Mitigating Power Side Channels during Compilation

Jingbo Wang
University of Southern California
Los Angeles, CA, USA

Chungha Sung
University of Southern California
Los Angeles, CA, USA

Chao Wang
University of Southern California
Los Angeles, CA, USA

ABSTRACT

The code generation modules inside modern compilers, which use a limited number of CPU registers to store a large number of program variables, may introduce side-channel leaks even in software equipped with state-of-the-art countermeasures. We propose a program analysis and transformation based method to eliminate such leaks. Our method has a type-based technique for detecting leaks, which leverages Datalog-based declarative analysis and domain-specific optimizations to achieve high efficiency and accuracy. It also has a mitigation technique for the compiler's backend, more specifically the register allocation modules, to ensure that leaky intermediate computation results are stored in different CPU registers or memory locations. We have implemented and evaluated our method in LLVM for the x86 instruction set architecture. Our experiments on cryptographic software show that the method is effective in removing the side channel while being efficient, i.e., our mitigated code is more compact and runs faster than code mitigated using state-of-the-art techniques.

CCS CONCEPTS

• **Security and privacy → Cryptanalysis and other attacks**; • **Software and its engineering → Compilers**; **Formal software verification**.

KEYWORDS

Side channel, information leak, countermeasure, power, register allocation, type inference, verification, code generation

ACM Reference Format:
Jingbo Wang, Chungha Sung, and Chao Wang. 2019. Mitigating Power Side Channels during Compilation. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26–30, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3338906.3338913

1 INTRODUCTION

Cryptography is an integral part of many security protocols, which in turn are used by numerous applications. However, despite the strong theoretical guarantee, cryptosystems in practice are vulnerable to side-channel attacks when non-functional properties such as timing, power and electromagnetic radiation are exploited to gain information about sensitive data [22, 25, 27, 44, 47, 56, 57, 67, 73, 85]. For example, if the power consumption of an encryption device depends on the secret key, techniques such as differential power analysis (DPA) may be used to perform attacks reliably [22, 27, 46, 55, 58]. Although there are methods for mitigating power side channels [2, 3, 14, 15, 34, 35, 83], they focus exclusively on the *Boolean level*, e.g., by targeting circuits or software code converted to a bit-level representation. This limits their usage; as a result, none of them was able to fit into modern compilers such as GCC and LLVM to directly handle the *word-level* intermediate representation (IR). In addition, code transformations in compilers may add new side channels, even if the input program is equipped with state-of-the-art countermeasures.

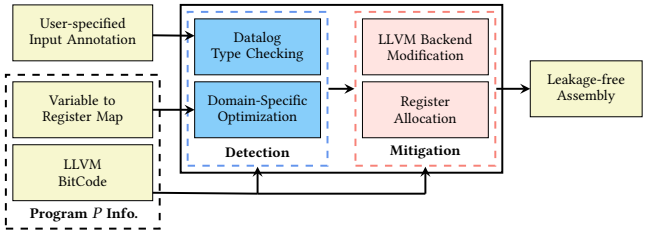


Figure 1: Overview of our secure compilation method

Specifically, compilers use a limited number of the CPU's registers to store a potentially-large number of intermediate computation results of a program. When two masked and hence desensitized values are put into the same register, the *masking* countermeasure may be removed accidentally. We will show, as part of this work, that even provably-secure techniques such as high-order masking [6, 8, 9] are vulnerable to such leaks. Indeed, we have found leaks in the compiled code produced by LLVM for both x86 and MIPS/ARM platforms, regardless of whether the input program is equipped with high-order masking.

To solve the problem, we propose a secure compilation method with two main contributions. First, we introduce a type-inference system to soundly and quickly detect power side-channel leaks. Second, we propose a mitigation technique for the compiler's backend to ensure that, for each pair of intermediate variables that may cause side-channel leaks, they are always stored in different registers or memory locations.

