vulnerability [11] to illustrate the issue. Function `tls1_cbc_remove_padding` removes the padding from a decrypted message that contains the plain text (secret), the Message Authentication Code (MAC) tag, and the padding. The size of the padding affects the execution time, which in turn reveals information about the size of the plain text. Specifically, `rec->data` holds the decrypted message together with the MAC tag and the padding, and is thus secret. Variables `i` and `ii` (line 6) contain the last item of array `rec->data`, which holds the padding size. Hence, the number of iterations of the `for` loop at line 9 depends on the secret-dependent variable `i`, which affects the execution time of the function. Similarly, the guard of `if` statement at line 10 depends on `ii`, which is also secret. Memory accesses also reveal information through timing due to the presence of caches. At line 10, the access to `rec->data[j]` reveals information about the value of index `j` by timing its presence in the cache.

B. WebAssembly

WebAssembly [1] is a stack-based typed low-level language serving as backend for both client-side computations, e.g., web browsers, and server-side computations [4] including stand-alone applications [12]. With some exceptions [8], WebAssembly code is compiler generated, e.g., via LLVM with support for C, C++, and Rust. Other languages, like Python and Julia, also provide support for WebAssembly. WASI Libc [12] is a library built on top of WASI system calls to enable I/O and memory management for WebAssembly programs.

The execution model of WebAssembly [1] consists of 1) an execution stack *es* that stores the instructions; 2) a value stack *vs* that holds the input arguments of the instructions, 3) a linear memory, and 4) the local and the global stores. WebAssembly has a structured control flow; for indirect calls (`call_indirect`), the call destination is an index to a function table; for conditional branch (`br_if`), the branch destination is an index i to enter (`loop`) or exit (`block`) the ith scope. Memory operations read from (`load`) and write to (`store`) the linear memory, and global variables are visible to all functions in a module. A function may also define local variables `lv`n including the function parameters. Modules are collections of functions with their own linear memory, and global variables [1].

Listing 2 shows an example WebAssembly module. The code is a simplified compiled version (using clang-10) of the C code in Listing 1. The code consists of a module (line 1-33), which imports a memory instance (`"_memory"`) from another module `$env` (line 3) and declares function `tls1_cbc_remove_padding` (line 4). The function takes four input parameters of type 32-bit integer and returns a 32-bit value (line 5). At line 6, the function declares five local variables and the rest of the function consists of the function body. The block at line 8 performs multiple initializations before the beginning of the loop (line 15). At line 10, instruction `local.tee` stores the top value of *vs* (here `rec->data + 1`) to `lv6` and pushes the same value back to *vs*. At line 15, the loop starts by loading `lv6` and `lv1` to *vs*. Instruction `i32.add` adds these two values and pushes back the result to *vs*. Finally, instruction `i32.load8_u` loads from the linear memory (`"_memory"`) the value at the index taken from the top of *vs*, i.e. the result of the addition. The loop body executes until instruction `br_if`, which reads one value from *vs*; if the value is non zero (`true`), the execution breaks out of the outermost block (lines 8-31), whereas if the value is zero (`false`), the execution continues to the next instruction, `br`, which unconditionally jumps back to the beginning of the loop (line 15).

Listing 2
WASM FUNCTION `TLS1_CBC_REMOVE_PADDING`

```
1  (module
2    ...
3    (import "env" "_memory" (memory (;0;) 2))
4    (func $tls1_cbc_remove_padding (type 2)
5          (param i32 i32 i32 i32) (result i32)
6      (local i32 i32 i32 i32 i32)
7      ...
8      block  ;; label = @1
9        ...
10       local.tee 6 ;; tee (rec->data + l)
11       ...
12       local.tee 1 ;; tee (1 - ii)
13       ...
14       block  ;; label = @2
15         loop  ;; label = @3
16           local.get 6 ;; get l
17           local.get 1 ;; get (j - l)
18           i32.add
19           i32.load8_u ;; load data[j] from memory
20           ...
21           i32.const 1
22           local.set 4 ;; store return value
23           ...
24           local.set 1 ;; j++
25           ...
26           br_if 2 (;@1;) ;; break if j >= l
27           br 0 (;@3;) ;; continue to loop
28         end
29       end
30       ...
31     end
32     ...))
```

WebAssembly programs may be vulnerable to timing side-channel attacks. The constant-time policy for WebAssembly concerns control-flow instructions, i.e. `br_if`, `if`, `br_table`, and `call_indirect`, and the memory operations, i.e. `load` and `store`.

C. Related Work

Several works have aimed at improving the security of WebAssembly [4], [9], [13], [14], [15]. CT-wasm [9] proposes a type system to check the constant-time policy. Type checking is very efficient but it suffers from the annotation burden and the conservative nature of the analysis. In

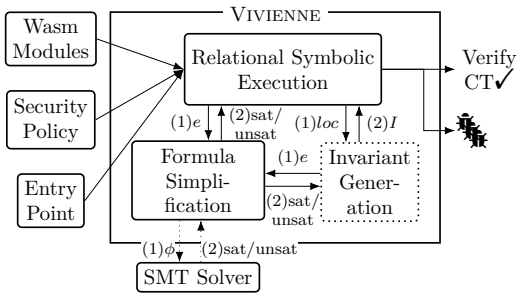


Fig. 1. Vivienne Architecture

$$
\begin{array}{lll}
v\ (values) & ::= h_n \mid l_n \mid c, \quad c \in \mathbb{Z}, n \in \mathbb{N}_0 \\
\rho\ (relational\ values) & ::= \langle v, v \rangle \\
e\ (expressions) & ::= \rho \mid \texttt{Add}(e,e) \mid \texttt{Sub}(e,e) \mid ... \\
 & \quad \mid \texttt{Le}(e,e) \mid \texttt{Load}(e,\mu) \\
i\ (instructions) & ::= \texttt{br\_if}\ l \mid ... \mid \texttt{load}, \quad l \in \mathbb{N}_0 \\
\hline
\mu\ (memory) & ::= \bot \mid \texttt{Store}(e,e,\mu) \\
st\ (stack) & ::= \varnothing \mid e :: st \\
pc\ (path\ condition) & ::= \texttt{true} \mid e \wedge pc \\
es\ (execution\ stack) & ::= \varnothing \mid i :: es \\
lv\ (local\ variables) & ::= \{lv_0 \mapsto e, ..., lv_n \mapsto e\}
\end{array}
$$

Fig. 2. Symbolic Data Structures

CT-wasm, this is reflected by the treatment of the whole memory as secret, e.g. requiring that every `load` operation returns a `high` value, which may require refactoring of the programs to make them amenable to the analysis (e.g., `poly1305_blocks` and `poly1305_update` functions of a WebAssembly TweetNaCl implementation [8]). Our approach aims at overcoming these limitations by means of RelSE, using a more accurate memory model and no extensive annotation burden. Moreover, we expect our analysis to yield less false positives because it relies on symbolic execution which is more precise than security type systems. For example, an expression such as `secret−secret` would be correctly identified as the constant `0`. However, as we will see, our solution comes with a computation cost due to the increased precision.

Almeida et al. [10] use product programs to verify constant-time for C implementations. A drawback of verifying the constant-time policy for high-level languages is that the analysis does not provide guarantees on the security of the generated code (see `ct_select` implementations [16]). Daniel et al. [16] verify constant-time programs at the binary level using RelSE. Web browsers using WebAssembly typically leverage Just-in-time (JIT) compilation, which does not result in binary file generation. Moreover, the verification of constant-time at the WebAssembly level provides opportunities for optimization due to WebAssembly's structured design. HACL* [17] uses a high-level specification language to generate a formally verified cryptographic library that is available in different languages including C and WebAssembly [3].

III. Vivienne: RelSE for WebAssembly

Vivienne analyzes WebAssembly implementations with respect to constant time. Figure 1 shows a high-level view of the tool. Vivienne takes three inputs: 1) the *WebAssembly modules* containing the functions to analyze, 2) the *security policy* annotating the memory regions and the parameters of the entry function, and 3) the *entry point* describing the entry function to analyze. Then, Vivienne performs RelSE on the entry function, reporting the discovered constant-time vulnerabilities (if any). We describe the different components of Vivienne using Listing 2 as a running example.

WebAssembly Modules The modules include the entry function to verify and its dependencies, possibly involving different modules. For example, the module in Listing 2 imports the memory from another module `$env` (line 3) and defines function `tls1_cbc_remove_padding` (lines 4-28).

Security Policy and Entry Point The security policy specifies the parts of the memory and the arguments of the entry function that contain public or secret values. Listing 3 reports the policy for function `tls1_cbc_remove_padding`. The policy specifies the bytes 2000 to 2039 (i.e. pointer `s`) and the memory of struct `rec` as public (not shown), and the bytes 2048 to 2111 (i.e. `rec->data`) as secret, thus reflecting the specification in Listing 1. Moreover, Vivienne requires the code of the modules (line 8) and the *entry function* (lines 9-11). The latter includes the security policy for its arguments which can be either concrete or symbolic values. Lines 9–11 specify the concrete and symbolic arguments for analyzing function `tls1_cbc_remove_padding` via RelSE. The function takes four arguments: 1) the memory index of `s`; 2) the memory index of struct `rec`; 3) the block size, which is a public symbolic value; and 4) the `MAC` size which is also a public symbolic value. Vivienne recognizes public (secret) symbolic values that start with letter `l` (`h`).

Listing 3
Security policy and Entry Function

```
1  (module $env
2   (memory (;0;) $memory (export "_memory") 2)
3   (public (i32.const 2000) (i32.const 2039));;s
4   ...
5   (secret (i32.const 2048) (i32.const 2111));;data
6  )
7  ;;definition of tls1_cbc_remove_padding-Listing 2
8  ...
9  (symb_exec "tls1_cbc_remove_padding"
10   (i32.sconst 2000) (i32.sconst 2040) ;; concrete
11   (i32.sconst l1)   (i32.sconst l2))  ;; symbolic
```

Relational Symbolic Execution Vivienne uses the above-mentioned inputs to initiate RelSE [18] for the entry

function. RelSE performs symbolic execution on relational states representing two program executions with identical public values but different secret values. We now describe the ingredients underpinning the constant-time analysis with VIVIENNE.

a) Symbolic State: A symbolic state σ consists of 1) the execution stack es, that contains the WebAssembly instructions, 2) the symbolic stack st, 3) the symbolic memory μ, 4) the symbolic local (and global) variables lv, and 5) the path condition pc. Figure 2 summarizes these five components of a symbolic state $\sigma = \langle es, st, \mu, lv, pc\rangle$. By convention, the values starting with h (l) are secret (public). Our symbolic analysis operates on pairs of symbolic values ρ. We write $\rho_{|l}$ ($\rho_{|r}$) to denote the first (second) element of a pair ρ. For public values, we have that $\rho_{|l} = \rho_{|r}$ and write $\langle v\rangle$, while for secret values $\rho_{|l}$ and $\rho_{|r}$ may differ. We lift this notation to expressions and the memory as expected.

b) Execution Path Exploration: We use small-step symbolic evaluation to analyze the instructions. At every step, the analysis takes a symbolic state as input and returns a list of symbolic states that correspond to the feasible execution paths. We visit the instructions in a depth-first search fashion and collect all path conditions pc to check path feasibility using an Satisfiability Modulo Theories (SMT) solver.

c) Symbolic Stack: The symbolic stack holds symbolic expressions e resulting from stack operations on symbolic values. Consider the `get` instructions at lines 16–17 in Listing 2 with the current symbolic memory μ and empty symbolic stack st. The program loads the symbolic expressions of `lv6` i.e. $\langle 2112\rangle$, and `lv1` i.e. $\texttt{Sub}(\langle 1\rangle, \texttt{Load}(\langle 2111\rangle, \mu))$ to the stack st. At line 18, the analysis of instruction `add` pops the two symbolic expressions off the stack st and pushes back the result, $\texttt{Add}(\langle 2112\rangle, \texttt{Sub}(\langle 1\rangle, \texttt{Load}(\langle 2111\rangle, \mu)))$.

d) Memory Operations: When analyzing a memory operation at index e, as in $\langle \texttt{load} :: es, e :: st, \mu, lv, pc\rangle$ or $\langle \texttt{store} :: es, e_1 :: e :: st, \mu, lv, pc\rangle$, the analysis generates a formula, $\phi = (T(e)|_{|r} \neq T(e)_{|l})$ to check that the index is not secret-dependent. The function $T : e \rightarrow \langle Exp, Exp\rangle$ translates the index expression e to a pair of SMT expressions Exp. If e only depends on public values, then for all valuations of e, $e_{|r} = e_{|l}$, thus ϕ is *unsatisfiable* and the memory operation is `safe`. However, if ϕ is *satisfiable*, then there are concrete values, such that the memory addresses for the two executions, $e_{|r}$ and $e_{|l}$, are different. This is only possible if expression e depends on secret values, and, thus, the solution to ϕ reveals a violation of constant time. In our example in Listing 2, load operation `load8_u` at line 19 has as index the top value of st, $\texttt{Add}(\langle 2112\rangle, \texttt{Sub}(\langle 1\rangle, \texttt{Load}(\langle 2111\rangle, \mu)))$. The policy in Listing 3 specifies $\texttt{Load}(\langle 2111\rangle, \mu)$ as secret, i.e. $\texttt{Load}(\langle 2111\rangle, \mu) = \langle h_1, h_1'\rangle$ with $h_1 \neq h_1'$. Thus, the generated formula $\phi = (2112+(1-h_1)) \neq (2112+(1-h_1'))$ is satisfiable for different values of h_1 and h_1'. This means that there exist the two concrete executions that differ with regards to the memory index, which violates constant-time.

e) Control-flow Instructions: Like memory operations, control-flow instructions require checking that boolean expression e, as in $\langle \texttt{br\_if}\ 0 :: es, e :: st, \mu, lv, pc\rangle$, is not secret-dependent. Our analysis generates a formula to check whether the two paths of the relational state take different branches. WebAssembly considers value *zero* as false and *any non-zero* value as true, hence the generated formula is $\phi = (T(e)_{|r} = 0) \wedge (T(e)_{|l} \neq 0)$. Formula ϕ is satisfiable only if there is a valuation of e such that the two executions follow different execution paths, indicating a violation of the constant-time policy.

Formula Simplification When RelSE needs to check the constant-time policy for an expression e, it first passes e to the simplification step (SS). SS translates the expression to a pair of SMT expressions, $e' = T(e)$, using the theory of bitvectors and arrays (32-bit indexed byte array), `QF_ABV`. The transformation includes simplification and memoization steps to reduce the recalculation overhead. Finally, based on the type of the query, namely memory operation or control-flow statement, this step generates formula ϕ. For our previous example, SS first translates expression $e = \texttt{Add}(\langle 2112\rangle, \texttt{Sub}(\langle 1\rangle, \texttt{Load}(\langle 2111\rangle, \mu)))$ to two SMT expressions $2112 + (1 - h_1)$ and $2112 + (1 - h_1')$, which are then simplified to $2113 - h_1$ and $2113 - h_1'$, hence the final formula becomes $\phi = (2113 - h_1) \neq (2113 - h_1')$. To solve the simplified formula, VIVIENNE invokes an SMT solver. For simple formulas, however, the resulting ϕ may already be a concrete boolean, e.g., `false`, allowing VIVIENNE skip a call to the SMT solver.

SMT Solver After the simplification step, VIVIENNE invokes an SMT solver for solving the simplified formula, ϕ. The SMT solver of VIVIENNE has two modes, one for small formulas and one for large and complex formulas. For small formulas, VIVIENNE uses a solver that provides bindings to the implementation language of VIVIENNE and thus, has a reduced communication cost. However, for larger formulas, the communication overhead is less significant compared to the benefit of using a more powerful SMT solver. In particular, for larger queries VIVIENNE uses a portfolio solver were many solvers take as input the same formula and the solver that finishes first returns the result. To decide over which solver mode to use, VIVIENNE uses the *number of expressions* in the formula.

Invariant Generation VIVIENNE has an optional invariant generation step for analyzing loops. When invariant generation is enabled and the analysis visits a loop at location *loc*, VIVIENNE starts a preprocessing step to automatically generate a relational invariant I. The invariant defines the variables (local variables, global variables, and memory) that are public, i.e. $I = \{\forall x \in V_p \subseteq V.\ x_{|l} = x_{|r}\}$, where V is the set of all variables modified in the loop and V_p is the subset of the modified variables that are public. To discover whether a variable is public or secret,

the preprocessing step queries the SMT solver about the security policies of the modified variables, V, after symbolically executing one loop iteration. That is, given a variable $x \in V$, the preprocessing step generates a query, $\phi = (x_{|l} \neq x_{|r})$. If the query is unsatisfiable, then the variable is assumed to be public and x is added to V_p, otherwise, it is assumed to be secret. In the special case of $x_{|l} = x_{|r} = c \in \mathbb{Z}$, the analysis assumes that x has a symbolic value c and adds the equality constraint $x = c$ to the invariant, I. After generating invariant I, the analysis continues with verifying this invariant. To do that, Vivienne 1) generates fresh symbolic variables (havoc) for all modified variables $x \in V$, 2) assumes that the invariant, I, holds, 3) performs RelSE on the loop body with the havoced values and discovers possible vulnerabilities, 4) verifies that the invariant holds by asserting I on the new relational state. If the generated invariant is not a loop invariant, then the last step will fail. After analyzing the loop body, the analysis continues outside the loop. The invariant verification algorithm is a generalization of standard (functional) invariant checking, hence we expect the loop analysis to be sound, as supported by the experiments.

Consider the loop at $loc = 15$ in Listing 2. Local variables 1 and 4 are modified in the loop body, i.e. $V = \{lv1, lv4\}$. Of these, $lv1$ stores `j` (line 24), which is secret because it depends on `rec->data[l-1]` and $lv4$ stores value 1, which is public. Thus, $V_p = \{lv4\}$, hence the invariant is $I = \{lv4_{|l} = lv4_{|r}\}$. To analyze the loop, Vivienne 1) havocs $lv1$ and $lv4$, 2) assumes the invariant I, i.e. that $lv4$ is initially public, 3) performs RelSE at the loop body to discover constant-time vulnerabilities, and 4) asserts the invariant I. Here, the program assigns $lv4$ only once in the loop body, at line 22, where, $lv4$ takes value one, which is public, and thus, the invariant I holds.

Output Vivienne outputs the discovered constant-time violations (🐞), if any, as well as the SMT solver-generated counterexamples that witness these violations.

Vivienne is implemented as an extension of the WebAssembly reference interpreter [19] in OCaml, using OCaml compiler 4.06. Vivienne uses the OCaml interface of z3 [20] to generate and simplify the constant-time formulas, and solve queries that have a small number of expressions. For larger formulas, Vivienne uses a portfolio solver consisting of four solvers, i.e. Boolector [21], Yices2 [22], CVC4 [23], and Z3 [20] running in parallel. Vivienne is publicly available online at https://github.com/romits800/Vivienne.

IV. Evaluation

We evaluate Vivienne with respect to three research questions:

RQ1: Can we use RelSE for constant-time analysis of real-world cryptographic implementations in WebAssembly? To investigate the effectiveness and efficiency of RelSE for constant-time analysis on WebAssembly programs, we use Vivienne to analyze the implementations of seven cryptographic libraries within a time limit of 90 minutes.

RQ2. To what extent do the automatically generated loop invariants affect the scalability and precision of RelSE? We evaluate Vivienne's support for automatic invariant generation on our benchmarks and compare it to the results of RQ1.

RQ3. How does Vivienne compare to existing approaches for constant-time analysis of WebAssembly? We compare Vivienne with CT-wasm [9] with regards to simplicity, permissiveness, and efficiency.

A. Experimental Setup and Overview of Benchmarks

We run the experiments on a machine running Debian GNU/Linux 10 (buster) on an IntelCore™i9-9920X processor 3.50GHz with 64GB of RAM. We used the LLVM-10 compiler with WASI libc [12] and two optimization levels (`-O0` and `-O3`) for compiling our C benchmarks to WebAssembly. Vivienne uses a time limit of 90 minutes for each benchmark and a threshold of 1500 expressions to trigger a call to the portfolio solver.

We evaluate Vivienne with seven cryptography libraries, including both constant-time and non-constant-time implementations. Some benchmarks have been used in prior works [9], [16] to evaluate constant-time policies, which provides us with common ground for comparison. We extract the security policies for the first two libraries from the type annotations of CT-wasm [24] and use the policies of Binsec/Rel [25] for the other libraries. The full details of our benchmarks are available at https://github.com/romits800/Vivienne_eval.

CT-wasm benchmarks (CTw): Three handwritten WebAssembly benchmarks from CT-wasm [9]. We verify the `encrypt` and `decrypt` functions of `Salsa20` and `TEA`, and the `transform` and `update` functions of `SHA256`.

TweetNaCl WebAssembly (Tw): WebAssembly implementation of TweetNaCl [8] previously verified by CT-wasm [9]. We verify `core_hsalsa20`, `core_salsa20`, and `crypto_onetimeauth`.

WHACL* (WH): A formally verified cryptography library compiled to WebAssembly [3]. We verify `Chacha20`, `Curve25519_51`, `Poly1305_32`, `Salsa20`, and `Hash_SHA2` in WHACL* v3.0.0. To our best knowledge, this is the first time WHACL* is verified.

Libsodium (L0, L3): A cryptography library written in C [2]. Vivienne verifies the constant-time implementations of `crypto_aead`, `crypto_auth`, `crypto_stream`, `crypto_onetimeauth`, `crypto_core`, and `crypto_hash` for Libsodium v.1.0.18-stable with optimization levels `-O0` and `-O3`.

BearSSL (B0, B3): An implementation of SSL/TLS in C. We verify the constant-time functions `aes_ct_cbcenc` and `des_ct_cbcenc` and the non constant-time functions `aes_big_cbcenc` and `des_tab_cbcenc`. B0 includes the

Bench.		VIVIENNE$_{\text{unroll}}$				VIVIENNE$_{\text{inv}}$			
BS	A	✓	✗	#FS	#SS	✓	✗	#FS	#SS
CTw	6	6/6	0/0	4K	0	6/6	0/0	814	412
Tw	3	3/3	0/0	181	0	3/3	0/0	320	164
WH	6	5/6	0/0	126K	0	6/6	0/0	70K	7K
B0	4	2/2	2/2	32K	40	1/2	0/2	10K	873
B3	4	2/2	2/2	2K	40	0/2	1/2	158K	3K
L0	8	8/8	0/0	113K	18	2/8	0/0	21K	347
L3	8	8/8	0/0	9K	18	3/8	0/0	3K	309
A0	8	5/5	3/3	683	31	No loops			
A3	8	5/5	3/3	55	9				
Lu0	1	0/0	0/1	25K	4K	0/0	1/1	539	217
Lu3	1	0/0	1/1	3K	3K	0/0	0/1	94	63
Sum	57	44/45	11/12	-	-	21/35	2/6	-	-

TABLE I
VERIFYING 57 CRYPTOGRAPHY FUNCTIONS WITH VIVIENNE, WITH UNROLLING AND WITH INVARIANT INFERENCE. THE NUMBERS IN RED DENOTE INCOMPLETE RESULTS.

functions with optimization level `-O0` and B3 is optimization `-O3`.

Almeida et al. [10] (A0, A3): Five constant-time and three non-constant-time implementations of `select` and `sort`. We analyze WebAssembly binaries compiled with optimization levels `-O0` and `-O3`.

Lucky 13 (Lu0, Lu3): A known timing vulnerability [11] of TLS implementations (see Listing 1). We analyze function `tls1_cbc_remove_padding` of OpenSSL 1.0.1 [26] with optimization levels `-O0` and `-O3`.

B. Results

This section discusses the evaluation results for each of the research questions. Table I presents the aggregated results of the analysis with VIVIENNE. The columns under `Bench` describe the benchmarks, i.e. the abbreviated library name, `BS`, and the number of analyzed algorithms, `A`. The next two columns present VIVIENNE's results with loop unrolling (VIVIENNE$_{\text{unroll}}$) and with loop invariant (VIVIENNE$_{\text{inv}}$). We report the number of verified constant-time implementations, ✓, the number of vulnerable implementations ✗, the number of formulas subject to simplification, `#FS`, and the number of queries that VIVIENNE propagates to the SMT solver, `#SS`. Note that `#SS` is the subset of `#FS` that requires a call to the SMT solver. We highlight in red the incomplete results. For example, VIVIENNE with loop unrolling (VIVIENNE$_{\text{unroll}}$) was able to verify successfully five out of six implementations of WH within the time limit of 90 minutes. Appendix A includes the full evaluation results for VIVIENNE$_{\text{unroll}}$, while the results for VIVIENNE$_{\text{inv}}$ are available as supplementary material online [27].

1) RQ1: Can we use RelSE for constant-time analysis of real-world cryptographic implementations in WebAssembly? : To evaluate the effectiveness of VIVIENNE in analyzing cryptographic libraries, we consider the rate of successfully analyzed algorithms for both secure (✓) and insecure (✗) implementations. The summarized results (Sum) in Table I show that VIVIENNE$_{\text{unroll}}$ analyzes successfully 44 out of 45 constant-time implementations and 11 out of 12 non-constant-time implementations for a total 55/57 implementations. This corresponds to 96% success rate while reporting no false positive. The two outliers are `Hacl_Curve25519_51_scalarmult` of WH and `tls1_cbc_remove_padding` of Lu0. The former contains a loop with 256 iterations, each generating 9108 queries. One of these queries affects an increasingly large part of the total execution time for an iteration. The corresponding formula models the satisfiability of a branch condition that depends on the stack pointer, which WHACL* stores in memory. As a result, the formula has to encode the whole memory, which contributes with 3054 new memory stores for every iteration, thus increasing the time for the generation and simplification of the formula. This can be inferred from the results of Table I, where the total six implementations of WH generate 126K formulas (`#FS`), of which 80896 correspond to `Hacl_Curve25519_51_scalarmult`.

The second outlier is `tls1_cbc_remove_padding` with `-O0`, which contains a loop with non-constant bound, as reported in line 9 in Listing 1. The lack of a constant bound forces VIVIENNE$_{\text{unroll}}$ to consider all possible values for `rec->data[l-1]`, which is an eight-byte value. This leads to maximum 256 iterations for every path that visits the loop. We find that the optimization level `-O0` includes a number of stack operations that modify the memory at every iteration. As we can see in Table I, this leads to 25K `#FS` and 4K `#SS`. The former requires on average 0.01 seconds (4 minutes in total) for simplification, whereas the latter requires 0.87 seconds (58 minutes in total) for SMT solving.

In summary, our results show that RelSE can be used to analyze real-world cryptographic implementations, while the memory operations and loops remain the main bottleneck for the SMT solver. VIVIENNE$_{\text{inv}}$ addresses the challenge of loops by generating relational loop invariants automatically. Further discussion about the SMT solver results of our analysis are available as supplementary material [27].

2) RQ2. To what extent do the automatically generated loop invariants affect the scalability and precision of RelSE?: Our results in Table I show that VIVIENNE$_{\text{inv}}$ is able to successfully analyze constant-time implementations for the first three benchmark libraries. It also analyzes successfully the implementations of WH and Lu0 that VIVIENNE$_{\text{unroll}}$ could not handle. Perhaps surprisingly, VIVIENNE$_{\text{inv}}$ performs poorly on the benchmarks B0, B3, L0, and L3, analyzing only 29% of the implementations. The main reason is that the havocing of modified variables during the invariant generation replaces constant values with unbounded symbolic values. This triggers a path explosion whenever a conditional instruction is analyzed with the new symbolic values. Moreover, it increases the search space for the solver and the complexity of queries whenever a symbolic value indexes the memory in store operations. In Table I, the number of solver queries, `#SS`, for VIVIENNE$_{\text{inv}}$ is larger than for VIVIENNE$_{\text{unroll}}$,

which reflects the increase in the complexity because the solver queries (`#SS`) report the formulas that cannot be resolved during the simplification stage. For the benchmarks Tw and B3, the number of queries, `#FS`, also increases due to path explosion. By contrast, for the benchmarks that VIVIENNE$_{\text{inv}}$ analyzes successfully, `#FS` decreases due to the reduction of loop iterations by the loop invariant.

In summary, VIVIENNE$_{\text{inv}}$ analyzes successfully 56% of the implementations, including two implementations for which VIVIENNE$_{\text{unroll}}$ failed. This shows that VIVIENNE$_{\text{inv}}$ complements VIVIENNE$_{\text{unroll}}$ for constant-time analysis.

