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
automatic software diversification
software masking
constant-resource programming
secure compilation

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, altering 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 constrained. 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 [50]. 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 [62, 57]. These attacks target both general purpose [62] and embedded devices [52, 32, 9, 59]. At the same time, the execution of embedded software may leak information about sensitive data to the adversary via side channels [43, 21, 22]. 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 associated 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 [21], which may be resource-forbidding, and thus, creates the need for overhead-aware approaches [71, 69].

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 [59] 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 [51, 69]. 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 [69] or in a re-randomization scheme [64] against advanced code-reuse attacks [8].

This paper contributes with:

- 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 4.2);
- 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 [17, 9, 28]. 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 [60] 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 [38] introduces uncertainty to the location and 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 [48] in small, often battery-operated devices. Automatic software diversification provides an efficient mitigation against CRAs [38, 69, 52]. 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 [38], where a centralized repository distributes precompiled code variants to each end user.

```
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

Side-Channel Attacks (SCAs): Usually, embedded devices use cryptographic algorithms, which are vulnerable to SCAs [35, 43, 11]. These attacks allow the adversary to extract information about secret values by measuring the execution time – called timing side channel (TSC) [14] or the power consumption – called power side channel (PSC) [54] 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 [74]. 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

```
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

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) [49], based on the Hamming Distance (HD) model [11], which is widely used for designing PSC attacks and defenses [11, 49]. 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 the implementation in Figure 3b leaks secret information. Thus, the embedded devices that use this variant may be vulnerable to PSCs. Notice that these leaks correspond to the implementation of the code in Figure 2a. If the compiler middle-end takes advantage of the associativity property of the exclusive-or operation to change the order of the exclusive-or operations, as we see in Figure 2b, then, there will be a leak regardless of the register allocation (see Section 3.1.3).

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 [47]. 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 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 handle 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 4.2. The latter demonstrates that fine-grained code diversification against CRAs and software

```
@ r0: pub, r1: key
@ BB#0:
  movs  r2, #0
  cmp   r1, r0
  bne   .LBB0_2
@ BB#1:
  movs  r0, #1
  b     .LBB0_3
.LBB0_2:
  mov   r0, r2
  movs  r1, #1
.LBB0_3:
  bx    lr
...
...
```

(a) Secure variant 1

```
@ BB#0:
  movs  r2, #0
  cmp   r1, r0
  bne   .LBB0_2
@ BB#1:
  movs  r3, #1
  mov   r0, r3
  b     .LBB0_3
.LBB0_2:
  mov   r3, r2
  mov   r0, r3
  movs  r3, #1
.LBB0_3:
  bx    lr
```

(b) Secure variant 2

Figure 5: Two secure program variants of Figure 4 for ARM Cortex M0

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 extracts 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 extracts 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) [34] or Correlational Power Analysis (CPA) [11, 49].

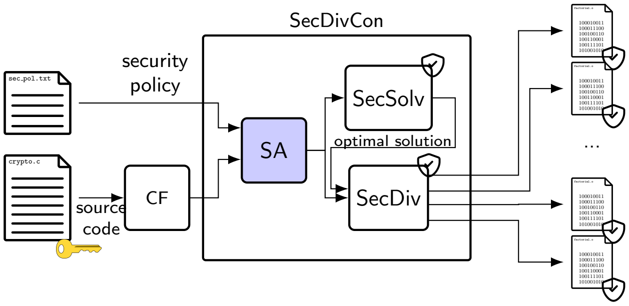

Figure 6: High-level view of SecDivCon

We adopt the leakage model for PSCs from Tsoupidi et al. [70] 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 generates 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 (Section 3.1.1). Subsequently, SecDivCon propagates the security policy to each program term using type inference (Section 3.1.2). In cases when the input program is not secure against SCAs, SecDivCon performs transformations (Section 3.1.3) 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. Security Policy

SecDivCon takes as input a filename that defines the security policy of the function under analysis. In particular, the security policy defines the security type of each function argument. For example, in Figure 2, we want to define that 1) the data of the first argument is known to the attacker (`Public`), 2) the second argument contains secret information (`Secret`), and 3) the third argument is a randomly generated value (`Random`). SecDivCon takes as input this information in a file as follows:

```
Public r0;
Secret r1;
Random r2
```

Given the definition of the types for the input arguments, SecDivCon implements a type-inference algorithm at a low-level intermediate representation format to extract explicit timing leaks and transitional power leaks.