Figure 1 illustrates our method, which takes a program P as input and returns the mitigated code as output. It has two steps. First, type inference is used to detect leaks by assigning each variable a *distribution* type. Based on the inferred types, we check each pair (v_1, v_2) of variables to see if they may cause leaks when stored in the same register. If the answer is yes, we constrain the compiler's register allocation modules to ensure that v_1 and v_2 are assigned to different registers or memory locations.

Our method differs from existing approaches in several aspects. First, it specifically targets power side-channel leaks caused by reuse of CPU registers in compilers, which have been largely overlooked by prior work. Second, it leverages Datalog, together with a number of domain-specific optimizations, to achieve high efficiency and

ESEC/FSE '19, August 26–30, 2019, Tallinn, Estonia

ACM ISBN 978-1-4503-5572-8/19/08…$15.00
https://doi.org/10.1145/3338906.3338913

accuracy during leak detection. Third, mitigation leverages the existing production-quality modules in LLVM to ensure that the compiled code is secure by construction.

Unlike existing techniques that translate the input program to a Boolean representation, our method works directly on the word-level IR and thus fits naturally into modern compilers. For each program variable, the leak is quantified using the well-known Hamming Weight (HW) and Hamming Distance (HD) leakage models [52, 53]. Correlation between these models and leaks on real devices has been confirmed in prior work (see Section 2). We also show, via experiments, that leaks targeted by our method exist even in program equipped with high-order masking [6, 8, 9].

To detect leaks quickly, we rely on type inference, which models the input program using a set of Datalog facts and codifies the type inference algorithm in a set of Datalog rules. Then, an off-the-shelf Datalog solver is used to deduce new facts. Here, a domain-specific optimization, for example, is to leverage the compiler's backend modules to extract a map from variables to registers and utilize the map to reduce the computational overhead, e.g., by checking pairs of some (instead of all) variables for leaks.

Our mitigation in the compiler's backend is systematic: it ensures that all leaks detected by type inference are eliminated. This is accomplished by constraining register allocation modules and then propagating the effect to subsequent modules, without having to implement any new backend module from scratch. Our mitigation is also efficient in that we add a number of optimizations to ensure that the mitigated code is compact and has low runtime overhead. While our implementation focuses on x86, the technique itself is general enough that it may be applied to other instruction set architectures (ISAs) such as ARM and MIPS as well.

We have evaluated our method on a number of cryptographic programs [8, 14], including well-known ciphers such as AES and MAC-Keccak. These programs are protected by masking countermeasures but, still, we have detected leaks in the LLVM compiled code. In contrast, the compiled code produced by our mitigation technique, also based on LLVM, is always leak free. In terms of runtime overhead, our method outperformed existing approaches such as high-order masking: our mitigated code not only is more secure and compact but also runs faster than code mitigated by high-order masking techniques [8, 9].

To summarize, this paper makes the following contributions:

- We show that register reuse implemented in modern compilers introduces new side-channel leaks even in software code already protected by masking.
- We propose a Datalog based type inference system to soundly and quickly detect these side-channel leaks.
- We propose a mitigation technique for the compiler's backend modules to systematically remove the leaks.
- We implement the method in LLVM and show its effectiveness on a set of cryptographic software programs.

The remainder of this paper is organized as follows. First, we illustrate the problem and the technical challenges for solving it in Section 2. Then, we review the background including the threat model and leakage model in Section 3. Next, we present our method for leak detection in Section 4 and leak mitigation in Section 5, followed by domain-specific optimizations in Section 6. We present our experimental results in Section 7, review the related work in Section 8, and give our conclusions in Section 9.

```
//'txt': PUBLIC, 'key': SECRET and 't' is HW-sensitive
uint32 Xor(uint32 txt, uint32 key) {uint32 t = txt ^ key; return t;}
//random variable 'mask1' splits 'key' to secure shares {mask1,mk}
uint64 SecXor(uint32 txt, uint32 key, uint32 mask1) {
   uint32 mk = mask1 ^ key; // mask1^key
   uint32 t = txt ^ mk; // txt^(mask1^key)
   return (mask1,t);
}
//'mask1' splits 'key' to shares {mask1,mk} a priori
//'mask2' splits the result to shares {mask2,t3} before return
uint64 SecXor2(uint32 txt, uint32 mk, unit32 mask1, unit32 mask2) {
   uint32 t1 = txt ^ mk; // txt^(mask1^key)
   uint32 t2 = t1 ^ mask2; // (txt^mask1^key)^mask2
   unit32 t3 = t2 ^ mask1; // (txt^mask1^key^mask2)^mask1
   return {mask2,t3};
}
```

