Optimized Code Generation against Power Side Channels

Rodothea Myrsini Tsoupidi
Royal Institute of Technology KTH, Sweden

Roberto Castañeda Lozano
Independent Researcher

Elena Troubitsyna
Royal Institute of Technology KTH, Sweden

Panagiotis Papadimitratos
Royal Institute of Technology KTH, Sweden

Abstract

Software masking, a software mitigation against power-side channel attacks, aims at removing the secret dependencies from the power traces that may leak cryptographic keys. However, high-level software mitigations often depend on general purpose compilers, which do not preserve non-functional properties. What is more, microarchitectural features, such as the memory bus and register reuse, may also reveal 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 turn off compiler optimization and/or perform local, post-compilation transformations. However, theses solution lead to inefficient code.

To deal with this issues, we propose Secure by Construction Code Generation (SecConCG), a secure constraint-based compiler backend to generate code that is secure. SecConCG can control 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 ten masked implementations on MIPS and ARM Cortex M0, SecConCG improves the generated code from 10% to 10x compared to non-optimized secure code at a small overhead of up to 7% compared to non-secure optimized code.

2012 ACM Subject Classification Security and privacy → Formal security models

Keywords and phrases constraint programming, code optimization, software masking, side channels

Digital Object Identifier 10.4230/LIPIcs.CP.2022.22

Acknowledgements We would like to thank Amir Mahmood Ahmadian for the discussions and advice on the type-inference algorithm and for reviewing the paper and Jingbo Wang for providing the FSE19 tool and support for using it.

1 Introduction

Cryptographic algorithms such as AES and RSA use mathematical properties to hide secret information such as encryption or decryption keys, which an unauthorized user should not obtain. However, the software implementation of these high-level algorithms may reveal information about their secret keys [7]. In particular, the attacker may observe side-channel information, such as the power consumption [7], during the execution of the algorithm to extract the secret keys. Many of these attacks do not require expensive equipment and provide an attractive way to attack exposed devices.

Software mitigations, such as constant-time programming and software masking, aim at protecting against these side-channel attacks. In particular, software masking hides secret information randomizing the secret values. While software masking can be an effective mitigation, it can be invalidated by compiler code generation [1, 12]. For example, consider Figure 1, a first-order masked C implementation of exclusive OR, where `key` is a secret value,

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

Figure 1 Masked exclusive OR implementation in C

`mask` is a uniformly random variable, and `pub` is a secret-independent value. At line 2, the algorithm creates the second share, `mk`, and at line 3, it performs the exclusive OR operation with the public value, `pub`. At a high-level, the code of Figure 1 is secure, but a binary implementation generated by a standard, security-unaware compiler may leak information about the secret `key` at the hardware level. For example, the random share `mk` may reuse the register of the `mask` variable, which results in a value transition of the reused register from `mask` to `mk`. This transition may be observable in the power trace of a device, leaking information about the secret `key`. What is more, memory operations that use the same bus may also reveal secret information [10].

To mitigate these compiler-introduced side-channel leaks at the binary level there are techniques based on compilation [12], and binary rewriting with hardware emulation [10]. The compiler-based methods need to be embedded in the compiler infrastructure [12]. Methods that depend on hardware emulation are typically accurate but have high overhead [10]. All these approaches generate programs with increased performance overhead [12, 10]. In particular, Rosita [10], a emulation-based approach, introduces an overhead of 24% to 64% on their mitigation. Wang et. al [12] perform their mitigation with no compiler optimizations (-O0) in LLVM to preserve the high-level properties of the mitigation. However, unoptimized code is highly inefficient. Moreover, unoptimized code may introduce additional leaks due to the heavy use of the program stack.

