Securing Optimized Code Against Power Side Channels

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

Roberto Castañeda Lozano
Independent Researcher
Stockholm, Sweden
rcas@acm.org

Elena Troubitsyna
Royal Institute of Technology KTH
Stockholm, Sweden
elenatro@kth.se

Panagiotis Papadimitratos
Royal Institute of Technology KTH
Stockholm, Sweden
papadim@kth.se

***Abstract*—Side-channel attacks impose a serious threat to cryptographic algorithms, including widely employed ones, such as AES and RSA. These attacks take advantage of the algorithm implementation in hardware or software to extract secret information via side channels. Software masking is a mitigation approach against power side-channel attacks aiming at hiding the secret-revealing dependencies from the power footprint of a vulnerable implementation. However, this type of software mitigation often depends on general-purpose compilers, which do not preserve non-functional properties. Moreover, microarchitectural features, such as the memory bus and register reuse, may also leak secret information. These abstractions are not visible at the high-level implementation of the program. Instead, they are decided at compile time. To remedy these problems, security engineers often sacrifice code efficiency by turning off compiler optimization and/or performing local, post-compilation transformations. This paper proposes Secure by Construction Code Generation (SecCG), a constraint-based compiler approach that generates optimized yet secure, which is protected against power side channels code. SecCG controls the quality of the mitigated program by efficiently searching the best possible low-level implementation according to a processor cost model. In our experiments with twelve masked cryptographic functions up to 100 lines of code on Mips32 and ARM Thumb, SecCG speeds up the generated code from 77% to 6.6 times compared to non-optimized secure code with an overhead of up to 13% compared to non-secure optimized code at the expense of a high compilation cost. For security and compiler researchers, this paper proposes a formal model to generate power side channel free low-level code. For software engineers, SecCG provides a practical approach to optimize performance critical and vulnerable cryptographic implementations that preserves security properties against power side channels.**

***Index Terms*—compilation, power side-channel attacks, code optimization, software masking**

I. Introduction

Cryptographic algorithms, symmetric/shared key or asymmetric/private key ones, rely on safeguarding the shared secret key or the private key, respectively. The exposure of these keys to unintended users compromises the security of these algorithms. Unfortunately, the software implementation of cryptographic algorithms may reveal information about their secret/private keys [1]. In particular, the attacker may observe what is termed *side-channel information*, notably observing the execution time [1] or the power consumption [2, 3], during the execution of the algorithm to extract information about the secret keys. These attacks are attractive especially because usually they do not require expensive equipment. This paper focuses on Power Side Channel (PSC) attacks.

Software masking is a widely-used approach to mitigate PSC attacks [4, 5], hiding secret information by splitting a secret into n randomized shares. The attacker has to retrieve all shares in order to acquire the secret value. While software masking can be an effective mitigation, compiler code generation may optimize it away. Moreover, Transition-Based Leakage (TBL) sources, such as register reuse or memory-access order, are decided at compile time by low-level compiler transformations [6, 7, 8].

To mitigate these compiler-induced power side-channel leaks at the binary level there are techniques based on compilation [7, 9, 10] and binary rewriting with hardware emulation [11, 12, 13]. All these approaches mitigate compiler-generated leakages using local transformations [13, 7, 11]. The methods that depend on hardware emulation are typically accurate but may introduce significant overhead [11] and are hardware specific. For example, Rosita [11], an emulation-based approach, propose a mitigation that introduces an overhead ranging from 21% to 64% for ARM Cortex M0. Wang et al. [7] perform their mitigation using a standard compiler with no high-level optimizations (-O0). This is a common practice for security research to ensure the absence of compiler-induced mitigation invalidation [6, 14]. However, unoptimized code is highly inefficient, and may even introduce additional leaks due to the heavy use of the program stack, as discussed in Section II.

Vu et al. present an approach that enables secure optimization of masked code at a higher level [14, 15]. This approach applies high-level compiler optimizations by disallowing secure-code removal and operand reordering (due to associativity of some operations) and are able to generate correctly masked code. However, they do not deal with TBLs.

Currently, the state-of-the-art approaches are unable to gen-

erate code that is both efficient and secure in the face of TBLs that enable PSC attacks. To address this challenge, this paper proposes Secure by Construction Code Generation (SecCG), an optimizing compiler approach that provably preserves security properties against PSC. At the middle-end, SecCG handles code generated using *register promotion* (promoting program variables from memory to registers) as a high-level optimization. Then, SecCG uses a constraint-based method to generate code that is secure against PSC attacks. SecCG controls the quality of the mitigated program by efficiently searching the best possible low-level implementation according to a processor cost model [16]. The security model of SecCG is hardware agnostic and can be extended with additional architectural constraints. SecCG is suitable for predictable architectures with no advanced microarchitectural features, such as caches or speculative execution. In our experiments with twelve masked implementations on Mips32 and ARM Thumb, SecCG improves the execution time of the generated code from 77% to a speedup of 6.6 compared to non-optimized code at a overhead of up to 13% compared to non-secure optimized code. This comes at a cost on compilation time and scalability, where SecCG optimizes successfully programs up to 100 lines of code. In summary, this paper makes the following contributions:

- a compiler approach to generate TBL-free, low-overhead assembly code for high-level software-masked programs;
- a constraint model for optimized and PSC-secure code generation;
- a proof that the constraint model guarantees the generation of secure code for a non-trivial leakage model; and
- experimental results on two architectures showing that the performance overhead of our mitigation is low and its efficiency benefits are significant, compared to current approaches.

II. Motivating Example

To motivate our approach, let us consider an example of a first-order masked implementation. First-order masking splits a secret value `k` into two shares, (`m`, `mk`), where `m` is a uniformly distributed random variable sampled at every execution of the algorithm; `mk` = `m`$\oplus$`k` is also uniformly distributed ($\oplus$ denotes the exclusive OR operation). Fig. 1 shows a first-order masked C implementation of exclusive OR, where `key` is a secret value (red), `mask` is a uniformly random variable (brown), and `pub` is a non-secret value (green). At line 2, the algorithm creates the second share, `mk`, and at line 3, it performs the exclusive OR operation with the secret-independent value, `pub`. At a high-level, the code of Fig. 1 is secure against power side channels but a binary implementation generated by a standard, security-unaware compiler may leak information about `key`. For example, hardware-register reuse and memory-bus access order may reveal secret information [7, 11, 6, 8]. These TBLs are a result of transitional effects, i.e., the power effect of bits switching between one and zero and vice versa.

Fig. 3a shows the ARM Thumb assembly code generated by the standard compiler LLVM [17] for the C code in Fig. 1. The first three `str` instructions store the function arguments that reside in registers `r0`–`r2` to the stack (lines 3-5). Line 6 loads (`ldr`) the value of `rand` from the stack into register `r1`. Line 7 performs the first exclusive OR (line 2 in Fig. 1) between registers `r1` and `r2` (`key`) and stores the result in register `r1`. Here, there is a transition for register `r1` from value `mask` to `mk`, which leaks the secret `key` (marked code at line 7). Line 8 stores the content of `r1` to the stack and the value of the memory bus that contains the `mask` at line 6 transitions to `mk`. This leads to another leak due to the transitional effect in the memory bus (marked code at lines 6 and 8). The rest of the code performs the second exclusive OR (line 10) and stores the final result on the stack (line 11).

Fig. 3b shows the mitigation produced by the security backend of SecCG that eliminates leakages that appear in the LLVM unoptimized code. The mitigation is based on *instruction scheduling* and *register allocation* transformations. In particular, changing the order of operands at line 7 results in a transition from `sec` to `mk` that leaks the value of `mask`, which is not secret (marked code at line 7). Changing the order of the instructions hides the memory-bus leakage. More specifically, because there are no data dependencies between lines 3-6, the `ldr` instruction that causes the leak in Fig. 3a may be scheduled earlier (line 4 in Fig. 3b). Then, another memory instruction that stores the secret value in memory (line 6 in Fig. 3b) is scheduled just before the store instruction at line 8. This causes a transition from `sec` to `mk` in the memory bus that leaks the value of `mask` (marked code at lines 6 and 8). These transformations are global, considering possible available memory instructions and register assignments to mitigate transitional leakages in the whole program and may (as in Fig. 3b) introduce no overhead.

However, unoptimized code leads to poor performance. In general, compiler optimizations may invalidate high-level software mitigations [14]. Fortunately, this is not the case for register promotion (*mem2reg* in LLVM), a simple high-level optimization that enables efficient register allocation by promoting program variables from memory to registers. This transformation replaces stack operations to register operations and preserves the operand order. In particular, aggressive optimizations (-O1 to -O3 in LLVM) may take advantage of the associativity property of $\oplus$ to change the order of the operands, converting $(\texttt{mask} \oplus \texttt{sec}) \oplus \texttt{pub}$ to $\texttt{mask} \oplus (\texttt{sec} \oplus \texttt{pub})$, which invalidates masking. Equipped with improved high-level code, the SecCG backend optimizes low-level transformations and generates optimized code. Fig.s 2a and 2b show the code of Fig. 1 compiled with register promotion. Fig. 2a leaks the same secret information as Fig. 3a due to register reuse, namely the first exclusive OR operation `eors`, but contains no memory-bus secret leak. To mitigate the register-reuse leak at line 2, SecCG changes the order of the arguments and the result is now stored in register `r2`.

As we see in Fig. 3a, unoptimized code may introduce additional leaks due to the heavy use of the program stack. Instead, SecCG uses register promotion to remove unnecessary memory accesses that may cause additional leaks. Then,

```
1  u32 Xor(u32 pub, u32 mask, u32 key) {
2    u32 mk = mask ^ key;
3    u32 t = pub ^ mk;
4    return t;
5  }
```

Fig. 1: Masked exclusive OR implementation in C

```
1  @ r0: pub, r1: mask, r2: key
2    eors  r1, r2
3    eors  r0, r1
4    ...
```

(a) Insecure (LLVM)

```
1  @ r0: pub, r1: mask, r2: key
2    eors  r2, r1
3    eors  r0, r2
4    ...
```

(b) Secure (SecCG)

Fig. 2: Compilation of function `Xor` applying register promotion

```
1  @ r0: pub, r1: mask, r2: key
2  ...
3  str     r0, [sp, #16] @ mem: pub
4  str     r1, [sp, #12] @ mem: rand
5  str     r2, [sp, #8]  @ mem: key
6  ldr     r1, [sp, #12] @ mem: rand
7  eors    r1, r2
   @ proc: rand <- rand ^ key
8  str     r1, [sp, #4]  @ mem: rand ^ key
9  ldr     r0, [sp, #16] @ mem: pub
10 eors    r0, r1
   @ proc: pub <- pub ^ rand ^ key
11 str     r0, [sp]
   @ mem: pub ^ rand ^ key
12 ...
```

(a) Insecure (LLVM)

```
1  @ r0: pub, r1: mask, r2: key
2  ...
3  str     r1, [sp, #12] @ mem: rand
4  ldr     r1, [sp, #12] @ mem: rand
5  str     r0, [sp, #16] @ mem: pub
6  str     r2, [sp, #8]  @ mem: key
7  eors    r2, r1
   @ proc: key <- sec ^ rand
8  str     r2, [sp, #4]  @ mem: key ^ rand
9  ldr     r0, [sp, #16] @ mem: pub
10 eors    r0, r2
   @ proc: pub <- pub ^ key ^ rand
11 str     r0, [sp]
   @ mem: pub ^ rand ^ key
12 ...
```

(b) Secure (SecCGwith no register promotion)

Fig. 3: Compilation of function `Xor` with no optimizations

SecCG's backend generates low-level optimized code that does not expose secret information through transitional leakages and does not introduce significant overhead compared to non-secure code.