3.1.2. 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 (`sec` $\oplus$ `mask`) $\oplus$ `mask` $=$ `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 [26, 72].

3.1.3. 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 [23]. 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, 70]. In the following paragraphs, we discuss the program transformations that SecDivCon implements before the solving stage protecting against TA or PA.

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

(a) Add Empty Block (b) Copy Unbalanced Block

Figure 7: Balancing transformations for Figure 4

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 transformation that balances secret-dependent blocks, using two methods: 1) Empty-Block Balancing (EBB), which inserts an empty block, and 2) Copy-Block Balancing (CBB), which copies the present secret-dependent block in an `else` statement.

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.

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, 70]. 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 [41]. A constraint model defines all legal instruction orders and register assignments [40]. 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`($paths_{sec}$):
$\forall p_1, p_2 \in paths_{sec}. \sum_{b \in p_1} \mathtt{cost}(b) = \sum_{b \in p_2} \mathtt{cost}(b)$

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

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 [29, 31] 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 [69]. 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 consists of two parts: 1) the identification of side-channel vulnerabilities in a diversification tool and 2) the evaluation of SecDivCon. The evaluation of SecDivCon focuses on two aspects: a) the effectiveness of the CR-preserving code generation and b) the effectiveness of SCA-aware code diversification. Section 4.1 describes the implementation of SecDivCon, experimental setup, and benchmarks, while Section 4.2 presents our findings with regard to SCA vulnerabilities in diversified code. Sections 4.3-4.5 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 [40], a combinatorial compiler backend that uses Constraint Programming (CP) [58] 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 [40]. 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 [70], 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 [27]. SecDivCon combines the SCA-aware mitigations with a diversification scheme [69] to generate multiple function variants. We target two architectures, 1) a generic MIPS32 processor and 2) the ARM Cortex M0 processor [4].

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 [70, 72] and five functions that exhibit secret-dependent timing variations and are used in cryptographic context [42]. Table 1 shows the benchmarks

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 MIPS32; 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

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 MIPS32 (M), 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. [72]. These benchmark programs consist of masked cryptographic core functions that may be vulnerable to PSC attacks. More specifically, P0 is a masked exclusive-OR implementation, P1-P3 are protected by Boolean masking [7], P4-P6 correspond to masked multiplication [56], P7-P9 correspond to masked S-box for AES [18], and P10 implements key whitening [7].

CR Programs: For evaluating the CR property we use Listing 4 and four benchmark programs used by Mantel and Starostin [42]. The code for these benchmarks in C including security-policy annotations is available by Winderix et al. [73]. The set of benchmark programs includes 1) C0 is simple unbalanced `if` statement (see Figure 4), 2) C1 computes the total market value of a share from the portfolio of the user, 3) C2 implements multiplication modulo 8, 4) C3 implements the square-and-multiply modular exponentiation, and 5) C4 implements Kruskal's algorithm [37] to compute the minimum spanning tree of a graph. These implementations are vulnerable to timing attacks [42].

4.2. Side-Channel Vulnerabilities in Code Diversification

Our approach is based on the intuition that automatic software diversification transformations may break side-channel mitigations. To confirm this intuition, we investigate the effect of code diversification using a freely-available[1] code diversification tool, Multicompiler (MCR) [30]. In particular, we analyze code variants generated by MCR to check if the code 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[2] by Wang et al. [72], whereas for TSC, we measure the execution time manually. Both tools focus on the x86 architecture. 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.

Hardware-Register Randomization: Hardware-register randomization [20] 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. [72] to extract information from the register allocation step in MCR. Subsequently, we use the tool by Wang et al. [72] to identify the leaks in the variants.

For each of the masked benchmarks, Table 2 shows the number of leaks that appear in the baseline, which uses the LLVM compiler [72] 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.

[1] MCR: `https://github.com/securesystemslab/multicompiler.git`

[2] FSE19 tool: `https://github.com/bobowang2333/FSE19`

Table 2
ROT vulnerabilities in diversified code by Multicompiler (MCR)

Prg	[72]	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%

Table 3
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

NOP Insertion: NOP insertion is a form of fine-grained software diversification that generates program variants that contain randomly inserted NOP operations. MCR implements NOP-insertion randomization [30]. 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[3]. To summarize, NOP insertion may break branch balancing for CR programs.