To summarize, current approaches to secure compilation against power side-channel attacks generate code that is either secure (does not leak secrets due to transitional effects) or efficient, but rarely both. To fill this gap, we propose SecConCG, a compiler approach that generates optimized code that preserves security properties. SecConCG can control the quality of the mitigated program by efficiently searching the best possible low-level implementation according to a processor cost model [3]. The mitigations are based on instruction scheduling and register allocation transformations. SecConCG is hardware agnostic but can be extended with additional architecture-specific constraints. In our experiments with ten masked implementations on MIPS and ARM Cortex M0, SecConCG improves the generated code from 10% to 10x compared to non-optimized secure code at a small overhead of up to 7% compared to non-secure optimized code. To conclude, this paper contributes a compiler method to generate leak-free, low-overhead assembly code for high-level software-masked programs.

2 Background

This section describes the background for our paper, including the Hamming Distance (HD) model (Section 2.1) and a constraint-based compiler backend model (Section 2.2).

2.1 Hamming-Distance Model

The Hamming Weight (HW) model [7] 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 0, otherwise. The HW of this data is the number of the bits that are set, i.e. $HW(d) = \sum_{i=0}^{N-1} d_i$. Without the need of detailed knowledge about the actual microelectronic design of a device, 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 equal to $HW(d_1 \oplus d_2)$, 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 $HW(d_1)$ [2]. This means that by masking (exclusive bitwise OR) a secret value k with a uniform random variable m, the hamming distance of the new variable, will have the same mean and variance as m. In this way masking hides the information of k from the power consumption traces.

2.2 Constraint-based Compiler Backend

A compiler backend implements low-level optimizations, namely instruction selection, instruction scheduling, and register allocation to optimize low-level code. A combinatorial compiler backend [3, 6] uses a combinatorial solving technique to optimize software using the aforementioned backend optimizations.

A constraint-based compiler backend generates a constraint model that corresponds to the program semantics, the low-level compiler transformations, and the hardware architecture. In particular, the compiler backend can be modeled as a Constraint Optimization Problem (COP), $P = \langle V, U, C, O \rangle$, where V is the set of the decision variables of the problem, U is the domain of each of these variables, C the set of constraints among the decision variables, and O is the optimization function. A combinatorial compiler backend aims at generating code that optimizes function O, which typically models the execution time or the code size of the analyzed code.

The constraint model represents the code semantics, the hardware constraints, and a number of standard program transformations. In the following, we focus on two low-level transformations, register allocation and instruction scheduling that are crucial for our mitigation. Each program consists of a set of code blocks B, each of which contains a number of optional operations, $o \in Operations$, that may be *active* or not. A code block $b \in B$ correspond often to a basic block, a piece of code with no branches apart from the entry and the exit of the block. However, a preprocessing step may split straight-line code into segments to improve the scalability by decomposing the problem. The machine-instruction set $Ins_o, o \in Operations$, is the set of hardware instructions that implement operation o. Each operation may consist of a number of operands $p \in Operands$, each of which may be implemented by different, equal-valued temporaries, $t \in Temps$.

The decision variables of the constraint problem are the following: $r(t) \in Regs_t,\ t \in Temps$ is the hardware register assigned to temporary t; $a(o) \in [0, 1],\ o \in Operations$ denotes whether operation o is active or not; $i(o) \in Ins_o,\ o \in Operations$ is the instruction that implements operation o; $c(o) \in [0, maxc],\ o \in Operations$ is the cycle at which an operation o is scheduled, bounded by $maxc$; $y(p) \in Temps_p,\ p \in Operands$ is the selected temporary among all possible temporaries for operand p. In addition to these, $l(t) \in [0, 1],\ t \in Temps$ 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, with $ls(t) < le(t),\ t \in Temps$. An important property of register allocation

is that the register live ranges of a specific hardware register r_i do not overlap. This means that $\forall t_1, t_2 \in Temps\ .\ r(t_1) = r(t_2) = r_i$, we have $ls(t_1) \geq le(t_2)$ or $ls(t_2) \geq le(t_1)$.