III. Threat Model and Modeling Background

This section describes the Hamming Distance (HD) model (Section III-A), the threat model (Section III-B), an HD-based type-inference algorithm (Section III-C), a constraint-based compiler backend model (Section III-D), and the running example for the constraint-based compiler backend (Section III-E),

A. Hamming-Distance Model

The Hamming Weight (HW) model [18, 2, 19] corresponds to the number of active bits in a data word. We assume the following encoding of the binary data, $d = \sum_{i=0}^{N-1} 2^i d_i$, where d_i is one if the i_{th} bit of an N-bit word is set and zero otherwise. The HW of this data is the number of bits that are set: $HW(d) = \sum_{i=0}^{N-1} d_i$. The HD leakage model assumes that the observed leakage when flipping the bits of a memory element from a value d_1 to a value d_2 is $HW(d_1 \oplus d_2)$, where $\oplus$ denotes the exclusive OR operation. If one of the values d_1 is a uniform random variable, then $d_1 \oplus d_2$ is also a uniform random variable and $HW(d_1 \oplus d_2)$ has the same mean and variance as $HW(d_1)$ [19]. This means that by masking (exclusive bitwise OR) a secret value k with a uniform random variable m, the HD of the new variable has the same mean and variance as m. In this way, masking hides the information of k from the power consumption traces.

We assume a program $P(\mathbf{IN}) = i_1; i_2; ..., i_n$ that takes as input a set of variables **IN** and consists of a sequence of n instructions i_j. We assume that the program has a leakage at every execution step when there is bit flipping in the hardware registers or the memory bus. We will use the terms by Papagiannopoulos and Veshchikov [13] and refer to the hardware-register transition leakage as Register-Overwrite Transition (ROT) and the memory-bus transition leakage as Memory-Remnant Effect (MRE). For MRE, we assume that both read and write operations make use of the same memory bus and that the source of the leakage is the transitional effect when writing the data to the memory bus. In our model, the memory address of the operations does not affect the leakage.

We represent the leakage as a set of observations in the power trace. To calculate the observed leakage $L(P(IN))$ for an instance IN of the input variables, we use the HD leakage model. We write $P = P'; i_n$ to denote a program $P = i_1; i_2; ...; i_{n-1}; i_n$, with a prefix $P' = i_1; i_2; ...; i_{n-1}$ (IN is omitted for simplicity). Equations 1-4 present a recursive definition of the leakage model, where for every point in the execution trace, the attacker observes the HW of any ROT or MRE transitions. In the formulas, an expression e is $e := r \mid v \mid \mathtt{bop}(e_1, e_2) \mid \mathtt{uop}(e_1) \mid \mathtt{mem}(e_a)$, where r is a register, v is a constant value, bop is a binary operation, uop is a unary operation, and $\mathtt{mem}(e_a)$ is a memory load operation that loads data from address e_a. An instruction is $i = r \leftarrow e \mid \mathtt{mem}(e_a, e)$, where $r \leftarrow e$ denotes that an expression is assigned to register r, and $\mathtt{mem}(e_a, e)$ is a store memory operation that stores data e at memory address e_a. To simplify the leakage equations, we transform the load operation from $r \leftarrow \mathtt{mem}(e_a)$ to a

$$L(P'; r \leftarrow e_2; P''; r \leftarrow e_1) \quad = L(P'; r \leftarrow e_2; P'') \cup \{HW(e_1 \oplus e_2)\}, \nexists i \in P''.\ i = r \leftarrow e_3 \tag{1}$$

$$L(P'; i_1; P''; \texttt{mem}(e_b, e_2)) \quad = L(P'; i_1; P'') \cup \{HW(e_1 \oplus e_2)\}, (i_1 = \texttt{mem}(e_a, e_1)) \ \wedge \ \nexists i \in P''.\ i = \texttt{mem}(e_c, e_3) \tag{2}$$

$$L(P'; r \leftarrow e) \quad = L(P') \cup \{HW(e \oplus r_{IN})\}, \nexists i \in P'.\ i = r \leftarrow e_3 \tag{3}$$

$$L(P'; \texttt{mem}(e_a, e_1)) \quad = L(P') \cup \{HW(e_1)\}, \nexists i \in P'.\ i = \texttt{mem}(e_b, e_3) \tag{4}$$

sequence $\texttt{mem}(e_a, v_{\texttt{mem}(e_a)}); r \leftarrow v_{\texttt{mem}(e_a)}$, where $v_{\texttt{mem}(e_a)}$ is the value in memory at address e_a. Equation 1 describes the leakage when two instructions write the value of their result to the same register and no other instruction between them writes to the same register. Note that the first equation deals also with instructions in the form $r_1 \leftarrow \texttt{bop}(r_2, r_3)$, where bop is a binary operation and $r_1 = r_2$. These two-address instructions are common in ARM Thumb and x86 architectures. Equation 2 describes the memory-bus leakage of a memory instruction that writes a value to the memory, given that another memory instructions precedes this memory instruction. Equation 3 describes the leakage of the first instruction that writes to register r. In this case, the leakage is equal to the HD between the new value and the initial value in register r, r_{IN}. Similarly, Equation 4 describes the leakage of the first memory operation. Here, we assume that the initial memory-bus content, mb_{IN}, is a constant value. For example, after executing the last instruction of program $P = r_1 \leftarrow v_1; \texttt{mem}(v_a, v_2); r_1 \leftarrow v_3; \texttt{mem}(v_b, r_1)$, the leakage is equal to $L(P) \overset{Eq.2}{=\!=} L(r_1 \leftarrow v_1; \texttt{mem}(v_a, v_2); r_1 \leftarrow v_3) \cup \{HW(v_3 \oplus v_2)\} \overset{Eq.1}{=\!=} L(r_1 \leftarrow v_1; \texttt{mem}(v_a, v_2)) \cup \{HW(v_3 \oplus v_2), HW(v_3 \oplus v_1)\} \overset{Eq.4}{=\!=} L(r_1 \leftarrow v_1) \cup \{HW(v_3 \oplus v_2), HW(v_3 \oplus v_1), HW(v_2)\} \overset{Eq.3}{=\!=} \{HW(v_3 \oplus v_2), HW(v_3 \oplus v_1), HW(v_2), HW(v_1 \oplus r_{1,IN})\}$, where $r_{1,IN}$ is the initial value of register r_1.

Here, we consider that a program is a straight-line function. Additional checks at the call site are necessary for ensuring the absence of leakage during function calls, for example to make sure that the initial memory-bus value is constant.

B. Threat Model

We assume that the software runs on an non-speculative hardware architecture. The attacker has access to the software implementation and the *public* data but not the *secret* data. The goal of the attacker is to extract information about the secret data by measuring the power consumption of the device that the code runs on. The attacker may accumulate a number of traces from multiple runs of the program and perform statistical analysis, such as Differential Power Analysis (DPA) [2]. At every execution, new *random* values are generated and the attacker has no knowledge of the values of these variables. Our goal is to eliminate any statistical dependencies between the secret data and the measured power traces.

We assume that input variables are `Secret`, `Public`, or `Random`. `Secret` variables contain sensitive values (e.g. cryptographic keys), which the attacker wants to retrieve information about. `Public` variables contain values that the attacker knows or may learn without causing a leakage. Finally, `Random` variables follow the uniform distribution in the domain of the corresponding program variable. We define the *Leakage Equivalence* security condition for the generated programs as follows:

Definition 1 (Leakage Equivalence)**.** *Given a program $P(\boldsymbol{IN})$ that has a set of secret input variables, $\boldsymbol{IN}_{sec} \subseteq \boldsymbol{IN}$, a set of random input variables, $\boldsymbol{IN}_{rand} \subseteq \boldsymbol{IN}$, and a set of public input variables, $\boldsymbol{IN}_{pub} \subseteq \boldsymbol{IN}$. We assume two instances of the input variables, IN and IN'. These two instances differ with regards to the set of secret variables IN_{sec} and ${IN'}_{sec}$, i.e. for all public variables, $\forall v \in IN_{pub}$ and $\forall v' \in {IN'}_{pub}$ we have $v = v'$. Let $r \in IN_{rand}$ and $r' \in {IN'}_{rand}$ be sampled from a uniform random distribution. Let $L_p = L(P(IN))$ and $L'_p = L(P(IN'))$. Then, we say that a program is leakage equivalent if the distributions of the leakage of the two executions do not differ, i.e.*

$$\sum_{l \in L_p} \mathbb{E}[l] = \sum_{l' \in L'_p} \mathbb{E}[l'] \ \wedge \ \sum_{l \in L_p} Var(l) = \sum_{l' \in L'_p} Var(l'),$$

where $\mathbb{E}[l]$ and $Var(l)$ are l's expected value and variance.

C. HD-based Vulnerability Detection

In our approach, we need a technique to identify whether two values result in a ROT or and MRE leak. There are different ways to identify whether there is a leak at some part of the code. One approach is to use symbolic execution [6, 8]. Symbolic execution executes different paths of a program symbolically and verifies or invalidates specific properties with the help of Satisfiability Modulo Theory (SMT) solvers. Symbolic execution is accurate but has scalability issues when the number of problem variables or program paths increases. On the other end, type-based approaches [20, 7] are typically efficient but at the price of accuracy. In particular, Wang et al. consider a hierarchy of three types based on the properties of the distribution they follow: *uniformly random distribution*, *secret independent distribution*, or finally *unknown distribution*. We call these, `Random`, `Public`, and `Secret`, respectively. The type-inference algorithm assigns a type to each program variable. To infer the program variable types, Wang et al. define a logic model and solve it using an SMT solver. The complexity of this approach is low compared to symbolic execution, at the price of lower accuracy. However, the accuracy is sufficient for loop-free, linearized programs, a format to which many masked and cryptographic implementations can

be transformed [7]. Because of this, our approach adapts the aforementioned type-inference analysis, with some accuracy improvements (see supplementary material [21]).