3) RQ3: How does VIVIENNE *compare to existing approaches for constant-time analysis of WebAssembly?:* To our best knowledge, CT-wasm [9] is the only constant-time analysis tool for WebAssembly. We consider three dimensions for comparison: 1) simplicity, 2) permissiveness, and 3) efficiency. Simplicity refers to the required user effort to verifying a target implementation. CT-wasm relies on type annotations for the program, which can be partially inferred [9]. By contrast, VIVIENNE requires only the security policies and entry-point function, otherwise no further modifications to the generated WebAssembly binary are needed. This reduces the user effort for analyzing a program. Permissiveness refers to the ability of the method to analyze and successfully verify cryptographic implementations. CT-wasm considers the whole memory as secret, which rules out any secure programs that store public values in memory. For example, CT-wasm required refactoring three functions of the TweetNaCl [8] library, i.e. `poly1305_blocks`, `poly1305_update`, and `poly1305_finish`, to make it amenable to verification. By contrast, VIVIENNE analyzes and verifies the whole implementation of (`crypto_onetimeauth`), with no modifications to the original code. Moreover, VIVIENNE could analyze and verify 57 WebAssembly implementations, including the two libraries CTw and Tw which were verified by CT-wasm [9]. With regards to efficiency, CT-wasm is clearly superior to VIVIENNE because it relies on type checking, while VIVIENNE performs expensive symbolic analysis and constraint solving. However, as we have seen in RQ1, VIVIENNE was still able to analyze real-world WebAssembly implementations within a reasonable time limit.

To summarize, VIVIENNE verifies a larger number of cryptographic implementations than CT-wasm with no need for refactoring and with minimal annotation efforts at the expense of an efficiency cost.

4) Discussion: Ideally, an accurate analysis should be implemented as close to the hardware as possible to avoid vulnerabilities introduced by compiler transformations. For VIVIENNE, the structured control flow of WebAssembly facilitates the analysis, while binary-level analyses face challenges with unstructured control flow and diversity of architectures [28], [16]. This raises the question of whether constant-time programs at WebAssembly level preserve the property at the machine level.

The machine code generated from a WebAssembly binary relies on the compiler of the respective runtime system. Unfortunately, a direct analysis of this machine code with tools like Binsec/Rel [16] is not possible due to the different calling conventions and implementation details of Binsec/Rel. A comparison of VIVIENNE's results at the WebAssembly level with Binsec/Rel's results at the machine level for the benchmarks L0, L3, B0, B3, A0, A3, Lu0, and Lu3 in Table I shows that both tools yield the same result on all benchmarks, except of the `select` implementations of the benchmarks of Almeida et al. [10]. The difference is manifested in the compilation of the `select` implementations at optimization level `-O3`, which Binsec/Rel identifies as insecure. In our experiments, LLVM-10 with flag `-O3` compiles all the C implementations of `select` (A3 in Table I) to one WebAssembly `select` instruction. The compilation from WebAssembly to machine code translates the WebAssembly `select` instruction either to a constant-time conditional assignment (safe), e.g. `cmov` for x86, or to a set of instructions that include a branch instruction (unsafe), depending on the target machine and the compiler implementation. To account for these differences, VIVIENNE provides a command-line option for treating the WebAssembly `select` instruction as unsafe.

V. Conclusion

This paper presented VIVIENNE, an open-source tool for analyzing constant-time for WebAssembly programs. VIVIENNE relies on RelSE and leverages the structure of WebAssembly to implement several optimizations, including automated invariant generation. We used VIVIENNE to analyze successfully 57 cryptographic implementations with minimal annotation overhead and no code refactoring. Moreover, VIVIENNE is the first tool to verify constant time for the WebAssembly implementation of HACL*.

Acknowledgments

We thank anonymous reviewers for their helpful feedback. This work is partially supported by the Wallenberg AI, Autonomous Systems, and Software Program (WASP) funded by Knut and Alice Wallenberg Foundation, the TrustFull project funded by the Swedish Foundation for Strategic Research (SSF), the JointForce project funded by the Swedish Research Council (VR), and Digital Futures.

References

[1] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the web up to speed with WebAssembly," in *Proc. of the Conf. on Programming Language Design and Implementation (PLDI)*, 2017, pp. 185–200.

[2] Libsodium Community, "The sodium cryptography library (Libsodium)," 2018. [Online]. Available: https://libsodium.gitbook.io/doc

[3] J. Protzenko, B. Beurdouche, D. Merigoux, and K. Bhargavan, "Formally Verified Cryptographic Web Applications in WebAssembly," in *2019 IEEE Symposium on Security and Privacy (SP)*, May 2019, pp. 1256–1274.

[4] D. Lehmann, J. Kinder, and M. Pradel, "Everything old is new again: Binary security of WebAssembly," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 217–234.

[5] D. Molnar, M. Piotrowski, D. Schultz, and D. A. Wagner, "The program counter security model: Automatic detection and removal of control-flow side channel attacks," in *Information Security and Cryptology - ICISC 2005, 8th International Conference, Seoul, Korea, December 1-2, 2005, Revised Selected Papers*, 2005, pp. 156–168.

[6] J. B. Almeida, M. Barbosa, G. Barthe, and F. Dupressoir, "Verifiable Side-Channel Security of Cryptographic Implementations: Constant-Time MEE-CBC," in *Fast Software Encryption*, ser. Lecture Notes in Computer Science, T. Peyrin, Ed. Berlin, Heidelberg: Springer, 2016, pp. 163–184.

[7] T. Pornin, "Bearssl, a smaller SSL/TLS library," last accessed May 14, 2021. [Online]. Available: https://bearssl.org/

[8] T. Stüber, "TorstenStueber/TweetNacl-WebAssembly," Oct. 2019. [Online]. Available: https://github.com/TorstenStueber/TweetNacl-WebAssembly

[9] C. Watt, J. Renner, N. Popescu, S. Cauligi, and D. Stefan, "CT-wasm: type-driven secure cryptography for the web ecosystem," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 77:1–77:29, Jan. 2019. [Online]. Available: http://doi.org/10.1145/3290390

[10] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, "Verifying constant-time implementations," in *25th USENIX security symposium (USENIX security 16)*. Austin, TX: USENIX Association, Aug. 2016, pp. 53–70. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/almeida

[11] N. J. Al Fardan and K. G. Paterson, "Lucky Thirteen: Breaking the TLS and DTLS Record Protocols," in *2013 IEEE Symposium on Security and Privacy*, May 2013, pp. 526–540, iSSN: 1081-6011.

[12] L. Clark, "Standardizing wasi: A system interface to run webassembly outside the web," *Mozilla Hacks–the Web developer blog, March*, 2019.

[13] C. Watt, A. Rossberg, and J. Pichon-Pharabod, "Weakening WebAssembly," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 133:1–133:28, Oct. 2019.

[14] M. Vassena, C. Disselkoen, K. v. Gleissenthall, S. Cauligi, R. G. Kıcı, R. Jhala, D. Tullsen, and D. Stefan, "Automatically eliminating speculative leaks from cryptographic code with blade," in *Proc. Symp. on Principles of Programming Languages (POPL 2021)*, 2021. [Online]. Available: http://arxiv.org/abs/2005.00294

[15] S. Narayan, C. Disselkoen, D. Moghimi, S. Cauligi, E. Johnson, Z. Gang, A. Vahldiek-Oberwagner, R. Sahita, H. Shacham, D. Tullsen, and D. Stefan, "Swivel: Hardening WebAssembly against Spectre," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/narayan

[16] L.-A. Daniel, S. Bardin, and T. Rezk, "Binsec/Rel: Efficient Relational Symbolic Execution for Constant-Time at Binary-Level," in *2020 IEEE Symposium on Security and Privacy (SP)*, May 2020, pp. 1021–1038, iSSN: 2375-1207.

[17] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche, "HACL*: A Verified Modern Cryptographic Library," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: Association for Computing Machinery, Oct. 2017, pp. 1789–1806. [Online]. Available: http://doi.org/10.1145/3133956.3134043

[18] G. P. Farina, S. Chong, and M. Gaboardi, "Relational Symbolic Execution," in *Proceedings of the 21st International Symposium on Principles and Practice of Declarative Programming*, ser. PPDP '19. New York, NY, USA: Association for Computing Machinery, Oct. 2019, pp. 1–14.

[19] W. C. Group, "Webassembly Reference Interpreter," 2018. [Online]. Available: https://github.com/WebAssembly/spec/tree/master/interpreter

[20] L. de Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, C. R. Ramakrishnan and J. Rehof, Eds. Berlin, Heidelberg: Springer, 2008, pp. 337–340.

[21] R. Brummayer and A. Biere, "Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, S. Kowalewski and A. Philippou, Eds. Berlin, Heidelberg: Springer, 2009, pp. 174–177.

[22] B. Dutertre, "Yices 2.2," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, A. Biere and R. Bloem, Eds. Cham: Springer International Publishing, 2014, pp. 737–744.

[23] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, "CVC4," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, G. Gopalakrishnan and S. Qadeer, Eds. Berlin, Heidelberg: Springer, 2011, pp. 171–177.

[24] "Ct-wasm," https://github.com/PLSysSec/ct-wasm-ports, accessed: 2021-06-11.

[25] "Binsec/rel," https://github.com/binsec/rel_bench, accessed: 2021-06-11.

[26] "Openssl dtls," https://github.com/openssl/openssl/blob/OpenSSL_1_0_1/ssl/d1_enc.c, accessed: 2021-06-11.

[27] R. M. Tsoupidi, M. Balliu, and B. Baudry, "Supplementary Material for Vivienne: Relational Verification of Cryptographic Implementations in WebAssembly," https://doi.org/10.5281/zenodo.5409477, 2021, accessed: 2021-09-02.

[28] M. Balliu, M. Dam, and R. Guanciale, "Automating Information Flow Analysis of Low Level Code," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14. New York, NY, USA: Association for Computing Machinery, Nov. 2014, pp. 1080–1091. [Online]. Available: https://doi.org/10.1145/2660267.2660322

Appendix A
Evaluation Results

Table IV and Table V show the complete results of the evaluation for VIVIENNE$_{\text{unroll}}$ and VIVIENNE$_{\text{inv}}$, respectively. The experiments for the two tables use a time limit of 90 minutes and the reported time values are in seconds and consist of the average and standard deviation after five runs. The first column shows the file name followed by the function that corresponds to the entry point for the analysis. Column `LoC` shows the number of WebAssembly instructions that the analysis accesses, column `AN time` is the analysis time in seconds. When `AN time` is -1, then VIVIENNE was not able to successfully analyze the respective implementations, whereas when `AN time` is * for VIVIENNE$_{\text{inv}}$, then this means that the invariant assertion failed for one of the loops. Column 🐞 shows the number of discovered timing vulnerabilities. #FS is the number of formulas during the analysis and next column shows the time in seconds for the simplification step. #SS is the number of formulas that VIVIENNE forwards to the SMT solver, followed by the average number of expressions in each formula, #Exprs, and the solving time `SS time`. #Exprs is the value that decides selecting the *bindings* solver or the *portfolio* solver. In these experiments, for #Expr $\leq$ 1500, VIVIENNE uses the *bindings* solver, otherwise the portfolio solver.

For example, the third entry for WHACL* in Table V shows the results for the analysis of function

`Hacl_Poly1305_32_poly1305_mac` from WHACL* module `poly1305`. VIVIENNE goes through 1440 different WebAssembly instructions, not considering the multiple accesses for loops. The analysis time is 1.55 seconds and the analysis did not discover any timing vulnerabilities, generated 700 formulas that took less than 0.01 seconds to simplify. Of these 700 formulas, 69 where forwarded to the SMT solver, whereas the rest were simple enough for the analysis to infer their result. The average number of expressions in these 69 formulas is 22 expressions and the solving time was less than 0.01 seconds.

A. VIVIENNE$_{unroll}$ and VIVIENNE$_{inv}$ comparison

By comparing Tables IV and V, we notice that the number of queries, #FS, is, in general, larger for VIVIENNE$_{\text{unroll}}$ than VIVIENNE$_{\text{inv}}$. The reason for this is that VIVIENNE$_{\text{unroll}}$ needs to make queries for memory operations and control-flow instructions at every iteration. However, constant-time cryptographic implementations typically use constant memory indexes and often branch on constant values. This means that these queries are simple and in most cases do not require invoking the SMT solver (low #SS). On the other hand, VIVIENNE$_{\text{inv}}$ has lower #FS than VIVIENNE$_{\text{unroll}}$ (in most cases) because of the use of an invariant simplifies the analysis of loops. However, VIVIENNE$_{\text{inv}}$ has increased #SS because first the invariant analysis requires querying the policies of modified variables in the loop that might not be constant values and second, it replaces constant values in `if` statements or memory indexes with symbolic unbound values that increase search space of the formula. In some cases, VIVIENNE$_{\text{inv}}$ has larger #FS than VIVIENNE$_{\text{unroll}}$, like in `br_aes_ct_cbcenc_run` of BearSSL -O3, where #FS=157984 for VIVIENNE$_{\text{inv}}$ and #SS=2793. This is due to path explosion as a result of the invariant-induced overapproximation.

To summarize, we can see three types of complexity sources in our RelSE analysis: 1) the number of loop iterations, 2) the number of execution paths, and 3) the formula complexity (depends often on the memory). VIVIENNE$_{\text{inv}}$ reduces 1) but may increases 2) and 3), whereas VIVIENNE$_{\text{unroll}}$ has higher 1) which may also increase 3), but typically lower 2). Depending on the combined effects of these three complexity sources, either of the two methods may perform better.

APPENDIX B
SMT SOLVER

Our approach uses an SMT solver with two modes, the first uses the Z3 OCaml bindings for reduced communication overhead and the second uses a portfolio solver that runs four solvers in parallel. The analysis selects which SMT solver mode to use depending on the number of expressions in the formula. Table III shows the share of formulas that VIVIENNE passes to the bindings and the portfolio solver. The table shows that for VIVIENNE$_{\text{inv}}$, the analysis passes the majority of the queries (72.3%) to the `Z3 Bindings` solver, which means that the queries from VIVIENNE$_{\text{inv}}$ have relatively low number of expressions, an indication on the complexity and the size of the query. For VIVIENNE$_{\text{unroll}}$, the opposite is true, as the analysis passes the majority of the queries (90.4%) to the `Portfolio Solver`. This means that VIVIENNE$_{\text{unroll}}$ passes to the solver mostly queries that contain a large number of expressions, an indication of complexity.

Solver	VIVIENNE$_{\text{unroll}}$	VIVIENNE$_{\text{inv}}$
Boolector	45.29%	91.94%
Yices 2	54.71%	6.11%
CVC4	0%	0.33%
Z3	0%	1.62%

TABLE II
PORTFOLIO SOLVER STATISTICS

Solver	VIVIENNE$_{\text{unroll}}$	VIVIENNE$_{\text{inv}}$
Z3 Bindings	6.0%	63.3%
Portfolio Solver	94.0%	36.7%

TABLE III
SOLVER STATISTICS

The portfolio solver consists of four solvers, namely Boolector, Yices 2, CVC4, and Z3, that run in parallel and the first to finish reports the result to VIVIENNE. Table II shows the share of answers from each of the four solvers to the queries to the portfolio solver. In Table II, we see that `Z3` and `CVC4` are not able to answer to a large number of queries for either VIVIENNE$_{\text{inv}}$ (3.36% and 0.25%) or VIVIENNE$_{\text{unroll}}$ (0%). For VIVIENNE$_{\text{unroll}}$, `Yices 2` answers the majority, i.e. 79.88% of the queries and `Boolector` answers to 20.12%. For VIVIENNE$_{\text{inv}}$, the opposite is true, namely `Boolector` answers the majority of the queries, i.e. 86.83%, whereas `Yices 2` answers 9.56% of the queries. The difference in the efficiency of the solvers for VIVIENNE$_{\text{inv}}$ and VIVIENNE$_{\text{unroll}}$, depends primarily on the (default) heuristics that they use, which can be beneficial for specific queries. Another parameter that affects the performance of the solvers is the hardware that these solvers run on (see Section IV-A) because the speed of the memory and the processor power may affect the performance of each solver. To summarize, our results show that `Boolector` and `Yices 2` are the best performing solvers in the portfolio, but there is no optimal solver for constant-time analysis of VIVIENNE.

bench/function	LoC	AN time	✱	#FS	FS time	#SS	#Exprs	SS time
CT-wasm								
salsa20/decrypt	515	0.09 ± 0.00	0	602	< 0.01	0		
salsa20/encrypt	512	0.10 ± 0.01	0	602	< 0.01	0		
sha256/transform	372	0.05 ± 0.01	0	926	< 0.01	0		
sha256/update	409	0.18 ± 0.01	0	1312	< 0.01	0		
tea/decrypt	80	< 0.01	0	72	< 0.01	0		
tea/encrypt	80	0.01 ± 0.00	0	72	< 0.01	0		
TweetNaCl								
core_hsalsa20/core_hsalsa20	356	< 0.01	0	46	< 0.01	0		
core_salsa20/core_salsa20	412	0.01 ± 0.00	0	54	< 0.01	0		
poly1305/crypto_onetimeauth	787	0.11 ± 0.00	0	81	< 0.01	0		
WHACL*								
chacha20/Hacl_Chacha20_chacha20_encrypt	1777	669.91 ± 3.53	0	9665	0.07 ± 2.77	0		
curve25519_51/Hacl_Curve25519_51_scalarmult	-1	-1	0	80896	0.07 ± 0.26	0		
poly1305/Hacl_Poly1305_32_poly1305_mac	1440	1.34 ± 0.01	0	829	< 0.01	0		
salsa20/Hacl_Salsa20_salsa20_encrypt	1887	162.86 ± 1.56	0	8596	0.02 ± 0.71	0		
sha256/Hacl_Hash_SHA2_hash_256	1147	1323.51 ± 7.13	0	14512	0.09 ± 4.56	0		
sha512/Hacl_Hash_SHA2_hash_512	1550	456.20 ± 4.14	0	12287	0.04 ± 1.62	0		
BearSSL -O0								
aes_big/br_aes_big_cbcenc_run	2089	13.04 ± 0.11	32	1111	< 0.01	32	3711	0.36 ± 0.37
aes_ct/br_aes_ct_cbcenc_run	4857	46.54 ± 0.76	0	4233	0.01 ± 0.13	0		
des_ct/br_des_ct_cbcenc_run	3841	1560.52 ± 6.80	0	23463	0.07 ± 1.23	0		
des_tab/br_des_tab_cbcenc_run	1920	24.94 ± 0.16	8	3301	0.01 ± 0.05	8	262	< 0.01
BearSSL -O3								
aes_big/br_aes_big_cbcenc_run	791	7.89 ± 0.09	32	218	< 0.01	32	3327	0.22 ± 0.22
aes_ct/br_aes_ct_cbcenc_run	1717	1.69 ± 0.01	0	493	< 0.01	0		
des_ct/br_des_ct_cbcenc_run	993	6.49 ± 0.03	0	952	0.01 ± 0.19	0		
des_tab/br_des_tab_cbcenc_run	581	3.20 ± 0.03	8	381	0.01 ± 0.15	8	262	< 0.01
Libsodium -O0								
aead/crypto_aead_chacha20poly1305_encrypt	7720	369.83 ± 1.33	0	11507	0.03 ± 0.48	16	4	0.04 ± 0.00
auth/crypto_auth_hmacsha256	13913	4856.64 ± 27.94	0	47679	0.10 ± 0.52	0		
chacha20/crypto_stream_chacha20	3313	228.04 ± 1.61	0	8756	0.03 ± 0.51	2	4	0.04 ± 0.00
poly1305/crypto_onetimeauth_poly1305_donna	3685	20.78 ± 0.09	0	1671	0.01 ± 0.07	0		
salsa20/crypto_core_salsa20	1628	11.99 ± 0.04	0	3513	< 0.01	0		
sha256/SHA256_Transform	11692	136.11 ± 0.95	0	8299	0.02 ± 0.06	0		
sha256/crypto_hash_sha256	13225	536.25 ± 3.84	0	18712	0.03 ± 0.11	0		
sha512/crypto_hash_sha512	13351	295.80 ± 3.18	0	12993	0.02 ± 0.08	0		
Libsodium -O3								
aead/crypto_aead_chacha20poly1305_encrypt	1971	45.06 ± 0.29	0	896	0.05 ± 0.67	16	4	0.04 ± 0.00
auth/crypto_auth_hmacsha256	3256	562.00 ± 4.19	0	4559	0.12 ± 5.32	0		
chacha20/crypto_stream_chacha20	956	0.29 ± 0.01	0	253	< 0.01	2	4	0.04 ± 0.00
poly1305/crypto_onetimeauth_poly1305_donna	940	11.20 ± 0.07	0	223	0.05 ± 0.58	0		
salsa20/crypto_core_salsa20	483	0.01 ± 0.00	0	52	< 0.01	0		
sha256/SHA256_Transform	2171	0.01 ± 0.00	0	479	< 0.01	0		
sha256/crypto_hash_sha256	2980	28.06 ± 0.66	0	1643	0.02 ± 0.66	0		
sha512/crypto_hash_sha512	2844	6.20 ± 0.06	0	1344	< 0.01	0		
Almeida -O0								
naive_select/ct_select_u32_naive	49	0.03 ± 0.00	1	9	< 0.01	3	15	< 0.01
select_v1/ct_select_u32_v1	149	< 0.01	0	14	< 0.01	0		
select_v2/ct_select_u32_v2	93	< 0.01	0	10	< 0.01	0		
select_v3/ct_select_u32_v3	70	< 0.01	0	9	< 0.01	0		
select_v4/ct_select_u32_v4	70	< 0.01	0	9	< 0.01	0		
sort/sort3	254	0.18 ± 0.00	1	298	< 0.01	14	68	< 0.01
sort_multiplex/sort3_multiplex	276	0.02 ± 0.00	0	89	< 0.01	0		
sort_negative/sort3_negative	209	0.16 ± 0.01	1	245	< 0.01	14	68	< 0.01
Almeida -O3								
naive_select/ct_select_u32_naive	5	< 0.01	0	0		0		
select_v1/ct_select_u32_v1	5	< 0.01	0	0		0		
select_v2/ct_select_u32_v2	5	< 0.01	0	0		0		
select_v3/ct_select_u32_v3	5	< 0.01	0	0		0		
select_v4/ct_select_u32_v4	5	< 0.01	0	0		0		
sort/sort3	84	0.07 ± 0.01	3	21	< 0.01	3	229	0.02 ± 0.01
sort_multiplex/sort3_multiplex	74	0.10 ± 0.00	3	17	< 0.01	3	229	0.02 ± 0.02
sort_negative/sort3_negative	74	0.09 ± 0.01	3	17	< 0.01	3	229	0.02 ± 0.02
lucky13 -O0								
tls1_cbc_remove_padding_lucky13/tls1_..._lucky13	-1	-1	5	24978	0.01 ± 0.06	4027	35698	0.87 ± 0.60
lucky13 -O3								
tls1_cbc_remove_padding_lucky13/tls1_..._lucky13	133	960.17 ± 15.52	5	3144	< 0.01	3106	3080	0.25 ± 1.03

TABLE IV
EVALUATION RESULTS WITH VIVIENNE$_{\text{UNROLL}}$

bench/function	LoC	AN time	#	#FS	FS time	#SS	#Exprs	SS time
CT-wasm								
salsa20/decrypt	515	38.99 ± 7.31	0	272	< 0.01	160	426	0.23 ± 0.90
salsa20/encrypt	512	57.78 ± 17.51	0	272	< 0.01	160	426	0.35 ± 1.55
sha256/transform	372	1.06 ± 0.03	0	97	< 0.01	36	323	0.02 ± 0.07
sha256/update	409	3.47 ± 0.03	0	123	< 0.01	44	2469	0.06 ± 0.07
tea/decrypt	80	0.17 ± 0.00	0	25	< 0.01	6	99	0.02 ± 0.03
tea/encrypt	80	0.17 ± 0.01	0	25	< 0.01	6	99	0.02 ± 0.03
TweetNaCl								
core_hsalsa20/core_hsalsa20	356	17.28 ± 0.18	0	98	< 0.01	66	291	0.25 ± 0.86
core_salsa20/core_salsa20	412	27.11 ± 5.04	0	106	< 0.01	66	291	0.40 ± 1.71
poly1305/crypto_onetimeauth	787	145.44 ± 0.38	0	116	< 0.01	32	221	4.54 ± 4.55
WHACL*								
chacha20/Hacl_Chacha20_chacha20_encrypt	1777	101.19 ± 0.88	0	2029	0.01 ± 0.46	100	95241	0.73 ± 2.36
curve25519_51/Hacl_Curve25519_51_scalarmult	44234	2007.77 ± 9.08	0	59780	0.03 ± 0.09	5676	80	0.01 ± 0.04
poly1305/Hacl_Poly1305_32_poly1305_mac	1440	1.55 ± 0.01	0	700	< 0.01	69	22	< 0.01
salsa20/Hacl_Salsa20_salsa20_encrypt	1887	230.22 ± 2.83	0	6449	0.03 ± 1.12	311	75631	0.11 ± 0.38
sha256/Hacl_Hash_SHA2_hash_256	1147	4.67 ± 0.05	0	720	< 0.01	197	257	0.01 ± 0.05
sha512/Hacl_Hash_SHA2_hash_512	1550	6.88 ± 0.07	0	832	< 0.01	211	244	0.01 ± 0.07
BearSSL -O0								
aes_big/br_aes_big_cbcenc_run	-1	-1	39	766	< 0.01	146	7296	8.15 ± 10.06
aes_ct/br_aes_ct_cbcenc_run	4857	19.51 ± 0.18	0	4337	< 0.01	50	10	< 0.01
des_ct/br_des_ct_cbcenc_run	-1	-1	14	3630	< 0.01	337	19742	9.07 ± 8.59
des_tab/br_des_tab_cbcenc_run	-1	-1	13	1563	< 0.01	340	12436	6.59 ± 7.45
BearSSL -O3								
aes_big/br_aes_big_cbcenc_run	791	45.45 ± 0.42	32	270	< 0.01	70	3684	0.63 ± 0.94
aes_ct/br_aes_ct_cbcenc_run	-1	-1	9	157984	0.02 ± 0.73	2793	4189	0.32 ± 0.06
des_ct/br_des_ct_cbcenc_run	-1	*	*	256	0.03 ± 0.34	129	3291	0.34 ± 0.27
des_tab/br_des_tab_cbcenc_run	-1	*	*	180	< 0.01	83	7209	0.83 ± 1.75
Libsodium -O0								
aead/crypto_aead_chacha20poly1305_encrypt	-1	-1	3	5207	0.02 ± 0.28	13	207810	5.11 ± 9.05
auth/crypto_auth_hmacsha256	-1	-1	74	473	0.01 ± 0.02	133	113452	13.01 ± 16.88
chacha20/crypto_stream_chacha20	3313	231.17 ± 3.25	0	8756	0.03 ± 0.51	2	4	0.04 ± 0.00
poly1305/crypto_onetimeauth_poly1305_donna	-1	-1	52	1623	0.01 ± 0.08	17	90626	39.19 ± 112.16
salsa20/crypto_core_salsa20	1628	13.58 ± 0.12	0	3513	< 0.01	0		
sha256/SHA256_Transform	-1	-1	102	410	0.01 ± 0.03	29	100360	9.32 ± 11.75
sha256/crypto_hash_sha256	-1	-1	110	541	0.01 ± 0.03	67	110077	14.73 ± 15.69
sha512/crypto_hash_sha512	-1	-1	6	270	0.01 ± 0.02	86	109908	0.99 ± 0.41
Libsodium -O3								
aead/crypto_aead_chacha20poly1305_encrypt	-1	*	*	376	0.04 ± 0.45	48	330717	6.63 ± 28.03
auth/crypto_auth_hmacsha256	-1	-1	3	669	0.03 ± 0.75	52	107103	1.49 ± 1.77
chacha20/crypto_stream_chacha20	956	0.31 ± 0.01	0	253	< 0.01	2	4	0.05 ± 0.00
poly1305/crypto_onetimeauth_poly1305_donna	-1	-1	6	326	0.03 ± 0.17	87	59566	16.92 ± 36.65
salsa20/crypto_core_salsa20	483	15.87 ± 0.06	0	106	< 0.01	66	291	0.23 ± 0.46
sha256/SHA256_Transform	2171	0.01 ± 0.00	0	479	< 0.01	0		
sha256/crypto_hash_sha256	-1	-1	0	632	0.04 ± 0.78	34	34559	0.27 ± 0.84
sha512/crypto_hash_sha512	-1	-1	4	68	0.01 ± 0.04	20	90629	0.62 ± 0.41
Almeida -O0								
naive_select/ct_select_u32_naive								
select_v1/ct_select_u32_v1								
select_v2/ct_select_u32_v2								
select_v3/ct_select_u32_v3					No loops			
select_v4/ct_select_u32_v4								
sort/sort3								
sort_multiplex/sort3_multiplex								
sort_negative/sort3_negative								
Almeida -O3								
naive_select/ct_select_u32_naive								
select_v1/ct_select_u32_v1								
select_v2/ct_select_u32_v2								
select_v3/ct_select_u32_v3					No loops			
select_v4/ct_select_u32_v4								
sort/sort3								
sort_multiplex/sort3_multiplex								
sort_negative/sort3_negative								
lucky13 -O0								
tls1_cbc_remove_padding_lucky13/tls1_..._lucky13	575	9.83 ± 0.04	5	539	< 0.01	217	701	0.03 ± 0.02
lucky13 -O3								
tls1_cbc_remove_padding_lucky13/tls1_..._lucky13	-1	*	*	94	< 0.01	63	472	0.03 ± 0.03

TABLE V
EVALUATION RESULTS WITH VIVIENNE$_{\text{INV}}$

Appendix D

Publication 4

Securing Optimized Code Against Power Side Channels

Rodothea Myrsini Tsoupidi
Royal Institute of Technology KTH
Stockholm, Sweden
tsoupidi@kth.se

Roberto Castañeda Lozano
Independent Researcher
Stockholm, Sweden
rcas@acm.org

Elena Troubitsyna
Royal Institute of Technology KTH
Stockholm, Sweden
elenatro@kth.se

Panagiotis Papadimitratos
Royal Institute of Technology KTH
Stockholm, Sweden
papadim@kth.se

***Abstract*—Side-channel attacks impose a serious threat to cryptographic algorithms, including widely employed ones, such as AES and RSA. These attacks take advantage of the algorithm implementation in hardware or software to extract secret information via side channels. Software masking is a mitigation approach against power side-channel attacks aiming at hiding the secret-revealing dependencies from the power footprint of a vulnerable implementation. However, this type of software mitigation often depends on general-purpose compilers, which do not preserve non-functional properties. Moreover, microarchitectural features, such as the memory bus and register reuse, may also leak secret information. These abstractions are not visible at the high-level implementation of the program. Instead, they are decided at compile time. To remedy these problems, security engineers often sacrifice code efficiency by turning off compiler optimization and/or performing local, post-compilation transformations. This paper proposes Secure by Construction Code Generation (SecCG), a constraint-based compiler approach that generates optimized yet protected against power side channels code. SecCG controls the quality of the mitigated program by efficiently searching the best possible low-level implementation according to a processor cost model. In our experiments with twelve masked cryptographic functions up to 100 lines of code on Mips32 and ARM Thumb, SecCG speeds up the generated code from 77% to 6.6 times compared to non-optimized secure code with an overhead of up to 13% compared to non-secure optimized code at the expense of a high compilation cost. For security and compiler researchers, this paper proposes a formal model to generate power side channel free low-level code. For software engineers, SecCG provides a practical approach to optimize performance critical and vulnerable cryptographic implementations that preserves security properties against power side channels.**

***Index Terms*—compilation, power side-channel attacks, code optimization, software masking, constraint programming**

I. Introduction

Cryptographic algorithms, symmetric/shared key or asymmetric/private key ones, rely on safeguarding the shared secret key or the private key, respectively. The exposure of these keys to unintended users compromises the security of these algorithms. Unfortunately, the software implementation of cryptographic algorithms may reveal information about their secret/private keys [1]. In particular, the attacker may observe what is termed *side-channel information*, notably observing the execution time [1] or the power consumption [2, 3], during the execution of the algorithm to extract information about the secret keys. These attacks are attractive especially because usually they do not require expensive equipment. This paper focuses on Power Side Channel (PSC) attacks.