A typical objective function of a combinatorial compiler backend aims at optimizing different metrics, including minimizing the *code size* and minimizing the *execution time*. The following equation is a generic objective function that sums up the weighted cost of each code block $\sum_{b \in B} weight(b) \cdot \mathbf{cost}(b)$. This weighted cost for each code block consists of the cost of the specific implementation and is a variable, whereas, *weight* is a static value that represents the contribution of the specific code block to the total cost.

3 SecConCG

Compiler code generation may generate code that is vulnerable to power side-channel attacks. To remedy this problem, we propose SecConCG, an approach to optimize code that does not expose secret information. SecConCG is a constraint-based secure optimizing compiler, i.e. it extends a constraint-based compiler backend with security constraints. It takes as inputs 1) a program in a low-level representation and 2) the security policy that describes which variables are `Secret`, `Random`, and `Public`.

3.1 Security Analysis

SecConCG performs a security analysis to extract the security types of each program variable and subsequently, generates constraints that lead the code generation to secure solutions. The security analysis identifies the security type, `Random`, `Public`, or `Secret` of each intermediate variables. In the compiler constraint model, the program variables are the operands and the result of each operation, i.e. the temporary variables $t \in Temps$ (see Section 2.2).

The security analysis uses a type-inference algorithm based on Wang et. al [12]. This algorithm assigns types to each temporary variables in a conservative way. Function $type(t) : Temps \rightarrow \{R, S, P\}$ returns the type that the type-inference algorithm assigns to temporary variable t. Here, and in the following of this section, we write the types as follows: type R corresponds to `Random`, S corresponds to `Secret`, and P corresponds to `Public`.

Given the type inference, the security analysis generates information that the solver uses to generate secure programs. According to the HD model, when the value of a hardware register changes from one value to another, the exclusive OR of the two values is exposed. $Rpairs$ is the set of pairs of temporary variables that when xor:ed together reveal secret information. This means that:

$$Rpairs = [(t_1, t_2) \mid t_1 \in Temps\ \wedge\ t_2 \in Temps\ \wedge\ (type(t_1) \in \{R, P\}) \wedge (type(t_2) \in \{R, P\})\ \wedge\ (type(t_1 \oplus t_2) = S)]. \quad (1)$$

If the type of a temporary variable, t, is `Secret`, then another random variable should precede the definition of the secret variable to hide the secret information. $Spairs$ is a set of pairs, each of which consists of a secret temporary variable and a set of random temporary variables that are able to hide the secret information, i.e. $type(t' \oplus t) = R$:

$$Spairs = [(t, ts) \mid t \in Temps\ \wedge\ type(t) = S\ \wedge\ ts = [t' \mid t' \in Temps\ \wedge\ type(t') = R \wedge type(t' \oplus t) = R]]. \quad (2)$$

Finally, memory operations may also reveal secret information. We assume the same 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. $Mmpairs$

are the memory to memory leakages, i.e. if the sequence of two instructions in memory leads to a secret leakage.

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

Here, $tm[o] \in Temps$ is the temporary that corresponds to the memory data of the operation.

The same leakage as we had in the case of a secret register write applies here. This means that 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 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. $type(tm[o'] \oplus tm[o]) = R$. The definition of $Mspairs$ is as follows:

$$Mspairs = [(o, os) \mid o \in MemOperations \ \wedge \ type(tm[o]) = S \ \wedge \\ os = [o' \mid o' \in MemOperations \ \wedge \ type(tm[o']) = R \ \wedge \\ type(tm[o'] \oplus tm[o]) = R]].$$

The security analysis provides $Rpairs$, $Spairs$, $Mmpairs$, and $Mspairs$ to the constraint model, which enables the implementation of constraints that constrain the code generation to generate secure programs.

3.2 Constraint Model

The constraint model, consisting of the combinatorial compiler, takes as input the four sets, $Rpairs$, $Spairs$, $Mmpairs$, and $Mspairs$ and uses them as data to generate appropriate constraints, which we will define in the following.