D. Constraint-based Compiler Backend

A compiler backend performs three main low-level transformations to generate low-level code: instruction selection, instruction scheduling, and register allocation. A combinatorial compiler backend [16, 22, 23] uses combinatorial solving techniques to optimize software using the aforementioned transformations. Different approaches may implement one or more low-level transformations. This section focuses on Constraint Programming (CP) [24] as a combinatorial solving technique.

1) Constraint Model: The constraint-based compiler backend generates a constraint model that captures the program semantics, the low-level compiler transformations, and the hardware architecture. This paper focuses on two compiler transformations, register allocation and instruction scheduling, that are crucial for our mitigation.

Compilers typically model the code using an unbounded number of *virtual* registers until the register allocation stage. Register allocation assigns each virtual register to a hardware register, when possible, or a memory slot on the stack (*spill*), otherwise. The latter has a negative effect on code efficiency. Therefore, register allocation transformations attempt to minimize this effect, while conforming to constraints, such as the number or hardware registers and the calling conventions.

Instruction scheduling decides on the order of the instructions in a program. A valid instruction schedule satisfies the data dependencies among instructions and the processor resource constraints.

A constraint-based compiler backend may be modeled as a Constraint Optimization Problem (COP), $P = \langle V, U, C, O \rangle$, where V is the set of decision variables of the problem, U is the domain of these variables, C is the set of constraints among the variables, and O is the objective function. A constraint-based backend aims at minimizing O, which typically models the code's execution time or size.

A program is modeled as a set of basic blocks B, pieces of code with no branches apart from the exit. Each block contains a number of optional operations, $o \in \mathit{Operations}$, that may be *active* or not. Ins_o denotes the set of hardware instructions that implement operation o. Each operation includes a number of operands $p \in \mathit{Operands}$, each of which may be implemented by different, equally-valued temporaries, $t \in \mathit{Temps}$. Temporaries are either not live or assigned to a register (hardware register or the stack).

Fig. 4 shows a simplified version of the constraint-based compiler backend model for Fig. 1. Temporaries `t0`, `t1`, and `t2` contain the input arguments `pub`, `mask`, and `key`, respectively. Copy operations (`o2, o3, o4, o6, o8`) enable copying program values from one register to another (or to the stack) and are critical for providing flexibility in register allocation. For example, `o2`, allows the copy of the value `pub` from `t0` to `t3`. In the final solution, a copy operation may not be active (shown by the dash in the set of instructions: `[ -, copy]`). The two `xor` operations (`o5, o7`) take two operands each, and each of these operands may use different but equally-valued temporary variables, e.g. `t1` and `t4`.

```
o1: in [t0 ← pub, t1 ← mask, t2 ← key]
o2: t3 ← [-, copy] t0
o3: t4 ← [-, copy] t1
o4: t5 ← [-, copy] t2
o5: t6 ← xor [t1,t4] [t2,t5]
o6: t7 ← [-, copy] t6
o7: t8 ← xor [t0,t3] [t6,t7]
o8: t9 ← [-, copy] t8
o9: out [t10 ← [t8,t9]]
```

Fig. 4: Simplified model of the function in Fig. 1

Fig. 5 shows a valid solution to the register allocation of the constraint model in Fig. 4. All copy operations are deactivated and `t0`, `t1`, and `t2` are assigned to registers `R0`, `R1`, `R2`. Temporary `t6` is assigned to `R1` and temporary `t8` is assigned to `R0`. This register assignment is problematic because it induces a transition in register `R1` from the initial value that holds the `mask` to the masked value `mask` $\oplus$ `key`, which leads to a leakage $L(\texttt{R1} \leftarrow \texttt{R1} \oplus \texttt{R2}; \texttt{R0} \leftarrow \texttt{R0} \oplus \texttt{R1}) \overset{Eq.3}{=\!=} L(\texttt{R1} \leftarrow \texttt{R1} \oplus \texttt{R2}) \cup \{HW(\texttt{pub} \oplus (\texttt{pub} \oplus \texttt{mask} \oplus \texttt{key}))\} \overset{Eq.3}{=\!=} \{HW(\texttt{mask} \oplus (\texttt{mask} \oplus \texttt{key})), HW(\texttt{mask} \oplus \texttt{key})\} = \{HW(\texttt{key}), HW(\texttt{key} \oplus \texttt{mask})\}$. The first element of the leakage reveals information about `key`.

The model of instruction scheduling assigns issue cycles to each operation. This assignment imposes an ordering of the operation and is constrained by the program semantics. For example, in Fig. 4, scheduling `o6` before `o5` is not allowed because `o6` depends on `o5` but scheduling `o4` before `o3` is possible. In Fig. 3b, the store instruction at line 6 (that corresponds to line 5 in Fig. 3a) is scheduled after the load instruction at line 4 (line 6 in Fig. 3a). This is allowed because there is no data dependency between these two instructions.

```
o1: in [t0:R0, t1:R1, t2:R2]
o5: t6:R1 ← xor t1:R1 t2:R2
o7: t8:R0 ← xor t0:R0 t6:R1
o9: out [t10:R0]
```

Fig. 5: Solution of the model in Fig. 4

The decision variables of the constraint problem are:

- $r(t) \in \mathit{Regs}_t$, $t \in \mathit{Temps}$ denotes the hardware register or stack slot assigned to temporary t;
- $a(o) \in [\texttt{false}, \texttt{true}]$, $o \in \mathit{Operations}$ denotes whether operation o is active or not;
- $i(o) \in \mathit{Ins}_o$, $o \in \mathit{Operations}$ is the instruction that implements operation o;
- $c(o) \in [0, \mathit{maxc}]$, $o \in \mathit{Operations}$ is the cycle at which an operation o is scheduled, bounded by maxc, a conservative upper bound of the execution time;

- $y(p) \in Temps_p,\ p \in Operands$ is the selected temporary among all possible temporaries for operand p.

In addition to these, $l(t) \in [\texttt{false}, \texttt{true}],\ t \in Temps$ represents whether a temporary is live or not, $ls(t) \in [0, maxc],\ t \in Temps$ represents the cycle at which t becomes live, and $le(t) \in [0, maxc],\ t \in Temps$ represents the last cycle at which t is live. An important constraint of register allocation is that the register live ranges of a specific hardware register r_i do not overlap:

$$\forall t_1, t_2 \in Temps\ .\ l(t_1)\ \wedge\ l(t_2)\ \wedge\ r(t_1) = r(t_2)\ \Longrightarrow\ ls(t_1) \geq le(t_2)\ \vee\ ls(t_2) \geq le(t_1). \quad (5)$$

Moreover, when a temporary is live, its last live cycle (le) is strictly greater than its live start (ls):

$$\forall t \in Temps\ .\ l(t)\ \Longrightarrow\ ls(t) < le(t). \quad (6)$$

2) Objective Function: A typical objective function of a constraint-based backend minimizes different metrics such as *code size* and *execution time*. These can be captured in a generic objective function that sums up the weighted cost of each basic block:

$$\sum_{b \in B} weight(b) \cdot \textbf{cost}(b).$$

The **cost** of each basic block consists of the cost of the specific implementation and is a variable, whereas *weight* is a constant value that represents the contribution of the specific basic block to the total cost. This cost model is accurate for simple hardware architectures. However, in the presence of advance microarchitectural features, such as complex cache hierarchy, branch prediction, and/or out-of-order execution, the cost model is not accurate.

E. Example in a Constraint-based Compiler Backend

Low-level transformations, like register allocation and instruction scheduling, affect the security of programs. Fig. 6a shows the high-level masked implementation of exclusive OR in C (same as Fig. 1). The code takes three inputs: `p` (a `Public` value), `k` (a `Secret` value), and `m` (a `Random` variable). The code computes first the exclusive OR of `m` and `k` and stores it in `mk`. Then, it computes the exclusive OR of `mk` with `p` and stores it in `rs`, which the function returns.

Fig. 6b shows a register allocation of function `Xor` that leads to a HD vulnerability. Both `m` and `mk` are stored in the same register, hence the content of `mk` replaces the previous value `m` in register `R1`. According to the leakage model, the attacker observes the exclusive OR between the initial and updated value of a hardware register. Using the register allocation of Fig. 6b, the leakage reveals information about the secret: $\text{HW}(\texttt{mk} \oplus \texttt{m}) = \text{HW}((\texttt{m} \oplus \texttt{k}) \oplus \texttt{m}) = \text{HW}(\texttt{k})$. Value `k` is a secret value, and thus, a leak occurs (circled in Fig. 6b).

A constraint-based compiler backend is able to generate all legal register allocations for a program. Fig. 6c shows an alternative register allocation for function `Xor`. Here, the result of `mk` is written in hardware register `R2`, giving a HD leakage $\text{HW}(\texttt{mk} \oplus \texttt{k}) = \text{HW}((\texttt{m} \oplus \texttt{k}) \oplus \texttt{k}) = \text{HW}(\texttt{m})$. The leakage here corresponds to the value of `m`, which is not a sensitive value. In a similar way, instruction scheduling may be able to remove leakages as seen in Fig. 3. By changing the schedule of the instructions, the model is often able to generate a PSC-free solution with no code quality overhead.

This example shows that low-level transformations can be responsible for the introduction of HD vulnerabilities and have thus to be taken into account to provide effective mitigations.

IV. SecCG

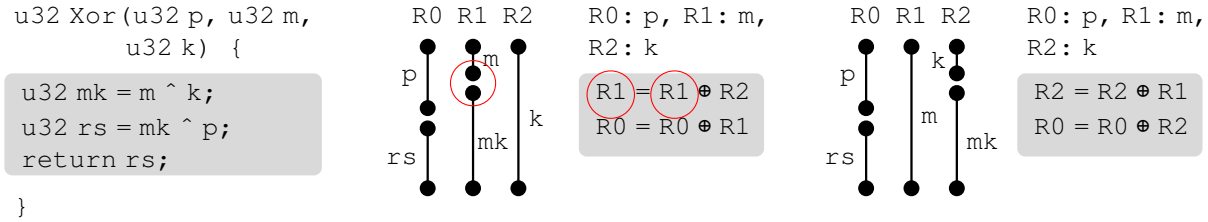
This section introduces SecCG, an approach to optimize code that is secure against PSC attacks. Fig. 7 shows the high-level view of SecCG. SecCG is a constraint-based optimizing secure compiler, i.e. it extends a constraint-based compiler backend with security constraints. It takes two inputs: 1) a C or C++ program, and 2) a security policy denoting which variables are `Secret`, `Random`, or `Public`. SecCG enables *register promotion* at the compiler middle end because this optimization preserves the high-level properties of the program and, at the same time, creates substantial opportunities for register allocation. Then, the constraint-based compiler backend, extended with security constraints, takes as input the program in a machine-level Intermediate Representation (IR) and the security policy. Next, SecCG performs a security analysis (see Section III-C). The results are used to impose constraints that prevent HD vulnerabilities. Given the secure model, the approach generates an optimized solution.