Name	Approach	HW-Sensitive	HD-Sensitive
Xor	No Masking	✓	✓
SecXor	First Order Masking	✗	✓
SecXor2	Specialized Hardware & Masking	✗	✓

Figure 2: Implementations of an XOR computation in the presence of HW and HD power side-channel leaks.

2 MOTIVATION

We use examples to illustrate why *register reuse* may lead to side-channel leaks and the challenges for removing them.

2.1 The HW and HD Leaks

Consider the program *Xor()* in Figure 2, which takes the public *txt* and the secret *key* as input and returns the Exclusive-OR of them as output. Since logical 1 and 0 bits in a CMOS circuit correspond to different leakage currents, they affect the power consumption of the device [52]; such leaks were confirmed by prior works [22, 58] and summarized in the Hamming Weight (HW) model. In program *Xor()*, variable *t* has a power side-channel leak because its register value depends on the secret *key*.

The leak may be mitigated by *masking* [2, 39] as shown in program *SecXor()*. The idea is to split a secret to *n* randomized shares before using them; unless the attacker has all *n* shares, it is theoretically impossible to deduce the secret. In *first-order* masking, the secret *key* may be split to *{mask1,mk}* where *mask1* is a random variable, *mk=mask1⊕key* is the bit-wise Exclusive-OR of *mask1* and *key*, and thus *mask1⊕mk=key*. We say that *mk* is *masked* and thus *leak free* because it is statistically independent of the value of *key*: if *mask1* is a uniform random number then so is *mk*. Therefore, when *mk* is aggregated over time, as in side-channel attacks, the result reveals no information of *key*.

Unfortunately, there can be leaks in *SecXor()* when the variables share a register and thus create second-order correlation. For example, the x86 assembly code of *mk=mask1⊕key* is `MOV mask1 %edx; XOR key %edx`, meaning the values stored in `%edx` are *mask1* and *mask1⊕key*, respectively. Since bit-flips in the register also affect the leakage current, they lead to side-channel leaks. This is captured by the Hamming Distance (HD) power model [22]: *HD(mask1,mask1⊕key) = HW(mask1 ⊕ (mask1 ⊕ key)) = HW(key)*, which reveals *key*. Consider, for example, where *key* is 0001_b and *mask1* is 1111_b in binary. If a register stores *mask1* (=1111_b) first and updates its value as *mask1⊕key* (=1110_b), the transition of the register (bit-flip) is 0001_b, which is same as the *key* value.

In embedded systems, specialized hardware [4, 50, 70] such as physically unclonable function (PUF) and true random number generator (TRNG) may produce *key* and *mask1* and map them to the memory address space; thus, these variables are considered leak

free. Specialized hardware may also directly produce the masked shares *{mask1,mk}* without producing the unmasked *key* in the first place. This more secure approach is shown in program *SecXor2()*, where masked shares are used to compute the result ($txt \oplus key$), which is also masked, but by *mask2* instead of *mask1*.

Inside *SecXor2()*, care should be taken to randomize the intermediate results by *mask2* first, before de-randomizing them by *mask1*. Thus, the CPU's registers never hold any unmasked result. However, there can still be HD leaks, for example, when the same register holds the following pairs at consecutive time steps: *(mask1,mk)*, *(mask1,t1)*, or *(mask2,t3)*.