Predicate `samereg` returns `true` if the two input temporaries are active (`l(t)` = 1) and they are assigned to the same register. Otherwise `samereg` returns `false`.

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

The following constraint ensures that a pair of random (or public) temporaries in $Rpairs$, are either not assigned to the same register or they are not subsequent (`subseq` constraint). We will define the `subseq` constraint in the following.

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

The following constraint ensures that for each pair $(\mathtt{t}_s, \mathtt{t}_{rs}) \in Spairs$, one of the random temporaries, $\mathtt{t}_r \in \mathtt{t}_{rs}$, precedes the secret temporary, $\mathtt{t}_s$, and another random temporary succeeds the secret temporary, if the $\mathtt{t}_s$ is live.

```
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))
```

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` is similar to `subseq` but considers two consecutive memory operations instead of temporaries. We will define the `msubseq` constraint in the following.

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

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 register equivalent, where instead of register operations, we have memory operations.

To define the `subseq` constraint, we first define a set of help problem variables, `lk`, and the help predicate `is_before`. First, predicate `is_before(t1, t2)` is `true` if t_1 is assigned to the same register as t_2 and t_1's life range ends, `le(t1)`, before the beginning of the life range of t_2, `ls(t2)`.

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

The following code snippet defines a new set of variables `lk`. Variable `lk(t)` contains the end live cycle of the temporary that occupied the same register at `t`, `r(t)`, right before `t` was assigned. Namely, 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))
```

Similar to the `subseq` constraint, `msubseq` requires the definition of assisting problem variables, `ok`, and predicate `is_before_mem`. Predicate `is_before_mem(o1, o2)` is `true`, when o_1 is scheduled before o_2.

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

Variable `ok(o)` is 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`, predicate `msubseq` is as follows:

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

4 Evaluation

This section describes the evaluation of SecConCG. For the evaluation of SecConCG, we pose the following research questions: RQ1: What is the overhead in execution time for

the generated code using SecConCG? RQ2: What is the speedup in execution time of the generated code compared with other techniques that use non-optimized code? RQ3: What is the solver overhead in solving time using SecConCG compared to the original constraint model?

Experimental Setup: This section describes the implementation details of SecConCG and the evaluation setup. The implementation of SecConCG is an extension of Unison [3], a combinatorial compiler backend that uses Constraint Programming (CP) to optimize software functions with regards to code size and execution time. The type-inference implementation is written in Haskell and is merely based on Wang et. al [12]. 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 frontend compiler for these experiments. We evaluate our method on two architectures, 1) ARM Cortex M0, a highly predictable architecture targeting small embedded devices, and 2) Mips, a widely used embedded architecture. We implemented the constraint model both as part of the specialized Gecode[1] constraint model and the Minizinc [8] model that Unison provides. The latter allows for solving the problem using multiple solvers. In total, we tried four solvers, Chuffed v0.10.3 [4], OR-Tools[2], Elsie Geas[3], and the specialized model written in Gecode v6.2. We ran the former three solvers activating the *free-search* option. Among all these solvers, Gecode and Chuffed were performing best. None of the two solvers was able to solve all the problems, but together, they could solve all the problems. To evaluate our approach, we use a set of small benchmark programs, up to 100 lines of C code, which were made available by Wang et. al [12]. Table 1 provides a description of these benchmarks, including the number of lines of code (LoC) and the results of the evaluation. These benchmark programs constitute masked implementations from previous work and are linearized.