Section IV-A presents the security analysis. Section IV-B presents the secure constraint model that extends the constraint-based compiler backend. Finally, Section IV-C presents the solving enhancements of SecCG.

A. Security Analysis

SecCG performs a security analysis to extract the security types of each program variable and, subsequently, generates constraints that prohibit insecure low-level implementations. The security analysis identifies the security type (`Random`, `Public`, or `Secret`) of each intermediate variable. In the compiler constraint model, the program variables correspond to the input arguments, the operands and the result of each operation. This is equivalent to the temporary variables, i.e. the virtual registers. Each operand can use a number of alternative temporary values $t \in Temps$ and each temporary value is assigned to a register (see Section III-D). The type-inference rules do not handle loops or conditional statements. However, cryptographic implementations that are free from PSCs are often linearizable [7].

The security analysis uses a type-inference algorithm based on Wang et al. [7]. We extend this algorithm with additional definitions that improve the accuracy of the type inference (see supplementary material [21]). In particular, we extend the type-inference algorithm with rules that consider additional properties of GF(2^n), like distributivity between exclusive or ($\oplus$) and multiplication in GF(2^n) ($\odot$). At the end of the analysis, all temporary variables have an inferred type. Fig. 8 shows the inferred security types for each of the temporaries



(a) Exclusive OR in C (b) Vulnerable register assignment (c) Secure register assignment

Fig. 6: The exclusive OR example, illustrating a HD vulnerability and alternative register assignments

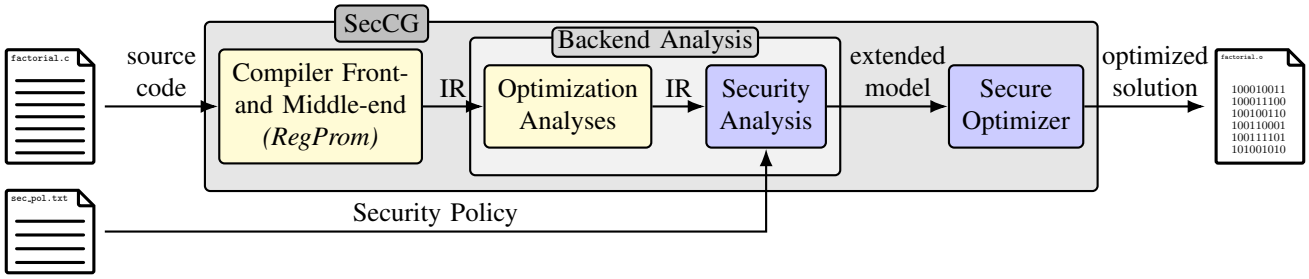


Fig. 7: High-level view of SecCG

in our running example. Temporaries `t0` and `t3` are `Public` (green), `t2` and `t5` are `Secret` (red), and `t1`, `t4` and `t6-t10` are `Random` (brown).

```
o1: in [t0:Public, t1:Random, t2:Secret]
o2: t3:Public ← [-, copy] t0
o3: t4:Random ← [-, copy] t1
o4: t5:Secret ← [-, copy] t2
o5: t6:Random ← xor [t1,t4] [t2,t5]
o6: t7:Random ← [-, copy] t6
o7: t8:Random ← xor [t0,t3] [t6,t7]
o8: t9:Random ← [-, copy] t8
o9: out [t10:Random ← [t8,t9]]
```

Fig. 8: Typed intermediate representation

The type-inference algorithm is conservative. Function $type(t) : Temps \rightarrow \{R, S, P\}$ returns the type assigned to temporary variable t. This section abbreviates the types as follows: type R corresponds to `Random`, S corresponds to `Secret`, and P corresponds to `Public`.

In the following, we define the data that the security analysis provides to the constraint model, which the latter requires to impose security constraints. According to the leakage model, when a hardware register changes from one value to another, the exclusive OR of the two values is exposed. $Rpairs$ is the set of temporary variable pairs that when xor:ed together reveal secret information:

$$Rpairs = \{(t_1, t_2) \mid t_1, t_2 \in Temps \ \wedge \\ type(t_1), type(t_2) \in \{R, P\} \ \wedge \\ type(t_1 \oplus t_2) = S\}. \quad (7)$$

In the running example (Fig. 8), $Rpairs = \{$(`t1`,`t6`), (`t1`,`t7`), (`t1`,`t8`), (`t1`,`t9`), (`t4`,`t6`), (`t4`,`t7`), (`t4`,`t8`), (`t4`,`t9`), (`t6`,`t7`), (`t6`,`t8`), (`t6`,`t9`), (`t7`,`t8`), (`t7`,`t9`), (`t8`,`t9`)$\}$. For every pair of temporaries in $Rpairs$, a constraint prohibits the contiguous assignment of the temporaries to the same register (`m` and `mk` in Fig. 6b).

$Rpairs$ do not consider secret values. Instead, if the type of a temporary variable t is `Secret`, we impose a different constraint because the secret information will always result in a leak. In this case, we impose the constraint that another random variable should precede and follow the definition of the secret variable to mask the secret information. $Spairs$ is a set of pairs, each of which consists of a secret temporary variable t and a set of random temporary variables ts that hide the secret information, i.e. $\forall t' \in ts \ . \ type(t' \oplus t) = R$:

$$Spairs = \{(t, ts) \mid t \in Temps \ \wedge \ type(t) = S \ \wedge \\ ts = \{t' \mid t' \in Temps \ \wedge \ type(t') = R \ \wedge \\ type(t' \oplus t) = R\}\}. \quad (8)$$

In the running example (Fig. 8), $Spairs = \{$(`t5`,$\{$`t4`,`t6`, `t7`,`t8`,`t9`$\}$)$\}$.

Memory operations may also reveal secret information. We assume the same leakage model (HD model) for the memory bus as for the register-reuse transitional effects. This means that the leakage corresponds to the exclusive OR of two subsequent memory operations. $Mmpairs$ includes the pairs of memory operations that result in memory-bus transitional leakage, i.e. pairs of memory operations that when scheduled subsequently lead to a secret leakage:

$$Mmpairs = \{(o_1, o_2) \mid o_1, o_2 \in MemOperations \ \wedge \\ type(tm(o_1)), type(tm(o_2)) \in \{R, P\} \ \wedge \\ type(tm(o_1) \oplus tm(o_2)) = S\}. \quad (9)$$

Here, $tm(o) \in Temps$ is the temporary that corresponds to the memory data of the operation. In the running example (Fig. 8),

$Mmpairs = \{(o3, o6), (o3, o8), (o6, o8)\}$, in case o3, o6, o8, are memory spills. Note that, for simplicity, Fig. 8 does not include all copies for memory spilling as we would need to duplicate the copies for first storing and then loading the variables.

The same leakage as in the case when a secret value was written to a register applies here. If a memory operation stores/loads a secret value to/from the memory, a random memory operation that is able to hide the secret information should precede and follow this operation. $Mspairs$ is a set of pairs, each of which consists of the memory operation that accesses secret data, o, and a set of memory operations that access random data and are able to hide the secret information, i.e. $type(tm(o') \oplus tm(o)) = R$:

$$\begin{aligned} Mspairs = \{(o, os) \mid o \in MemOperations \ \wedge \\ type(tm(o)) = S \ \wedge \\ os = \{o' \mid o' \in MemOperations \ \wedge \\ type(tm(o')) = R \ \wedge \\ type(tm(o') \oplus tm(o)) = R\}\}. \end{aligned} \quad (10)$$

In the example (Fig. 8), $Mspairs = \{(o4, \{o3, o6, o8\})\}$, in case o4, o3, o6, and o8 are spilled in memory.

The security analysis provides $Rpairs$, $Spairs$, $Mmpairs$, and $Mspairs$ to the constraint model, which enables constraining code generation to generate secure implementations.

B. Constraint Model

The constraint model takes as input the four sets computed by the security analysis ($Rpairs$, $Spairs$, $Mmpairs$, and $Mspairs$) and uses them to generate appropriate constraints that prohibit insecure solutions.

Predicate samereg tells whether the two input temporaries are active (l(t) = 1) and are assigned to the same register.

```
pred samereg(t1,t2):
  l(t1) ∧ l(t2) ∧ (r(t1) = r(t2))
```

In Fig. 5, samereg(t0,t8) = l(t0) ∧ l(t8) ∧ (r(t0) = r(t8)) = true, samereg(t2,t6) = false (r(t2) ≠ r(t6)), and samereg(t1,t7) = false (t7 is not live).

1) Rpairs Constraints: The following constraint ensures that a pair of temporaries in $Rpairs$ are either not assigned to the same register or they are not subsequent (subseq constraint, defined in Section IV-B5).

```
forall (t1,t2) in Rpairs:
  samereg(t1, t2) ⟹
    (¬subseq(t1,t2) ∧ ¬subseq(t2,t1))
```

In Fig. 5, this constraint is not satisfied for t1 and t6 because samereg(t2,t6) = true and subseq(t2,t6) = true.