Software masking is a widely-used approach to mitigate PSC attacks [4, 5], hiding secret information by splitting a secret into n randomized shares. The attacker has to retrieve all shares in order to acquire the secret value. While software masking can be an effective mitigation, compiler code generation may optimize it away. Moreover, Transition-Based Leakage (TBL) sources, such as register reuse or memory-access order, are decided at compile time by low-level compiler transformations [6, 7, 8].

To mitigate these compiler-induced power side-channel leaks at the binary level there are techniques based on compilation [7, 9, 10] and binary rewriting with hardware emulation [11, 12, 13]. All these approaches mitigate compiler-generated leakages using local transformations [13, 7, 11]. The methods that depend on hardware emulation are typically accurate but may introduce significant overhead [11] and are hardware specific. For example, Rosita [11], an emulation-based approach, propose a mitigation that introduces an overhead ranging from 21% to 64% for ARM Cortex M0. Wang et al. [7] perform their mitigation using a standard compiler with no high-level optimizations (-O0). This is a common practice for security research to ensure the absence of compiler-induced mitigation invalidation [6, 14]. However, unoptimized code is highly inefficient, and may even introduce additional leaks due to the heavy use of the program stack, as discussed in Section II.

Vu et al. present an approach that enables secure optimization of masked code at a higher level [14, 15]. This approach applies high-level compiler optimizations by disallowing secure-code removal and operand reordering (due to associativity of some operations) and are able to generate correctly masked code. However, they do not deal with TBLs.

Currently, the state-of-the-art approaches are unable to gen-

erate code that is both efficient and secure in the face of TBLs that enable PSC attacks. To address this challenge, this paper proposes Secure by Construction Code Generation (SecCG), an optimizing compiler approach that provably preserves security properties against PSC. At the middle-end, SecCG handles code generated using *register promotion* (promoting program variables from memory to registers) as a high-level optimization. Then, SecCG uses a constraint-based method to generate code that is secure against PSC attacks. SecCG controls the quality of the mitigated program by efficiently searching the best possible low-level implementation according to a processor cost model [16]. The security model of SecCG is hardware agnostic and can be extended with additional architectural constraints. SecCG is suitable for predictable architectures with no advanced microarchitectural features, such as caches or speculative execution. In our experiments with twelve masked implementations on Mips32 and ARM Thumb, SecCG improves the execution time of the generated code from 77% to a speedup of 6.6 compared to non-optimized code at a overhead of up to 13% compared to non-secure optimized code. This comes at a cost on compilation time and scalability, where SecCG optimizes successfully programs up to 100 lines of code. In summary, this paper makes the following contributions:

- a compiler approach to generate TBL-free, low-overhead assembly code for high-level software-masked programs;
- a constraint model for optimized and PSC-secure code generation;
- a proof that the constraint model guarantees the generation of secure code for a non-trivial leakage model; and
- experimental results on two architectures showing that the performance overhead of our mitigation is low and its efficiency benefits are significant, compared to current approaches.

II. Motivating Example

To motivate our approach, let us consider an example of a first-order masked implementation. First-order masking splits a secret value `k` into two shares, (`m`, `mk`), where `m` is a uniformly distributed random variable sampled at every execution of the algorithm; `mk` = `m`$\oplus$`k` is also uniformly distributed ($\oplus$ denotes the exclusive OR operation). Fig. 1 shows a first-order masked C implementation of exclusive OR, where `key` is a secret value (red), `mask` is a uniformly random variable (brown), and `pub` is a non-secret value (green). At line 2, the algorithm creates the second share, `mk`, and at line 3, it performs the exclusive OR operation with the secret-independent value, `pub`. At a high-level, the code of Fig. 1 is secure against power side channels but a binary implementation generated by a standard, security-unaware compiler may leak information about `key`. For example, hardware-register reuse and memory-bus access order may reveal secret information [7, 11, 6, 8]. These TBLs are a result of transitional effects, i.e., the power effect of bits switching between one and zero and vice versa.

Fig. 3a shows the ARM Thumb assembly code generated by the standard compiler LLVM [17] for the C code in Fig. 1. The first three `str` instructions store the function arguments that reside in registers `r0-r2` to the stack (lines 3-5). Line 6 loads (`ldr`) the value of `rand` from the stack into register `r1`. Line 7 performs the first exclusive OR (line 2 in Fig. 1) between registers `r1` and `r2` (`key`) and stores the result in register `r1`. Here, there is a transition for register `r1` from value `mask` to `mk`, which leaks the secret `key` (marked code at line 7). Line 8 stores the content of `r1` to the stack and the value of the memory bus that contains the `mask` at line 6 transitions to `mk`. This leads to another leak due to the transitional effect in the memory bus (marked code at lines 6 and 8). The rest of the code performs the second exclusive OR (line 10) and stores the final result on the stack (line 11).

Fig. 3b shows the mitigation produced by the security backend of SecCG that eliminates leakages that appear in the LLVM unoptimized code. The mitigation is based on *instruction scheduling* and *register allocation* transformations. In particular, changing the order of operands at line 7 results in a transition from `sec` to `mk` that leaks the value of `mask`, which is not secret (marked code at line 7). Changing the order of the instructions hides the memory-bus leakage. More specifically, because there are no data dependencies between lines 3-6, the `ldr` instruction that causes the leak in Fig. 3a may be scheduled earlier (line 4 in Fig. 3b). Then, another memory instruction that stores the secret value in memory (line 6 in Fig. 3b) is scheduled just before the store instruction at line 8. This causes a transition from `sec` to `mk` in the memory bus that leaks the value of `mask` (marked code at lines 6 and 8). These transformations are global, considering possible available memory instructions and register assignments to mitigate transitional leakages in the whole program and may (as in Fig. 3b) introduce no overhead.

However, unoptimized code leads to poor performance. In general, compiler optimizations may invalidate high-level software mitigations [14]. Fortunately, this is not the case for register promotion (*mem2reg* in LLVM), a simple high-level optimization that enables efficient register allocation by promoting program variables from memory to registers. This transformation replaces stack operations to register operations and preserves the operand order. In particular, aggressive optimizations (-O1 to -O3 in LLVM) may take advantage of the associativity property of $\oplus$ to change the order of the operands, converting (`mask` $\oplus$ `sec`) $\oplus$ `pub` to `mask` $\oplus$ (`sec` $\oplus$ `pub`), which invalidates masking. Equipped with improved high-level code, the SecCG backend optimizes low-level transformations and generates optimized code. Fig.s 2a and 2b show the code of Fig. 1 compiled with register promotion. Fig. 2a leaks the same secret information as Fig. 3a due to register reuse, namely the first exclusive OR operation `eors`, but contains no memory-bus secret leak. To mitigate the register-reuse leak at line 2, SecCG changes the order of the arguments and the result is now stored in register `r2`.

As we see in Fig. 3a, unoptimized code may introduce additional leaks due to the heavy use of the program stack. Instead, SecCG uses register promotion to remove unnecessary memory accesses that may cause additional leaks. Then,

```
1  u32 Xor(u32 pub, u32 mask, u32 key) {
2    u32 mk = mask ^ key;
3    u32 t = pub ^ mk;
4    return t;
5  }
```

Fig. 1: Masked exclusive OR implementation in C

```
1  @ r0: pub, r1: mask, r2: key
2    eors  r1, r2
3    eors  r0, r1
4    ...
```

(a) Insecure (LLVM)

```
1  @ r0: pub, r1: mask, r2: key
2    eors  r2, r1
3    eors  r0, r2
4    ...
```

(b) Secure (SecCG)

Fig. 2: Compilation of function `Xor` applying register promotion

```
1  @ r0: pub, r1: mask, r2: key
2  ...
3  str     r0, [sp, #16] @ mem: pub
4  str     r1, [sp, #12] @ mem: rand
5  str     r2, [sp, #8]  @ mem: key
6  ldr     r1, [sp, #12] @ mem: rand
7  eors    r1, r2
   @ proc: rand <- rand ^ key
8  str     r1, [sp, #4]  @ mem: rand ^ key
9  ldr     r0, [sp, #16] @ mem: pub
10 eors    r0, r1
   @ proc: pub <- pub ^ rand ^ key
11 str     r0, [sp]
   @ mem: pub ^ rand ^ key
12 ...
```

(a) Insecure (LLVM)

```
1  @ r0: pub, r1: mask, r2: key
2  ...
3  str     r1, [sp, #12] @ mem: rand
4  ldr     r1, [sp, #12] @ mem: rand
5  str     r0, [sp, #16] @ mem: pub
6  str     r2, [sp, #8]  @ mem: key
7  eors    r2, r1
   @ proc: key <- sec ^ rand
8  str     r2, [sp, #4]  @ mem: key ^ rand
9  ldr     r0, [sp, #16] @ mem: pub
10 eors    r0, r2
   @ proc: pub <- pub ^ key ^ rand
11 str     r0, [sp]
   @ mem: pub ^ rand ^ key
12 ...
```

(b) Secure (SecCGwith no register promotion)

Fig. 3: Compilation of function `Xor` with no optimizations

SecCG's backend generates low-level optimized code that does not expose secret information through transitional leakages and does not introduce significant overhead compared to non-secure code.

III. Threat Model and Modeling Background

This section describes the Hamming Distance (HD) model (Section III-A), the threat model (Section III-B), an HD-based type-inference algorithm (Section III-C), a constraint-based compiler backend model (Section III-D), and the running example for the constraint-based compiler backend (Section III-E),

A. Hamming-Distance Model

The Hamming Weight (HW) model [18, 2, 19] corresponds to the number of active bits in a data word. We assume the following encoding of the binary data, $d = \sum_{i=0}^{N-1} 2^i d_i$, where d_i is one if the i_{th} bit of an N-bit word is set and zero otherwise. The HW of this data is the number of bits that are set: $HW(d) = \sum_{i=0}^{N-1} d_i$. The HD leakage model assumes that the observed leakage when flipping the bits of a memory element from a value d_1 to a value d_2 is $HW(d_1 \oplus d_2)$, where $\oplus$ denotes the exclusive OR operation. If one of the values d_1 is a uniform random variable, then $d_1 \oplus d_2$ is also a uniform random variable and $HW(d_1 \oplus d_2)$ has the same mean and variance as $HW(d_1)$ [19]. This means that by masking (exclusive bitwise OR) a secret value k with a uniform random variable m, the HD of the new variable has the same mean and variance as m. In this way, masking hides the information of k from the power consumption traces.

We assume a program $P(\mathbf{IN}) = i_1; i_2; ..., i_n$ that takes as input a set of variables **IN** and consists of a sequence of n instructions i_j. We assume that the program has a leakage at every execution step when there is bit flipping in the hardware registers or the memory bus. We will use the terms by Papagiannopoulos and Veshchikov [13] and refer to the hardware-register transition leakage as Register-Overwrite Transition (ROT) and the memory-bus transition leakage as Memory-Remnant Effect (MRE). For MRE, we assume that both read and write operations make use of the same memory bus and that the source of the leakage is the transitional effect when writing the data to the memory bus. In our model, the memory address of the operations does not affect the leakage.

We represent the leakage as a set of observations in the power trace. To calculate the observed leakage $L(P(IN))$ for an instance IN of the input variables, we use the HD leakage model. We write $P = P'; i_n$ to denote a program $P = i_1; i_2; ...; i_{n-1}; i_n$, with a prefix $P' = i_1; i_2; ...; i_{n-1}$ (IN is omitted for simplicity). Equations 1-4 present a recursive definition of the leakage model, where for every point in the execution trace, the attacker observes the HW of any ROT or MRE transitions. In the formulas, an expression e is $e ::= r \mid v \mid \texttt{bop}(e_1, e_2) \mid \texttt{uop}(e_1) \mid \texttt{mem}(e_a)$, where r is a register, v is a constant value, `bop` is a binary operation, `uop` is a unary operation, and $\texttt{mem}(e_a)$ is a memory load operation that loads data from address e_a. An instruction is $i = r \leftarrow e \mid \texttt{mem}(e_a, e)$, where $r \leftarrow e$ denotes that an expression is assigned to register r, and $\texttt{mem}(e_a, e)$ is a store memory operation that stores data e at memory address e_a. To simplify the leakage equations, we transform the load operation from $r \leftarrow \texttt{mem}(e_a)$ to a

$$L(P'; r \leftarrow e_2; P''; r \leftarrow e_1) = L(P'; r \leftarrow e_2; P'') \cup \{HW(e_1 \oplus e_2)\}, \nexists i \in P''.\ i = r \leftarrow e_3 \quad (1)$$

$$L(P'; i_1; P''; \texttt{mem}(e_b, e_2)) = L(P'; i_1; P'') \cup \{HW(e_1 \oplus e_2)\}, (i_1 = \texttt{mem}(e_a, e_1)) \ \wedge \ \nexists i \in P''.\ i = \texttt{mem}(e_c, e_3) \quad (2)$$

$$L(P'; r \leftarrow e) = L(P') \cup \{HW(e \oplus r_{IN})\}, \nexists i \in P'.\ i = r \leftarrow e_3 \quad (3)$$

$$L(P'; \texttt{mem}(e_a, e_1)) = L(P') \cup \{HW(e_1)\}, \nexists i \in P'.\ i = \texttt{mem}(e_b, e_3) \quad (4)$$

sequence $\texttt{mem}(e_a, v_{\texttt{mem}(e_a)}); r \leftarrow v_{\texttt{mem}(e_a)}$, where $v_{\texttt{mem}(e_a)}$ is the value in memory at address e_a. Equation 1 describes the leakage when two instructions write the value of their result to the same register and no other instruction between them writes to the same register. Note that the first equation deals also with instructions in the form $r_1 \leftarrow \texttt{bop}(r_2, r_3)$, where `bop` is a binary operation and $r_1 = r_2$. These two-address instructions are common in ARM Thumb and x86 architectures. Equation 2 describes the memory-bus leakage of a memory instruction that writes a value to the memory, given that another memory instructions precedes this memory instruction. Equation 3 describes the leakage of the first instruction that writes to register r. In this case, the leakage is equal to the HD between the new value and the initial value in register r, r_{IN}. Similarly, Equation 4 describes the leakage of the first memory operation. Here, we assume that the initial memory-bus content, mb_{IN}, is a constant value. For example, after executing the last instruction of program $P = r_1 \leftarrow v_1; \texttt{mem}(v_a, v_2); r_1 \leftarrow v_3; \texttt{mem}(v_b, r_1)$, the leakage is equal to $L(P) \overset{Eq.2}{=\!=} L(r_1 \leftarrow v_1; \texttt{mem}(v_a, v_2); r_1 \leftarrow v_3) \cup \{HW(v_3 \oplus v_2)\} \overset{Eq.1}{=\!=} L(r_1 \leftarrow v_1; \texttt{mem}(v_a, v_2)) \cup \{HW(v_3 \oplus v_2), HW(v_3 \oplus v_1)\} \overset{Eq.4}{=\!=} L(r_1 \leftarrow v_1) \cup \{HW(v_3 \oplus v_2), HW(v_3 \oplus v_1), HW(v_2)\} \overset{Eq.3}{=\!=} \{HW(v_3 \oplus v_2), HW(v_3 \oplus v_1), HW(v_2), HW(v_1 \oplus r_{1,IN})\}$, where $r_{1,IN}$ is the initial value of register r_1.

Here, we consider that a program is a straight-line function. Additional checks at the call site are necessary for ensuring the absence of leakage during function calls, for example to make sure that the initial memory-bus value is constant.

B. Threat Model

We assume that the software runs on an non-speculative hardware architecture. The attacker has access to the software implementation and the *public* data but not the *secret* data. The goal of the attacker is to extract information about the secret data by measuring the power consumption of the device that the code runs on. The attacker may accumulate a number of traces from multiple runs of the program and perform statistical analysis, such as Differential Power Analysis (DPA) [2]. At every execution, new *random* values are generated and the attacker has no knowledge of the values of these variables. Our goal is to eliminate any statistical dependencies between the secret data and the measured power traces.

We assume that input variables are `Secret`, `Public`, or `Random`. `Secret` variables contain sensitive values (e.g. cryptographic keys), which the attacker wants to retrieve information about. `Public` variables contain values that the attacker knows or may learn without causing a leakage. Finally, `Random` variables follow the uniform distribution in the domain of the corresponding program variable. We define the *Leakage Equivalence* security condition for the generated programs as follows:

Definition 1 (Leakage Equivalence)**.** *Given a program $P(\boldsymbol{IN})$ that has a set of secret input variables, $\boldsymbol{IN}_{sec} \subseteq \boldsymbol{IN}$, a set of random input variables, $\boldsymbol{IN}_{rand} \subseteq \boldsymbol{IN}$, and a set of public input variables, $\boldsymbol{IN}_{pub} \subseteq \boldsymbol{IN}$. We assume two instances of the input variables, IN and IN'. These two instances differ with regards to the set of secret variables IN_{sec} and IN'_{sec}, i.e. for all public variables, $\forall v \in IN_{pub}$ and $\forall v' \in IN'_{pub}$ we have $v = v'$. Let $r \in IN_{rand}$ and $r' \in IN'_{rand}$ be sampled from a uniform random distribution. Let $L_p = L(P(IN))$ and $L'_p = L(P(IN'))$. Then, we say that a program is leakage equivalent if the distributions of the leakage of the two executions do not differ, i.e.*

$$\sum_{l \in L_p} \mathbb{E}[l] = \sum_{l' \in L'_p} \mathbb{E}[l'] \ \wedge \ \sum_{l \in L_p} Var(l) = \sum_{l' \in L'_p} Var(l'),$$

where $\mathbb{E}[l]$ and $Var(l)$ are l's expected value and variance.

C. HD-based Vulnerability Detection

In our approach, we need a technique to identify whether two values result in a ROT or and MRE leak. There are different ways to identify whether there is a leak at some part of the code. One approach is to use symbolic execution [6, 8]. Symbolic execution executes different paths of a program symbolically and verifies or invalidates specific properties with the help of Satisfiability Modulo Theory (SMT) solvers. Symbolic execution is accurate but has scalability issues when the number of problem variables or program paths increases. On the other end, type-based approaches [20, 7] are typically efficient but at the price of accuracy. In particular, Wang et al. consider a hierarchy of three types based on the properties of the distribution they follow: *uniformly random distribution*, *secret independent distribution*, or finally *unknown distribution*. We call these, `Random`, `Public`, and `Secret`, respectively. The type-inference algorithm assigns a type to each program variable. To infer the program variable types, Wang et al. define a logic model and solve it using an SMT solver. The complexity of this approach is low compared to symbolic execution, at the price of lower accuracy. However, the accuracy is sufficient for loop-free, linearized programs, a format to which many masked and cryptographic implementations can

be transformed [7]. Because of this, our approach adapts the aforementioned type-inference analysis, with some accuracy improvements (see supplementary material [21]).

D. Constraint-based Compiler Backend

A compiler backend performs three main low-level transformations to generate low-level code: instruction selection, instruction scheduling, and register allocation. A combinatorial compiler backend [16, 22, 23] uses combinatorial solving techniques to optimize software using the aforementioned transformations. Different approaches may implement one or more low-level transformations. This section focuses on Constraint Programming (CP) [24] as a combinatorial solving technique.

1) Constraint Model: The constraint-based compiler backend generates a constraint model that captures the program semantics, the low-level compiler transformations, and the hardware architecture. This paper focuses on two compiler transformations, register allocation and instruction scheduling, that are crucial for our mitigation.

Compilers typically model the code using an unbounded number of *virtual* registers until the register allocation stage. Register allocation assigns each virtual register to a hardware register, when possible, or a memory slot on the stack (*spill*), otherwise. The latter has a negative effect on code efficiency. Therefore, register allocation transformations attempt to minimize this effect, while conforming to constraints, such as the number or hardware registers and the calling conventions.

Instruction scheduling decides on the order of the instructions in a program. A valid instruction schedule satisfies the data dependencies among instructions and the processor resource constraints.

A constraint-based compiler backend may be modeled as a Constraint Optimization Problem (COP), $P = \langle V, U, C, O \rangle$, where V is the set of decision variables of the problem, U is the domain of these variables, C is the set of constraints among the variables, and O is the objective function. A constraint-based backend aims at minimizing O, which typically models the code's execution time or size.

A program is modeled as a set of basic blocks B, pieces of code with no branches apart from the exit. Each block contains a number of optional operations, $o \in \mathit{Operations}$, that may be *active* or not. Ins_o denotes the set of hardware instructions that implement operation o. Each operation includes a number of operands $p \in \mathit{Operands}$, each of which may be implemented by different, equally-valued temporaries, $t \in \mathit{Temps}$. Temporaries are either not live or assigned to a register (hardware register or the stack).

Fig. 4 shows a simplified version of the constraint-based compiler backend model for Fig. 1. Temporaries `t0`, `t1`, and `t2` contain the input arguments `pub`, `mask`, and `key`, respectively. Copy operations (`o2, o3, o4, o6, o8`) enable copying program values from one register to another (or to the stack) and are critical for providing flexibility in register allocation. For example, `o2`, allows the copy of the value `pub` from `t0` to `t3`. In the final solution, a copy operation may not be active (shown by the dash in the set of instructions: `[ -, copy]`). The two `xor` operations (`o5, o7`) take two operands each, and each of these operands may use different but equally-valued temporary variables, e.g. `t1` and `t4`.

```
o1: in [t0 ← pub, t1 ← mask, t2 ← key]
o2: t3 ← [-, copy] t0
o3: t4 ← [-, copy] t1
o4: t5 ← [-, copy] t2
o5: t6 ← xor [t1,t4] [t2,t5]
o6: t7 ← [-, copy] t6
o7: t8 ← xor [t0,t3] [t6,t7]
o8: t9 ← [-, copy] t8
o9: out [t10 ← [t8,t9]]
```

Fig. 4: Simplified model of the function in Fig. 1

Fig. 5 shows a valid solution to the register allocation of the constraint model in Fig. 4. All copy operations are deactivated and `t0`, `t1`, and `t2` are assigned to registers `R0`, `R1`, `R2`. Temporary `t6` is assigned to `R1` and temporary `t8` is assigned to `R0`. This register assignment is problematic because it induces a transition in register `R1` from the initial value that holds the `mask` to the masked value `mask` $\oplus$ `key`, which leads to a leakage $L(\texttt{R1} \leftarrow \texttt{R1} \oplus \texttt{R2}; \texttt{R0} \leftarrow \texttt{R0} \oplus \texttt{R1}) \overset{Eq.3}{=\!=} L(\texttt{R1} \leftarrow \texttt{R1} \oplus \texttt{R2}) \cup \{HW(\texttt{pub} \oplus (\texttt{pub} \oplus \texttt{mask} \oplus \texttt{key}))\} \overset{Eq.3}{=\!=} \{HW(\texttt{mask} \oplus (\texttt{mask} \oplus \texttt{key})), HW(\texttt{mask} \oplus \texttt{key})\} = \{HW(\texttt{key}), HW(\texttt{key} \oplus \texttt{mask})\}$. The first element of the leakage reveals information about `key`.

The model of instruction scheduling assigns issue cycles to each operation. This assignment imposes an ordering of the operation and is constrained by the program semantics. For example, in Fig. 4, scheduling `o6` before `o5` is not allowed because `o6` depends on `o5` but scheduling `o4` before `o3` is possible. In Fig. 3b, the store instruction at line 6 (that corresponds to line 5 in Fig. 3a) is scheduled after the load instruction at line 4 (line 6 in Fig. 3a). This is allowed because there is no data dependency between these two instructions.

```
o1: in [t0:R0, t1:R1, t2:R2]
o5: t6:R1 ← xor t1:R1 t2:R2
o7: t8:R0 ← xor t0:R0 t6:R1
o9: out [t10:R0]
```

Fig. 5: Solution of the model in Fig. 4

The decision variables of the constraint problem are:

- $r(t) \in \mathit{Regs}_t,\ t \in \mathit{Temps}$ denotes the hardware register or stack slot assigned to temporary t;
- $a(o) \in [\texttt{false}, \texttt{true}],\ o \in \mathit{Operations}$ denotes whether operation o is active or not;
- $i(o) \in \mathit{Ins}_o,\ o \in \mathit{Operations}$ is the instruction that implements operation o;
- $c(o) \in [0, \mathit{max}c],\ o \in \mathit{Operations}$ is the cycle at which an operation o is scheduled, bounded by $\mathit{max}c$, a conservative upper bound of the execution time;

- $y(p) \in \mathit{Temps}_p,\ p \in \mathit{Operands}$ is the selected temporary among all possible temporaries for operand p.

In addition to these, $l(t) \in [\texttt{false}, \texttt{true}],\ t \in \mathit{Temps}$ represents whether a temporary is live or not, $ls(t) \in [0, maxc],\ t \in \mathit{Temps}$ represents the cycle at which t becomes live, and $le(t) \in [0, maxc],\ t \in \mathit{Temps}$ represents the last cycle at which t is live. An important constraint of register allocation is that the register live ranges of a specific hardware register r_i do not overlap:

$$\forall t_1, t_2 \in \mathit{Temps}\ .\ l(t_1)\ \wedge\ l(t_2)\ \wedge\ r(t_1) = r(t_2)\ \Longrightarrow\ ls(t_1) \geq le(t_2)\ \vee\ ls(t_2) \geq le(t_1). \quad (5)$$

Moreover, when a temporary is live, its last live cycle (le) is strictly greater than its live start (ls):

$$\forall t \in \mathit{Temps}\ .\ l(t)\ \Longrightarrow\ ls(t) < le(t). \quad (6)$$

2) Objective Function: A typical objective function of a constraint-based backend minimizes different metrics such as *code size* and *execution time*. These can be captured in a generic objective function that sums up the weighted cost of each basic block:

$$\sum_{b \in B} \mathit{weight}(b) \cdot \mathbf{cost}(b).$$

The **cost** of each basic block consists of the cost of the specific implementation and is a variable, whereas *weight* is a constant value that represents the contribution of the specific basic block to the total cost. This cost model is accurate for simple hardware architectures. However, in the presence of advance microarchitectural features, such as complex cache hierarchy, branch prediction, and/or out-of-order execution, the cost model is not accurate.