Progr.	Description	LoC	ARM Cortex M0			Mips32		
			PO	SU	SO	PO	SU	SO
P0	Listing 1	5	0%	1.69	0.67	0%	7.33	1.38
P1	AES Shift Rows [1]	11	0%	1.83	0.87	0%	10.00	1.93
P2	Messerges Boolean [1]	12	0%	1.70	2.42	0%	6.86	3.65
P3	Goubin Boolean [1]	12	0%	1.66	2.54	0%	6.11	4.41
P4	SecMultOpt_wires_1 [9]	25	0%	1.59	0.42	0%	1.83	7.34
P5	SecMult_wires_1 [9]	25	0%	1.57	0.14	0%	1.75	7.53
P6	SecMultLinear_wires_1 [9]	32	0%	1.65	2.58	0%	2.55	8.59
P7	CPRR13-lut_wires_1 [5]	81	7%	1.10	0.03	0%	2.44	16.23
P8	CPRR13-OptLUT_wires_1 [5]	84	5%	1.08	0.03	0%	2.49	28.06
P9	CPRR13-1_wires_1 [5]	104	6%	1.11	0.08	≈0%	2.10	25.35

Table 1 Optimality overhead (PO) over the optimal solution (by Unison), the speedup (SU) compared to a secure non-optimized solution (by FSE19), and the solver overhead (SO) compared to the Unison model

RQ1: Optimality Overhead: SecConCG builds on a combinatorial compiler backend to generate a program that satisfies security constraints for software masking. This means that our approach may compromise some of the code optimality of the non-mitigated code to mitigate the software masking leaks. To evaluate 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 [3] generates compared with SecConCG's optimized

[1] Gecode: https://www.gecode.org
[2] OR-Tools: https://developers.google.com/optimization/
[3] Elsie Geas: `https://bitbucket.org/gkgange/geas/src/master/`

and secure code. We calculate the overhead as follows: $PO = 100 \cdot (Cycles_{SecConCG} - Cycles_{Unison})/Cycles_{Unison}$. Table 1 shows the results of our evaluation. For each of the benchmark programs, the table shows the overhead for ARM Cortex M0 and Mips. The results show zero overhead for Mips, and a maximum 7% overhead in ARM Cortex M0. More specifically, ARM Cortex M0 has an non-zero overhead for programs P7-P9. The observed overhead in ARM Cortex M0 depends partially on the mitigation itself that may require redundant or non-optimal operations. However, the main reason for the overhead that appears in larger benchmarks is that the constraint model becomes larger compared to the original model due to the introduction of additional constraints and problem variables. This leads to an increase in the search space and hinders the constraint model to locate feasible solutions.

RQ2: Execution-time Improvement: To evaluate the execution-time improvement of our approach against other approaches, we compare SecConCG with the work by Wang et. al[12]. Wang et. al perform their vulnerability identification approach on the non-optimized code from LLVM 3.6. This is a common approach by different security mitigations, because compilation passes might violate the security properties of a program. However, non-optimized code leads to high performance overhead. This evaluation, compares the execution time in number of cycles (based on the cost model that we use) of the mitigated code by Wang et. al [12] and SecConCG. Table 1 shows the results of the evaluation for ARM Cortex M0 and Mips. In Table 1 the calculation of the improvement is as follows: $SU = Cycles_{SecConCG}/Cycles_{Baseline}$ For ARM Cortex M0, the improvement ranges from 8% for P8 to 83% for P1. We notice that for the smaller benchmarks, SecConCG achieves increased improvement over the baseline, whereas, for the largest benchmarks P7-P9, the improvement is smaller. The main reason for this, is the increased size of the program under analysis that consists of one large function that is difficult to decompose. As a future work, we plan to investigate opportunities for improved block decomposition. For Mips, the improvement ranges from 75% to 10x. 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 SecConCG does not require memory spilling. This experiment shows a clear difference between the two architectures with regards to the improvement over the baseline, with SecConCG achieving a larger improvement for Mips than for ARM Cortex M0. The main reason for this difference is the characteristics of each architecture. ARM Cortex M0 has twelve general purpose registers, whereas Mips has 32 general purpose registers.. This allows the constraint model to select among multiple registers to assign to a temporary, avoiding non-secure register reuse. In addition to that, ARM Cortex M0 implements the Thumb instruction set, which contains many instructions, where one of the operands shares the same register as the result. This increases the complexity of the constraint model further.