4.3. 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.

Table 4
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

Table 5
Security Evaluation using a WCET tool to compare the execution time: T denotes a symbolic value, v 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 = v	✓
C2	a0,a1,a2,a3 = T	✓	a0,a1,a2,a3 = T	✓
C3	a0,a1,a2,a3 = T[a]	✓	a0,a1,a3 = T, a2 = v	✓
C4	a0,a1,a2,a3 = v[a,b]	✓	a0,a1,a2 = T, a3 = v	✓

[a] Verified only the secret-dependent branches to improve scalability and accuracy
[b] The concrete values correspond to addresses of the inputs

4.3.1. *Performance Overhead*

The CR-preserving code generation extends Unison [40] 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 (see Section 3.2.1) of the constraint-based compiler backend [40]. Table 3 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 [70]. 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.

4.3.2. *Compilation Overhead*

The introdution of new constraints to satisfy the constant-resource property in the constraint model may lead to

[3] Here, we do not investigate multi-variant execution, where different variants are loaded dynamically, which may hinder timing attacks by randomizing the execution time.

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 4 shows the compilation-time overhead of SecDivCon (🔒) compared to Unison (non-CR-preserving code optimization) [40] (🔓). 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 [70] 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.

4.3.3. 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 [12, 68]. 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[4] that generates the WCET and BCET for a sequence of binary instructions [39]. 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[5]. For each of the benchmark programs, Table 5 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.

4.4. 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 generates 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 generates 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, which increases the available variants. However, increasing the optimality gap, increases also the search space, which increases the solver

[4] CM0 WCET: `https://github.com/kth-step/HolBA/tree/dev_symbexec_form`

[5] 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.

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

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 4.3) 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 [24] 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 [30, 51], 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

[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	-

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, 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 [62]. 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 [69]. Return-into-libc (RILC) [67] 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) [8], 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 [64] that may be performed at boot time [52]. 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 [16], 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. [46] show that advanced randomization techniques, such as plaintext shuffling, are vulnerable [46, 45], 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.

Security Policy: SecDivCon is suitable for analyzing functions that process secret values and hence, are vulnerable to side-channel attacks. The security policy defines, which of the input values are `secret`, and which of them are `public` or `random`. The type of each value depends on the usage of the respective algorithm in a security-critical application. SecDivCon requires only annotating the function arguments and applies a type-inference algorithm to extract the type of each intermediate variable.

6. Related Work

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
[30]	CRA	Div	25%	x86
[51]	CRA	Div	0%	x86
[19]	TSC	Div	8x	x86
[55]	TSC	Obf	16x	x86
[53]	CRA	Div, RR	-	AVR
[2]	CRA	CFI	∼80%	ARM
[48]	CRA	CFI	5x	ARM
[75]	TSC, MS	CT	-	C
[36]	CRA	Div, RR	7%	x86
[59]	CRA	Div,CFI	70%	ARM
[63]	PSC	SM	64%	ARM
[73]	TSC, IL	BB	60%	MSP430
[64]	CRA	Div, RR	6%	ARM
[10]	TSC	CT	5x	x86
[25]	CRA	HWCFI	24%	ARM
SDC	TSC, PSC, CRA	Div, SM/BB	70%[a]	MIPS, ARM

[a] Diversification overhead is controlled

This section presents the related work with regards to Code-Reuse Attacks and Side-Channel Attacks. Table 8 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 [38]. Many software diversification approaches target x86 systems [30, 51, 36], while others target embedded systems [53, 59, 69, 64]. 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 [51, 69].

Re-randomization approaches [64, 53, 36] repeat the randomization process in specific timing intervals to protect against advanced CRAs, such as JIT-ROP [65], BROP, and side-channel-based diversification deciphering [61]. These approaches may introduce additional binary-size overhead [16] and performance overhead. However, this performance overhead is typically low, for example, HARM [64] 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 [15]. Software-based CFI systems [2, 48, 59] typically result in high overhead, whereas hardware-assisted methods may lead to reduced overhead [25]. However, hardware-assisted CFI approaches often depend on specialized hardware mechanisms [15].