E. Example in a Constraint-based Compiler Backend

Low-level transformations, like register allocation and instruction scheduling, affect the security of programs. Fig. 6a shows the high-level masked implementation of exclusive OR in C (same as Fig. 1). The code takes three inputs: `p` (a `Public` value), `k` (a `Secret` value), and `m` (a `Random` variable). The code computes first the exclusive OR of `m` and `k` and stores it in `mk`. Then, it computes the exclusive OR of `mk` with `p` and stores it in `rs`, which the function returns.

Fig. 6b shows a register allocation of function `Xor` that leads to a HD vulnerability. Both `m` and `mk` are stored in the same register, hence the content of `mk` replaces the previous value `m` in register `R1`. According to the leakage model, the attacker observes the exclusive OR between the initial and updated value of a hardware register. Using the register allocation of Fig. 6b, the leakage reveals information about the secret: $\mathrm{HW}(\mathtt{mk} \oplus \mathtt{m}) = \mathrm{HW}((\mathtt{m} \oplus \mathtt{k}) \oplus \mathtt{m}) = \mathrm{HW}(\mathtt{k})$. Value `k` is a secret value, and thus, a leak occurs (circled in Fig. 6b).

A constraint-based compiler backend is able to generate all legal register allocations for a program. Fig. 6c shows an alternative register allocation for function `Xor`. Here, the result of `mk` is written in hardware register `R2`, giving a HD leakage $\mathrm{HW}(\mathtt{mk} \oplus \mathtt{k}) = \mathrm{HW}((\mathtt{m} \oplus \mathtt{k}) \oplus \mathtt{k}) = \mathrm{HW}(\mathtt{m})$. The leakage here corresponds to the value of `m`, which is not a sensitive value. In a similar way, instruction scheduling may be able to remove leakages as seen in Fig. 3. By changing the schedule of the instructions, the model is often able to generate a PSC-free solution with no code quality overhead.

This example shows that low-level transformations can be responsible for the introduction of HD vulnerabilities and have thus to be taken into account to provide effective mitigations.

IV. SecCG

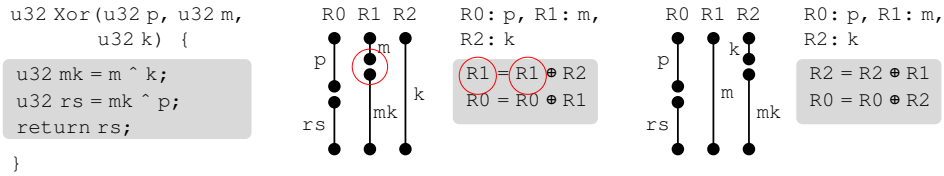
This section introduces SecCG, an approach to optimize code that is secure against PSC attacks. Fig. 7 shows the high-level view of SecCG. SecCG is a constraint-based optimizing secure compiler, i.e. it extends a constraint-based compiler backend with security constraints. It takes two inputs: 1) a C or C++ program, and 2) a security policy denoting which variables are `Secret`, `Random`, or `Public`. SecCG enables *register promotion* at the compiler middle end because this optimization preserves the high-level properties of the program and, at the same time, creates substantial opportunities for register allocation. Then, the constraint-based compiler backend, extended with security constraints, takes as input the program in a machine-level Intermediate Representation (IR) and the security policy. Next, SecCG performs a security analysis (see Section III-C). The results are used to impose constraints that prevent HD vulnerabilities. Given the secure model, the approach generates an optimized solution.

Section IV-A presents the security analysis. Section IV-B presents the secure constraint model that extends the constraint-based compiler backend. Finally, Section IV-C presents the solving enhancements of SecCG.

A. Security Analysis

SecCG performs a security analysis to extract the security types of each program variable and, subsequently, generates constraints that prohibit insecure low-level implementations. The security analysis identifies the security type (`Random`, `Public`, or `Secret`) of each intermediate variable. In the compiler constraint model, the program variables correspond to the input arguments, the operands and the result of each operation. This is equivalent to the temporary variables, i.e. the virtual registers. Each operand can use a number of alternative temporary values $t \in \mathit{Temps}$ and each temporary value is assigned to a register (see Section III-D). The type-inference rules do not handle loops or conditional statements. However, cryptographic implementations that are free from PSCs are often linearizable [7].

The security analysis uses a type-inference algorithm based on Wang et al. [7]. We extend this algorithm with additional definitions that improve the accuracy of the type inference (see supplementary material [21]). In particular, we extend the type-inference algorithm with rules that consider additional properties of GF(2^n), like distributivity between exclusive or ($\oplus$) and multiplication in GF(2^n) ($\odot$). At the end of the analysis, all temporary variables have an inferred type. Fig. 8 shows the inferred security types for each of the temporaries



(a) Exclusive OR in C (b) Vulnerable register assignment (c) Secure register assignment

Fig. 6: The exclusive OR example, illustrating a HD vulnerability and alternative register assignments

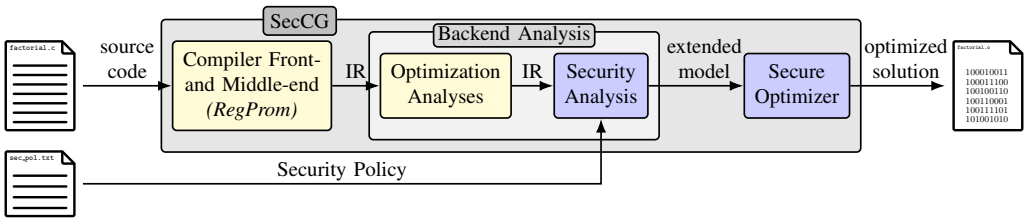


Fig. 7: High-level view of SecCG

in our running example. Temporaries `t0` and `t3` are `Public` (green), `t2` and `t5` are `Secret` (red), and `t1`, `t4` and `t6-t10` are `Random` (brown).

```
o1: in [t0:Public, t1:Random, t2:Secret]
o2: t3:Public ← [-, copy] t0
o3: t4:Random ← [-, copy] t1
o4: t5:Secret ← [-, copy] t2
o5: t6:Random ← xor [t1,t4] [t2,t5]
o6: t7:Random ← [-, copy] t6
o7: t8:Random ← xor [t0,t3] [t6,t7]
o8: t9:Random ← [-, copy] t8
o9: out [t10:Random ← [t8,t9]]
```

Fig. 8: Typed intermediate representation

The type-inference algorithm is conservative. Function $type(t) : Temps \rightarrow \{R, S, P\}$ returns the type assigned to temporary variable t. This section abbreviates the types as follows: type R corresponds to `Random`, S corresponds to `Secret`, and P corresponds to `Public`.

In the following, we define the data that the security analysis provides to the constraint model, which the latter requires to impose security constraints. According to the leakage model, when a hardware register changes from one value to another, the exclusive OR of the two values is exposed. $Rpairs$ is the set of temporary variable pairs that when xor:ed together reveal secret information:

$$\begin{aligned} Rpairs = \{(t_1, t_2) \mid\ & t_1, t_2 \in Temps \ \wedge \\ & type(t_1), type(t_2) \in \{R, P\} \ \wedge \\ & type(t_1 \oplus t_2) = S\}. \end{aligned} \tag{7}$$

In the running example (Fig. 8), $Rpairs = \{$(`t1`,`t6`), (`t1`,`t7`), (`t1`,`t8`), (`t1`,`t9`), (`t4`,`t6`), (`t4`,`t7`), (`t4`,`t8`), (`t4`,`t9`), (`t6`,`t7`), (`t6`,`t8`), (`t6`,`t9`), (`t7`,`t8`), (`t7`,`t9`), (`t8`,`t9`)$\}$. For every pair of temporaries in $Rpairs$, a constraint prohibits the contiguous assignment of the temporaries to the same register (`m` and `mk` in Fig. 6b).

$Rpairs$ do not consider secret values. Instead, if the type of a temporary variable t is `Secret`, we impose a different constraint because the secret information will always result in a leak. In this case, we impose the constraint that another random variable should precede and follow the definition of the secret variable to mask the secret information. $Spairs$ is a set of pairs, each of which consists of a secret temporary variable t and a set of random temporary variables ts that hide the secret information, i.e. $\forall t' \in ts\ .\ type(t' \oplus t) = R$:

$$\begin{aligned} Spairs = \{(t, ts) \mid\ & t \in Temps \ \wedge \ type(t) = S \ \wedge \\ & ts = \{t' \mid t' \in Temps \ \wedge \ type(t') = R \ \wedge \\ & type(t' \oplus t) = R\}\}. \end{aligned} \tag{8}$$

In the running example (Fig. 8), $Spairs = \{$(`t5`,$\{$`t4`,`t6`, `t7`,`t8`,`t9`$\}$)$\}$.

Memory operations may also reveal secret information. We assume the same leakage model (HD model) for the memory bus as for the register-reuse transitional effects. This means that the leakage corresponds to the exclusive OR of two subsequent memory operations. $Mmpairs$ includes the pairs of memory operations that result in memory-bus transitional leakage, i.e. pairs of memory operations that when scheduled subsequently lead to a secret leakage:

$$\begin{aligned} Mmpairs = \{(o_1, o_2) \mid\ & o_1, o_2 \in MemOperations \ \wedge \\ & type(tm(o_1)), type(tm(o_2)) \in \{R, P\} \ \wedge \\ & type(tm(o_1) \oplus tm(o_2)) = S\}. \end{aligned} \tag{9}$$

Here, $tm(o) \in Temps$ is the temporary that corresponds to the memory data of the operation. In the running example (Fig. 8),

$Mmpairs = \{(o3, o6), (o3, o8), (o6, o8)\}$, in case o3, o6, o8, are memory spills. Note that, for simplicity, Fig. 8 does not include all copies for memory spilling as we would need to duplicate the copies for first storing and then loading the variables.

The same leakage as in the case when a secret value was written to a register applies here. If a memory operation stores/loads a secret value to/from the memory, a random memory operation that is able to hide the secret information should precede and follow this operation. $Mspairs$ is a set of pairs, each of which consists of the memory operation that accesses secret data, o, and a set of memory operations that access random data and are able to hide the secret information, i.e. $type(tm(o') \oplus tm(o)) = R$:

$$\begin{aligned} Mspairs = \{(o, os) \mid o \in MemOperations \ \wedge \\ type(tm(o)) = S \ \wedge \\ os = \{o' \mid o' \in MemOperations \ \wedge \\ type(tm(o')) = R \ \wedge \\ type(tm(o') \oplus tm(o)) = R\}\}. \end{aligned} \quad (10)$$

In the example (Fig. 8), $Mspairs = \{(o4, \{o3, o6, o8\})\}$, in case o4, o3, o6, and o8 are spilled in memory.

The security analysis provides $Rpairs$, $Spairs$, $Mmpairs$, and $Mspairs$ to the constraint model, which enables constraining code generation to generate secure implementations.

B. Constraint Model

The constraint model takes as input the four sets computed by the security analysis ($Rpairs$, $Spairs$, $Mmpairs$, and $Mspairs$) and uses them to generate appropriate constraints that prohibit insecure solutions.

Predicate `samereg` tells whether the two input temporaries are active (`l(t) = 1`) and are assigned to the same register.

```
pred samereg(t1,t2):
  l(t1) ∧ l(t2) ∧ (r(t1) = r(t2))
```

In Fig. 5, `samereg(t0,t8) = l(t0) ∧ l(t8) ∧ (r(t0) = r(t8)) = true`, `samereg(t2,t6) = false` (`r(t2) ≠ r(t6)`), and `samereg(t1,t7) = false` (`t7` is not live).

1) Rpairs Constraints: The following constraint ensures that a pair of temporaries in $Rpairs$ are either not assigned to the same register or they are not subsequent (`subseq` constraint, defined in Section IV-B5).

```
forall (t1,t2) in Rpairs:
  samereg(t1, t2) ⟹
    (¬subseq(t1,t2) ∧ ¬subseq(t2,t1))
```

In Fig. 5, this constraint is not satisfied for `t1` and `t6` because `samereg(t2,t6) = true` and `subseq(t2,t6) = true`.

2) Spairs Constraints: The following constraint ensures that for each pair $(t_s, t_{rs}) \in Spairs$, if t_s is live, one of the random temporaries $t_r \in t_{rs}$ precedes the secret temporary t_s and another random temporary succeeds t_s.

```
forall (ts,trs) in Spairs:
  exists tr in trs:
      l(ts) ⟹ (l(tr) ∧ subseq(tr,ts)) ∧
  exists tr in trs:
      l(ts) ⟹ (l(tr) ∧ subseq(ts,tr))
```

Fig. 9 shows a solution to the model in Fig. 4, where both the $Rpairs$ and the $Spairs$ constraints are satisfied. `t5` is active but is assigned to the same register as `t4`, which precedes `t5` and thus eliminates the leakage. Similarly, `t6` follows the assignment of `t5` and thus hides the secret value.

```
o1: in [t0:R0, t1:R1, t2:R2]
o3: t4:R3 ← t1:R1
o4: t5:R3 ← t2:R2
o5: t6:R3 ← xor t1:R1 t5:R3
o7: t8:R0 ← xor t0:R0 t6:R3
o9: out [t10:R0]
```

Fig. 9: Solution of the model in Fig. 4

3) Mmpairs Constraints: The following constraint ensures that a pair of non-secret memory operations in $Mmpairs$, are either not active or not subsequent memory operations (`msubseq` constraint). Constraint `msubseq` (defined in Section IV-B5) is similar to `subseq` but considers consecutive memory operations instead of temporaries.

```
forall (o1,o2) in Mmpairs:
  a(o1) ∧ a(o2) ⟹
    (¬msubseq(o1,o2) ∧ ¬msubseq(o2,o1))
```

4) Mspairs Constraints: Finally, the following constraint ensures that for each pair $(o_s, o_{rs}) \in Mspairs$ a random memory operation $o_r \in o_{rs}$ precedes the secret-dependent memory operation o_s.

```
forall (os,ors) in Mspairs:
  exists or in ors:
    a(os) ⟹ (a(or) ∧ msubseq(or,os)) ∧
  exists or in ors:
    a(os) ⟹ (a(or) ∧ msubseq(or,os))
```

This constraint works similarly as the equivalent register constraint, where instead of register operations, we have memory operations. In our example, we need to have memory spilling, i.e. store to the stack, and then load from the stack (only one of the operations is shown in Fig. 9).

5) Modeling `subseq`: To define the `subseq` constraint, we first define an auxiliary predicate `is_before` and a set of auxiliary problem variables `lk`. Predicate `is_before(t1, t2)` tells whether t_1 is assigned to the same register as t_2 and t_1's life range ends (`le(t1)`) before the beginning of the life range of t_2 (`ls(t2)`).

```
pred is_before(t1,t2): same_reg(t2, t1) ∧
                       (le(t2) ≤ ls(t1))
```

Variable `lk(t)` captures the end live cycle of the temporary that occupied the same register as `t` (`r(t)`) right before `t`

was assigned. If `t' = lk(t)`, then the values of `t` and `t'` result in a transitional effect that may reveal information to the attacker.

```
forall t in Temps: lk(t) = max(
  [ite(is_before(t',t),le(t'),-1)
                  | forall t' in Temps])
```

Then, the definition of the `subseq` predicate is as follows:

```
pred subseq(t1,t2):
  samereg(t1,t2) ∧ (lk(t2) = le(t1))
```

Theorem 1 (Subseq Constraint)**.** *The `subseq` constraint is true only for pairs of temporary variables that are subsequently assigned to the same register:*

$subseq(t_1,t_2) \iff P = P';t_1 \leftarrow e_1;P'';t_2 \leftarrow e_2;P''' \wedge r(t_1) = r(t_2) \wedge \forall i \in P'' \,.\, i = t \leftarrow e \implies r(t) \neq r(t_1)$.

Proof. ($\Leftarrow$) Assume $P = P';t_1 \leftarrow e_1;P'';t_2 \leftarrow e_2;P''' \wedge r(t_1) = r(t_2) \wedge \forall i \in P'' \,.\, i = t = e \wedge r(t) \neq r(t_1)$. We consider all register assignments in P: $P = ...;t_i \leftarrow e_i;...;t_1 \leftarrow e_2;...;t_2 \leftarrow e_2;...;t_j \leftarrow e_j...$; all these assignments are live because they appear in the final program. For all assignments t_j following t_2 (and thus also t_i) we have that $le(t_j) > ls(t_2)$, which implies that $\texttt{is\_before}(t_j,t_i) = \texttt{false}$, and thus all t_j contribute with -1 to `max` in $lk(t_2)$. The same applies for all registers that are assigned to a different register, they contribute with -1 because $\texttt{is\_before}(t_j,t_i) = \texttt{false}$. Then, $lk(t_2) = max(le(t) | t \in \{t_{i_1},t_{i_2},..,t_1\})$, where all $\{t_{i_1},t_{i_2},..,t_1\}$ are assigned the same register, $r(t_2)$. Because these temporaries are assigned to the same register, their live ranges do not overlap (Equation 5), i.e. $\forall t,t' \in \{t_{i_1},t_{i_2},..,t_1\} \,.\, ls(t) \geq le(t') \vee ls(t') \geq le(t)$. Because $t_1 \leftarrow e_1$ is scheduled last $\forall t \in \{t_{i_1},t_{i_2},..,t_{i_n},t_1\} \,.\, ls(t_1) \geq le(t)$. Also, from Equation 6, $le(t_1) > ls(t_1)$. This implies that $\forall t \in \{t_{i_1},t_{i_2},..,t_{i_n}\} \,.\, le(t_1) > le(t)$, so we have $lk(t_2) = le(t_1)$ and $\forall t \in \{t_{i_1},t_{i_2},..,t_{i_n}\} \,.\, lk(t_2) > le(t)$. Therefore only for t_1, $\texttt{subseq}(t_1,t_2) = \texttt{true}$.

($\Rightarrow$) Assume $\texttt{subseq}(t_1,t_2)$. This implies that $\texttt{samereg}(t_1,t_2) \wedge lk(t_2) = le(t_1)$. Constraint $\texttt{samereg}(t_1,t_2)$ implies that $r(t_1) = r(t_2)$ and $l(t_1) \wedge l(t_2)$, which means that they appear in the final code, P, and are assigned to the same register. Because $lk(t_2) = le(t_1)$, t_1 is scheduled before t_2 or $P = P';t_1 \leftarrow e_1;P'';t_2 \leftarrow e_2;P'''$. Now, we only need to prove that there is no other assignment of $r(t_1)$ in P'', i.e. $\forall i \in P'' \,.\, t \leftarrow e \wedge r(t) \neq r(t_1)$. If $\exists i \in P'' \,.\, t \leftarrow e \wedge r(t) = r(t_1)$, then, because live ranges do not overlap, $le(t) > le(t_1)$, which means that $lk(t_2) = le(t), \neq le(t_1)$, which is invalid. □

For the definition of `msubseq`, we define an auxiliary predicate `is_before_mem` and auxiliary problem variables `ok`. Predicate $\texttt{is\_before\_mem}(o_1, o_2)$ tells whether o_1 is scheduled before o_2.

```
pred is_before_mem(o1,o2):
    a(o1) ∧ (c(o1) ≤ c(o2))
```

In Fig. 9, $\texttt{is\_before\_mem}(o_4, o_3)$ is `true`.

Variable `ok(o)` captures the issue cycle of memory operation $o' \in$ `MemOperations` that was issued before o.

```
forall o in MemOperations: ok(o) = max(
  [ite(is_before_mem(o', o), c(o'), -1)
          | forall o' in MemOperations])
```

Similar to predicate `subseq`, `msubseq` is as follows:

```
pred msubseq(o1,o2):
  a(o1) ∧ a(o2) ∧ ok(o2) = c(o1)
```

Theorem 2 (Msubsec Constraint)**.** *The `msubseq` constraint is true only for two instructions that are subsequently accessing the memory:* $msubseq(o_1,o_2) \iff P = P';o_1;P'';o_2;P''' \wedge \nexists o \in P'' \,.\, o = mem(e'',e_3)$*, where o_1 and o_2 are memory operations, $o_1 = mem(e,e_1)$ and $o_2 = mem(e',e_2)$.*

Proof. Similar to Theorem 1. □

Theorem 3 shows that SecCG generates secure code for our threat model.

Theorem 3 (Secure Modeling)**.** *A program P, generated by SecCG, satisfies the leakage equivalence condition in Definition 1. This means that given two input instances IN, IN′ that differ only with regards to the secret variables, $IN_{sec} \subseteq IN$, $IN'_{sec} \subseteq IN'$, the distributions of the leakages do not differ.*

Proof. We assume that the type-inference algorithm overapproximates the actual distribution of each variable. Then, we perform structural induction on the program P to prove that security constraints we introduce lead to secure programs. The proof is available as supplementary material [21]. □

C. Solving Enhancements

Large problems in combinatorial solving can quickly become difficult to handle due to state-space explosion. A solution to this problem is structural decomposition of the problem into subproblems. In code generation, a natural structural decomposition scheme consists of splitting the problem into basic blocks [16]. However, SecCG's security analysis [7] requires linearized code that corresponds to one large basic block. There are already approaches on splitting large code blocks into smaller artificial code blocks for improving the scalability of the solver [16]. Typically, in decomposition schemes, the solver first solves each partial solution (basic blocks) and then composes a full solution consisting of the partial solutions. However, this solution becomes challenging with the addition of security constraints that relate different parts of the code, introducing new inter-block dependencies. These dependencies may lead to conflicts between the partial solutions resulting in the rejection of the full solution. To deal with this problem, SecCG propagates only part of the partial solutions, leaving some parts of the full solution unsolved. In particular, SecCG does not propagate the register assignments to temporaries that correspond to earliest and latest assigned hardware registers in each basic block, as well as their

corresponding issue cycles. Subsequently, SecCG solves the unsolved parts as part of the full problem.

The second main enhancement to the solving procedure concerns the final step of the solving process. In SecCG we make use of Large Neighborhood Search (LNS) [25], a form of local search for constraint programming. In particular, at the end of the decomposition phase, SecCG uses the best found solution to perform local search and locate better solutions.

V. Evaluation

For the evaluation of SecCG, we pose the following research questions:

RQ1: What is the overhead in execution time for the generated code using SecCG? Here, we want to evaluate the introduced overhead of secure solutions compared to optimized but insecure solutions. To do that, we compare the best known solution [16] with our approach SecCG.

RQ2: What is the improvement in execution time of the generated code over non-optimized code and other TBL-secure approaches? Here, we compare our results with LLVM-3.8 with no optimization (-O0) and the work by Wang et al. [7].

RQ3: What is the overhead in compilation time using SecCG? Here, we want to evaluate the introduced compilation overhead of secure solutions compared to insecure solutions. To do that, we compare the compilation time for retrieving the best known solution [16] with SecCG's compilation time.

A. Preliminaries

The following sections describe the implementation details and the experimental setup of the evaluation of SecCG. The implementation of SecCG and the experiments and benchmarks for the evaluation are available at https://github.com/romits800/seccon_experiments.git.

1) Implementation Details: SecCG is implemented as an extension of Unison [16], a constraint-based compiler backend that uses CP to optimize software functions with regards to code size and execution time. In particular, Unison combines two low-level optimizations, instructions scheduling and register allocation, and achieves optimizing medium-size functions with improvement compared to LLVM. Unison uses two global constraints for modeling the backend transformations; 1) the *geometric packing constraint* for register allocation and 2) the *cumulative* constraint for instruction scheduling. The type-inference implementation is written in Haskell and is based on Wang et al. [7] with precision improvements (see supplementary material [21]).

2) Experimental Setup: All experiments run on an Intel®Core™i9-9920X processor at 3.50GHz with 64GB of RAM running Debian GNU/Linux 10 (buster). We use LLVM-3.8 as the front-end compiler for these experiments. To preserve the high-level security properties of the compiled programs, we apply only one optimization, register promotion, (`-mem2reg` in LLVM), which lifts program variables from the stack to registers. We evaluate our method on two architectures: ARM Thumb, targeting processor ARM Cortex M0, a highly predictable processor targeting small embedded devices; and Mips32, a widely-used embedded architecture.

We implemented the constraint model both as part of the specialized Gecode [26] constraint model and the Minizinc [27] model that Unison provides. The Minizinc model allows for solving the problem using multiple solvers. In total, we tried four solvers, Chuffed v0.10.3 [28], OR-Tools [29], Elsie Geas[1], and the specialized model written in Gecode v6.2. We ran the former three solvers activating the *free-search* option. For the specialized model in Gecode, apart from the security model, SecCG includes the modified search enhancements that we describe in Section IV-C. Among all these solvers, Gecode and Chuffed performed best. None of them was able to solve all the problems but together they could solve most of the problems. In the smaller benchmarks, P0-P6, we run a portfolio solver including Gecode and Chuffed. For the larger benchmarks, we run every solver separately for reducing the risk of out-of-memory errors when running both solvers in parallel. The presented results are the result of five runs for SecCG and Unison, whereas for the calculation of the execution time for LLVM -O0, we run the compilation 1000 times to account for possible fluctuations in the compilation time on the evaluation machine.

3) Benchmarks: To evaluate our approach, we use a set of small benchmark programs, up to 100 lines of C code and one program exceeding 900 lines of C code. Table I provides a description of these benchmarks, including the number of lines of code (LoC), and the program variables, i.e. the input variables (IN) and the number of secret (IN_{sec}), public (IN_{pub}), and random (IN_{rand}) input variables. Benchmarks P1 to P6 and P8 to P11 were made available by Wang et al.[2] [7], whereas P0 and P7 are implemented by the authors of this paper. These benchmark programs constitute different masked implementations from previous work and are linearized. Wang et al. [7] use a larger number of benchmarks to evaluate their approach. However, our approach depends on a combinatorial optimizing compiler, Unison, which scales to up to medium size functions, namely, up to approximately 200 intermediate instructions for ARM Cortex M0 and Mips32 architectures [16]. In addition to this, SecCG adds additional constraints that increase the complexity of the model (see Section V-D). Therefore, we selected the smallest benchmarks for our experiments. As a future work, we plan to investigate non-linearized implementations, but this comes at the expense of analysis precision and potentially increased performance overhead.

B. RQ1: Optimality Overhead

SecCG builds on a constraint-based compiler backend to generate a program that satisfies security constraints for software masking. This means that our approach might compromise some of the code quality of the non-mitigated optimized code to mitigate the software masking leaks. To evaluate

[1]Elsie Geas: https://bitbucket.org/gkgange/geas/src/master/

[2]FSE19 tool: https://github.com/bobowang2333/FSE19

TABLE I: Benchmark Description

Prg	Description	LoC	Input Variables (IN)		
			pub	sec	rand
P0	Xor (Listing 1)	5	1	1	1
P1	AES Shift Rows [6]	11	0	2	2
P2	Messerges Boolean [6]	12	0	1	2
P3	Goubin Boolean [6]	12	0	1	2
P4	SecMultOpt_wires_1 [4]	25	1	1	3
P5	SecMult_wires_1 [4]	25	1	1	3
P6	SecMultLinear_wires_1 [4]	32	1	1	3
P7	Whitening [6]	58	16	16	16
P8	CPRR13-lut_wires_1 [5]	81	1	1	7
P9	CPRR13-OptLUT_wires_1 [5]	84	1	1	7
P10	CPRR13-1_wires_1 [5]	104	1	1	7
P11	KS_transitions_1 [30]	964	1	16	32

TABLE II: Optimal solution by Unison and SecCG (SCG) in cycles; Oh stands for overhead; OM stands for *out of memory*

Prg	ARM Cortex M0			Mips32		
	[16]	SCG	Oh (%)	[16]	SCG	Oh (%)
P0	6	6	0	3	3	0
P1	8	9	13	5	5	0
P2	10	10	0	7	7	0
P3	13	13	0	9	9	0
P4	28	28	0	75	75	0
P5	28	28	0	75	75	0
P6	30	30	0	73	73	0
P7	125	128	2	184	184	0
P8	85	85	0	151	151	0
P9	79	82	4	151	151	0
P10	85	81	-5	281	281	0
P11	2635	OM	-	1335	OM	-

the overhead of our method compared to non-secure optimization, we compare the execution time of the optimized solution (optimal or suboptimal solution) that Unison [16] generates compared with SecCG's optimized and TBL-secure code. The overhead is computed as $(cycles(SecCG) - cycles(Unison))/cycles(Unison)$.

Table II shows the mean execution time for each of the benchmark programs and architectures. In particular, for each of the architectures, we compare the execution time in number of cycles of the solution that Unison produces against SecCG's solution. The final column shows the overhead of SecCG compared to Unison.

The results show zero overhead for Mips32, and a maximum 13% overhead in ARM Cortex M0. The zero overhead for most of the benchmarks shows that the Pareto front of optimal solutions synthesized by Unison includes code variants that are secure. This result is in agreement with previous work [31], which shows the existence of multiple optimal (or best found) solutions. For ARM Cortex M0, programs P1, P7, and P9 have a non-zero positive overhead. The observed overhead in ARM Cortex M0 is due to three main reasons: 1) the mitigation itself that may require the introduction of redundant operations in the generated code, 2) the scalability issue that appears in larger functions due to the addition of new security constraints in the order of $|Temps|^2$, and 3) the decomposition mode that may fail to compose solutions (Section IV-C). Program P10 shows a slight improvement. This improvement is due to the introduction of LNS at the end of the solving stage (see Section IV-C), which is not present in Unison. The last benchmark program, P11, demonstrates the scalability limits of our approach. The operating system terminates the solving process because the process attempts to allocate more than the available memory (out-of-memory error).