2.2 Identifying the HD Leaks

To identify these leaks, we need to develop a scalable method. While there are techniques for detecting flaws in various masking implementations [9, 10, 17, 18, 23, 30, 33, 34, 39, 43, 66, 68, 69, 71], none of them was scalable enough for use in real compilers, or targeted the HD leaks caused by register reuse.

First, we check if there are sensitive, unmasked values stored in the CPU registers. Here, *mask* means a value is made statistically independent of the secret using randomization. We say a value is *HW-sensitive* if, statistically, it depends on the secret. For example, in Figure 2, *key* is HW-sensitive whereas $mk{=}mask1 \oplus key$ is masked. If there were $nk{=}mask1 \vee key$, it would be HW-sensitive because the masking is not perfect.

Second, we check if there is any pair of values (v_1, v_2) that, when stored in the same register, may cause an HD leak. That is, $HD(v_1, v_2) = HW(v_1 \oplus v_2)$ may statistically depend on the secret. For example, in Figure 2, *mk* and *mask1* form a HD-sensitive pair.

Formal Verification. Deciding if a variable is HW-sensitive, or two variables are HD-sensitive, is hard in general, since it corresponds to *model counting* [35, 83]. This can be illustrated by the truth table in Table 1 for functions *t1*, *t2* and *t3* over secret bit *k* and random bits *m1*, *m2* and *m3*. First, there is no HW leak because, regardless of whether *k*=0 or 1, there is a 50% chance of *t1* and *t2* being 1 and a 25% chance of *t3* being 1. This can be confirmed by counting the number of 1's in the top and bottom halves of the table.

When two values $(t1, t2)$ are stored in the same register, however, the bit-flip may depend on the secret. As shown in the column $HD(t1, t2)$ of the table, when $k = 0$, the bit is never flipped; whereas when $k = 1$, the bit is always flipped. The existence of HD leak for $(t1, t2)$ can be decided by model counting over the function $f_{t1 \oplus t2}(k, m1, m2, m3)$: the number of solutions is 0/8 for $k = 0$ but 8/8 for $k = 1$. In contrast, there is no HD leak for $(t2, t3)$ because the number of satisfying assignments (solutions) is always 2/8 regardless of whether $k = 0$ or $k = 1$.

Type Inference. Since model counting is expensive, we develop a fast, sound, and static type system to identify the HD-sensitive pairs in a program. Following Zhang et al. [83], we assign each variable one of three types: `RUD`, `SID` or `UKD` (details in Section 3). Briefly, `RUD` means random uniform distribution, `SID` means secret independent distribution, and `UKD` means unknown distribution. Therefore, a variable may have a leak only if it is the `UKD` type.

In Table 1, for example, given $t1 \leftarrow m1 \oplus m2$, where *m1* and *m2* are random (`RUD`), it is easy to see that *t1* is also random (`RUD`). For $t3 \leftarrow t2 \wedge m3$, where *t2*, *m3* are `RUD`, however, *t3* may not always be random, but we can still prove that *t3* is `SID`; that is, *t3* is statistically independent of *k*. This type of *syntactical* inference is fast because it does not rely on any *semantic* information, although in general, it is not as accurate as the model counting based approach. Nevertheless, such inaccuracy does not affect the soundness of our mitigation.

Table 1: Truth table showing that (1) there is no HW leak in *t1,t2,t3* but (2) there is an HD leak when *t1,t2* share a register.