2) Spairs Constraints: The following constraint ensures that for each pair $(t_s, t_{rs}) \in Spairs$, if t_s is live, one of the random temporaries $t_r \in t_{rs}$ precedes the secret temporary t_s and another random temporary succeeds t_s.

```
forall (ts,trs) in Spairs:
  exists tr in trs:
      l(ts) ⟹ (l(tr) ∧ subseq(tr,ts)) ∧
  exists tr in trs:
      l(ts) ⟹ (l(tr) ∧ subseq(ts,tr))
```

Fig. 9 shows a solution to the model in Fig. 4, where both the $Rpairs$ and the $Spairs$ constraints are satisfied. t5 is active but is assigned to the same register as t4, which precedes t5 and thus eliminates the leakage. Similarly, t6 follows the assignment of t5 and thus hides the secret value.

```
o1: in [t0:R0, t1:R1, t2:R2]
o3: t4:R3 ← t1:R1
o4: t5:R3 ← t2:R2
o5: t6:R3 ← xor t1:R1 t5:R3
o7: t8:R0 ← xor t0:R0 t6:R3
o9: out [t10:R0]
```

Fig. 9: Solution of the model in Fig. 4

3) Mmpairs Constraints: The following constraint ensures that a pair of non-secret memory operations in $Mmpairs$, are either not active or not subsequent memory operations (msubseq constraint). Constraint msubseq (defined in Section IV-B5) is similar to subseq but considers consecutive memory operations instead of temporaries.

```
forall (o1,o2) in Mmpairs:
  a(o1) ∧ a(o2) ⟹
    (¬msubseq(o1,o2) ∧ ¬msubseq(o2,o1))
```

4) Mspairs Constraints: Finally, the following constraint ensures that for each pair $(o_s, o_{rs}) \in Mspairs$ a random memory operation $o_r \in o_{rs}$ precedes the secret-dependent memory operation o_s.

```
forall (os,ors) in Mspairs:
  exists or in ors:
    a(os) ⟹ (a(or) ∧ msubseq(or,os)) ∧
  exists or in ors:
    a(os) ⟹ (a(or) ∧ msubseq(or,os))
```

This constraint works similarly as the equivalent register constraint, where instead of register operations, we have memory operations. In our example, we need to have memory spilling, i.e. store to the stack, and then load from the stack (only one of the operations is shown in Fig. 9).

5) Modeling subseq: To define the subseq constraint, we first define an auxiliary predicate is_before and a set of auxiliary problem variables lk. Predicate is_before(t1, t2) tells whether t1 is assigned to the same register as t2 and t1's life range ends (le(t1)) before the beginning of the life range of t2 (ls(t2)).

```
pred is_before(t1,t2): same_reg(t2, t1) ∧
                       (le(t2) ≤ ls(t1))
```

Variable lk(t) captures the end live cycle of the temporary that occupied the same register as t (r(t)) right before t

was assigned. If t' = lk(t), then the values of t and t' result in a transitional effect that may reveal information to the attacker.

```
forall t in Temps: lk(t) = max(
  [ite(is_before(t',t),le(t'),-1)
                  | forall t' in Temps])
```

Then, the definition of the subseq predicate is as follows:

```
pred subseq(t1,t2):
  samereg(t1,t2) ∧ (lk(t2) = le(t1))
```

Theorem 1 (Subseq Constraint)**.** *The* subseq *constraint is true only for pairs of temporary variables that are subsequently assigned to the same register:*

$subseq(t_1,t_2) \iff P = P';t_1 \leftarrow e_1;P'';t_2 \leftarrow e_2;P''' \wedge r(t_1) = r(t_2) \wedge \forall i \in P'' \,.\, i = t \leftarrow e \implies r(t) \neq r(t_1)$.

Proof. ($\Leftarrow$) Assume $P = P';t_1 \leftarrow e_1;P'';t_2 \leftarrow e_2;P''' \wedge r(t_1) = r(t_2) \wedge \forall i \in P'' \,.\, i = t = e \wedge r(t) \neq r(t_1)$. We consider all register assignments in P: $P = ...;t_i \leftarrow e_i;...;t_1 \leftarrow e_2;...;t_2 \leftarrow e_2;...;t_j \leftarrow e_j...$; all these assignments are live because they appear in the final program. For all assignments t_j following t_i we have that $le(t_j) > ls(t_2)$, which implies that is_before(t_j,t_i) = false, and thus all t_j contribute with -1 to max in $lk(t_2)$. The same applies for all registers that are assigned to a different register, they contribute with -1 because is_before(t_j,t_i) = false. Then, $lk(t_2) = max(le(t)|t \in \{t_{i_1},t_{i_2},..,t_1\})$, where all $\{t_{i_1},t_{i_2},..,t_1\}$ are assigned the same register, $r(t_2)$. Because these temporaries are assigned to the same register, their live ranges do not overlap (Equation 5), i.e. $\forall t,t' \in \{t_{i_1},t_{i_2},..,t_1\} \,.\, ls(t) \geq le(t') \vee ls(t') \geq le(t)$. Because $t_1 \leftarrow e_1$ is scheduled last $\forall t \in \{t_{i_1},t_{i_2},..,t_{i_n},t_1\} \,.\, ls(t_1) \geq le(t)$. Also, from Equation 6, $le(t_1) > ls(t_1)$. This implies that $\forall t \in \{t_{i_1},t_{i_2},..,t_{i_n}\} \,.\, le(t_1) > le(t)$, so we have $lk(t_2) = le(t_1)$ and $\forall t \in \{t_{i_1},t_{i_2},..,t_{i_n}\} \,.\, lk(t_2) > le(t)$. Therefore only for t_1, subseq(t_1,t_2) = true.

($\Rightarrow$) Assume subseq(t_1,t_2). This implies that samereg$(t_1,t_2) \wedge lk(t_2) = le(t_1)$. Constraint samereg$(t_1,t_2)$ implies that $r(t_1) = r(t_2)$ and $l(t_1) \wedge l(t_2)$, which means that they appear in the final code, P, and are assigned to the same register. Because $lk(t_2) = le(t_1)$, t_1 is scheduled before t_2 or $P = P';t_1 \leftarrow e_1;P'';t_2 \leftarrow e_2;P'''$. Now, we only need to prove that there is no other assignment of $r(t_1)$ in P'', i.e. $\forall i \in P'' \,.\, t \leftarrow e \wedge r(t) \neq r(t_1)$. If $\exists i \in P'' \,.\, t \leftarrow e \wedge r(t) = r(t_1)$, then, because live ranges do not overlap, $le(t) > le(t_1)$, which means that $lk(t_2) = le(t), \neq le(t_1)$, which is invalid.

□

For the definition of msubseq, we define an auxiliary predicate is_before_mem and auxiliary problem variables ok. Predicate is_before_mem(o1, o2) tells whether o1 is scheduled before o2.

```
pred is_before_mem(o1,o2):
    a(o1) ∧ (c(o1) ≤ c(o2))
```

In Fig. 9, is_before_mem(o4, o3) is true.

Variable ok(o) captures the issue cycle of memory operation $o' \in$ MemOperations that was issued before o.

```
forall o in MemOperations: ok(o) = max(
  [ite(is_before_mem(o', o), c(o'), -1)
          | forall o' in MemOperations])
```

Similar to predicate subseq, msubseq is as follows:

```
pred msubseq(o1,o2):
  a(o1) ∧ a(o2) ∧ ok(o2) = c(o1)
```

Theorem 2 (Msubsec Constraint)**.** *The* msubseq *constraint is true only for two instructions that are subsequently accessing the memory:* $msubseq(o_1,o_2) \iff P = P';o_1;P'';o_2;P''' \wedge \nexists o \in P'' \,.\, o = mem(e'',e_3)$*, where* o_1 *and* o_2 *are memory operations,* $o_1 = mem(e,e_1)$ *and* $o_2 = mem(e',e_2)$.

Proof. Similar to Theorem 1. □

Theorem 3 shows that SecCG generates secure code for our threat model.

Theorem 3 (Secure Modeling)**.** *A program P, generated by SecCG, satisfies the leakage equivalence condition in Definition 1. This means that given two input instances IN, IN′ that differ only with regards to the secret variables,* $IN_{sec} \subseteq IN$, $IN'_{sec} \subseteq IN'$*, the distributions of the leakages do not differ.*

Proof. We assume that the type-inference algorithm overapproximates the actual distribution of each variable. Then, we perform structural induction on the program P to prove that security constraints we introduce lead to secure programs. The proof is available as supplementary material [21]. □

C. Solving Enhancements

Large problems in combinatorial solving can quickly become difficult to handle due to state-space explosion. A solution to this problem is structural decomposition of the problem into subproblems. In code generation, a natural structural decomposition scheme consists of splitting the problem into basic blocks [16]. However, SecCG's security analysis [7] requires linearized code that corresponds to one large basic block. There are already approaches on splitting large code blocks into smaller artificial code blocks for improving the scalability of the solver [16]. Typically, in decomposition schemes, the solver first solves each partial solution (basic blocks) and then composes a full solution consisting of the partial solutions. However, this solution becomes challenging with the addition of security constraints that relate different parts of the code, introducing new inter-block dependencies. These dependencies may lead to conflicts between the partial solutions resulting in the rejection of the full solution. To deal with this problem, SecCG propagates only part of the partial solutions, leaving some parts of the full solution unsolved. In particular, SecCG does not propagate the register assignments to temporaries that correspond to earliest and latest assigned hardware registers in each basic block, as well as their

corresponding issue cycles. Subsequently, SecCG solves the unsolved parts as part of the full problem.

The second main enhancement to the solving procedure concerns the final step of the solving process. In SecCG we make use of Large Neighborhood Search (LNS) [25], a form of local search for constraint programming. In particular, at the end of the decomposition phase, SecCG uses the best found solution to perform local search and locate better solutions.

V. Evaluation

For the evaluation of SecCG, we pose the following research questions:

RQ1: What is the overhead in execution time for the generated code using SecCG? Here, we want to evaluate the introduced overhead of secure solutions compared to optimized but insecure solutions. To do that, we compare the best known solution [16] with our approach SecCG.

RQ2: What is the improvement in execution time of the generated code over non-optimized code and other TBL-secure approaches? Here, we compare our results with LLVM-3.8 with no optimization (-O0) and the work by Wang et al. [7].

RQ3: What is the overhead in compilation time using SecCG? Here, we want to evaluate the introduced compilation overhead of secure solutions compared to insecure solutions. To do that, we compare the compilation time for retrieving the best known solution [16] with SecCG's compilation time.

A. Preliminaries

The following sections describe the implementation details and the experimental setup of the evaluation of SecCG. The implementation of SecCG and the experiments and benchmarks for the evaluation are available at https://github.com/romits800/seccon_experiments.git.

1) Implementation Details: SecCG is implemented as an extension of Unison [16], a constraint-based compiler backend that uses CP to optimize software functions with regards to code size and execution time. In particular, Unison combines two low-level optimizations, instructions scheduling and register allocation, and achieves optimizing medium-size functions with improvement compared to LLVM. Unison uses two global constraints for modeling the backend transformations; 1) the *geometric packing constraint* for register allocation and 2) the *cumulative* constraint for instruction scheduling. The type-inference implementation is written in Haskell and is based on Wang et al. [7] with precision improvements (see supplementary material [21]).