To summarize, SecCG does not introduce significant overhead over the non-secure optimized solution that Unison generates. This means that in most cases, there is space for generating secure code without affecting the quality of the generated code.

C. RQ2: Execution-Time Improvement

To evaluate the execution-time speedup of our approach, we compare SecCG with the code generated by LLVM without optimizations (-O0). We also compare SecCG with the work by Wang et al. [7]. Wang et al. identify and mitigate ROT leaks on non-optimized code from LLVM 3.6. This is a common approach by different security mitigations, because compilation passes may violate the security properties of a program. During their mitigation, Wang et al. may remove unused code [7], which reduces the overhead.

We compare SecCG with the approach by Wang et al. [7] for three main reasons, 1) their tool is available freely, 2) they propose an architecture-agnostic approach that applies to both Mips32 and ARM Thumb, and 3) they mitigate transitional effect caused by register reuse, a subset of our mitigation. Table III compares the execution time in number of cycles (based on a LLVM-derived cost model) of LLVM, the mitigated code by Wang et al. [7] and SecCG, for each of the programs and architectures. Speedup is computed as $cycles(SecCG)/cycles(LLVMO0)$.

For ARM Cortex M0, the speedup ranges from 2.9 for P5 to 6.3 for P2 and a geometric mean of 3.9 speedup. We notice that for all benchmarks, SecCG achieves significant improvement over the baseline. The main reason for this, is that the increased size of the program under analysis reduces the ability of the solver to find optimal solutions.

For Mips32, the improvement ranges from 77% to 6.6 speedup and a geometric mean of 3.15 speedup. The improvement is larger for smaller benchmarks due to the large overhead of `load` and `store` instructions that are present in the absence of optimizations in the baseline. In contrast to the non-optimized code, the code generated by SecCG reduces memory spilling. In particular, the generic cost model for Mips32 that we use (derived from LLVM) has an one cycle overhead compared to linear instructions. For larger programs, P4-P10, the speedup is smaller but still significant.

This experiment shows that for both architectures SecCG achieves improvement ranging from 77% up to a speedup of 6.6 with geometric-mean speedups 3.9 and 3.15 for ARM Cortex M0 and Mips32, respectively. Although not completely comparable with SecCG because of the use of different benchmarks and mitigations, Vu et al. show an improvement over non-optimized code (-O0) that ranges from 20% to a speedup of 12.6, with a geometric mean of 2.8 [15]. Compared to the

approach by Wang et al., the speedup that SecCG achieves ranges from 1.95 (24%) to 7.6 for ARM Cortex M0 and from 1.36 (36%) to 7.7 for Mips32. The geometric-mean speedups are 3.52 for ARM Cortex M0 and 2.9 for Mips32.

To summarize, for both Mips32 and ARM Cortex M0, SecCG improves the non-optimized LLVM code. We notice large improvements for both Mips32 and ARM Cortex M0 ranging from 77% to 6.6 speedup. SecCG generates also improved code compared to the work by Wang et al. [7].

TABLE III: Execution-time comparison between the non-optimized baseline and SecCG (SCG); Su is the speedup of SecCG with LLVM with -O0 as baseline; OM stands for *out of memory*

Prg	ARM Cortex M0				Mips32			
	O0	[7]	SCG	Su	O0	[7]	SCG	Su
P0	20	22	6	3.33	19	23	3	6.33
P1	39	32	9	4.33	33	21	5	6.60
P2	63	76	10	6.30	43	43	7	6.14
P3	52	56	13	4.00	47	47	9	5.22
P4	87	96	28	3.11	139	139	75	1.85
P5	81	90	28	2.89	133	133	75	1.77
P6	112	69	30	3.73	189	188	73	2.59
P7	609	786	128	4.76	382	430	184	2.08
P8	293	166	85	3.45	371	253	151	2.46
P9	301	303	82	3.67	371	371	151	2.46
P10	333	176	81	4.11	593	383	281	2.11
P11	4504	6742	OM	-	3688	3237	OM	-

D. RQ3: Compilation Overhead

To evaluate the compilation overhead of our approach, we compare SecCG with Unison [16] and non-optimized LLVM. The main reason for the compilation overhead of SecCG compared to LLVM is the combinatorial nature of the backend compiler. Compared to Unison, SecCG introduces compilation overhead due to the security constraints among temporaries and operations in the combinatorial model. In particular, the `subseq` constraint introduces a large number of constraints and variables that are in the order of $|Temp|^2$. The constraints between memory operations (`msubseq`) are typically fewer because memory operations are a subset of all operations. In general, the actual overhead depends on the program logic and the security policy. The compilation slowdown is computed as $comp\_time(SecCG)/comp\_time(Unison)$.

Table IV compares the compilation time of SecCG, Unison, and LLVM -O0. The last column for each architecture in Table IV presents the slowdown of SecCG compared to Unison. In Mips32, we can see an increasing overhead in the compilation time of SecCG compared to Unison with the increase of the function size. The largest compilation overhead is for P10 and corresponds to 57.4 slowdown compared to Unison. The compilation time for non-optimized LLVM ranges from 0.01 to 0.04 seconds. Comparing SecCG with LLVM, the slowdown ranges from 300 for P0 to 300K for P10 (the slowdown does not appear in Table IV)

In the case of ARM Cortex M0, we observe a similar trend. We observe the largest slowdown for P9 which corresponds to 27.9 slowdown. However, the compilation time increases faster than for Mips32. Compared with LLVM, SecCG results in a slowdown that ranges from 29 for P0 to 400K for P9 (does not appear in Table IV). The main reasons for the observable difference between the two architectures are 1) the ARM Thumb architecture is more constrained[3] and 2) interestingly, most instances for Mips32 are solved quickly by Chuffed, whereas most instances for ARM Cortex M0 are only solved by Gecode.

To summarize, the compilation time for SecCG is multiple times slower than Unison because of the introduction of security constraints. In addition to this, SecCG is slower than LLVM. Therefore, we believe that SecCG is mostly suitable for compiling small cryptographic kernels that are both critical for the performance and the PSC security, such as secure field multiplication for AES [4].

TABLE IV: Compilation overhead for SecCG (SCG) compared to Baseline (Unison) in seconds; Sd stands for slowdown of SecCG compared to Unison [16]; OM stands for "out of memory"

Prg	ARM Cortex M0				Mips32			
	O0	[16]	SCG	Sd	O0	[16]	SCG	Sd
P0	0.01	0.17	0.29	1.7	0.01	0.43	2.9	6.6
P1	0.01	0.23	3.4	14.7	0.01	0.52	5.1	9.9
P2	0.01	0.32	1.2	3.7	0.01	0.69	6.5	9.5
P3	0.01	7.7	22.9	3.0	0.01	0.84	8.9	10.6
P4	0.01	1K	1K	1.0	0.01	1.2	16.0	13.5
P5	0.01	1K	1K	1.0	0.01	1.2	16.0	13.7
P6	0.01	1K	1K	1.0	0.01	1.3	18.6	14.3
P7	0.02	1.0K	4K	4.7	0.02	6.3	0.1K	17.2
P8	0.01	0.1K	3K	19.4	0.01	35.0	1K	31.3
P9	0.01	0.1K	4K	27.9	0.01	37.0	1K	27.6
P10	0.02	0.4K	7K	18.0	0.01	47.8	3K	57.4
P11	0.04	5K	OM	-	0.04	52K	OM	-

E. Threat to Validity

Our model considers the HD leakage model and generates code that mitigates these leakages. The security guaranties for our model depend on the HD leakage model. The HD model has been used both for designing defenses [7] and attacks [19]. However, the HD model does not express precisely the actual leakage model for some devices [32]. Moreover, an HD-based mitigation at the assembly level may not hold in the presence of advance microarchitectural features, such as out-of-order execution and write buffers. In addition to this, SecCG does not handle transitional effects through value interaction in the pipeline stage registers and in the memory. We leave further improvement of the hardware model as a future work.

SecCG is not a verified compiler approach like CompCert [33]. Unison, the constraint-based backend that SecCG depends on is based on a formal model that implements standard optimizations but the external solvers and the tool implementation are not verified. Verification of constraint solvers is an active research topic [34].

[3]ARM Cortex M0 has fewer general-purpose registers than MIPS32 and includes two-address instructions, which restrict register allocation.

VI. Related Work

The following sections discuss the related work, with regards to mitigations against side-channel attacks, mitigations against TBLs, and combinatorial compilation approaches. Athanasiou et al. consider two types of PSC leakage sources, Value-Based Leakage (VBL) and Transition-Based Leakage (TBL). VBL occur due to the absence or compiler-induced removal of masking. For example, a compiler transformation may convert a masked expression $\texttt{pub} \oplus (\texttt{mask} \oplus \texttt{key})$ to $\texttt{mask} \oplus (\texttt{pub} \oplus \texttt{key})$, which preserves the code semantics but breaks the masking mitigation. On the other end, TBL is a result of low-level microarchitectural features such as register reuse, memory overwrite, or interactions between values in the hardware. In the following, we will use these two terms to describe different mitigations.

TABLE V: Mitigation approaches against side-channel attacks; SCG stands of SecCG, FE, ME, BE stands for front end, middle end, and back end, respectively; ASM stands for assembly

Pub.	Mitigation	Transf.	InL	OutL	ML	Avail.
[36]	VBL	FE, ME	DSL	-	Custom	✗
[37]	TSC, MS, RS	-	DSL	ASM	Custom	✓
[38]	TSC, MS	-	DSL	ASM	Custom	✓
[39]	TSC, MS	-	DSL	C	F_{low}	✓
[40]	TSC	ME	DSL	C	Custom	✓
[13]	TBL	BE	AVR	AVR	Binary	✓
[41]	IFL	BE	C	ASM	CompCert	?
[7]	TBL	BE	C, C++	ASM	LLVM	✓
[35]	TBL	-	ARM	ARM	Binary	✓✗
[15]	VBL, TSC, FI	ALL	C, C++	ASM	LLVM	✗
[11]	TBL	-	ARM	ARM	Binary	✓
SCG	TBL	ME, BE	C, C++	ASM	LLVM	✓

Side-Channel Compiler Mitigation Approaches: General purpose optimizing compilers perform transformations that may invalidate high-level security mitigations or introduce security flaws [42]. Table V presents a non-exhaustive list of related work that present compiler-based or binary-rewriting approaches against side-channel attacks. For each publication (Publication), Table V, shows the mitigations of each approach (Mitigation), the compiler level that each approach perform the mitigation (Transformation), the input language (InL), the output language (OutL), the Mitigation Level (ML) of each approach that is either a compiler or binary. The last column (Avail.) denotes with ✗ that the artifact is not available, with ✓ that the artifact is available, with ✓✗ that part of the artifact is available, and finally, with **?** where it is not clear whether the artifact is available or not.

Multiple approaches present compiler-based mitigations against Timing Side Channels (TSCs) [37, 38, 39, 40, 15], proof of Memory Safety (MS) [37, 38, 39], or Residual Program State (RS) [37]. Besson et al. present the notion of Information-Flow Leakage (IFL) in compiler optimizations that guarantees that the compiler does not introduce new vulnerabilities [41]. They evaluate their approach on two passes of CompCert, dead-store elimination and register allocation, using a threat model that considers observation points at function boundaries. In contrast, the SecCG backend generates a program secure against ROT and MRE leaks at each execution point. In addition to this, SecCG does not guarantee the preservation of the property but rather the absence of TBLs. If that is not possible, the model is unsatisfiable and SecCG fails to generate a program. The latter outcome has not appeared in our experiments[4] but there is no guarantee that it will not happen. For remedying this problem, one may try to activate a pass in SecCG that introduces additional copies of masked values, deactivate some high-level optimizations, and/or deactivate the ROT or MRE constraints.

A recent approach [14, 15] generates high-quality code that deals with VBLs, Fault Injection (FI), and TSC attacks. To achieve this, Vu et al. [14] introduce the concept of *opaque observations* that disallows the compiler to remove security mitigations or rearrange operands in instructions, such as masking instructions. In their later work [15], they improve the performance of their optimizing compiler by reducing the requirement for serialization. To achieve this, they require source-code annotation that may be challenging for non-trivial programs [15]. Eldib and Wang [36] propose a high-level program synthesis approach to automatically generate masked implementations free from VBLs. Both approaches generate code that mitigates VBLs and thus, do not protect against TBLs.

TABLE VI: TBL-aware approaches

Pub.	Mitigation	Target	Processor
[13]	ROT, MOT, MRE, RNL	AVR	ATMega163
[7]	ROT	*	*
[35]	ROT	ARM	ARM Cortex-M3
[11]	ROT, MOT, MRE, IPI, OT	ARM	ARM Cortex-M0
SecCG	ROT, MRE	*	*

Code Hardening Against Transition-Based Leakages: There is a number of approaches that deal with different types of TBL-related PSCs [13, 7, 35, 11]. Table VI shows the mitigation approaches against TBLs. For each of the related works, Table VI, presents the leakage types each of them mitigates (Mitigation), the target architecture (Target), and the target processor (Processor). In the last two columns * denotes that these approaches may target multiple architectures and processors.

Papagiannopoulos and Veshchikov [13] perform experiments to identify possible sources of leakage in binary AVR code on a ATMega163. They identify sources of leakage including ROT, Memory-Overwrite Transition (MOT), which occurs when overwriting a value in memory, MRE, which occurs when overwriting a value in the memory bus, and Register Neighbor Leakage (RNL), which occurs when the values of neighboring registers interact with each other [13]. Papagiannopoulos and Veshchikov [13] observe that ROT and MRE leakages may be exploited with a small number of runs,

[4] There were unsatisfiable instances due to associativity-related VBLs when using aggressive high-level compiler optimizations (O1, O2, and O3)

500, whereas MOT requites much more (40K). Rosita [11] is a recent approach to mitigate transitional effects that may lead to power side-channel attacks using an emulation-based technique. Rosita performs an iterative process to identify power leakages in software implementations for ARM Cortex M0 and identifies transitional effects due to ROT, MOT, MRE, Instruction-Pair Interaction (IPI), and Other Transitions (OT). IPI occurs when pairs of instructions interact with each other and OT corresponds to interactions between data of different instructions. The mitigation introduces a performance overhead of 21% to 64%. In comparison, SecCG is a generic compiler-based approach that may be applied to multiple hardware architectures and introduces smaller overhead. However, a direct comparison would be unfair because Rosita mitigates more leakage sources.

Wang et al. [7] uses a rule-based system [20, 7] to identify leaks in a masked implementation and perform local register allocation and instruction selection transformations to mitigate these leaks in LLVM. They identify transitional effects due to register reuse, ROT. Their approach is scalable and the mitigation introduces small performance overhead compared to non-optimized code. However, they depend on a non-optimized compilation in order to preserve the security properties of the high-level program, which leads to code generation that is secure against ROT but not optimized. Athanasiou et al. [35] use the same rule-based system to mitigate ROT leakages on binary ARM code targeting the ARM Cortex M3 processor. They are able to reduce the number of potentially vulnerable register pairs given the instruction order. Athanasiou et al. confirm that aggressive compiler optimization passes introduce VBLs. SecCG uses a rule-based system but models a constraint model that is able to generate optimized code that is secure.

Other approaches perform mitigations at whole-system design time [43, 44]. The availability of open hardware architectures and, more specifically, RISC-V, has enabled approaches, such as Coco, which apply software-hardware co-design techniques to mitigate power side-channel attacks [44].

In summary, there are compiler-based and binary rewriting approaches to mitigate TBLs but all these approaches perform local transformations that introduce performance overhead. Instead SecCG trades quality for compilation time and is suitable for performance critical and vulnerable cryptographic functions.

Combinatorial Compiler Approaches: Compiler backend optimizations, like instruction selection, instruction scheduling, and register allocation are known to be hard combinatorial problems. Hence, solving such problems completely does not scale for large program sizes. Therefore, popular compilers, like GCC [45] and LLVM [17], use heuristics that throughout the years have proved to improve program performance. However, these heuristics do not guarantee finding the optimal solution to these backend optimizations.

For critical code and code aimed for compiler-demanding architectures, combinatorial methods may find an optimized version of the code that leads to reduced power consumption and/or high performance benefits. Different works [46, 16, 22, 23] aim to optimize critical code at different levels, like loops [22], locally [23] or at function level [16]. The optimization goals range from execution time, code size, or estimated energy consumption [22, 16, 23]. The main drawback of these approaches is scalability [46]. However, a recent work, Unison [16], allows the optimization of functions of up to almost 1000 instructions.

A different combinatorial approach for generating optimal program code is superoptimization [47]. Given a code sequence, superoptimization approaches attempt to find an equivalent code sequence that reduces the overall execution time and is provably equivalent to the initial code. Souper [48], a state-of-the-art superoptimization approach, performs middle-end optimizations to LLVM IR code. Middle-end optimizations typically do not take decisions on the register allocation or the instruction scheduling. Instead, they enable algorithmic-level code optimizations. Crow [49] is an approach based on Souper that performs software diversification as a security mitigation approach.

To summarize, many combinatorial compiler backend techniques allow low-level code optimization but, to our knowledge, none of them considers the preservation of security properties against TBLs.

VII. Limitations

This paper proposes an architecture-agnostic method to generate high quality code against register-reuse and memory-bus transitional effects. We aim specifically at small-size embedded devices that have a predictable cost model and implement single-issued, non-speculative architectures. Our approach has clear scalability issues, however, we plan to investigate its use in non-linearized functions.

Secondly, our approach is limited to two optimizations, namely register allocation and instruction scheduling. Other backend optimizations, such as instruction selection may be beneficial for removing HD leakages for CISC architectures like x86. Another useful optimization for mitigating optimized implementations may be expression reassociation (`-reassociate` in LLVM).

SecCG generates programs that are MRE- and ROT-leak free. The generated code is straight-line code and thus satisfies the constant-time programming discipline (in the absence of caches). However, analyzing programs that contain operations with operand-dependent latencies (e.g. division) may violate this property. In addition to this, the generated code may contain other types of TBLs, which depends on the actual processor implementation [13].

VIII. Conclusion and Future Work

This paper proposes a constraint-based compiler backend to generate code that is both optimized and secure against power side-channel attacks. We prove that the generated code is secure according to a non-trivial leakage model, and show that our approach achieves high code improvement against non-optimized approaches ranging from 77% to a speedup of 6.6 for two embedded architectures, Mips32 and ARM Cortex

M0. At the same time, our approach introduces a maximum overhead of 13% from the optimal code. This comes at the expense of increased compilation time and reduced scalability.

There are several future directions for our work. We are planning to work on extending the type-inference algorithm to include function calls and loops. Moreover, by improving the accuracy of the hardware model of SecCG to model precisely a specific device, we will be able to improve the leakage model and compare our approach to approaches like Rosita [11]. Finally, we believe that combining our approach with optimizing high-level approaches [14, 15] may further improve the quality of the generated code.

Acknowledgment

We would like to thank Jingbo Wang for providing support for the FSE19 tool and Amir M. Ahmadian for the fruitful discussions and his significant feedback on the paper. Finally, we would like to thank the anonymous reviewers for their constructive and valuable feedback.

References

[1] P. C. Kocher, "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems," in *Annual International Cryptology Conference*. Springer, 1996, pp. 104–113.

[2] P. Kocher, J. Jaffe, and B. Jun, "Differential Power Analysis," in *Annual International Cryptology Conference*. Springer, 1999, pp. 388–397.

[3] M. Joye, P. Paillier, and B. Schoenmakers, "On Second-Order Differential Power Analysis," in *Cryptographic Hardware and Embedded Systems*. Springer, 2005, pp. 293–308.

[4] M. Rivain and E. Prouff, "Provably secure higher-order masking of aes," in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2010, pp. 413–427.

[5] J.-S. Coron, E. Prouff, M. Rivain, and T. Roche, "Higher-Order Side Channel Security and Mask Refreshing," in *Fast Software Encryption*. Springer, 2014, pp. 410–424.

[6] A. G. Bayrak, F. Regazzoni, D. Novo, and P. Ienne, "Sleuth: Automated Verification of Software Power Analysis Countermeasures," in *International Conference on Cryptographic Hardware and Embedded Systems*. Springer, 2013, pp. 293–310.

[7] J. Wang, C. Sung, and C. Wang, "Mitigating power side channels during compilation," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 590–601.

[8] H. Eldib, C. Wang, and P. Schaumont, "Formal Verification of Software Countermeasures against Side-Channel Attacks," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 24, no. 2, pp. 1–24, 2014.

[9] G. Barthe, S. Blazy, R. Hutin, and D. Pichardie, "Secure Compilation of Constant-Resource Programs," in *CSF 2021 - 34th IEEE Computer Security Foundations Symposium*. IEEE, 2021, pp. 1–12.

[10] P. Borrello, D. C. D'Elia, L. Querzoni, and C. Giuffrida, "Constantine: Automatic Side-Channel Resistance Using Efficient Control and Data Flow Linearization," *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pp. 715–733, 2021.

[11] M. A. Shelton, N. Samwel, L. Batina, F. Regazzoni, M. Wagner, and Y. Yarom, "Rosita: Towards Automatic Elimination of Power-Analysis Leakage in Ciphers," *Proceedings 2021 Network and Distributed System Security Symposium*, 2021, appears in NDSS 2022.

[12] N. Veshchikov and S. Guilley, "Use of Simulators for Side-Channel Analysis," in *2017 IEEE European Symposium on Security and Privacy Workshops (EuroS PW)*, 2017, pp. 104–112.

[13] K. Papagiannopoulos and N. Veshchikov, "Mind the Gap: Towards Secure 1st-Order Masking in Software," in *International Workshop on Constructive Side-Channel Analysis and Secure Design*. Springer, 2017, pp. 282–297.

[14] S. T. Vu, K. Heydemann, A. de Grandmaison, and A. Cohen, "Secure delivery of program properties through optimizing compilation," in *Proceedings of the 29th International Conference on Compiler Construction*, 2020, pp. 14–26.

[15] S. T. Vu, A. Cohen, A. De Grandmaison, C. Guillon, and K. Heydemann, "Reconciling optimization with secure compilation," *Proceedings of the ACM on Programming Languages*, vol. 5, no. OOPSLA, pp. 1–30, 2021.

[16] R. Castañeda Lozano, M. Carlsson, G. H. Blindell, and C. Schulte, "Combinatorial Register Allocation and Instruction Scheduling," *ACM Trans. Program. Lang. Syst.*, vol. 41, no. 3, pp. 17:1–17:53, 2019.

[17] C. Lattner and V. Adve, "LLVM: a compilation framework for lifelong program analysis amp; transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 2004, pp. 75–86.

[18] T. S. Messerges, E. A. Dabbish, and R. H. Sloan, "Investigations of power analysis attacks on smartcards." *Smartcard*, vol. 99, pp. 151–161, 1999.

[19] E. Brier, C. Clavier, and F. Olivier, "Correlation Power Analysis with a Leakage Model," in *International workshop on cryptographic hardware and embedded systems*. Springer, 2004, pp. 16–29.

[20] P. Gao, J. Zhang, F. Song, and C. Wang, "Verifying and Quantifying Side-channel Resistance of Masked Software Implementations," *ACM Transactions on Software Engineering and Methodology*, vol. 28, no. 3, pp. 16:1–16:32, 2019.

[21] R. M. Tsoupidi, R. Castañeda Lozano, E. Troubitsyna, and P. Papadimitratos, "Supplemental material: Securing optimized code against power side channels," 2022. [Online]. Available: https://github.com/romits800/seccon_experiments/blob/main/supp_material/main_appendix.pdf

[22] M. Eriksson and C. Kessler, "Integrated Code Generation for Loops," *ACM Transactions on Embedded Computing Systems*, vol. 11S, no. 1, pp. 19:1–19:24, 2012.

[23] C. H. Gebotys, "An efficient model for DSP code generation: Performance, code size, estimated energy," in *Proceedings of the tenth International Symposium on System Synthesis*. IEEE, 1997, pp. 41–47.

[24] F. Rossi, P. Van Beek, and T. Walsh, *Handbook of constraint programming*. Elsevier, 2006.

[25] P. Shaw, "Using Constraint Programming and Local Search Methods to Solve Vehicle Routing Problems," in *International conference on principles and practice of constraint programming*. Springer, 1998, pp. 417–431.

[26] Gecode Team, "Gecode: Generic constraint development environment," 2022. [Online]. Available: https://www.gecode.org

[27] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack, "MiniZinc: Towards a Standard CP Modelling Language," in *International Conference on Principles and Practice of Constraint Programming*. Springer, 2007, pp. 529–543.

[28] G. G. Chu, "Improving combinatorial optimization," Ph.D. dissertation, The University of Melbourne, Australia, 2011.

[29] Google Developers, "Google OR-Tools," 2022. [Online]. Available: https://developers.google.com/optimization/

[30] G. Barthe, S. Belaïd, F. Dupressoir, P.-A. Fouque, B. Grégoire, and P.-Y. Strub, "Verified Proofs of Higher-Order Masking," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2015, pp. 457–485.

[31] R. M. Tsoupidi, R. Castañeda Lozano, and B. Baudry, "Constraint-based Diversification of JOP Gadgets," *Journal of Artificial Intelligence Research*, vol. 72, pp. 1471–1505, 2021.

[32] Y. Oren, M. Renauld, F.-X. Standaert, and A. Wool, "Algebraic Side-Channel Attacks Beyond the Hamming Weight Leakage Model," in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2012, pp. 140–154.

[33] X. Leroy, "A Formally Verified Compiler Back-end," *Journal of Automated Reasoning*, vol. 43, no. 4, p. 363, 2009.

[34] S. Gocht, C. McCreesh, and J. Nordström, "An auditable constraint programming solver," in *International Conference on Principles and Practice of Constraint Programming*, 2022.

[35] K. Athanasiou, T. Wahl, A. A. Ding, and Y. Fei, "Automatic detection and repair of transition-based leakage in software binaries," in *Software Verification*. Springer, 2020, pp. 50–67.

[36] H. Eldib and C. Wang, "Synthesis of Masking Countermeasures against Side Channel Attacks," in *International Conference on Computer Aided Verification*. Springer, 2014, pp. 114–130.

[37] B. Bond, C. Hawblitzel, M. Kapritsos, K. R. M. Leino, J. R. Lorch, B. Parno, A. Rane, S. Setty, and L. Thompson, "Vale: Verifying {High-Performance} Cryptographic Assembly Code," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 917–934.

[38] J. B. Almeida, M. Barbosa, G. Barthe, A. Blot, B. Grégoire, V. Laporte, T. Oliveira, H. Pacheco, B. Schmidt, and P.-Y. Strub, "Jasmin: High-Assurance and High-Speed Cryptography," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1807–1823.

[39] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche, "HACL*: A Verified Modern Cryptographic Library," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1789–1806.

[40] S. Cauligi, G. Soeller, F. Brown, B. Johannesmeyer, Y. Huang, R. Jhala, and D. Stefan, "FaCT: A Flexible, Constant-Time Programming Language," in *2017 IEEE Cybersecurity Development (SecDev)*, 2017, pp. 69–76.

[41] F. Besson, A. Dang, and T. Jensen, "Information-Flow Preservation in Compiler Optimisations," in *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*, 2019, pp. 230–23 012.

[42] V. D'Silva, M. Payer, and D. Song, "The Correctness-Security Gap in Compiler Optimization," in *2015 IEEE Security and Privacy Workshops*, 2015, pp. 73–87.

[43] D. Šijačić, J. Balasch, B. Yang, S. Ghosh, and I. Verbauwhede, "Towards Efficient and Automated Side Channel Evaluations at Design Time," *Journal of Cryptographic Engineering*, vol. 10, no. 4, pp. 305–319, 2020.

[44] B. Gigerl, V. Hadzic, R. Primas, S. Mangard, and R. Bloem, "Coco:{Co-Design} and {Co-Verification} of masked software implementations on {CPUs}," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 1469–1468.

[45] R. M. Stallman, *Using the GNU Compiler Collection: a GNU manual for GCC version 4.3.3*. CreateSpace, 2009.

[46] R. Castañeda Lozano and C. Schulte, "Survey on Combinatorial Register Allocation and Instruction Scheduling," *ACM Computing Surveys*, vol. 52, no. 3, pp. 62:1–62:50, 2019.

[47] C. W. Fraser, "A compact, machine-independent peephole optimizer," in *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM, 1979, pp. 1–6.

[48] R. Sasnauskas, Y. Chen, P. Collingbourne, J. Ketema, G. Lup, J. Taneja, and J. Regehr, "Souper: A synthesizing superoptimizer," *arXiv preprint arXiv:1711.04422*, 2017.