k	m1	m2	m3	t1= m1⊕m2	t2= t1⊕k	t3= t2∧m3	HD(t1,t2) =t1⊕t2	HD(t2,t3) =t2⊕t3
0	0	0	0	0	0	0	**0**	0
0	0	0	1	0	0	0	**0**	0
0	0	1	0	1	1	0	**0**	1
0	0	1	1	1	1	1	**0**	0
0	1	0	0	1	1	0	**0**	1
0	1	0	1	1	1	1	**0**	0
0	1	1	0	0	0	0	**0**	0
0	1	1	1	0	0	0	**0**	0
1	0	0	0	0	1	0	**1**	1
1	0	0	1	0	1	1	**1**	0
1	0	1	0	1	0	0	**1**	0
1	0	1	1	1	0	0	**1**	0
1	1	0	0	1	0	0	**1**	0
1	1	0	1	1	0	0	**1**	0
1	1	1	0	0	1	0	**1**	1
1	1	1	1	0	1	1	**1**	0
UKD	RUD	RUD	RUD	RUD	RUD	SID	UKD	SID*

* Our Datalog based type inference rules can infer it as SID instead of UKD

Furthermore, we rely on a Datalog based declarative analysis framework [20, 48, 78, 79, 84] to implement and refine the type inference rules, which can infer $HD(t2, t3)$ as `SID` instead of `UKD`. We also leverage domain-specific optimizations, such as precomputing certain Datalog facts and using compiler's backend information, to reduce cost and improve accuracy.

2.3 Mitigating the HD Leaks

To remove the leaks, we constrain the register allocation algorithm using our inferred types. We focus on LLVM and x86, but the method is applicable to MIPS and ARM as well. To confirm this, we inspected the assembly code produced by LLVM for the example (*t1,t2,t3*) in Table 1 and found HD leaks on all three architectures. For x86, in particular, the assembly code is shown in Figure 3a, which uses %eax to store all intermediate variables and thus has a leak in *HD(t1,t2)*.

Figure 3b shows our mitigated code, where the HD-sensitive variables *t1* and *t2* are stored in different registers. Here, *t1* resides in %eax and memory -20(%rbp) whereas *t2* resides in %ecx and memory -16(%rbp). The stack and a value of %eax are shown in Figure 3c, both before and after mitigation, when the leak may occur at lines 8-9. Since the value of *k* is used only once in the example, i.e., for computing *t2*, overwriting its value stored in the original memory location -16(%rbp) does not affect subsequent execution. If *k* were to be used later, our method would have made a copy in memory and direct uses of *k* to that memory location.

Register allocation in real compilers is a highly optimized process. Thus, care should be taken to maintain correctness and performance. For example, the naive approach of assigning all HD-sensitive variables to different registers does not work because the number of registers is small (x86 has 4 general-purpose registers while MIPS has 24) while the number of sensitive variables is often large, meaning many variables must be *spilled* to memory.

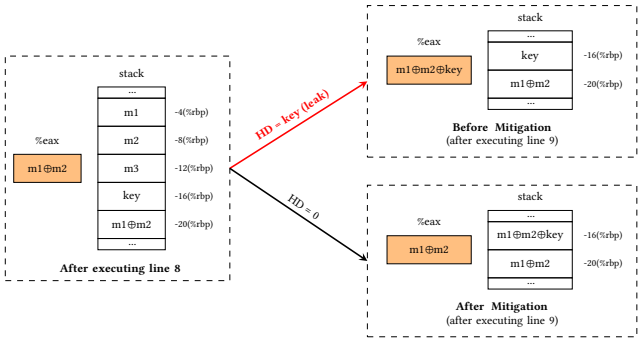
The instruction set architecture also add constraints. In x86, for example, %eax is related to %ah and %al and thus cannot be assigned independently. Furthermore, binary operations such as *Xor* may require that the result and one operand share the same register or memory location. Therefore, for $mk{=}mask1 \oplus key$, it means that either *mk* and *mask1* share a register, which causes a leak in *HD(mk,*

```
1  // assembly for Table1
2  movl %edi, -4(%rbp)
3  movl %esi, -8(%rbp)
4  movl %edx, -12(%rbp)
5  movl %ecx, -16(%rbp)
6  movl -4(%rbp), %eax
7  xorl -8(%rbp), %eax
8  movl %eax, -20(%rbp)
9  xorl -16(%rbp), %eax
10 movl %eax, -24(%rbp)
11 andl -12(%rbp), %eax
12 movl %eax, -28(%rbp)
13
14 popq %rbp
```