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 [72, 63, 49, 5]. Wang et al. [72] identify leaks in masked implementation using a type-inference algorithm, and then, perform register-allocation transformations to mitigate these leaks in LLVM. Rosita [63] 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 [70] based on type inference [26] 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 [44] 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. [10] 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 [47]. 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. [13] perform transformations to balance secret-dependent branches by balancing the branch bodies at the C level. Winderix et al. [73] 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 [55], which uses control-flow obfuscation to mitigate TSC attacks. However, Joshi et al. [33] 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. [19] 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. [75] 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 consider the effect on CRAs.

7. Conclusion and Future Work

This paper presents SecDivCon – a constraint-based approach that combines code diversification with side-channel mitigations. It enables the secure-by-design generation of optimized 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.

In 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 their support with their tool. We would also like to thank Andreas Lindner for his support with verifying SecDivCon using WCET analysis for ARM Cortex M0. In addition, we would like to thank 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 Arteaga 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 parts of his work in this context. E. Troubitsyna acknowledges the support of the Swedish Foundation for Strategic Research (SSF) in this work.

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] Bayrak, A.G., Regazzoni, F., Novo, D., Ienne, P., 2013. Sleuth: Automated Verification of Software Power Analysis Countermeasures, in: Cryptographic Hardware and Embedded Systems - CHES 2013, Springer. pp. 293–310. doi:10.1007/978-3-642-40349-1_17.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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].

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] Coron, J.S., Prouff, E., Rivain, M., Roche, T., 2014. Higher-Order Side Channel Security and Mask Refreshing, in: Fast Software Encryption, Springer. pp. 410–424. doi:10.1007/978-3-662-43933-3_21.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] Foundation, L., . Tool interface standard (tis) portable formats specification version 1.1. URL: https://refspecs.linuxfoundation.org/elf/TIS1.1.pdf. accessed February 2023.

[25] 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.

[26] 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.

[27] Gecode Team, 2022. Gecode: Generic constraint development environment. URL: https://www.gecode.org.

[28] 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.

[29] Hebrard, E., O'Sullivan, B., Walsh, T., 2007. Distance Constraints in Constraint Satisfaction, in: International Joint Conference on Artificial Intelligence - IJCAI 2007, p. 6.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] Kruskal, J.B., 1956. On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem. Proceedings of the American Mathematical Society 7, 48–50. doi:10.2307/2033241. publisher:

American Mathematical Society.

[38] 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.

[39] Lindner, A., Guanciale, R., Dam, M., 2023. Proof-producing symbolic execution for binary code verification. arXiv:2304.08848.

[40] 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.

[41] 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.

[42] 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.

[43] Messerges, T.S., Dabbish, E.A., Sloan, R.H., 1999. Investigations of power analysis attacks on smartcards. Smartcard 99, 151–161.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] 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.

[52] 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.

[53] 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.

[54] 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.

[55] 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.

[56] Rivain, M., Prouff, E., 2010. Provably Secure Higher-Order Masking of AES, in: Cryptographic Hardware and Embedded Systems, CHES 2010, Springer. pp. 413–427. doi:10.1007/978-3-642-15031-9_28.

[57] 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.

[58] Rossi, F., Van Beek, P., Walsh, T., 2006. Handbook of constraint programming. Elsevier.

[59] 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.

[60] Salwan, J., 2020. ROPgadget Tool. URL: http://shell-storm.org/project/ROPgadget/.

[61] 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.

[62] 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.

[63] 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.

[64] 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).

[65] 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.

[66] Soares, L., Canesche, M., Pereira, F.M.Q., 2023. Side-Channel Elimination via Partial Control-Flow Linearization. ACM Transactions on Programming Languages and Systems URL: https://dl.acm.org/doi/10.1145/3594736, doi:10.1145/3594736. just Accepted.

[67] 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.

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

[69] Tsoupidi, R.M., Castañeda Lozano, R., Baudry, B., 2021. Constraint-based diversification of jop gadgets. Journal of Artificial Intelligence Research 72, 1471–1505.

[70] 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.

[71] 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.

[72] 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.

[73] 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.

[74] 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.

[75] 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

```
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

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.

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 secret-dependent loops or public-dependent loops nested in secret-dependent branches. The reason for the latter is that SecDivCon does not implement a loop-insertion transformation. In the presense of these patterns, the compiler will fail to generate a balanced program because the constraints are infeasible. To handle secret-dependent loops, SecDivCon needs to consider additional transformations, such as loop insertion, and converting secret-dependent loops to non-terminating loops [66]. This is part of future work.