[49] J. Cabrera Arteaga, O. Floros, O. Vera Perez, B. Baudry, and M. Monperrus, "Crow: Code diversification for webassembly," in *MADWeb, NDSS 2021*, 2021.

Appendix E

Publication 5

Thwarting Code-Reuse and Side-Channel Attacks in Embedded Systems

Rodothea Myrsini Tsoupidi[a,*], Elena Troubitsyna[a] and Panagiotis Papadimitratos[a]

[a]Royal Institute of Technology KTH, Stockholm, Sweden

ARTICLE INFO

Keywords:
compiler-based mitigation
software diversification
software masking
constant-resource programming

ABSTRACT

Embedded devices are increasingly present in our everyday life. They often process critical information and hence, rely on cryptographic protocols to achieve security. However, embedded devices remain particularly vulnerable to attackers seeking to hijack their operation and extract sensitive information by exploiting side channels and code reuse. Code-Reuse Attacks (CRAs) can steer the execution of a program to malicious outcomes leveraging existing on-board code without direct access to the device memory. Moreover, Side-Channel Attacks (SCAs) may reveal secret information to the attacker based on mere observation of the device. Thwarting CRAs and SCAs against embedded devices is especially challenging because embedded devices are usually resource-constraint. Fine-grained code diversification can hinder CRAs by introducing uncertainty to the binary code; while software mechanisms can thwart timing or power SCAs. The resilience to either attack may come at the price of the overall efficiency. Moreover, a unified approach that preserves these mitigations against both CRAs and SCAs is not available. In this paper, we propose a novel Secure Diversity by Construction (SecDivCon) approach that tackles this challenge. SecDivCon is a combinatorial compiler-based approach that combines software diversification against CRAs with software mitigations against SCAs. SecDivCon restricts the performance overhead introduced by the generated code that thwarts the attacks and hence, offers a secure-by-design approach enabling control over the performance-security trade-off. Our experiments, using 16 benchmark programs, show that SCA-aware diversification is effective against CRAs, while preserving SCA mitigation properties at a low, controllable overhead. Given the combinatorial nature of our approach, SecDivCon is suitable for small, performance-critical functions that are sensitive to SCAs. SecDivCon may be used as a building block to whole-program code diversification or in a re-randomization scheme of cryptographic code.

1. Introduction

Nowadays, numerous embedded devices, sensors and Internet of Things (IoT) devices process and control a large variety of sensitive information. They are typically resource-constrained and vulnerable to attacks that aim to manipulate their operation and/or extract sensitive information [47]. Memory corruption vulnerabilities induce a serious security threat. Mitigations such as data execution prevention have eradicated code injection attacks. Nonetheless, Code-Reuse Attacks (CRAs) achieve hijacking the control flow of a program using a chain of executable code snippets [58, 53]. These attacks target both general purpose [58] and embedded devices [49, 30, 8, 55]. At the same time, the execution of embedded software may leak information about sensitive data to the adversary via side channels [40, 19, 20]. Side-Channel Attacks (SCAs) allow an attacker to extract information from the target device by recording side-channel information, such as execution time or power consumption, which may depend on secret values.

Mitigating CRAs and SCAs is a double-edged challenge. In the literature, there are solutions tailored to each of these attacks for embedded devices. However, there are two main drawbacks associate with combining individual mitigations. First, there is no guarantee that the sequential application of the mitigations preserves the properties of each of them (see Section 2.1). Second, the mitigation result may accumulate the introduced overhead from each approach [19], which may be forbidding, and thus, creates the need for overhead-aware approaches [66, 64].

In this paper, we address this challenge by proposing a novel approach that combines fine-grained code diversification against CRAs with software mitigations against SCAs. Fine-grained code diversification [55] is a mitigation against CRAs that introduces uncertainty to the binary code implementation, which makes the attacker payload nonfunctional. An important advantage of fine-grained software diversification compared to other mitigations against CRAs is its reduced performance overhead [48, 64]. Typical mitigations against SCAs include software countermeasures that prohibit the flow of secret information to the attacker, such as software masking and Constant Resource (CR) programming (see Section 2.1). The compilation process may not propagate correctly these software mitigations and, thus, the compiler needs to be aware of these properties.

Secure Diversity by Construction (SecDivCon) is a combinatorial compiler-based approach that combines code diversification against CRAs with mitigations against Timing Side Channel (TSC) and Power Side Channel (PSC) attacks. Moreover, SecDivCon uses an accurate cost model for predictable architectures that allows control over the overall performance overhead of the generated code. SecDivCon is appropriate for diversifying small cryptographic core functions

tsoupidi@kth.se (R.M. Tsoupidi); elenatro@kth.se (E. Troubitsyna); papadim@kth.se (P. Papadimitratos)
ORCID(s): 0000-0002-8345-2752 (R.M. Tsoupidi); 0000-0002-3267-5374 (P. Papadimitratos)

that may impose security threats through SCAs. Function-level diversification may be used for whole-program diversification [64] or in a re-randomization scheme [60] against advanced code-reuse attacks [7].

This paper contributes:

- an entirely novel (to the best of our knowledge) composable framework that combines automatic fine-grained code diversification and side-channel mitigations;
- a constraint-based model to generate optimized code against TSCs (Section 3);
- a secure-by-design compiler-based approach that preserves the properties of multiple software mitigations and enables control over the trade-off between performance and security (Section 4);
- evidence that fine-grained automatic code diversification introduces side-channel leaks (Section 4.3);
- evidence that restraining diversity to preserve security measures against SCAs does not have a negative effect on CRA mitigations (Section 4.5).

Reproducibility: The source code and the evaluation process are available online: `https://github.com/romits800/secdivcon_experiments`.

2. Problem Statement and Threat Model

Section 2.1 presents the attacks that we consider and motivates our approach, which combines security mitigations against CRAs and SCAs. Section 2.2 presents the threat model, and finally, Section 2.3 defines the problem.

2.1. Background and Motivation

Code-Reuse Attacks (CRAs): CRAs exploit memory corruption vulnerabilities to hijack the control flow of the victim program and take control over the system [16, 8, 26]. The attacker selects pieces of executable code from the victim program memory, so-called gadgets, and stitches these gadgets together in a chain that results in a malicious attack. Code-reuse gadgets typically end with a control-flow instruction, such as indirect branch, return, or call, which allows the attacker to build a chain of gadgets. Figure 1a shows a code-reuse gadget that we extracted using ROPGadget [56] from an ARM Cortex M0 binary. At address 0x0044, the gadget copies the value of `r2` to register `r0` (line 1), then jumps to the next instruction (line 2) and finally, jumps to the value of register `lr` to the next gadget. As demonstrated in Figure 1a, code-reuse gadgets consist of common instruction sequences that are frequently available in compiled programs.

The main approaches against CRAs are Control-Flow Integrity (CFI) and code randomization. CFI [1] enforces the dynamic execution of the program to conform with the permitted execution paths, whereas automatic code diversification [35] introduces uncertainty to the location and

```
1 0x0044 : mov r0, r2
2 0x0046 : b #0x48
3 0x0048 : bx lr
```

(a) Gadget 1

```
1 0x0044 : mov r0, r3
2 0x0046 : bx lr
3 0x0048 : ...
```

(b) Gadget 2

Figure 1: Two diversified gadgets in ARM Thumb extracted from Figure 4 using ROPGadget

```
1 u32 Xor(u32 pub, u32 key, u32 mask) {
2   u32 mk = mask ^ key;
3   u32 t = pub ^ mk;
4   return t;
5 }
```

(a) Original C code

```
2   u32 t1 = pub ^ key;
3   u32 t2 = mask ^ t1;
4   return t2;
5 }
```

(b) Compiler-induced masking removal

Figure 2: Masked exclusive OR implementation

instruction sequence of the gadgets in the program memory. CFI may be impractical for small, resource-constrained devices due to the diversity of embedded hardware and the increased overhead [45] in small, often battery-operated devices. Automatic software diversification provides an efficient mitigation against CRAs [35, 64, 49]. Figure 1 shows two gadgets in two diversified program variants. Figure 1a and 1b illustrate that they differ in the first instruction, which copies the content of the register `r2`/`r3` to `r0`. An attacker that has designed an attack that uses the first gadget at address `0x0044` to move an attacker-controlled value from `r2` to `r0` will fail if the victim uses the second gadget. There are different ways to diversify software and distribute it to the end users. In this paper, we consider the app store model [35], where a centralized repository distributes precompiled code variants to each end user.

Side-Channel Attacks (SCAs): Usually, embedded devices use cryptographic algorithms, which are vulnerable to SCAs [33, 40, 10]. These attacks allow the adversary to extract information about secret values by measuring the execution time – called timing side channel (TSC) [13] or the power consumption – called power side channel (PSC) [51] of the target device. For example, a publicly installed camera or a smartwatch may be physically exposed to malicious actors that are able to measure the power consumption of the device or the execution time of cryptographic tasks to infer cryptographic keys and retrieve information about sensitive data.

Power Side Channels (PSCs): PSC attack is a SCA that uses the power traces of the target device to extract secret information [69]. A mitigation approach to protect against PSCs is software masking. Consider the code in Figure 2a. Function `Xor` applies software masking to an exclusive or (xor) operation. The program takes three inputs, `pub`, which is a public value, `key`, which is a secret value, and `mask`, which is a randomly generated value. At line 2, the code performs an exclusive or operation between `mask` and `key` to

```
1 @ r0: pub, r1: key, r2: mask
2 eors    r1, r2
3 eors    r0, r1
4 bx      lr
```

(a) Secure

```
2 eors    r2, r1
3 eors    r0, r2
4 bx      lr
```

(b) Insecure

Figure 3: Two program variants of Figure 2a for ARM Cortex M0

randomize the secret value. At line 3, the implementation performs an additional exclusive or operation between the previous result and value `pub`. Figure 3 shows two machine implementations of the code in Figure 2a in ARM Thumb. The first implementation in Figure 3a performs the first xor operation at line 2 and stores the result in register `r1` and the then, performs the second xor operation at line 3 and stores the result in register `r0`. The second implementation in Figure 3b is identical to the first one, apart from the first xor operation at line 2, where the result is copied to register `r2`. The power leakage of the program depends on register-value transitions, Register-Overwrite Transition (ROT) [46], based on the Hamming Distance (HD) model [10], which is widely used for designing PSC attacks and defenses [10, 46]. The leakage using the HD model depends on the exclusive or of the previous value of a register and the new value. Thus, as shown in Figure 3a, the leakage depends on two transitions of registers `r0` and `r1`, with values $\texttt{r1}_{old} \oplus \texttt{r1}_{new} = \texttt{key} \oplus (\texttt{key} \oplus \texttt{mask}) = \texttt{mask}$ and $\texttt{r0}_{old} \oplus \texttt{r0}_{new} = \texttt{pub} \oplus (\texttt{key} \oplus \texttt{mask} \oplus \texttt{pub}) = \texttt{mask} \oplus \texttt{key}$. None of the values depends on a secret value because both values are randomized with `mask`. However, in Figure 3b we have a different leakage, $\texttt{r2}_{old} \oplus \texttt{r2}_{new} = \texttt{mask} \oplus (\texttt{key} \oplus \texttt{mask}) = \texttt{key}$ and $\texttt{r0}_{old} \oplus \texttt{r0}_{new} = \texttt{pub} \oplus (\texttt{key} \oplus \texttt{mask} \oplus \texttt{pub}) = \texttt{mask} \oplus \texttt{key}$. The first value leaks information about the `key`, which is secret (highlighted in Figure 3b). This means that implementation Figure 3b leaks secret information. Thus, the embedded devices that use this variant may be vulnerable to PSCs.

Timing Side Channels (TSCs): TSC attack is another type of SCA, where the attacker measures the execution time during the execution of a program to infer secret information. For example, Figure 4 shows a simple program that contains a timing vulnerability. In particular, at line 3 there is a branch that compares the value of `key` and the value of `pub`. The attacker knows and may control the value of `pub`, whereas `key` is a secret value. If the result of the comparison is `true`, then the observed execution time will be longer than when the result is `false`. Thus, an attacker who can measure the execution time of the code and knows the value of `pub` is able to infer information about the value of `key`.

The Constant Resource (CR) policy is a software-based mitigation approach against TSC attacks that aims at eliminating timing leaks [44]. The CR policy allows secret-dependent branches, as long as the different execution paths require the same execution time. The implementation of CR code is hardware specific because the same instruction may take a different number of cycles in different processor

```
1  u8 check_bit(u8 pub, u8 key) {
2    u8 t = 0;
3    if (pub == key) t = 1;
4    return t;
5  }
```

Figure 4: Program with secret-dependent branching

```
1  @ r0: pub, r1: key
2  @ BB#0:
3    movs   r2, #0
4    cmp    r1, r0
5    bne    .LBB0_2
6  @ BB#1:
7    movs   r0, #1
8    b      .LBB0_3
9  .LBB0_2:
10   mov    r0, r2
11   movs   r1, #1
12 .LBB0_3:
13   bx     lr
14 ...
15 ...
```

(a) Secure variant 1

```
2  @ BB#0:
3    movs   r2, #0
4    cmp    r1, r0
5    bne    .LBB0_2
6  @ BB#1:
7    movs   r3, #1
8    mov    r0, r3
9    b      .LBB0_3
10 .LBB0_2:
11   mov    r3, r2
12   mov    r0, r3
13   movs   r3, #1
14 .LBB0_3:
15   bx     lr
```

(b) Secure variant 2

Figure 5: Two secure program variants of Figure 4 for ARM Cortex M0

implementations. Figure 5 shows two machine implementations for ARM Cortex M0 that preserve the CR policy of the program in Figure 4, where the `if` branch in Figure 4 is balanced with an `else` branch. In Figure 5a, the first basic block (lines 3-5) initializes `t` (line 3) and compares these two input values (lines 4-5). If the result of the comparison is `true` (taken branch), the execution jumps to the third branch, `.LBB0_2`, and the branch operation takes three cycles. If the result of the comparison is `false` (not-taken branch), the execution continues to the second branch (`@BB#1`) and the branch operation takes just one cycle. To balance the two branches, the code generation considers the branch overhead for taken branches and the latency of every instruction, which is three cycles for the unconditional branch, `b`, and one cycle for the move instruction, `mov`. In particular $t(\texttt{@BB\#1}) + 2 = t(\texttt{.LBB0\_2})$, where $t(b)$ is the execution time of the body of basic block b and $+2$ corresponds to the branch overhead on a taken branch. Figure 5b shows another machine implementation of the code in Figure 5a that also preserves the same constraint as Figure 5a. The main differences in Figure 5b concern the register assignment. For example, Figure 5b introduces additional `mov` instructions (lines 8, 11, and 12) to transfer values from one hardware register to another. Without the constraint that enforces the equality of execution time for the two branches, a randomization procedure, may break the CR policy, for example, by adding one No Operation (NOP) instruction in `.LBB0_2`.

Combined Mitigation: Embedded devices that manipulate sensitive data are vulnerable to both SCAs and CRAs. Mitigating SCAs and CRAs in these devices is essential for protecting sensitive data and the system. A low-overhead approach against CRA is fine-grained code diversification, while software mitigations hinder SCAs in cryptographic software. Avoiding diversifying cryptographic libraries may lead to CRAs, as shown in recent work by Ahmed et al. [3], where a CRAs attack may use gadgets from OpenSSL, a cryptographic library. Similarly, diversifying cryptographic code may break software mitigations against SCAs, as we show in Section 4.3. The latter demonstrates that fine-grained code diversification against CRAs and software mitigations against SCAs constitute conflicting mitigations. Therefore, there is a need for combined approaches that protect against the combination of these attacks.

Figures. 3 and 5 show two different machine-code implementations of programs in Figures 2 and 4, respectively. Each of these functions includes code-reuse gadgets that end with instruction `bx lr`. An attacker may select these gadgets to perform a CRA. Generating multiple versions of each program is a form of diversification that hinders attacks by altering the attacker's building blocks. At the same time, these variants should preserve SCA mitigations. For example, the variant in Figure 3b is not secure against PSC attacks. To tackle this problem, we propose SecDivCon, which generates diverse variants protected against CRAs that are also secure against SCAs.

2.2. Threat Model

We assume that the code implementation contains a memory vulnerability that allows the attacker to perform a CRA, in particular a static Return Oriented Programming (ROP) or Jump Oriented Programming (JOP) attack. We further assume that the attacker does not have direct access to the memory of the device. We consider two types of attacker models for SCAs, Timing Attacker (TA) that measures the execution time of the program and Power Attacker (PA) that records the power consumption of the program:

TA: The attacker has access to the software implementation and the *public* data but not the *secret* data. The attacker is able to extract information about the secret data by measuring the execution time of the code on the target device. The measurements are done remotely.

PA: The attacker has access to the software implementation and the *public* data but not the *secret* data. At every execution, the program under execution generates new *random* values and the attacker has no knowledge of these values. The attacker is able to extract information about the secret data by measuring the power consumption of the device that the code runs on. The attacker may accumulate a number of power traces from multiple runs of the program and perform statistical analysis, such as Differential Power Analysis (DPA) [32] or Correlational Power Analysis (CPA) [10, 46].

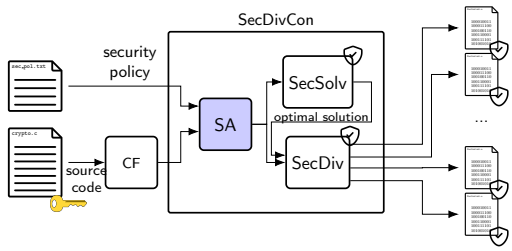


Figure 6: High-level view of SecDivCon

We adopt the leakage model for PSCs from Tsoupidi et al. [65] and the leakage model for the CR-policy from Barthe et al. [6].

2.3. Problem Statement

Our goal is to generate code secure against the attacker models TA and PA. First, we define formally code diversification. We consider a program p and a set, S, of program implementations, $p_i \in S$, that are functionally equivalent ($\sim$) with the original program, i.e. $\forall p_i \in S . p \sim p_i$ and each other, $\forall p_i, p_j \in S . p_i \sim p_j$. To protect against SCAs, we define a set of constraints C_{sec}. A program implementation p_i is secure against SCAs (PSC or TSC attacks) if $p_i \in sol(C_{sec})$.

To protect small embedded devices that are vulnerable to CRAs and SCAs, SecDivCon generates a pool of diverse solutions, S_{sec}, that is a subset of S, and all solutions are secure against SCAs, namely they satisfy C_{sec}, or $p_i, p_j \in S_{sec} \subseteq S \implies p_i, p_j \in sol(C_{sec}) \wedge p_i \sim p_j$, The goal of SecDivCon is to generate set S_{sec}.

3. SecDivCon

SecDivCon uses a combinatorial compiler backend to combine SCA mitigations with code diversification against CRAs. Figure 6 shows a high-level view of SecDivCon. The input to SecDivCon is 1) the security policy, namely which input values are `secret`, `public`, or `random`, and 2) the input function in a low-level intermediate representation generated by a general purpose compiler frontend (CF).

The first stage of SecDivCon is the Security Analysis (SA) module (Section 3.1), which performs code analysis and generates the input data for the second stage, SecSolver (Section 3.2). SecSolver solves the SCA-aware constraint-based backend model and generates the best-found solution according to the cost function. Then, SecSolver passes this solution together with the constraint model to the third stage, SecDiv (Section 3.3), a constraint-based diversification method that is able to generate multiple SCA-aware solutions. The following sections describe each of the stages of SecDivCon.

3.1. Security Analysis Module

The SA module takes as input the security policy and the input function. Subsequently, SecDivCon propagates the security policy to each program term using type inference (Section 3.1.1). In cases when the input program is not secure against SCAs, SecDivCon performs transformations (Section 3.1.2) that enable the generation of secure code. The output of the analysis is the extended constraint model of the input program, which includes data that is necessary for the security constraints (Section 3.2).

3.1.1. Type Inference

For both attacker models, TA and PA, SecDivCon uses type inference to propagate a type, i.e. `secret`, `public`, or `random`, to each program variable. For example, in Figure 4, intermediate variable `t` takes the type `public`. Similarly, in Figure 2a, intermediate variables `mk` and `t` are assigned type `random` (because `mask` randomizes the value of `key`). Software masking introduces additional challenges to the type inference algorithm, which has to capture properties such as $(\texttt{sec} \oplus \texttt{mask}) \oplus \texttt{mask} = \texttt{sec}$. To achieve this, the inference algorithm uses additional environment structures that keep track of the random and secret values that an intermediate variable may contain. The type-inference algorithm that considers random values is based on previous work [24, 67].

3.1.2. Code Transformations

The implementations of C or C++ programs that are given as input to SecDivCon may not be secure. Furthermore, the general-purpose middle-end compiler transformations that SecDivCon uses may break some of the high-level mitigations. In particular, SecDivCon needs to preserve the CR property against TA. However, some of the secret-dependent branches may not be balanced in the source code, or the high-level compiler optimizations may remove dead basic blocks [21]. Similarly, SecDivCon needs to generate secure masked code against PA. The input code is masked, however, high-level optimizations are known to invalidate some masking countermeasures [5, 65]. In the following paragraphs, we discuss the program transformations that SecDivCon implements before the solving stage protecting against TA or PA.

Timing Attacker (TA): CR programs may contain secret-dependent branches. However, these branches should not result in any execution-time differences. Yet, sometimes, the source code of the input program contains unbalanced secret-dependent branches. Figure 4 shows a program that branches on the secret value (line 3). If the condition is `true`, the execution takes at least one cycle (line 4), whereas if the condition is `false`, it takes zero cycles. To deal with these programs, we introduce a balancing block, using two methods, 1) inserting an empty block (EBB), and 2) copying one block (CBB).

EBB: To balance an unbalanced secret-dependent block, EBB adds an empty block that contains NOP operations. Figure 7a shows a secret-dependent branch that is balanced using an empty basic block (lines 5-7). At a later stage, the constraint solver fills this basic block with an appropriate number of NOP instructions to balance the secret-dependent branch. In contrast to CBB, this transformation works also when we want to balance an unbalanced path with more than one basic blocks.

```
1 u32 check_bit(u32 pub, u32 key) {
2   u32 t = 0;
3   if (pub == key)
4     t = 1;
5   else
6     // nop;
7   return t;
8 }
```

(a) Add Empty Block

```
1 u32 check_bit(u32 pub, u32 key) {
2   u32 _t, t = 0;
3   if (pub == key)
4     t = 1;
5   else
6     _t = 1;
7   return t;
8 }
```

(b) Copy Unbalanced Block

Figure 7: Balancing transformations for Figure 4

CBB: Another way to balance a secret-dependent block that consists of one basic block is by copying the unbalanced block instructions. Figure 7b shows a secret-dependent branch, where the else branch is a copy of the if-branch body with inactive instructions. Here, SecDivCon copies the body of the secret-dependent branch to a new `else` body, which contains all the operations of the original block but assigned to unused variables (lines 5-7).

Power Attacker (PA): Previous work has shown that high-level compiler optimizations may break software masking against PSC attacks [5, 65]. For example, Figure 2b shows the result of high-level compiler optimizations (-O1 to -O3) on masked code. The code performs first the xor operation between the public value `pub` and the secret value `key` (line 2) and then, performs the second xor operation with the result of the first and the `mask`. Performing the operations in this order fails to randomize the secret value and leads to a PSC leak at line 2. To mitigate this type of transformation, SecDivCon transforms the code to the original operand order (see Figure 2a).

3.2. Security Constraint Model

SecSolv (see Figure 6) takes as input the data from the SA module and applies constraints in order to generate SCA-aware code. First, we will give an overview of a combinatorial compiler backend (Section 3.2.1) and then proceed with the SCA-aware model (Section 3.2.2).

3.2.1. Constraint-based Compiler Backend

We consider a constraint-based compiler backend that implements two low-level optimizations, instruction scheduling and register allocation [38]. A constraint model defines all legal instruction orders and register assignments [37]. More formally, a constraint-based compiler backend may be modeled as a Constraint Optimization Problem (COP),

$P = \langle V, U, C, O \rangle$, where V is the set of decision variables of the problem, U is the domain of these variables, C is the set of constraints among the variables, and O is the objective function. A constraint-based compiler backend aims at minimizing O, which typically models the execution time or size of the code.

A program is modeled as a set of basic blocks B. Each basic block contains a number of optional operations that may be *active* or not. An active operation appears in the final generated binary code, whereas inactive operations are not present in the final code. A set of hardware instructions may implement each operation that consists of a number of operands. Each operand may be implemented by different, equally-valued virtual registers, which are the result of copying the content of a register to another register or memory (copies). The model maps each virtual register to a set of hardware registers and memory locations. The solver assigns each virtual register with one hardware register or memory location. Every assignment p of the problemc variables that satisfies the constraints, C, is a solution to P, $p \in sol(P)$ and represents a compiled program.

A typical objective function of a constraint-based backend minimizes different metrics such as *code size* and *execution time*. These can be captured in a generic objective function that sums up the weighted cost of each basic block:

$$\sum_{b \in B} weight(b) \cdot \mathbf{cost}(b).$$

The **cost** of each basic block is a variable that differs among solutions, whereas *weight* is a constant value that represents the contribution of the specific basic block to the total cost. This cost model is accurate for predictable hardware architectures, such as microcontrollers. These architectures do not include cache hierarchy, dynamic branch prediction, and/or out-of-order execution, which reduce predictability.

3.2.2. Side-Channel Mitigation Constraints

The constraint-based solver aims at optimizing code given an accurate cost model for predictable microcontrollers. However, SecDivCon aims at generating SCA-secure code. Given the constraint problem $P = \langle V, U, C, O \rangle$ that describes the combinatorial compiler backend, we extend the constraints C, with a set of constraints C_{sec} that capture the properties of the SCA mitigations. Then, the problem becomes $P_{sec} = \langle V, U, C \cup C_{sec}, O \rangle$ and the goal is to find the solution that optimizes the cost function, O, while satisfying all constraints. The following paragraphs describe briefly the constraints for the two attacker models.

Timing Attacker (TA): For TA, the SA module generates a list of sets of paths, $paths_{sec}$. Each element in the list contains the set of possible paths starting from one secret-dependent branch. To generate the set of paths SA applies a path-finding algorithm (see Section A).

The constraints that guarantee the preservation of the CR policy are based on the paths ($paths_{sec}$) that SA provides to the solver. For each set of paths that depends on a secret value, we define a constraint `balance_blocks`, which guaranties that all paths in the set have the same execution time.

`balance_blocks`($paths_{sec}$):
$\forall p_1, p_2 \in paths_{sec}. \sum_{b \in p_1} \text{cost}(b) = \sum_{b \in p_2} \text{cost}(b)$

In particular, for each set of paths that depend on a `secret` value (sec_i), we apply the `balance_blocks` constraint, i.e. $\forall paths_{sec_i} \in paths_{sec}$. `balance_blocks`($paths_{sec_i}$). In this case, we have one security constraint, i.e. $C_{sec} =$ {`balance_blocks`}.

Power Attacker (PA): The model against PSCs depends on the constraint model in previous work [65]. This model focuses on two leakage sources, namely, ROT and Memory-Remnant Effect (MRE).

ROT leakages occur when there is a value transition in a hardware register, namely when a new value replaces the previous value of a register. When this transition depends on a secret value, we have a secret leak. The constraint model enforces the absence of these leaks in the generated code by constraining register allocation. More specifically, for ROT leaks, SA generates all pairs of intermediate variables, $(t_1, t_2) \in RPairs$, that should not be assigned to the same register ($r(t)$) subsequently (`subseq`).

`conflict_rassign`($RPairs$):
$\forall t_1, t_2 \in RPairs.\ r(t_1) = r(t_2) \implies \neg\text{subseq}(t_1, t_2)$

Similarly, MRE corresponds to a leak when there is a secret-dependent transition at the memory bus, i.e. when a load or store operation overwrites the previous value in the bus. The constraints that ensure secure code generation are similar with those against ROT and enforce the instruction order of memory operations. In particular, for MRE leaks, SA generates all pairs of memory operations, $(o_1, o_2) \in MPairs$, that should not be scheduled one after the other (`msubseq`).

`conflict_order`($MPairs$):
$\forall o_1, o_2 \in MPairs.\ \neg\text{msubseq}(o_1, o_2)$

In this case, we have two security constraints that protect against ROT and MRE leaks, i.e. $C_{sec} =$ {`conflict_rassign`, `conflict_order`}.

3.3. Secure Code Diversification

Constraint-based diversification [27, 29] aims at generating different solutions for a given problem rather than one solution. For optimization problems, there is often the requirement to generate good or optimal solutions with regard to the optimality function, O. Constraint-based diversification defines the notion of *distance measure*, δ, which is a constraint between problem solutions and measures how *different* two solutions of the problem are.

3.3.1. Diversification Problem

Given our SCA-aware optimization problem $P_{sec} = \langle V, U, C \cup C_{sec}, O \rangle$, the diversification problem attempts to find the set of distinct solutions S that are solutions of $P'_{sec} = \langle V, U, C \cup C_{sec} \rangle$ and the distance between the

solutions satisfies δ, i.e. $S = \{p \mid p \in sol(P'_{sec}) \wedge \forall p' \in S \,.\, p' \neq p \implies \delta(p, p')\}$.