2) Experimental Setup: All experiments run on an Intel®Core™i9-9920X processor at 3.50GHz with 64GB of RAM running Debian GNU/Linux 10 (buster). We use LLVM-3.8 as the front-end compiler for these experiments. To preserve the high-level security properties of the compiled programs, we apply only one optimization, register promotion, (`-mem2reg` in LLVM), which lifts program variables from the stack to registers. We evaluate our method on two architectures: ARM Thumb, targeting processor ARM Cortex M0, a highly predictable processor targeting small embedded devices; and Mips32, a widely-used embedded architecture.

We implemented the constraint model both as part of the specialized Gecode [26] constraint model and the Minizinc [27] model that Unison provides. The Minizinc model allows for solving the problem using multiple solvers. In total, we tried four solvers, Chuffed v0.10.3 [28], OR-Tools [29], Elsie Geas[1], and the specialized model written in Gecode v6.2. We ran the former three solvers activating the *free-search* option. For the specialized model in Gecode, apart from the security model, SecCG includes the modified search enhancements that we describe in Section IV-C. Among all these solvers, Gecode and Chuffed performed best. None of them was able to solve all the problems but together they could solve most of the problems. In the smaller benchmarks, P0-P6, we run a portfolio solver including Gecode and Chuffed. For the larger benchmarks, we run every solver separately for reducing the risk of out-of-memory errors when running both solvers in parallel. The presented results are the result of five runs for SecCG and Unison, whereas for the calculation of the execution time for LLVM -O0, we run the compilation 1000 times to account for possible fluctuations in the compilation time on the evaluation machine.

3) Benchmarks: To evaluate our approach, we use a set of small benchmark programs, up to 100 lines of C code and one program exceeding 900 lines of C code. Table I provides a description of these benchmarks, including the number of lines of code (LoC), and the program variables, i.e. the input variables (IN) and the number of secret (IN_{sec}), public (IN_{pub}), and random (IN_{rand}) input variables. Benchmarks P1 to P6 and P8 to P11 were made available by Wang et al.[2] [7], whereas P0 and P7 are implemented by the authors of this paper. These benchmark programs constitute different masked implementations from previous work and are linearized. Wang et al. [7] use a larger number of benchmarks to evaluate their approach. However, our approach depends on a combinatorial optimizing compiler, Unison, which scales to up to medium size functions, namely, up to approximately 200 intermediate instructions for ARM Cortex M0 and Mips32 architectures [16]. In addition to this, SecCG adds additional constraints that increase the complexity of the model (see Section V-D). Therefore, we selected the smallest benchmarks for our experiments. As a future work, we plan to investigate non-linearized implementations, but this comes at the expense of analysis precision and potentially increased performance overhead.

B. RQ1: Optimality Overhead

SecCG builds on a constraint-based compiler backend to generate a program that satisfies security constraints for software masking. This means that our approach might compromise some of the code quality of the non-mitigated optimized code to mitigate the software masking leaks. To evaluate

[1]Elsie Geas: https://bitbucket.org/gkgange/geas/src/master/

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

TABLE I: Benchmark Description

Prg	Description	LoC	Input Variables (IN)		
			pub	sec	rand
P0	Xor (Listing 1)	5	1	1	1
P1	AES Shift Rows [6]	11	0	2	2
P2	Messerges Boolean [6]	12	0	1	2
P3	Goubin Boolean [6]	12	0	1	2
P4	SecMultOpt_wires_1 [4]	25	1	1	3
P5	SecMult_wires_1 [4]	25	1	1	3
P6	SecMultLinear_wires_1 [4]	32	1	1	3
P7	Whitening [6]	58	16	16	16
P8	CPRR13-lut_wires_1 [5]	81	1	1	7
P9	CPRR13-OptLUT_wires_1 [5]	84	1	1	7
P10	CPRR13-1_wires_1 [5]	104	1	1	7
P11	KS_transitions_1 [30]	964	1	16	32

TABLE II: Optimal solution by Unison and SecCG (SCG) in cycles; Oh stands for overhead; OM stands for *out of memory*

Prg	ARM Cortex M0			Mips32		
	[16]	SCG	Oh (%)	[16]	SCG	Oh (%)
P0	6	6	0	3	3	0
P1	8	9	13	5	5	0
P2	10	10	0	7	7	0
P3	13	13	0	9	9	0
P4	28	28	0	75	75	0
P5	28	28	0	75	75	0
P6	30	30	0	73	73	0
P7	125	128	2	184	184	0
P8	85	85	0	151	151	0
P9	79	82	4	151	151	0
P10	85	81	-5	281	281	0
P11	2635	OM	-	1335	OM	-

the overhead of our method compared to non-secure optimization, we compare the execution time of the optimized solution (optimal or suboptimal solution) that Unison [16] generates compared with SecCG's optimized and TBL-secure code. The overhead is computed as $(cycles(SecCG) - cycles(Unison))/cycles(Unison)$.

Table II shows the mean execution time for each of the benchmark programs and architectures. In particular, for each of the architectures, we compare the execution time in number of cycles of the solution that Unison produces against SecCG's solution. The final column shows the overhead of SecCG compared to Unison.

The results show zero overhead for Mips32, and a maximum 13% overhead in ARM Cortex M0. The zero overhead for most of the benchmarks shows that the Pareto front of optimal solutions synthesized by Unison includes code variants that are secure. This result is in agreement with previous work [31], which shows the existence of multiple optimal (or best found) solutions. For ARM Cortex M0, programs P1, P7, and P9 have a non-zero positive overhead. The observed overhead in ARM Cortex M0 is due to three main reasons: 1) the mitigation itself that may require the introduction of redundant operations in the generated code, 2) the scalability issue that appears in larger functions due to the addition of new security constraints in the order of $|Temps|^2$, and 3) the decomposition mode that may fail to compose solutions (Section IV-C). Program P10 shows a slight improvement. This improvement is due to the introduction of LNS at the end of the solving stage (see Section IV-C), which is not present in Unison. The last benchmark program, P11, demonstrates the scalability limits of our approach. The operating system terminates the solving process because the process attempts to allocate more than the available memory (out-of-memory error).

To summarize, SecCG does not introduce significant overhead over the non-secure optimized solution that Unison generates. This means that in most cases, there is space for generating secure code without affecting the quality of the generated code.

C. RQ2: Execution-Time Improvement

To evaluate the execution-time speedup of our approach, we compare SecCG with the code generated by LLVM without optimizations (-O0). We also compare SecCG with the work by Wang et al. [7]. Wang et al. identify and mitigate ROT leaks on non-optimized code from LLVM 3.6. This is a common approach by different security mitigations, because compilation passes may violate the security properties of a program. During their mitigation, Wang et al. may remove unused code [7], which reduces the overhead.

We compare SecCG with the approach by Wang et al. [7] for three main reasons, 1) their tool is available freely, 2) they propose an architecture-agnostic approach that applies to both Mips32 and ARM Thumb, and 3) they mitigate transitional effect caused by register reuse, a subset of our mitigation. Table III compares the execution time in number of cycles (based on a LLVM-derived cost model) of LLVM, the mitigated code by Wang et al. [7] and SecCG, for each of the programs and architectures. Speedup is computed as $cycles(SecCG)/cycles(LLVMO0)$.

For ARM Cortex M0, the speedup ranges from 2.9 for P5 to 6.3 for P2 and a geometric mean of 3.9 speedup. We notice that for all benchmarks, SecCG achieves significant improvement over the baseline. The main reason for this, is that the increased size of the program under analysis reduces the ability of the solver to find optimal solutions.

For Mips32, the improvement ranges from 77% to 6.6 speedup and a geometric mean of 3.15 speedup. The improvement is larger for smaller benchmarks due to the large overhead of `load` and `store` instructions that are present in the absence of optimizations in the baseline. In contrast to the non-optimized code, the code generated by SecCG reduces memory spilling. In particular, the generic cost model for Mips32 that we use (derived from LLVM) has an one cycle overhead compared to linear instructions. For larger programs, P4-P10, the speedup is smaller but still significant.

This experiment shows that for both architectures SecCG achieves improvement ranging from 77% up to a speedup of 6.6 with geometric-mean speedups 3.9 and 3.15 for ARM Cortex M0 and Mips32, respectively. Although not completely comparable with SecCG because of the use of different benchmarks and mitigations, Vu et al. show an improvement over non-optimized code (-O0) that ranges from 20% to a speedup of 12.6, with a geometric mean of 2.8 [15]. Compared to the

approach by Wang et al., the speedup that SecCG achieves ranges from 1.95 (24%) to 7.6 for ARM Cortex M0 and from 1.36 (36%) to 7.7 for Mips32. The geometric-mean speedups are 3.52 for ARM Cortex M0 and 2.9 for Mips32.

To summarize, for both Mips32 and ARM Cortex M0, SecCG improves the non-optimized LLVM code. We notice large improvements for both Mips32 and ARM Cortex M0 ranging from 77% to 6.6 speedup. SecCG generates also improved code compared to the work by Wang et al. [7].

TABLE III: Execution-time comparison between the non-optimized baseline and SecCG (SCG); Su is the speedup of SecCG with LLVM with -O0 as baseline; OM stands for *out of memory*

Prg	ARM Cortex M0				Mips32			
	O0	[7]	SCG	Su	O0	[7]	SCG	Su
P0	20	22	6	3.33	19	23	3	6.33
P1	39	32	9	4.33	33	21	5	6.60
P2	63	76	10	6.30	43	43	7	6.14
P3	52	56	13	4.00	47	47	9	5.22
P4	87	96	28	3.11	139	139	75	1.85
P5	81	90	28	2.89	133	133	75	1.77
P6	112	69	30	3.73	189	188	73	2.59
P7	609	786	128	4.76	382	430	184	2.08
P8	293	166	85	3.45	371	253	151	2.46
P9	301	303	82	3.67	371	371	151	2.46
P10	333	176	81	4.11	593	383	281	2.11
P11	4504	6742	OM	-	3688	3237	OM	-

D. RQ3: Compilation Overhead

To evaluate the compilation overhead of our approach, we compare SecCG with Unison [16] and non-optimized LLVM. The main reason for the compilation overhead of SecCG compared to LLVM is the combinatorial nature of the backend compiler. Compared to Unison, SecCG introduces compilation overhead due to the security constraints among temporaries and operations in the combinatorial model. In particular, the `subseq` constraint introduces a large number of constraints and variables that are in the order of $|Temp|^2$. The constraints between memory operations (`msubseq`) are typically fewer because memory operations are a subset of all operations. In general, the actual overhead depends on the program logic and the security policy. The compilation slowdown is computed as $comp\_time(SecCG)/comp\_time(Unison)$.