RQ3: Solver Overhead: SecConCG builds on Unison to generate a program that satisfies security constraints for software masking. The introduction of the new constraints and variables to the model, may introduce a solving overhead. To evaluate the overhead of SecConCG, we compare the solving time of SecConCG with the solving time of Unison. We calculate the overhead as follows: $SO = (st_{SecConCG} - st_{Unison})/st_{Unison}$, where st is the solving time. Table 1 shows the results of our evaluation. For each of the benchmark programs, the table shows the overhead for ARM Cortex M0 and Mips. The results show an increase in the overhead for Mips, when the size of the program increases. The overhead ranges from 2.3x to 26x slowdown in the solver. This is due to the introduction of new variables ($t^2, t \in Temps$) and logical constraints that do not propagate well, leading to a

solving overhead. On the other hand, the overhead in ARM Cortex M0 is fluctuating, ranging from 3% to 3.58x. The reason for this, is that ARM Cortex M0 is a constrained architecture, with few registers, which makes the optimization problem difficult to solve for Unison, leading to reaching internal time limits in the solver. Therefore, introducing additional constraints reduces the search space. In summary, in this experiment, we observe that the solving-time overhead depends on the architecture and lead to a 26x slowdown to 30% overhead.

5 Related Work

This section presents the related work with regards to binary-code hardening against side-channel attacks, combinatorial compiler backends, and optimized security compilation approaches.

Code Hardening Against Side-Channel Attacks: Software masking is a software approach to mitigate power side channels. However, a compiler that translates a program to machine code may introduce power leaks [12, 10]. Wang et. al [12] identify leaks in masked implementation using a type system and perform local register allocation and instruction selection transformations to mitigate these leaks in LLVM. They identify transitional effects due to register reuse. Their approach is efficient allowing the analysis of large linearized functions and the mitigations introduce small overhead. However, they depend on a non-optimized compilation in order to preserve the security properties of the high-level program leading in the code generation of non-optimized but secure code.

Combinatorial Compiler Backend: Compiler backend optimizations, like instruction selection, instruction scheduling, and register allocation are known to be combinatorial problems. However, solving combinatorial problems is difficult and results in low scalability and may result in high solving time. Therefore, general-purpose compilers use heuristics that throughout the years have proved to improve the performance of programs in modern compilers. However, these heuristics do not provide any optimality guarantees. For critical code and code aimed for compiler-demanding architectures, combinatorial methods may be able to find an optimized version of the code that may lead to reduced power consumption and/or high performance benefits. Different works [3, 6] aim to optimize critical code at different levels, like locally [6] or at function level [3]. The optimization goals range from execution time, code size, or estimated energy consumption [3, 6]. The main drawback of these approaches is scalability. However, a recent work, Unison [3], allows the optimization of functions with almost 1000 instructions.

Optimized Secure Compilation: A recent work [11] deals with the problem that modern compilers do not guarantee preservation of security properties and optimization passes may invalidate security mitigations at the software level. Vu et. al [11] are able to perform high-level optimizations, which is not in the scope of our approach. The work by Vu et. al deals with the same problem as our approach but in a different level of abstraction considering a weaker leakage model. We believe that the combination of our approach with their approach could lead into more efficient secure code.