(a) Before Mitigation

```
1  // assembly for Table1
2  movl %edi, -4(%rbp)
3  movl %esi, -8(%rbp)
4  movl %edx, -12(%rbp)
5  movl %ecx, -16(%rbp)
6  movl -4(%rbp), %eax
7  xorl -8(%rbp), %eax
8  movl %eax, -20(%rbp)
9  xorl %eax, -16(%rbp)
10 movl -16(%rbp), %ecx
11 andl -12(%rbp), %ecx
12 movl %ecx, -28(%rbp)
13 movl -28(%rbp), %eax
14 popq %rbp
```

(b) After Mitigation



(c) Diagram for stack and register %eax

Figure 3: The assembly code before and after mitigation.

mask1)=HW(key), or *mk* and *key* share a register, which causes a leak in *HW(key)* itself. Thus, while modifying the backend, multiple submodules must be constrained together to ensure the desired register and memory isolations (see Section 5).

2.4 Leaks in High-order Masking

Here, a question is whether the HD leak can be handled by second-order masking (which involves two variables). The answer is no, because even with high-order masking techniques such as Barthe et al. [8–10], the compiled code may still have HD leaks introduced by register reuse. We confirmed this through experiments, where the code compiled by LLVM for high-order masked programs from [8] was found to contain HD leaks.

Figure 4 illustrates this problem on a second-order arithmetic masking of the multiplication of `txt` (public) and `key` (secret) in a finite field. Here, the symbol $*$ denotes multiplication. While there are a lot of details, at a high level, the program relies on the same idea of *secret sharing*: random variables are used to split the secret *key* to three shares, before these shares participate in the computation. The result is a masked triplet *(res0,res1,res2)* such that *(res0⊕res1⊕res2)=key∗txt*.

The x86 assembly code in Figure 4 has leaks because the same register %edx stores both *mask0* ⊕ *mask1* and *mask0* ⊕ *mask1* ⊕ *key*. Let the two values be denoted $\%edx_1$ and $\%edx_2$, we have $HD(\%edx_1, \%edx_2) = HW(key)$. Similar leaks exist in the LLVM-generated assembly code of this program for ARM and MIPS as well, but we omit them for brevity.

3 PRELIMINARIES

We first define the threat model and then review the leakage models used for quantifying the power side channel.

```
1  uint8 SecondOrderMaskingMultiply(uint8 txt, uint8 key) {
2      int mask0, mask1, mask2, mask3, mask4, mask5, mask6; //random
3      int t1 = mask0 ^ mask1 ^ key;
4      int t2 = mask2 ^ mask3 ^ txt;
5      int t3 = (mask4 ^ mask0 * mask3) ^ mask1 * mask2;
6      int t4 = (mask5 ^ mask0 * t2) ^ t1 * mask2;
7      int t5 = (mask6 ^ mask1 * t2) ^ t1 * mask3;
8      res0 = (mask0 * mask2 ^ mask4) ^ mask5;
9      res1 = (mask1 * mask3 ^ t3) ^ mask6;
10     res2 = (t1 * t2 ^ t4) ^ t5;
11     return {res0, res1, res2};
12 }
```

```
movzbl -41(%rbp), %edx // mask0 is loaded to %edx
movzbl -43(%rbp), %esi // mask1 is loaded to %esi
xorl %esi, %edx   // mask0^mask1 is stored to %edx (%edx1)
movzbl -44(%rbp), %esi // key is loaded to %esi
xorl %esi, %edx   // mask0^mask1^key is stored to %edx (%edx2)
movb %dl, %al
movb %al, -50(%rbp)
```

Figure 4: Second-order masking of multiplication in a finite field, and the LLVM-generated x86 assembly code of Line 3.