Table IV compares the compilation time of SecCG, Unison, and LLVM -O0. The last column for each architecture in Table IV presents the slowdown of SecCG compared to Unison. In Mips32, we can see an increasing overhead in the compilation time of SecCG compared to Unison with the increase of the function size. The largest compilation overhead is for P10 and corresponds to 57.4 slowdown compared to Unison. The compilation time for non-optimized LLVM ranges from 0.01 to 0.04 seconds. Comparing SecCG with LLVM, the slowdown ranges from 300 for P0 to 300K for P10 (the slowdown does not appear in Table IV)

In the case of ARM Cortex M0, we observe a similar trend. We observe the largest slowdown for P9 which corresponds to 27.9 slowdown. However, the compilation time increases faster than for Mips32. Compared with LLVM, SecCG results in a slowdown that ranges from 29 for P0 to 400K for P9 (does not appear in Table IV). The main reasons for the observable difference between the two architectures are 1) the ARM Thumb architecture is more constrained[3] and 2) interestingly, most instances for Mips32 are solved quickly by Chuffed, whereas most instances for ARM Cortex M0 are only solved by Gecode.

To summarize, the compilation time for SecCG is multiple times slower than Unison because of the introduction of security constraints. In addition to this, SecCG is slower than LLVM. Therefore, we believe that SecCG is mostly suitable for compiling small cryptographic kernels that are both critical for the performance and the PSC security, such as secure field multiplication for AES [4].

TABLE IV: Compilation overhead for SecCG (SCG) compared to Baseline (Unison) in seconds; Sd stands for slowdown of SecCG compared to Unison [16]; OM stands for "out of memory"

Prg	ARM Cortex M0				Mips32			
	O0	[16]	SCG	Sd	O0	[16]	SCG	Sd
P0	0.01	0.17	0.29	1.7	0.01	0.43	2.9	6.6
P1	0.01	0.23	3.4	14.7	0.01	0.52	5.1	9.9
P2	0.01	0.32	1.2	3.7	0.01	0.69	6.5	9.5
P3	0.01	7.7	22.9	3.0	0.01	0.84	8.9	10.6
P4	0.01	1K	1K	1.0	0.01	1.2	16.0	13.5
P5	0.01	1K	1K	1.0	0.01	1.2	16.0	13.7
P6	0.01	1K	1K	1.0	0.01	1.3	18.6	14.3
P7	0.02	1.0K	4K	4.7	0.02	6.3	0.1K	17.2
P8	0.01	0.1K	3K	19.4	0.01	35.0	1K	31.3
P9	0.01	0.1K	4K	27.9	0.01	37.0	1K	27.6
P10	0.02	0.4K	7K	18.0	0.01	47.8	3K	57.4
P11	0.04	5K	OM	-	0.04	52K	OM	-

E. Threat to Validity

Our model considers the HD leakage model and generates code that mitigates these leakages. The security guaranties for our model depend on the HD leakage model. The HD model has been used both for designing defenses [7] and attacks [19]. However, the HD model does not express precisely the actual leakage model for some devices [32]. Moreover, an HD-based mitigation at the assembly level may not hold in the presence of advance microarchitectural features, such as out-of-order execution and write buffers. In addition to this, SecCG does not handle transitional effects through value interaction in the pipeline stage registers and in the memory. We leave further improvement of the hardware model as a future work.

SecCG is not a verified compiler approach like CompCert [33]. Unison, the constraint-based backend that SecCG depends on is based on a formal model that implements standard optimizations but the external solvers and the tool implementation are not verified. Verification of constraint solvers is an active research topic [34].

[3]ARM Cortex M0 has fewer general-purpose registers than MIPS32 and includes two-address instructions, which restrict register allocation.

VI. Related Work

The following sections discuss the related work, with regards to mitigations against side-channel attacks, mitigations against TBLs, and combinatorial compilation approaches. Athanasiou et al. consider two types of PSC leakage sources, Value-Based Leakage (VBL) and Transition-Based Leakage (TBL). VBL occur due to the absence or compiler-induced removal of masking. For example, a compiler transformation may convert a masked expression $\texttt{pub} \oplus (\texttt{mask} \oplus \texttt{key})$ to $\texttt{mask} \oplus (\texttt{pub} \oplus \texttt{key})$, which preserves the code semantics but breaks the masking mitigation. On the other end, TBL is a result of low-level microarchitectural features such as register reuse, memory overwrite, or interactions between values in the hardware. In the following, we will use these two terms to describe different mitigations.

TABLE V: Mitigation approaches against side-channel attacks; SCG stands of SecCG, FE, ME, BE stands for front end, middle end, and back end, respectively; ASM stands for assembly

Pub.	Mitigation	Transf.	InL	OutL	ML	Avail.
[36]	VBL	FE, ME	DSL	-	Custom	✗
[37]	TSC, MS, RS	-	DSL	ASM	Custom	✓
[38]	TSC, MS	-	DSL	ASM	Custom	✓
[39]	TSC, MS	-	DSL	C	F_{low}	✓
[40]	TSC	ME	DSL	C	Custom	✓
[13]	TBL	BE	AVR	AVR	Binary	✓
[41]	IFL	BE	C	ASM	CompCert	?
[7]	TBL	BE	C, C++	ASM	LLVM	✓
[35]	TBL	-	ARM	ARM	Binary	✓✗
[15]	VBL, TSC, FI	ALL	C, C++	ASM	LLVM	✗
[11]	TBL	-	ARM	ARM	Binary	✓
SCG	TBL	ME, BE	C, C++	ASM	LLVM	✓

Side-Channel Compiler Mitigation Approaches: General purpose optimizing compilers perform transformations that may invalidate high-level security mitigations or introduce security flaws [42]. Table V presents a non-exhaustive list of related work that present compiler-based or binary-rewriting approaches against side-channel attacks. For each publication (Publication), Table V, shows the mitigations of each approach (Mitigation), the compiler level that each approach perform the mitigation (Transformation), the input language (InL), the output language (OutL), the Mitigation Level (ML) of each approach that is either a compiler or binary. The last column (Avail.) denotes with ✗ that the artifact is not available, with ✓ that the artifact is available, with ✓✗ that part of the artifact is available, and finally, with **?** where it is not clear whether the artifact is available or not.

Multiple approaches present compiler-based mitigations against Timing Side Channels (TSCs) [37, 38, 39, 40, 15], proof of Memory Safety (MS) [37, 38, 39], or Residual Program State (RS) [37]. Besson et al. present the notion of Information-Flow Leakage (IFL) in compiler optimizations that guarantees that the compiler does not introduce new vulnerabilities [41]. They evaluate their approach on two passes of CompCert, dead-store elimination and register allocation, using a threat model that considers observation points at function boundaries. In contrast, the SecCG backend generates a program secure against ROT and MRE leaks at each execution point. In addition to this, SecCG does not guarantee the preservation of the property but rather the absence of TBLs. If that is not possible, the model is unsatisfiable and SecCG fails to generate a program. The latter outcome has not appeared in our experiments[4] but there is no guarantee that it will not happen. For remedying this problem, one may try to activate a pass in SecCG that introduces additional copies of masked values, deactivate some high-level optimizations, and/or deactivate the ROT or MRE constraints.

A recent approach [14, 15] generates high-quality code that deals with VBLs, Fault Injection (FI), and TSC attacks. To achieve this, Vu et al. [14] introduce the concept of *opaque observations* that disallows the compiler to remove security mitigations or rearrange operands in instructions, such as masking instructions. In their later work [15], they improve the performance of their optimizing compiler by reducing the requirement for serialization. To achieve this, they require source-code annotation that may be challenging for non-trivial programs [15]. Eldib and Wang [36] propose a high-level program synthesis approach to automatically generate masked implementations free from VBLs. Both approaches generate code that mitigates VBLs and thus, do not protect against TBLs.

TABLE VI: TBL-aware approaches

Pub.	Mitigation	Target	Processor
[13]	ROT, MOT, MRE, RNL	AVR	ATMega163
[7]	ROT	*	*
[35]	ROT	ARM	ARM Cortex-M3
[11]	ROT, MOT, MRE, IPI, OT	ARM	ARM Cortex-M0
SecCG	ROT, MRE	*	*

Code Hardening Against Transition-Based Leakages: There is a number of approaches that deal with different types of TBL-related PSCs [13, 7, 35, 11]. Table VI shows the mitigation approaches against TBLs. For each of the related works, Table VI, presents the leakage types each of them mitigates (Mitigation), the target architecture (Target), and the target processor (Processor). In the last two columns * denotes that these approaches may target multiple architectures and processors.

Papagiannopoulos and Veshchikov [13] perform experiments to identify possible sources of leakage in binary AVR code on a ATMega163. They identify sources of leakage including ROT, Memory-Overwrite Transition (MOT), which occurs when overwriting a value in memory, MRE, which occurs when overwriting a value in the memory bus, and Register Neighbor Leakage (RNL), which occurs when the values of neighboring registers interact with each other [13]. Papagiannopoulos and Veshchikov [13] observe that ROT and MRE leakages may be exploited with a small number of runs,

[4] There were unsatisfiable instances due to associativity-related VBLs when using aggressive high-level compiler optimizations (O1, O2, and O3)

500, whereas MOT requites much more (40K). Rosita [11] is a recent approach to mitigate transitional effects that may lead to power side-channel attacks using an emulation-based technique. Rosita performs an iterative process to identify power leakages in software implementations for ARM Cortex M0 and identifies transitional effects due to ROT, MOT, MRE, Instruction-Pair Interaction (IPI), and Other Transitions (OT). IPI occurs when pairs of instructions interact with each other and OT corresponds to interactions between data of different instructions. The mitigation introduces a performance overhead of 21% to 64%. In comparison, SecCG is a generic compiler-based approach that may be applied to multiple hardware architectures and introduces smaller overhead. However, a direct comparison would be unfair because Rosita mitigates more leakage sources.

Wang et al. [7] uses a rule-based system [20, 7] to identify leaks in a masked implementation and perform local register allocation and instruction selection transformations to mitigate these leaks in LLVM. They identify transitional effects due to register reuse, ROT. Their approach is scalable and the mitigation introduces small performance overhead compared to non-optimized code. However, they depend on a non-optimized compilation in order to preserve the security properties of the high-level program, which leads to code generation that is secure against ROT but not optimized. Athanasiou et al. [35] use the same rule-based system to mitigate ROT leakages on binary ARM code targeting the ARM Cortex M3 processor. They are able to reduce the number of potentially vulnerable register pairs given the instruction order. Athanasiou et al. confirm that aggressive compiler optimization passes introduce VBLs. SecCG uses a rule-based system but models a constraint model that is able to generate optimized code that is secure.