6 Conclusion

This paper proposes a constraint model to be embedded in a combinatorial compiler backend that allows the automatic generation of optimized code that is secure against power side-channel attacks. We show that our approach achieves high code improvement against non-optimized approaches ranging from 8% to 10x for two embedded architectures, MIPS

and ARM Cortex M0. At the same time, our approach introduces a maximum overhead of 7% from the optimal solution.

References

1. Ali Galip Bayrak, Francesco Regazzoni, David Novo, and Paolo Ienne. Sleuth: Automated Verification of Software Power Analysis Countermeasures. In Guido Bertoni and Jean-Sébastien Coron, editors, *Cryptographic Hardware and Embedded Systems - CHES 2013*, Lecture Notes in Computer Science, pages 293–310, Berlin, Heidelberg, 2013. Springer. doi:10.1007/978-3-642-40349-1_17.
2. Eric Brier, Christophe Clavier, and Francis Olivier. Correlation Power Analysis with a Leakage Model. In Marc Joye and Jean-Jacques Quisquater, editors, *Cryptographic Hardware and Embedded Systems - CHES 2004*, Lecture Notes in Computer Science, pages 16–29, Berlin, Heidelberg, 2004. Springer. doi:10.1007/978-3-540-28632-5_2.
3. Roberto Castañeda Lozano, Mats Carlsson, Gabriel Hjort Blindell, and Christian Schulte. Combinatorial Register Allocation and Instruction Scheduling. *ACM Trans. Program. Lang. Syst.*, 41(3):17:1–17:53, July 2019. URL: http://doi.acm.org/10.1145/3332373, doi:10.1145/3332373.
4. Geoffrey G. Chu. *Improving combinatorial optimization*. PhD thesis, The University of Melbourne, Australia, 2011.
5. Jean-Sébastien Coron, Emmanuel Prouff, Matthieu Rivain, and Thomas Roche. Higher-Order Side Channel Security and Mask Refreshing. In Shiho Moriai, editor, *Fast Software Encryption*, Lecture Notes in Computer Science, pages 410–424, Berlin, Heidelberg, 2014. Springer. doi:10.1007/978-3-662-43933-3_21.
6. C.H. Gebotys. An efficient model for DSP code generation: performance, code size, estimated energy. In *Proceedings. Tenth International Symposium on System Synthesis (Cat. No.97TB100114)*, pages 41–47, September 1997. ISSN: 1080-1820. doi:10.1109/ISSS.1997.621674.
7. Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. In Michael Wiener, editor, *Advances in Cryptology — CRYPTO' 99*, Lecture Notes in Computer Science, pages 388–397, Berlin, Heidelberg, 1999. Springer. doi:10.1007/3-540-48405-1_25.
8. Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. MiniZinc: Towards a Standard CP Modelling Language. In Christian Bessière, editor, *Principles and Practice of Constraint Programming – CP 2007*, Lecture Notes in Computer Science, pages 529–543, Berlin, Heidelberg, 2007. Springer. doi:10.1007/978-3-540-74970-7_38.
9. Matthieu Rivain and Emmanuel Prouff. Provably Secure Higher-Order Masking of AES. In Stefan Mangard and François-Xavier Standaert, editors, *Cryptographic Hardware and Embedded Systems, CHES 2010*, Lecture Notes in Computer Science, pages 413–427, Berlin, Heidelberg, 2010. Springer. doi:10.1007/978-3-642-15031-9_28.
10. Madura A. Shelton, Niels Samwel, Lejla Batina, Francesco Regazzoni, Markus Wagner, and Yuval Yarom. Rosita: Towards Automatic Elimination of Power-Analysis Leakage in Ciphers. *Proceedings 2021 Network and Distributed System Security Symposium*, 2021. appears in NDSS 2022. URL: http://arxiv.org/abs/1912.05183, doi:10.14722/ndss.2021.23137.
11. Son Tuan Vu, Karine Heydemann, Arnaud de Grandmaison, and Albert Cohen. Secure delivery of program properties through optimizing compilation. In *Proceedings of the 29th International Conference on Compiler Construction*, CC 2020, pages 14–26, New York, NY, USA, February 2020. Association for Computing Machinery. URL: http://doi.org/10.1145/3377555.3377897, doi:10.1145/3377555.3377897.
12. Jingbo Wang, Chungha Sung, and Chao 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*, ESEC/FSE 2019,

410 pages 590–601, New York, NY, USA, August 2019. Association for Computing Machinery.
411 doi:10.1145/3338906.3338913.