To generate the set of diverse programs S, SecDiv (see Figure 6) takes the best solution from the code generation part and generates multiple solutions using the security-aware constraint model (SecSolver in Figure 6), similar to previous work [64]. In particular, SecDiv generates solutions in the neighbourhood of this solution that satisfy $C_{opt} = O < g \cdot o$, where g is the maximum allowed optimality gap and o the cost of the best-found solution.

3.3.2. Diversifying Transformations

The diversifying transformations that SecDiv supports are 1) hardware register assignment, 2) register copying, 3) memory spilling, 4) constant rematerialization, 5) instruction order, 6) NOP insertion, and 7) operand order in two-address instructions. Hardware register assignment permits changing the register assignment for instruction operands. Register copying enables copying the content of a register to another register for future uses of the register value. Memory spilling allows copying values from a register to the stack and from the stack to a register. Spilling affects the size of the stack and thus leads to stack size diversification, however, spilling increases execution-time overhead. Rematerialization allows re-executing an instruction instead of copying its result. SecDiv may also alter the instruction order as long as there are not data dependencies and insert NOP instructions by delaying the issue cycle of an instruction. Finally, SecDiv may alter the operand order in two-address instructions.

4. Evaluation

The evaluation of SecDivCon consists of three parts, 1) evaluation of the CR-preserving code generation that we propose in this paper, 2) evaluation of introduced leaks in mitigated source code by current diversification tools, and 3) evaluation of SCA-aware code diversification. Section 4.1, describes the implementation, experimental setup, and benchmarks, while Sections 4.2-4.5 present the evaluation of SecDivCon.

4.1. Evaluation Setup

In the following parts, we present the implementation, experimental setup and benchmarks we use to evaluate SecDivCon.

4.1.1. Implementation

We implement SecDivCon as an extension of Unison [37], a combinatorial compiler backend that uses Constraint Programming (CP) [54] to optimize software functions. To do this, Unison combines two low-level optimizations, instruction scheduling and register allocation, and achieves optimizing medium-size functions with improvement over LLVM [37]. SecDivCon takes as input the function in LLVM's Machine Intermediate Representation (MIR) and outputs multiple versions of the function that satisfy the compiler and security constraints. For generating PSC-free code, we adapt the model of SecCG [65], which generates optimal code that is secure against ROT and MRE leakages. For generating CR code, we implement the path-extraction algorithm (see Section A) in Haskell as part of Unison's presolving process. We implement the path-balancing constraints (see Section 3.2.2) as part of the constraint model, which is written using the Gecode C++ library [25]. SecDivCon combines the SCA-aware mitigations with a diversification scheme [64] to generate multiple function variants. We target two architectures, 1) a generic Mips32 processor and 2) the ARM Cortex M0 processor [4].

Table 1
Benchmark description; N_i is the number of machine instructions that are input to the compiler backend; N_b is the number of basic blocks; A stands for ARM Cortex M0; and M stands for Mips; I_p, I_s, and I_r is the number of public, secret, and random input arguments, respectively

Prg	Description	N_i		N_b		Input Vars		
		A	M	A	M	I_p	I_s	I_r
P0	SecXor	7	7	1	1	1	1	1
P1	AES Shift Rows	8	8	1	1	0	2	2
P2	Messerges Boolean	11	11	1	1	0	1	2
P3	Goubin Boolean	13	13	1	1	0	1	2
P4	SecMultOpt_wires	18	18	1	1	1	1	3
P5	SecMult_wires	18	18	1	1	1	1	3
P6	SecMultLinear_wires	19	19	1	1	1	1	3
P7	CPRR13-lut_wires	48	48	1	1	1	1	7
P8	CPRR13-OptLUT_wires	48	48	1	1	1	1	7
P9	CPRR13-1_wires	52	52	1	1	1	1	7
P10	Whitening	113	88	1	1	16	16	16
C0	If check (Figure 4)	10	9	3	3	1	1	-
C1	Share's Value	23	26	6	5	1[a]	2[a]	-
C2	Mult. Modulo 8	28	24	8	6	1	1	-
C3	Modulo Exponentiation	51	36	7	7	1	2	-
C4	Kruskal	51	55	9	9	1[a]	3[a]	-

[a]The input is an address to an array of secret values

4.1.2. Experimental Setup

All experiments run on an Intel®Core™i9-9920X processor with maximum frequency 3.50GHz per core and 64 GB of RAM running Debian GNU/Linux 10 (buster). We use LLVM-3.8 as the front-end and middle-end compiler for these experiments. We repeat all experiments five times, with different random seeds (where applicable) and report the mean value for each metric in the results.

4.1.3. Benchmarks

Our approach concerns programs that handle secret information and are, thus, vulnerable to SCAs. Therefore, we have selected eleven masked cryptographic core functions that may be vulnerable to PSCs [65, 67] and five functions that exhibit secret-dependent timing variations and are used in cryptographic context [39]. Table 1 shows the benchmarks with information about the origin of the function, the function size in number of instructions (N_i) and the number of basic blocks (N_b) for ARM Thumb (A) and Mips (M),

Table 2
Optimality overhead in cycles for CR-preserving code-generation; 🔒 denotes secure variants and 🔓 non-secure variants; Oh stands for Overhead

Prg	ARM Cortex M0			Mips32		
	Cycles		Oh (%)	Cycles		Oh (%)
	🔒	🔓		🔒	🔓	
C0	26	20	30	13	10	30
C1	1406	1220	15	1105	857	28
C2	1039	803	29	975	571	70
C3	3012	1984	51	7641	5843	30
C4	16130	13590	18	10429	8905	17

Table 3
Compilation-time overhead in seconds for CR-preserving code generation; 🔒 denotes secure variants and 🔓 non-secure variants; Oh stands for Overhead

Prg	ARM Cortex M0			Mips32		
	t (s)		Oh (%)	t(s)		Oh (%)
	🔒	🔓		🔒	🔓	
C0	0.32	0.19	68	0.81	0.55	47
C1	1.68	0.91	84	4.42	2.56	72
C2	8.48	0.81	946	2.65	1.33	99
C3	57.26	23.46	144	8.02	4.97	61
C4	150.80	92.72	62	27.52	7.93	247

and finally, the input variables, (I_p public, I_s secret, and I_r random input variables).

Masked Programs: The masked programs that we use in this evaluation consist of eleven programs, P0 to P10, most of which originate from the work by Wang et al. [67]. These benchmark programs consist of masked cryptographic core functions that are vulnerable to PSC attacks.

CR Programs: For evaluating the CR property we use Listing 4 and four benchmark programs used by Mantel and Starostin [39]. The code for these benchmarks in C including security-policy annotations is available by Winderix et al. [68]. These implementations are vulnerable to timing attacks [39].

4.2. Effectiveness and Efficiency of TSC-Aware Code Generation

This section evaluates the CR-preserving code generation in three dimensions, 1) performance overhead, 2) compilation overhead, and 3) security.

4.2.1. Performance Overhead

The CR-preserving code generation extends Unison [37] with constraints that enforce the CR property. For optimizing code against TSCs, SecDivCon optimizes the generated code given the compiler-backend constraints and the newly introduced security constraints. Generating CR-preserving programs introduces performance overhead due to the introduction of new basic blocks and/or NOP padding for balancing secret-dependent branches. To estimate the overhead on the generated code, we utilize the cost model

Table 4
Security Evaluation using a WCET tool to compare the execution time: T denotes a symbolic value, υ denotes a set of concrete values, and a_i corresponds to the i_{th} input argument

Prg	ARM Cortex M0		Mips32	
	Input	🔒	Input	🔒
C0	a0,a1 =T	✓	a0,a1 = T	✓
C1	a0,a1,a2,a3 = T[a]	✓	a0,a1,a3 = T, a2 = υ	✓
C2	a0,a1,a2,a3 = T	✓	a0,a1,a2,a3 = T	✓
C3	a0,a1,a2,a3 = T[a]	✓	a0,a1,a3 = T, a2 = υ	✓
C4	a0,a1,a2,a3 = υ[a,b]	✓	a0,a1,a2 = T, a3 = υ	✓

[a]Verified only the secret-dependent branches to improve scalability and accuracy
[b]The concrete values correspond to addresses of the inputs

(see Section 3.2.1) of the constraint-based compiler backend [37]. Table 2 shows the performance overhead of the CR-preserving code generation backend of SecDivCon (🔒) compared to Unison that is not security aware (🔓). SecDivCon has a maximum overhead of 70% over Unison for C2. The introduced overhead is due to the introduction of new basic blocks and the extension of other basic blocks in order to balance secret-dependent execution paths. In contrast to the CR-preserving code generation, PSC-aware code generation does not introduce significant execution-time overhead [65]. One reason for this is that the CR policy affects directly the execution time of the generated code because it enforces secret-dependent block balance by increasing the execution time of all secret-dependent paths to reach the longest path.

4.2.2. Compilation Overhead

The introdution of new constraints to satisfy the constant-resource property in the constraint model may lead to increased compilation time compared to non-secure compilation in Unison. To evaluate the compilation-time overhead, we compare the compilation time of SecDivCon with Unison measuring the solving time. Table 3 shows the compilation-time overhead of SecDivCon (🔒) compared to Unison (non-CR-preserving code optimization) [37] (🔓). For ARM Cortex M0, the compilation time is at most ten times slower in SecDivCon compared to Unison for C2. For Mips, we observe lower slowdown up to 3.5 times for C4. PSC-aware code generation [65] demonstrates a similar difference in the compilation-time slowdown between the two architectures. Here, the introduced compilation-time slowdown is mainly due to the introduced constraints for balancing the cost of different paths, which introduces inter-block dependencies that delay the solving process. At the same time, we notice larger absolute compilation times for ARM cortex M0 than for Mips. This is due to the characteristics of the ARM Thumb architecture compared to Mips32, including a smaller number of general-purpose hardware registers and two-address instructions.

4.2.3. Security Evaluation

SecDivCon uses a constraint model to generate secure variants. To verify the effectiveness of SecDivCon against timing side channels, we use two Worst-Case Execution Time (WCET) tools for the two architectures we are investigating. WCET is typically a sound overapproximation of the execution time of the program, whereas Best-Case Execution Time (BCET) is a sound underapproximation of the execution time. For Mips, we use KTA [11, 63]. KTA is a tool that extracts the best- and worst-case execution time for a binary program. For evaluating ARM Cortex M0, we use a symbolic-execution-based WCET tool[1] that generates the WCET and BCET for a sequence of binary instructions [36]. To verify that SecDivCon generates CR programs, we test the generated binaries using a WCET tool. We give as inputs symbolic values that range over all integer values ($\top$) for `secret` values and `public` values that do not affect the control flow, whereas for `public` values that affect the control flow (e.g. loop bounds), we provide concrete values. If the returned BCET and WCET are equal, then we have evidence that the program's execution time is secret independent for the given concrete inputs. Performing the same experiment using multiple concrete inputs gives an indication that the program satisfies the CR property. More specifically, we compare the WCET and the BCET of the function for different concrete values of the public inputs. If $\forall p \in IN_{test}.wcet_p = bcet_p$ for all concrete public inputs, IN_{test}, we say that the program is constant resource modulo inputs[2]. For each of the benchmark programs, Table 4 shows the type of input value we use (Input) and the result of the comparison between WCET and BCET (🔒). For the experiment, we provide different values for the concrete value v. Symbol ✓ denotes that the experiments for all inputs result in the same WCET and BCET. The result of this experiment indicates that the generated code does not violate the CR property.

4.3. Effect of Code Diversification on Side-Channel Mitigations

We investigate how diversification approaches affect side-channel mitigations. In particular, we investigate to what extent a freely-available[3] code diversification tool, Multicompiler (MCR) [28] violates software mitigations against SCAs. To do that, we use MCR to diversify benchmark programs that implement security mitigations against SCAs at source-code level. Then we verify whether the generated program variants (for the respective benchmarks) satisfy the software mitigations against PSC or TSC attacks. For PSC, we use a tool[4] by Wang et al. [67], whereas for TSC, we measure the execution time manually. For these experiments, we generate 50 random variants by providing 50 different random seeds to MCR. MCR supports randomization at multiple layers of the compilation process, including hardware-register randomization and NOP-insertion. These randomizing transformations may affect PSC and TSC mitigations, respectively. In the following paragraphs, we investigate how hardware-register randomization affects ROT leakages and how NOP-insertion affects the CR property.

Table 5
Rate of variants that contain ROT vulnerabilities in MCR

Prg	[67]	MCR	
	#leaks	#leaks (% of variants)	≥ one leak
P0	1	1 (62%) 0 (38%)	62%
P1	0	2 (92%) 1 (2%) 0 (6%)	94%
P2	1	2 (100%)	100%
P3	1	1 (70%) 0 (30%)	70%
P4	1	1 (100%)	100%
P5	1	1 (96%) 0 (4%)	96%
P6	3	4 (52%) 5 (48%)	100%
P7	14	14 (100%)	100%
P8	16	16 (100%)	100%
P9	12	12 (100%)	100%
P10	5	5 (100%)	100%

Hardware-Register Randomization: Hardware-register randomization [18] is a form of fine-grained software diversification that generates program variants that differ with regard to the register assignment at the register-allocation stage of the compilation process. Among other transformations, MCR implements hardware-register randomization. To identify the number of ROT leaks of each of the variants, we implement parts of the tool by Wang et al. [67] to extract information from the register allocation step in MCR. Subsequently, we use the tool by Wang et al. [67] to identify the leaks in the variants.

For each of the masked benchmarks, Table 5 shows the number of leaks that appear in the baseline, which uses the LLVM compiler [67] and the rate of variants that contain different numbers of leaks after diversification with MCR. The last column shows the rate of variants that have at least one leak. Overall, there are leaking variants in all programs, ranging from 62% for P0 and 100% for P2, P4, P6-P10. For programs P0 to P6, the number of leaks differs for the generated variant population. In particular, MCR may introduce leaks in P1, P2, and P6 that the baseline does not generate. Inversely, MCR may generate variants that are leak-free for P0, P1, P3, and P5. This means that the hardware-register randomization transformation allows the generation of leak-free variants.

To summarize, we observe that randomization may break masking mitigations, whereas, in many cases, there is a space for generating leak-free variants.

NOP Insertion: NOP insertion is a form of fine-grained software diversification that generates program variants that

[1] CM0 WCET: https://github.com/kth-step/HolBA/tree/dev_symbexec_form

[2] Note that possible overapproximations of the WCET or underapproximations of the BCET may lead to inequality of BCET and WCET, regardless of the program satisfying the CR property.

[3] MCR: https://github.com/securesystemslab/multicompiler.git

[4] FSE19 tool: https://github.com/bobowang2333/FSE19

Table 6
Number of variants (N) and diversification time (t) in seconds for SCA-aware (🔒) and non SCA-aware (🔓) diversification in ARM Cortex M0 and Mips32; TO stands for time limit (ten minutes); SecDivCon controls the execution-time overhead, here we show the results for a maximum execution-time overhead of 0% and 10%.

Prg	ARM Cortex M0								Mips32							
	0%				10%				0%				10%			
	🔒		🔓		🔒		🔓		🔒		🔓		🔒		🔓	
	N	t (s)	N	t (s)	N	t (s)	N	t (s)	N	t (s)	N	t (s)	N	t (s)	N	t (s)
P0	1	-	2	0.00	8	8.09	18	150.88	17	0.03	18	0.01	17	0.03	18	0.01
P1	5	0.17	16	0.05	109	194.53	200	9.47	200	0.36	200	0.12	200	1.70	200	0.26
P2	2	0.00	2	0.00	84	397.67	65	196.98	200	0.44	200	0.12	200	3.65	200	0.41
P3	39	217.16	9	0.17	200	73.97	200	15.90	200	0.70	200	0.20	200	4.98	200	0.51
P4	200	28.83	200	2.70	200	27.21	200	2.09	200	85.91	200	3.03	200	74.03	200	3.80
P5	200	28.76	200	2.71	200	27.36	200	2.10	200	86.31	200	3.05	200	73.48	200	3.78
P6	200	31.18	200	2.48	200	29.01	200	2.25	200	134.76	200	3.75	200	215.73	200	3.94
P7	51	TO	200	18.60	51	TO	200	22.69	40	TO	200	20.32	32	TO	200	55.32
P8	69	TO	200	15.47	58	TO	200	20.55	47	TO	200	20.40	34	TO	200	65.83
P9	185	TO	200	18.16	165	TO	200	23.24	6	TO	200	306.09	8	TO	200	171.21
P10	53	TO	200	23.27	20	TO	200	35.28	36	TO	200	16.44	15	TO	200	17.34
C0	200	0.65	4	41.15	200	0.38	162	247.78	200	0.32	19	0.04	200	0.46	200	1.77
C1	200	1.22	200	1.69	200	2.96	200	2.76	200	0.88	200	0.39	200	7.66	200	5.47
C2	200	0.53	200	0.25	200	2.51	200	1.51	200	0.99	200	0.43	200	4.43	200	1.66
C3	200	6.39	200	5.24	200	8.55	200	8.53	200	4.07	200	2.69	200	22.09	200	19.88
C4	200	14.11	200	10.25	200	27.53	200	17.19	200	10.27	200	7.85	200	27.87	200	18.97

contain randomly inserted NOP operations. MCR implements NOP-insertion randomization [28]. The source code of programs C0 to C3 does not comply with the CR policy. To identify CR violations, we consider C0, C1 and C3 because they are simple to verify manually. We modify the C implementations of C0, C1 and C3 to balance the secret-dependent branches and consider a simple timing model for the processor that considers one cycle per instruction. The results are that 88% of C0, 74% of C1, and 72% of C3 are unbalanced. MCR inserts NOP operations randomly without information about secret balancing and, thus, generates non-CR-preserving code[5]. To summarize, NOP insertion may break branch balancing for CR programs.

4.4. SCA-Mitigation Effect on Code Diversification

To evaluate the effect of SCA mitigations on code diversification, we compare the effect of SCA-aware diversification with SCA-unaware diversification. We evaluate SecDivCon in two axes, 1) diversity and 2) diversification scalability.

Table 6 shows the number of variants (N) and the diversification time in seconds (t(s)) for each of the benchmarks and each of the configurations of the diversification experiments. The diversification time consists of the time it takes to generate diverse program variants given an initial optimized solution. We use a time limit of ten minutes. In addition, we use upper bound (200) on the number of variants, because of the increasing complexity of the pairwise gadget-overlap rate (see Section 4.5) that depends on all pairs of generated variants. For each of the two architectures, ARM Cortex M0 and Mips32, we perform SCA-aware (🔒) diversification and SCA-unaware (🔓) diversification using 0% (optimal based on the cost model) and 10% optimality gap. The optimality gap, p, depends on the cost model of the combinatorial compiler backend and the input best-found solution. The optimality gap results in a constraint that ensures that the cost of each generated variant is at most p% worse than the best-found solution.

In the upper part of Table 6, we see that for ARM Thumb there is limited diversity for small benchmarks (P0-P3), especially when restricting the solutions to the optimal/best-found ones (0% optimality gap). Increasing the optimality gap to 10% enables SecDivCon to generate a larger number of program variants. For both cases, the presence of PSC-mitigating constraints reduces the number of available variants. The opposite occurs for P3, where SecDivCon is able to generate more variants compared to PSC-unaware code diversification. This is due to the introduction of additional transformations (random variable copies) in PSC-aware compilation, which increases the search space and diversification ability of SecDivCon.

For larger benchmarks (P4-P6), SecDivCon is able to generate all the requested variants (200). Looking at the diversification time of these benchmarks, we notice a clear overhead of PSC-aware compared to PSC-unaware diversification. The overhead is up to a slowdown of 55 times for P6 in Mips32. For the largest benchmarks, P7-P10, SecDivCon reaches the time limit (TO) and the number of generated variants is significantly less than for the PSC-insecure variant generation. Interestingly, increasing the optimality gap to 10% decreases the number of generated variants. As we see in small benchmarks, increasing the optimality gap allows for non-optimal (according to the model) solutions,

[5]Here, we do not investigate multi-variant execution, where different variants are loaded dynamically, which may hinder timing attacks by randomizing the execution time.

which increases the available variants. However, increasing the optimality gap, increases also the search space, which increases the solver overhead for locating solutions. This results in a reduction of the generated solutions.

We observe similar trends for both Mips32 and ARM Thumb. The main difference is that among small benchmarks, only P0 with 0% optimality gap appears to lead to reduced diversity in Mips32. At the same time, the difference in diversity between secure and non-secure variants is smaller in Mips32 (17 compared to 18 in P0) than for ARM Thumb (1 compared to 2 in P0). The reason for this is that Mips32 provides a larger number of general-purpose registers that may replace vulnerable register combinations for ROT leakages.

The lower part of Table 6 shows the results for TSC-aware diversification. Here, SecDivCon is able to generate 200 function variants for all benchmarks. Interestingly, for C0, the number of variants for TSC-unaware diversification is less than 200 because our CR mitigation introduces performance overhead (see Section 4.2) and thus, increased diversification capacity. The diversification-time overhead is less than for PSC-aware diversification, reaching up to a slowdown of eight times (C0, 0% optimality gap, Mips32). In all cases, SecDivCon was able to generate 200 variants in less than 30 seconds.

To summarize, we observe a clear effect on the diversification time and available diversity in SecDivCon compared to SCA-unaware code diversification. This effect is more significant in PSC-aware diversification, where there is a general decrease in diversity and increase in the diversification-time slowdown. TSC-aware diversification appears to affect mainly diversification time, whereas in some cases, the CR countermeasure increases the available diversity. Nonetheless, in almost all cases, SecDivCon generates program variants within ten minutes.

4.5. Effect of Security Constraints on Code-Reuse Attacks

This section evaluates the effect of SCA-aware diversification on the effectiveness against CRAs. To evaluate the effectiveness of SecDivCon against CRAs, we measure the rate of code-reuse gadgets that are relocated or transformed among different variants. We perform this evaluation at the generated binary ELF [22] files. This evaluation uses ROPgadget[6], a tool that extracts code-reuse gadgets from a binary and Capstone, a lightweight disassembly framework. We extract the gadgets from the `.text` section of the generated ELF files. Similarly to previous work [28, 48], we assess the gadget-overlap rate $srate(p_i, p_j)$ for each pair of variants $p_i, p_j \in S$ in the set of generated variants, S, to evaluate the effectiveness of SecDivCon against CRAs. This metric returns the rate of the gadgets of variant p_i that appear at the same address in the second variant p_j. The procedure for computing $srate(p_i, p_j)$ is as follows: 1) run ROPgadget on variant p_i to find the set of gadgets $gad(p_i)$ in variant p_i, and 2) for every $g \in gad(p_i)$, check whether there exists a gadget identical to g at the same address in the second variant p_j. Before the comparison, we remove all NOP instructions. The smaller the *srate* is, the fewer gadgets are shared among program variants, and thus, the highest the effect against CRAs. Note that *srate* does not check the semantic equivalence of the gadgets, and hence, there may be false negatives, namely pairs of gadgets that are syntactically different but semantically equivalent. We use a time limit of ten minutes and an upper bound on the number of variants to generate because of the increasing complexity of the pairwise gadget-overlap rate that depends on all pairs of generated variants.

Table 7 shows the rate of shared code-reuse gadgets among the generated variants for ARM Cortex M0 and Mips32. For each processor, Table 7, shows the results for two configurations that allow variants to introduce at most 0% to 10% execution-time overhead. We compare SCA-aware variants (🔒) and SCA-unaware variants (🔓). For each of these cases, Table 7 shows the *srate*, i.e. rate of pairs of variants, in the form of a histogram with three buckets. The buckets represent the rate of variant pairs that share 1) 0% of their gadgets (0 in Table 7), 2) (0%, 20%] of the gadgets (20 in Table 7), or 3) (20%, 100] of the gadgets (100 in Table 7). The goal of SecDivCon is to generate variants that share as few gadgets as possible, i.e. the variant pairs share no gadgets (0 in Table 7).

In Table 7, we observe a general difference between the two processors, with SecDivCon achieving lower gadget survival rate for Mips32 than ARM Cortex M0. We describe the results for the two processors in the following.

In ARM Cortex M0, with 0% allowed execution-time overhead, for both SCA-aware and SCA-unaware diversification, the mode of the pairwise survival rate for the majority of the benchmarks lies within (0%, 20%]. For SCA-aware diversification for 13 benchmarks the mode of the distribution is under (0%, 20%] and for two is 0% (P10 and C4). The results for SCA-unaware diversification are similar, with P9 having improved gadget elimination and P10, C0, C3, and C4 having reduced gadget-elimination ability than SecDivCon. Increasing the optimality gap to 10% results in reduced survival rate (improvement). In particular, for SCA-aware diversification, five benchmarks have a distribution with the mode in 0%, ten have their mode under (0%, 20%], and one under (20%, 100%]. Here, the results for SecDivCon are similar to SCA-unaware diversification, with C0 showing better results in SCA-unaware diversification.

In contrast, for Mips32, most experiments (apart for C0, C1, and C3 with 0% optimality gap) have their mode under 0% survival rate, which means that the majority of variant pairs do not share any gadgets. The reason why Mips32 appears to achieve higher gadget relocation/diversification is the characteristics of the architecture with many general purpose registers. ARM Cortex M0, on the other hand, has significantly fewer general-purpose hardware registers and multiple 2-address instructions that are highly constrained.

To summarize, the results show relatively low gadget survival rate for both ARM Cortex M0 and Mips32, whereas,

[6]ROPgadget: https://github.com/JonathanSalwan/ROPgadget

Table 7
CRA gadget-overlap rate in pairs of variants; 🔒 denotes secure variants and 🔓 non-secure variants

Prg	ARM Cortex M0												Mips32											
	0%						10%						0%						10%					
	🔒			🔓			🔒			🔓			🔒			🔓			🔒			🔓		
	0	20	100	0	20	100	0	20	100	0	20	100	0	20	100	0	20	100	0	20	100	0	20	100
P0	-	-	-	-	-	**100**	23	-	**77**	21	-	**79**	**100**	-	-	**100**	-	-	**100**	-	-	**100**	-	-
P1	-	**53**	47	-	**78**	23	14	**55**	31	12	**71**	16	**89**	-	11	**94**	-	6	**95**	5	-	**94**	6	-
P2	-	**100**	-	-	**100**	-	3	**66**	31	5	**64**	32	**93**	6	1	**94**	6	1	**97**	3	-	**97**	2	-
P3	-	**71**	29	-	**73**	27	2	**83**	15	11	**81**	9	**99**	1	-	**99**	1	-	**99**	1	-	**99**	1	-
P4	-	**86**	14	-	**75**	25	5	**87**	7	10	**83**	7	**100**	-	-	**100**	-	-	**99**	-	-	**99**	1	-
P5	-	**86**	14	-	**75**	25	5	**87**	7	10	**83**	7	**100**	-	-	**100**	-	-	**99**	-	-	**99**	1	-
P6	-	**90**	10	-	**87**	13	8	**86**	6	15	**80**	4	**100**	-	-	**100**	-	-	**99**	1	-	**99**	1	-
P7	1	**92**	8	-	**95**	5	6	**87**	7	7	**88**	5	**98**	2	-	**100**	-	-	**99**	1	-	**100**	-	-
P8	2	**88**	11	-	**93**	6	19	**75**	6	3	**91**	5	**97**	2	1	**100**	-	-	**98**	2	1	**100**	-	-
P9	-	**83**	17	32	**63**	5	10	**85**	5	46	**50**	4	**78**	10	12	**100**	-	-	**95**	1	4	**99**	1	-
P10	**57**	42	2	-	**95**	5	**75**	22	2	**64**	35	1	**79**	19	1	**94**	6	-	**66**	26	8	**99**	1	-
C0	36	**61**	3	-	45	**55**	43	**54**	3	**55**	30	15	29	**68**	2	**95**	-	5	**80**	18	2	**86**	14	-
C1	34	**66**	-	-	**100**	-	**95**	4	1	**93**	6	1	29	**71**	-	41	**59**	-	**92**	5	2	**95**	3	2
C2	-	**69**	31	-	**68**	32	**42**	37	21	**53**	33	13	**90**	10	-	**82**	18	-	**92**	8	-	**80**	20	-
C3	18	**56**	26	-	1	**99**	**93**	6	2	**93**	4	3	8	**92**	-	-	**100**	-	**97**	1	2	**96**	3	2
C4	**57**	41	1	-	**97**	3	**96**	4	-	**95**	5	-	**94**	6	-	**94**	6	-	**100**	-	-	**99**	1	-

this survival rate does not appear to increase (worse) for SCA-aware diversification. This means that combining SCA mitigations with diversification against CRAs does not reduce the mitigation capability of fine-grained diversification against CRAs.

5. Discussion

This section discusses the application of SecDivCon against more advanced attacks and the potential extension of SecDivCon to support additional mitigations.