Other approaches perform mitigations at whole-system design time [43, 44]. The availability of open hardware architectures and, more specifically, RISC-V, has enabled approaches, such as Coco, which apply software-hardware co-design techniques to mitigate power side-channel attacks [44].

In summary, there are compiler-based and binary rewriting approaches to mitigate TBLs but all these approaches perform local transformations that introduce performance overhead. Instead SecCG trades quality for compilation time and is suitable for performance critical and vulnerable cryptographic functions.

Combinatorial Compiler Approaches: Compiler backend optimizations, like instruction selection, instruction scheduling, and register allocation are known to be hard combinatorial problems. Hence, solving such problems completely does not scale for large program sizes. Therefore, popular compilers, like GCC [45] and LLVM [17], use heuristics that throughout the years have proved to improve program performance. However, these heuristics do not guarantee finding the optimal solution to these backend optimizations.

For critical code and code aimed for compiler-demanding architectures, combinatorial methods may find an optimized version of the code that leads to reduced power consumption and/or high performance benefits. Different works [46, 16, 22, 23] aim to optimize critical code at different levels, like loops [22], locally [23] or at function level [16]. The optimization goals range from execution time, code size, or estimated energy consumption [22, 16, 23]. The main drawback of these approaches is scalability [46]. However, a recent work, Unison [16], allows the optimization of functions of up to almost 1000 instructions.

A different combinatorial approach for generating optimal program code is superoptimization [47]. Given a code sequence, superoptimization approaches attempt to find an equivalent code sequence that reduces the overall execution time and is provably equivalent to the initial code. Souper [48], a state-of-the-art superoptimization approach, performs middle-end optimizations to LLVM IR code. Middle-end optimizations typically do not take decisions on the register allocation or the instruction scheduling. Instead, they enable algorithmic-level code optimizations. Crow [49] is an approach based on Souper that performs software diversification as a security mitigation approach.

To summarize, many combinatorial compiler backend techniques allow low-level code optimization but, to our knowledge, none of them considers the preservation of security properties against TBLs.

VII. Limitations

This paper proposes an architecture-agnostic method to generate high quality code against register-reuse and memory-bus transitional effects. We aim specifically at small-size embedded devices that have a predictable cost model and implement single-issued, non-speculative architectures. Our approach has clear scalability issues, however, we plan to investigate its use in non-linearized functions.

Secondly, our approach is limited to two optimizations, namely register allocation and instruction scheduling. Other backend optimizations, such as instruction selection may be beneficial for removing HD leakages for CISC architectures like x86. Another useful optimization for mitigating optimized implementations may be expression reassociation (`-reassociate` in LLVM).

SecCG generates programs that are MRE- and ROT-leak free. The generated code is straight-line code and thus satisfies the constant-time programming discipline (in the absence of caches). However, analyzing programs that contain operations with operand-dependent latencies (e.g. division) may violate this property. In addition to this, the generated code may contain other types of TBLs, which depends on the actual processor implementation [13].

VIII. Conclusion and Future Work

This paper proposes a constraint-based compiler backend to generate code that is both optimized and secure against power side-channel attacks. We prove that the generated code is secure according to a non-trivial leakage model, and show that our approach achieves high code improvement against non-optimized approaches ranging from 77% to a speedup of 6.6 for two embedded architectures, Mips32 and ARM Cortex

M0. At the same time, our approach introduces a maximum overhead of 13% from the optimal code. This comes at the expense of increased compilation time and reduced scalability.

There are several future directions for our work. We are planning to work on extending the type-inference algorithm to include function calls and loops. Moreover, by improving the accuracy of the hardware model of SecCG to model precisely a specific device, we will be able to improve the leakage model and compare our approach to approaches like Rosita [11]. Finally, we believe that combining our approach with optimizing high-level approaches [14, 15] may further improve the quality of the generated code.

ACKNOWLEDGMENT

We would like to thank Jingbo Wang for providing support for the FSE19 tool and Amir M. Ahmadian for the fruitful discussions and his significant feedback on the paper. Finally, we would like to thank the anonymous reviewers for their constructive and valuable feedback.

REFERENCES

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

[2] P. Kocher, J. Jaffe, and B. Jun, "Differential Power Analysis," in *Annual International Cryptology Conference*. Springer, 1999, pp. 388–397.

[3] M. Joye, P. Paillier, and B. Schoenmakers, "On Second-Order Differential Power Analysis," in *Cryptographic Hardware and Embedded Systems*. Springer, 2005, pp. 293–308.

[4] M. Rivain and E. Prouff, "Provably secure higher-order masking of aes," in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2010, pp. 413–427.

[5] J.-S. Coron, E. Prouff, M. Rivain, and T. Roche, "Higher-Order Side Channel Security and Mask Refreshing," in *Fast Software Encryption*. Springer, 2014, pp. 410–424.

[6] A. G. Bayrak, F. Regazzoni, D. Novo, and P. Ienne, "Sleuth: Automated Verification of Software Power Analysis Countermeasures," in *International Conference on Cryptographic Hardware and Embedded Systems*. Springer, 2013, pp. 293–310.

[7] J. Wang, C. Sung, and C. Wang, "Mitigating power side channels during compilation," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 590–601.

[8] H. Eldib, C. Wang, and P. Schaumont, "Formal Verification of Software Countermeasures against Side-Channel Attacks," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 24, no. 2, pp. 1–24, 2014.

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

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

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

[12] N. Veshchikov and S. Guilley, "Use of Simulators for Side-Channel Analysis," in *2017 IEEE European Symposium on Security and Privacy Workshops (EuroS PW)*, 2017, pp. 104–112.

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

[14] S. T. Vu, K. Heydemann, A. de Grandmaison, and A. Cohen, "Secure delivery of program properties through optimizing compilation," in *Proceedings of the 29th International Conference on Compiler Construction*, 2020, pp. 14–26.

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

[16] R. Castañeda Lozano, M. Carlsson, G. H. Blindell, and C. Schulte, "Combinatorial Register Allocation and Instruction Scheduling," *ACM Trans. Program. Lang. Syst.*, vol. 41, no. 3, pp. 17:1–17:53, 2019.

[17] C. Lattner and V. Adve, "LLVM: a compilation framework for lifelong program analysis amp; transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 2004, pp. 75–86.

[18] T. S. Messerges, E. A. Dabbish, and R. H. Sloan, "Investigations of power analysis attacks on smartcards." *Smartcard*, vol. 99, pp. 151–161, 1999.

[19] E. Brier, C. Clavier, and F. Olivier, "Correlation Power Analysis with a Leakage Model," in *International workshop on cryptographic hardware and embedded systems*. Springer, 2004, pp. 16–29.

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

[21] R. M. Tsoupidi, R. Castañeda Lozano, E. Troubitsyna, and P. Papadimitratos, "Supplemental material: Securing optimized code against power side channels," 2022. [Online]. Available: https://github.com/romits800/seccon_experiments/blob/main/supp_material/main_appendix.pdf

[22] M. Eriksson and C. Kessler, "Integrated Code Generation for Loops," *ACM Transactions on Embedded Computing Systems*, vol. 11S, no. 1, pp. 19:1–19:24, 2012.

[23] C. H. Gebotys, "An efficient model for DSP code generation: Performance, code size, estimated energy," in *Proceedings of the tenth International Symposium on System Synthesis*. IEEE, 1997, pp. 41–47.

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

[25] P. Shaw, "Using Constraint Programming and Local Search Methods to Solve Vehicle Routing Problems," in *International conference on principles and practice of constraint programming*. Springer, 1998, pp. 417–431.

[26] Gecode Team, "Gecode: Generic constraint development environment," 2022. [Online]. Available: https://www.gecode.org

[27] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack, "MiniZinc: Towards a Standard CP Modelling Language," in *International Conference on Principles and Practice of Constraint Programming*. Springer, 2007, pp. 529–543.

[28] G. G. Chu, "Improving combinatorial optimization," Ph.D. dissertation, The University of Melbourne, Australia, 2011.

[29] Google Developers, "Google OR-Tools," 2022. [Online]. Available: https://developers.google.com/optimization/

[30] G. Barthe, S. Belaïd, F. Dupressoir, P.-A. Fouque, B. Grégoire, and P.-Y. Strub, "Verified Proofs of Higher-Order Masking," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2015, pp. 457–485.

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

[32] Y. Oren, M. Renauld, F.-X. Standaert, and A. Wool, "Algebraic Side-Channel Attacks Beyond the Hamming Weight Leakage Model," in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2012, pp. 140–154.

[33] X. Leroy, "A Formally Verified Compiler Back-end," *Journal of Automated Reasoning*, vol. 43, no. 4, p. 363, 2009.

[34] S. Gocht, C. McCreesh, and J. Nordström, "An auditable constraint programming solver," in *International Conference on Principles and Practice of Constraint Programming*, 2022.

[35] K. Athanasiou, T. Wahl, A. A. Ding, and Y. Fei, "Automatic detection and repair of transition-based leakage in software binaries," in *Software Verification*. Springer, 2020, pp. 50–67.

[36] H. Eldib and C. Wang, "Synthesis of Masking Countermeasures against Side Channel Attacks," in *International Conference on Computer Aided Verification*. Springer, 2014, pp. 114–130.

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

[38] J. B. Almeida, M. Barbosa, G. Barthe, A. Blot, B. Grégoire, V. Laporte, T. Oliveira, H. Pacheco, B. Schmidt, and P.-Y. Strub, "Jasmin: High-Assurance and High-Speed Cryptography," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1807–1823.

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

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

[41] F. Besson, A. Dang, and T. Jensen, "Information-Flow Preservation in Compiler Optimisations," in *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*, 2019, pp. 230–23 012, iSSN: 2374-8303.

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

[43] D. Šijačić, J. Balasch, B. Yang, S. Ghosh, and I. Verbauwhede, "Towards Efficient and Automated Side Channel Evaluations at Design Time," *Journal of Cryptographic Engineering*, vol. 10, no. 4, pp. 305–319, 2020.

[44] B. Gigerl, V. Hadzic, R. Primas, S. Mangard, and R. Bloem, "Coco:{Co-Design} and {Co-Verification} of masked software implementations on {CPUs}," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 1469–1468.

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

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

[47] C. W. Fraser, "A compact, machine-independent peephole optimizer," in *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM, 1979, pp. 1–6.

[48] R. Sasnauskas, Y. Chen, P. Collingbourne, J. Ketema, G. Lup, J. Taneja, and J. Regehr, "Souper: A synthesizing superoptimizer," *arXiv preprint arXiv:1711.04422*, 2017.

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