Whole-Program Mitigation: Our threat model considers static gadget-based code-reuse attacks, such as ROP attacks [58]. SecDivCon proposes a fine-grained function-level diversification approach as a mitigation against these attacks. Combining the generated variants for each function allows for whole-program diversification [64]. Return-into-libc (RILC) [62] attacks where the gadgets correspond to entire functions may be defeated by combining whole-program diversification with function shuffling and/or coarse-grained diversification approaches, such as Address Space Layout Randomization (ASLR).

Advanced Attacks: Advanced code-reuse attacks, such as Blind ROP (BROP) [7], may use a memory vulnerability to read the program memory and find gadgets dynamically in the diversified code. BROP attacks read the program memory using a memory vulnerability and depend on the reset of the system after a system crash. An efficient approach against BROP is re-randomization [60] that may be performed at boot time [49]. Runtime re-randomization switches program variants at runtime at an interval within which the attacker should not be able to complete an attack. The main drawbacks of re-randomization is that 1) it may lead to high memory footprint for the binary [15], which may be forbidding in resource-constrained devices, and 2) it contributes to additional performance overhead. Nonetheless, SecDivCon performs fine-grained automatic diversification that may be used in a re-randomization scheme, enabling improved protection against advanced code-reuse attacks.

Apart from classical power analyses, such as DPA and CPA, recently, the advancement of deep learning has allowed more powerful attacks. Ngo et al. [43] show that advanced randomization techniques, such as plaintext shuffling, are vulnerable [43, 42], when the implementation leaks secret values. They also show that first-order masking can be defeated with deep-learning based analysis, however, the masking property is preserved at the source-code level, thus ROT or MRE leakages may be present after compilation [5]. We leave the evaluation of our approach against these attacks as future work.

Implement Additional Mitigations: SecDivCon combines code diversification and side-channel attack mitigations to protect embedded devices. However, additional mitigations may be necessary to protect a device against other types of attacks. An essential step for combining different mitigations is to determine whether these mitigations are conflicting. In case they are, the designer may describe the new mitigations as constraints to extend the constraint model of SecDivCon. This allows SecDivCon to generate secure code.

6. Related Work

This section presents the related work with regards to Code-Reuse Attacks and Side-Channel Attacks. Table 8

Table 8
Related work; CRA stands for code-reuse attacks; TSC stands for timing side-channel attacks; MS stands for memory safety; PSC stands for power side-channel attacks; IL stands for interrupt-latency SCA; Div stands for diversification; Obf stands for obfuscation; CFI stands for control-flow integrity; CT stands for constant-time discipline; SM stands for software masking; BB stands for basic-block balance; RR stands for re-randomization; HWCFI stands for hardware-assisted CFI; PO corresponds to the upper bound of the performance overhead; SDC stands for SecDivCon.

Pub.	Attack	Mitigation	PO	Target
[28]	CRA	Div	25%	x86
[48]	CRA	Div	0%	x86
[17]	TSC	Div	8x	x86
[52]	TSC	Obf	16x	x86
[50]	CRA	Div, RR	-	AVR
[2]	CRA	CFI	~80%	ARM
[45]	CRA	CFI	5x	ARM
[70]	TSC, MS	CT	-	C
[34]	CRA	Div, RR	7%	x86
[55]	CRA	Div,CFI	70%	ARM
[59]	PSC	SM	64%	ARM
[68]	TSC, IL	BB	60%	MSP430
[60]	CRA	Div, RR	6%	ARM
[9]	TSC	CT	5x	x86
[23]	CRA	HWCFI	24%	ARM
SDC	TSC, PSC, CRA	Div, SM/BB	70%[a]	Mips, ARM

[a]Diversification overhead is controlled

shows a representative subset of compiler-based or binary-rewrite contributions against CRAs and SCAs in the literature. For each of these works, Table 8 shows the publication citation reference (Pub.), the attack it is mitigating (Attack), the type of mitigation the publication is proposing (Mitigation), the maximum performance overhead the approach introduces (PO), and the target language/architecture (Target).

6.1. Mitigations against Code-Reuse Attacks

In the literature, there are two main approaches against CRAs, software diversification and CFI.

Automatic software diversity has been proposed as an efficient mitigation against CRAs [35]. Many software diversification approaches target x86 systems [28, 48, 34], while others target embedded systems [50, 55, 64, 60]. The main characteristic of these approaches is that they lead to relatively low performance overhead. For example, fine-grained diversification approaches may lead to 0% performance overhead [48, 64].

Re-randomization approaches [60, 50, 34] repeat the randomization process in specific timing intervals to protect against advanced CRAs, such as JIT-ROP [61], BROP, and side-channel-based diversification deciphering [57]. These approaches may introduce additional binary-size overhead [15] and performance overhead. However, this performance overhead is typically low, for example, HARM [60] introduces up to 6% additional overhead.

CFI mitigates CRAs by ensuring that the dynamic execution of the program adheres to the intended program control flow [14]. Software-based CFI systems [2, 45, 55] typically result in high overhead, whereas hardware-assisted methods may lead to reduced overhead [23]. However, hardware-assisted CFI approaches often depend on specialized hardware mechanisms [14].

To summarize, there are multiple approaches to mitigate CRAs, but none of them considers or evaluates the effect on mitigations against SCAs. Comparing code diversification and CFI approaches, the former typically lead to lower overhead. This is the main motivation for selecting code diversification as a mitigation against CRAs.

6.2. Code Hardening Against Side-Channel Attacks

Software masking is a software approach to mitigate PSCs. However, a compiler that translates a program to machine code may introduce power leaks [67, 59, 46, 5]. Wang et al. [67] identify leaks in masked implementation using a type-inference algorithm, and then, perform register-allocation transformations to mitigate these leaks in LLVM. Rosita [59] performs an iterative process to identify power leakages in software implementations for ARM Cortex M0, with a performance overhead of up to 64%. Our recent approach [65] based on type inference [24] presents an approach with execution overhead up to 13%. SecDivCon adapts this approach to generate diverse code variants that preserve software masking.

The constant-time programming discipline [41] is a widely-used programming discipline that prevents TSC attacks. It prohibits the use of secret values in branch decisions, memory indexes, and variable-latency instructions (such as division in many architectures). Borrello et al. [9] linearize code to translate a program to a constant-time equivalent including branches, loops, and memory accesses. The main drawback of this approach is the introduction of execution-time overhead of up to five times. The constant-time programming discipline leads to secure code as it ensures that there are no secret-dependent timing variations, however it is restrictive because it does not allow secret-dependent branches and makes the code difficult to read and implement [44]. Barthe et al. [6] present CR programming, an alternative, more relaxed form of constant-time programming that allows branches on secret values as long as the diverse execution paths take identical time to execute. Similarly, Brown et al. [12] perform transformations to balance secret-dependent branches by balancing the branch bodies at the C level. Winderix et al. [68] balance secret-dependent branches with equivalent-latency NOPs to mitigate TSC and Interrupt Latency Side-Channel Attacks. The latter attacks distinguish which path of a branch the program follows based on the latencies of the instructions in each block. A different approach against timing attacks is Raccoon [52], which uses control-flow obfuscation to mitigate TSC attacks. However, Joshi et al. [31] has shown that obfuscation may introduce code-reuse gadgets. Hence,

Raccoon may increase the attack surface of CRAs. Moreover, this mitigation introduces an overhead of up to 16 times, which is prohibiting for resource-constraint devices. Crane et al. [17] present a compiler-based diversification approach that inserts timing noise to obfuscate cache-based timing attacks on cryptographic algorithms. However, this approach introduces a performance overhead of up to 8x, which is higher than SecDivCon that introduces an overhead of up to 70% for generating constant-resource programs.

Finally, HACL* by Zinzindohoué et al. [70] is a verified cryptographic library that generates C code that is memory safe and constant time. Although memory safety hinders memory corruption vulnerabilities in the generated library, HACL* does not prohibit memory vulnerabilities in the rest of the code, which may enable CRAs. Thus, mitigations against CRAs may still be necessary.

In summary, there are compiler-based and binary rewriting approaches to mitigate PSC attacks and TSC attacks, however, none of these approaches are effective against CRAs and/or considers the effect on CRAs.

7. Conclusion and Future Work

This paper presents SecDivCon – a constraint-based approach that is able to combine code diversification with side-channel mitigations. It enables secure-by-design generation of the optimised code for small predictable hardware architectures. Our evaluation shows that the introduction of SCA mitigation-preserving constraints impacts the scalability of diversification but it does not have a negative effect against code-reuse attacks.

As a future work, we plan to investigate how to improve SecDivCon's scalability and extend the CR-preserving model with additional transformations that allow the analysis of secret-dependent branches that contain bounded loops.

Acknowledgment

We would like to thank Jingbo Wang for the support with their tool. We would also like to thank Andreas Lindner for his support with verifying SecDivCon using WCET analysis forq ARM Cortex M0. In addition, we would like to thanks Roberto Castañeda Lozano for his technical support on Unison, SecDivCon's underlying constraint-based compiler backend. Finally, we would like to thank Nicolas Harrand, Amir M. Ahmadian, and Javier Cabrera for their feedback on this paper. P. Papadimitratos acknowledges the support of the Swedish Science Foundation (VR) and the Knut and Alice Wallenberg (KAW) Foundation that funded in parts his work in this context

References

[1] Abera, T., Asokan, N., Davi, L., Ekberg, J.E., Nyman, T., Paverd, A., Sadeghi, A.R., Tsudik, G., 2016a. C-FLAT: Control-Flow Attestation for Embedded Systems Software, in: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 743–754. doi:10.1145/2976749.2978358.

[2] Abera, T., Asokan, N., Davi, L., Ekberg, J.E., Nyman, T., Paverd, A., Sadeghi, A.R., Tsudik, G., 2016b. C-FLAT: Control-Flow Attestation for Embedded Systems Software, in: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 743–754. doi:10.1145/2976749.2978358.

[3] Ahmed, S., Xiao, Y., Snow, K.Z., Tan, G., Monrose, F., Yao, D.D., 2020. Methodologies for Quantifying (Re-)randomization Security and Timing under JIT-ROP, in: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, pp. 1803–1820.

[4] ARM, . Cortex-M0 - Technical Reference Manual. URL: https://developer.arm.com/documentation/ddi0432/c/. accessed: November 2022.

[5] Athanasiou, K., Wahl, T., Ding, A.A., Fei, Y., 2020. Automatic detection and repair of transition-based leakage in software binaries, in: Software Verification. Springer, pp. 50–67.

[6] Barthe, G., Blazy, S., Hutin, R., Pichardie, D., 2021. Secure Compilation of Constant-Resource Programs, in: CSF 2021 - 34th IEEE Computer Security Foundations Symposium, IEEE. pp. 1–12.

[7] Bittau, A., Belay, A., Mashtizadeh, A., Mazières, D., Boneh, D., 2014. Hacking Blind, in: 2014 IEEE Symposium on Security and Privacy, pp. 227–242. ISSN: 2375-1207.

[8] Bletsch, T., Jiang, X., Freeh, V.W., Liang, Z., 2011. Jump-oriented Programming: A New Class of Code-reuse Attack, in: Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, ACM. pp. 30–40.

[9] Borrello, P., D'Elia, D.C., Querzoni, L., Giuffrida, C., 2021. Constantine: Automatic Side-Channel Resistance Using Efficient Control and Data Flow Linearization. Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security , 715–733doi:10.1145/3460120.3484583.

[10] Brier, E., Clavier, C., Olivier, F., 2004. Correlation Power Analysis with a Leakage Model, in: Cryptographic Hardware and Embedded Systems - CHES 2004, Springer. pp. 16–29. doi:10.1007/978-3-540-28632-5_2.

[11] Broman, D., 2017. A Brief Overview of the KTA WCET Tool. doi:10.48550/arXiv.1712.05264. number: arXiv:1712.05264 arXiv:1712.05264 [cs].

[12] Brown, C., Barwell, A.D., Marquer, Y., Zendra, O., Richmond, T., Gu, C., 2022. Semi-automatic ladderisation: improving code security through rewriting and dependent types, in: Proceedings of the 2022 ACM SIGPLAN International Workshop on Partial Evaluation and Program Manipulation, pp. 14–27. doi:10.1145/3498886.3502202.

[13] Brumley, B.B., Tuveri, N., 2011. Remote Timing Attacks Are Still Practical, in: Atluri, V., Diaz, C. (Eds.), Computer Security – ESORICS 2011, Springer. pp. 355–371. doi:10.1007/978-3-642-23822-2_20.

[14] Burow, N., Carr, S.A., Nash, J., Larsen, P., Franz, M., Brunthaler, S., Payer, M., 2017. Control-Flow Integrity: Precision, Security, and Performance. ACM Computing Surveys 50, 16:1–16:33. doi:10.1145/3054924.

[15] Cabrera Arteaga, J., Laperdrix, P., Monperrus, M., Baudry, B., 2022. Multi-variant Execution at the Edge, in: Proceedings of the 9th ACM Workshop on Moving Target Defense, pp. 11–22.

[16] Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.R., Shacham, H., Winandy, M., 2010. Return-oriented Programming Without Returns, in: Proceedings of the 17th ACM Conference on Computer and Communications Security, ACM. pp. 559–572.

[17] Crane, S., Homescu, A., Brunthaler, S., Larsen, P., Franz, M., 2015a. Thwarting Cache Side-Channel Attacks Through Dynamic Software Diversity, in: Proceedings 2015 Network and Distributed System Security Symposium, Internet Society. doi:10.14722/ndss.2015.23264.

[18] Crane, S., Liebchen, C., Homescu, A., Davi, L., Larsen, P., Sadeghi, A., Brunthaler, S., Franz, M., 2015b. Readactor: Practical Code Randomization Resilient to Memory Disclosure, in: 2015 IEEE Symposium on Security and Privacy, pp. 763–780. doi:10.1109/SP.2015.52.

[19] Deogirikar, J., Vidhate, A., 2017. Security attacks in iot: A survey, in: 2017 International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud)(I-SMAC), IEEE. pp. 32–37.

[20] Devi, M., Majumder, A., 2021. Side-Channel Attack in Internet of Things: A Survey, in: Mandal, J.K., Mukhopadhyay, S., Roy, A. (Eds.), Applications of Internet of Things, Springer. pp. 213–222. doi:10.1007/978-981-15-6198-6_20.

[21] D'Silva, V., Payer, M., Song, D., 2015. The Correctness-Security Gap in Compiler Optimization, in: 2015 IEEE Security and Privacy Workshops, pp. 73–87. doi:10.1109/SPW.2015.33.

[22] Foundation, L., . Tool interface standard (tis) portable formats specification version 1.1. URL: https://refspecs.linuxfoundation.org/elf/TIS1.1.pdf. accessed February 2023.

[23] Fu, A., Ding, W., Kuang, B., Li, Q., Susilo, W., Zhang, Y., 2022. FH-CFI: Fine-grained hardware-assisted control flow integrity for ARM-based IoT devices. Computers & Security 116, 102666. doi:10.1016/j.cose.2022.102666.

[24] Gao, P., Zhang, J., Song, F., Wang, C., 2019. Verifying and Quantifying Side-channel Resistance of Masked Software Implementations. ACM Transactions on Software Engineering and Methodology 28, 16:1–16:32. doi:10.1145/3330392.

[25] Gecode Team, 2022. Gecode: Generic constraint development environment. URL: https://www.gecode.org.

[26] Gilles, O., Viguier, F., Kosmatov, N., Pérez, D.G., 2022. Control-flow integrity at risc: Attacking risc-v by jump-oriented programming. URL: https://arxiv.org/abs/2211.16212, doi:10.48550/ARXIV.2211.16212.

[27] Hebrard, E., O'Sullivan, B., Walsh, T., 2007. Distance Constraints in Constraint Satisfaction, in: International Joint Conference on Artificial Intelligence - IJCAI 2007, p. 6.

[28] Homescu, A., Neisius, S., Larsen, P., Brunthaler, S., Franz, M., 2013. Profile-guided Automated Software Diversity, in: Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), IEEE Computer Society. pp. 1–11. doi:10.1109/CGO.2013.6494997.

[29] Ingmar, L., Garcia de la Banda, M., Stuckey, P.J., Tack, G., 2020. Modelling diversity of solutions, in: Proceedings of the thirty-fourth AAAI conference on artificial intelligence.

[30] Jaloyan, G.A., Markantonakis, K., Akram, R.N., Robin, D., Mayes, K., Naccache, D., 2020. Return-Oriented Programming on RISC-V, in: Proceedings of the 15th ACM Asia Conference on Computer and Communications Security, pp. 471–480. doi:10.1145/3320269.3384738.

[31] Joshi, H.P., Dhanasekaran, A., Dutta, R., 2015. Trading off a vulnerability: does software obfuscation increase the risk of rop attacks. Journal of Cyber Security and Mobility , 305–324.

[32] Kocher, P., Jaffe, J., Jun, B., 1999. Differential Power Analysis, in: Advances in Cryptology — CRYPTO' 99, Springer. pp. 388–397. doi:10.1007/3-540-48405-1_25.

[33] Kocher, P.C., 1996. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems, in: Advances in Cryptology — CRYPTO '96, Springer. pp. 104–113. doi:10.1007/3-540-68697-5_9.

[34] Koo, H., Chen, Y., Lu, L., Kemerlis, V.P., Polychronakis, M., 2018. Compiler-Assisted Code Randomization, in: 2018 IEEE Symposium on Security and Privacy (SP), pp. 461–477.

[35] Larsen, P., Homescu, A., Brunthaler, S., Franz, M., 2014. SoK: Automated Software Diversity, in: 2014 IEEE Symposium on Security and Privacy, pp. 276–291. doi:10.1109/SP.2014.25.

[36] Lindner, A., Guanciale, R., Dam, M., 2023. Proof-producing symbolic execution for binary code verification. arXiv:2304.08848.

[37] Castañeda Lozano, R., Carlsson, M., Blindell, G.H., Schulte, C., 2019. Combinatorial Register Allocation and Instruction Scheduling. ACM Trans. Program. Lang. Syst. 41, 17:1–17:53. doi:10.1145/3332373.

[38] Castañeda Lozano, R., Schulte, C., 2019. Survey on Combinatorial Register Allocation and Instruction Scheduling. ACM Computing Surveys 52, 62:1–62:50. doi:10.1145/3200920.

[39] Mantel, H., Starostin, A., 2015. Transforming Out Timing Leaks, More or Less, in: Computer Security – ESORICS 2015, Springer International Publishing. pp. 447–467. doi:10.1007/978-3-319-24174-6_23.

[40] Messerges, T.S., Dabbish, E.A., Sloan, R.H., 1999. Investigations of power analysis attacks on smartcards. Smartcard 99, 151–161.

[41] Molnar, D., Piotrowski, M., Schultz, D., Wagner, D.A., 2005. The program counter security model: Automatic detection and removal of control-flow side channel attacks, in: Information Security and Cryptology - ICISC 2005, 8th International Conference, Seoul, Korea, December 1-2, 2005, Revised Selected Papers, pp. 156–168.

[42] Ngo, K., Dubrova, E., Guo, Q., Johansson, T., 2021a. A Side-Channel Attack on a Masked IND-CCA Secure Saber KEM Implementation. IACR Transactions on Cryptographic Hardware and Embedded Systems , 676–707doi:10.46586/tches.v2021.i4.676-707.

[43] Ngo, K., Dubrova, E., Johansson, T., 2021b. Breaking Masked and Shuffled CCA Secure Saber KEM by Power Analysis, in: Proceedings of the 5th Workshop on Attacks and Solutions in Hardware Security, pp. 51–61.

[44] Ngo, V.C., Dehesa-Azuara, M., Fredrikson, M., Hoffmann, J., 2017. Verifying and Synthesizing Constant-Resource Implementations with Types, in: 2017 IEEE Symposium on Security and Privacy (SP), pp. 710–728. doi:10.1109/SP.2017.53. iSSN: 2375-1207.

[45] Nyman, T., Ekberg, J.E., Davi, L., Asokan, N., 2017. CFI CaRE: Hardware-Supported Call and Return Enforcement for Commercial Microcontrollers, in: Dacier, M., Bailey, M., Polychronakis, M., Antonakakis, M. (Eds.), Research in Attacks, Intrusions, and Defenses, Springer International Publishing. pp. 259–284.

[46] Papagiannopoulos, K., Veshchikov, N., 2017. Mind the Gap: Towards Secure 1st-Order Masking in Software, in: International Workshop on Constructive Side-Channel Analysis and Secure Design, Springer. pp. 282–297.

[47] Papp, D., Ma, Z., Buttyan, L., 2015. Embedded systems security: Threats, vulnerabilities, and attack taxonomy, in: 2015 13th Annual Conference on Privacy, Security and Trust (PST), pp. 145–152. doi:10.1109/PST.2015.7232966.

[48] Pappas, V., Polychronakis, M., Keromytis, A.D., 2012. Smashing the Gadgets: Hindering Return-Oriented Programming Using In-place Code Randomization, in: 2012 IEEE Symposium on Security and Privacy, pp. 601–615. doi:10.1109/SP.2012.41. iSSN: 1081-6011.

[49] Pastrana, S., Tapiador, J., Suarez-Tangil, G., Peris-López, P., 2016a. AVRAND: A Software-Based Defense Against Code Reuse Attacks for AVR Embedded Devices, in: Detection of Intrusions and Malware, and Vulnerability Assessment. Springer International Publishing. volume 9721, pp. 58–77. doi:10.1007/978-3-319-40667-1_4. series Title: Lecture Notes in Computer Science.

[50] Pastrana, S., Tapiador, J., Suarez-Tangil, G., Peris-López, P., 2016b. AVRAND: A Software-Based Defense Against Code Reuse Attacks for AVR Embedded Devices, in: Detection of Intrusions and Malware, and Vulnerability Assessment, pp. 58–77. doi:10.1007/978-3-319-40667-1_4.

[51] Randolph, M., Diehl, W., 2020. Power Side-Channel Attack Analysis: A Review of 20 Years of Study for the Layman. Cryptography 4, 15. URL: https://www.mdpi.com/2410-387X/4/2/15, doi:10.3390/cryptography4020015. number: 2 Publisher: Multidisciplinary Digital Publishing Institute.

[52] Rane, A., Lin, C., Tiwari, M., 2015. Raccoon: Closing Digital {Side-Channels} through Obfuscated Execution, in: 26th USENIX Security Symposium (USENIX Security 15), pp. 431–446.

[53] Roemer, R., Buchanan, E., Shacham, H., Savage, S., 2012. Return-Oriented Programming: Systems, Languages, and Applications. ACM Transactions on Information and System Security 15, 2:1–2:34. doi:10.1145/2133375.2133377.

[54] Rossi, F., Van Beek, P., Walsh, T., 2006. Handbook of constraint programming. Elsevier.

[55] Salehi, M., Hughes, D., Crispo, B., 2019. MicroGuard: Securing Bare-Metal Microcontrollers against Code-Reuse Attacks, in: 2019 IEEE Conference on Dependable and Secure Computing (DSC), pp.

1–8. doi:10.1109/DSC47296.2019.8937667.
[56] Salwan, J., 2020. ROPgadget Tool. URL: http://shell-storm.org/project/ROPgadget/.
[57] Seibert, J., Okhravi, H., Söderström, E., 2014. Information Leaks Without Memory Disclosures: Remote Side Channel Attacks on Diversified Code, in: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, pp. 54–65.
[58] Shacham, H., 2007. The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86), in: Proceedings of the 14th ACM Conference on Computer and Communications Security, ACM. pp. 552–561.
[59] Shelton, M.A., Samwel, N., Batina, L., Regazzoni, F., Wagner, M., Yarom, Y., 2021. Rosita: Towards Automatic Elimination of Power-Analysis Leakage in Ciphers. Proceedings 2021 Network and Distributed System Security Symposium doi:10.14722/ndss.2021.23137. appears in NDSS 2022.
[60] Shi, J., Guan, L., Li, W., Zhang, D., Chen, P., Chen, P., 2022. HARM: Hardware-assisted continuous re-randomization for microcontrollers, in: 2022 IEEE european symposium on security and privacy (EuroS P).
[61] Snow, K.Z., Monrose, F., Davi, L., Dmitrienko, A., Liebchen, C., Sadeghi, A., 2013. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization, in: 2013 IEEE Symposium on Security and Privacy, pp. 574–588.
[62] Tran, M., Etheridge, M., Bletsch, T., Jiang, X., Freeh, V., Ning, P., 2011. On the Expressiveness of Return-into-libc Attacks, in: Sommer, R., Balzarotti, D., Maier, G. (Eds.), Recent Advances in Intrusion Detection, Springer. pp. 121–141.
[63] Tsoupidi, R.M., 2017. Two-phase WCET analysis for cache-based symmetric multiprocessor systems. Master's thesis. Royal Institute of Technology KTH.
[64] Tsoupidi, R.M., Castañeda Lozano, R., Baudry, B., 2021. Constraint-based diversification of jop gadgets. Journal of Artificial Intelligence Research 72, 1471–1505.
[65] Tsoupidi, R.M., Castañeda Lozano, R., Troubitsyna, E., Papadimitratos, P., 2023. Securing optimized code against power side channels, in: CSF 2023 - 36th IEEE Computer Security Foundations Symposium, IEEE. To appear.
[66] Vu, S.T., Cohen, A., De Grandmaison, A., Guillon, C., Heydemann, K., 2021. Reconciling optimization with secure compilation. Proceedings of the ACM on Programming Languages 5, 1–30.
[67] Wang, J., Sung, C., Wang, C., 2019. Mitigating power side channels during compilation, in: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 590–601. doi:10.1145/3338906.3338913.
[68] Winderix, H., Mühlberg, J.T., Piessens, F., 2021. Compiler-Assisted Hardening of Embedded Software Against Interrupt Latency Side-Channel Attacks, in: 2021 IEEE European Symposium on Security and Privacy (EuroS P), pp. 667–682. doi:10.1109/EuroSP51992.2021.00050.
[69] Xu, R., Zhu, L., Wang, A., Du, X., Choo, K.K.R., Zhang, G., Gai, K., 2018. Side-Channel Attack on a Protected RFID Card. IEEE Access 6, 58395–58404. doi:10.1109/ACCESS.2018.2870663. conference Name: IEEE Access.
[70] Zinzindohoué, J.K., Bhargavan, K., Protzenko, J., Beurdouche, B., 2017. HACL*: A Verified Modern Cryptographic Library, in: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pp. 1789–1806.

A. Path-Finding Algorithm

When the branch condition has type `secret`, i.e. depends on a `secret` value, SecDivCon performs an analysis to discover all paths starting from the branch condition (source) to a common node (sink). We assume that the program is split into basic blocks, pieces of code with at most one branch (apart from function calls) at the end of the block. To identify all possible paths, we generate the Control-Flow Graph (CFG) between the basic blocks of the program.

```
1  GET_PATHS(n, BCFG):
2    t.empty() # Queue - First path
3    P.empty() # Priority queue - Paths
4    t.insert(n)
5    P.insert(t)
6    W.empty() # final paths
7    while (¬P.isempty() and ¬P.hasCycle()):
8      p ← P.top() # Top path
9      h ← p.pop() # Last element of path
10     succ ← BCFG.successors(h)
11     if (succ = ∅): # exit node
12       W.push(p)
13       P.remove(p)
14     elif (succ = {s}):
15       p.push(s)
16       P.replace(p)
17       # if this is a sink, we terminate
18       if (W.extend(P).hasSink()):
19         return W.extend(P)
20     elif (succ = {s1,s2}):
21       p1 ← p.copy()
22       p2 ← p.copy()
23       p1.push(s1)
24       p2.push(s2)
25       P.remove(p)
26       P.insert(p1)
27       P.insert(p2)
28    return W
```

Figure 8: Path extraction

Figure 8 shows the algorithm for extracting the paths that start from a basic block *n* (the secret-dependent branch), given the CFG (`BCFG`). We use two data structures, a priority queue, `P`, which contains all paths under analysis, and a queue, `t` that represents the current path and starts with the first basic block, *n* (line 4). The priority queue uses the block order as the priority, with smaller numbers having priority. At line 5, `P` is initialized with `t`. We store the final results in `W` (line 6). At line 7, we start a loop that terminates when there are no paths left to analyze in `P` or when we find a cycle. At lines 8 and 9, we get the top element of the top path from `P`. Subsequently, the algorithm finds all successor nodes in the CFG, which correspond to possible basic blocks that follow the current basic block (line 10). Then, the algorithm performs different actions depending on the successor nodes. First, if the current node, `h` does not have any successors, it means that `h` is an exit node, thus, `h` is the last node in the current path. Lines 12 and 13 add the path to `W` and remove it from the paths under analysis. If `h` has one successor, `s`, then we push the successor to the path and update `P` (lines 14-16). Here, we need to check if the new node leads to the current paths having a sink, i.e. the same final node (line 17). The last case is when the branch is conditional and there are two possible destinations. Here, we need to generate two paths `p1`

and p2 for each of the two destinations and insert them to P for further analysis (lines 20-27). When the analysis finishes and the algorithm exits the loop, then it returns W.

This analysis does not support loops.