3.1 The Threat Model

We assume the attacker has access to the software code, but not the secret data, and the attacker's goal is to gain information of the secret data. The attacker may measure the power consumption of a device that executes the software, at the granularity of each machine instruction. A set of measurement traces is aggregated to perform statistical analysis, e.g., as in DPA attacks. In mitigation, our goal is to eliminate the statistical dependence between secret data and the (aggregated) measurement data.

Let P be the program under attack and the triplet $(\mathbf{x}, \mathbf{k}, \mathbf{r})$ be the input: sets $\mathbf{x}$, $\mathbf{k}$ and $\mathbf{r}$ consist of *public*, *secret*, and *random (mask)* variables, respectively. Let x, k_1, k_2, and r be valuations of these input variables. Then, $\sigma_t(P, x, k_1, r)$ denotes, at time step t, the power consumption of a device executing P under input x, k_1 and r. Similarly, $\sigma_t(P, x, k_2, r)$ denotes the power consumption of the device executing P under input x, k_2 and r. Between steps t and $t+1$, one instruction in P is executed.

We say P has a leak if there are t, x, k_1 and k_2 such that the distribution of $\sigma_t(P, x, k_1, r)$ differs from that of $\sigma_t(P, x, k_2, r)$. Let random variables in $\mathbf{r}$ be uniformly distributed in the domain R, and let the probability of each $r \in R$ be $Pr(r)$, we expect $\forall t, x, k_1, k_2$.

$$\sum_{r \in R} \sigma_t(P, x, k_1, r)\ Pr(r) = \sum_{r \in R} \sigma_t(P, x, k_2, r)\ Pr(r) \qquad (1)$$

For efficiency reasons, in this work, we identify *sufficient conditions* under which Formula 1 is implied. Toward this end, we focus on the leaks of individual variables, and pairs of variables, in P instead of the sum σ_t: if we remove all individual leaks, the leak-free property over the sum $\sigma_t(P, x, k, r)$ is implied.

3.2 The Leakage Model

In the Hamming Weight (HW) model [52, 53], the leakage associated with a register value, which corresponds to an intermediate variable in the program, depends on the number of 1-bits. Let the value be $D = \sum_{i=0}^{n-1} d_i 2^i$ where d_0 is the least significant bit, d_{n-1} is the most significant bit, and each bit d_i, where $0 \le i < n$, is either 0 or 1. The Hamming Weight of D is $HW(D) = \sum_{i=0}^{n-1} d_i$.

In the Hamming Distance (HD) model [52, 53], the leakage depends not only on the current register value D but also a reference value D'. Let $D' = \sum_{i=0}^{n-1} d'_i 2^i$. We define the Hamming Distance between D and D' as $HD(D, D') = \sum_{i=0}^{n-1} d_i \oplus d'_i$, which is equal to

$HW(D \oplus D')$, the Hamming Weight of the bit-wise XOR of D and D'. Another interpretation is to regard $HW(D)$ as a special case of $HD(D, D')$, where all bits in the reference value D' are set to 0.

The HW/HD models have been validated on real devices [22, 27, 46, 55, 58]. The correlation between power variance and number of 1-bits may be explained using the *leakage current* of a CMOS transistor, which is the foundation of modern computing devices. Broadly speaking, a CMOS transistor has two kinds of leakage currents: *static* and *dynamic*. Static leakage current exists all the time but the volume depends on whether the transistor is on or off, i.e., a logical 1. Dynamic leakage current occurs only when a transistor is switched (0-1 or 1-0 flip). While static leakage current is captured by the HW model, dynamic leakage current is captured by the HD model (for details refer to Mangard [52].)

3.3 The Data Dependency

We consider two dependency relations: *syntactical* and *statistical*. Syntactical dependency is defined over the program structure: a function $f(k, \ldots)$ syntactically depends on the variable k, denoted $\mathcal{D}_{syn}(f, k)$, if k appears in the expression of f; that is, k is in the support of f, denoted $k \in supp(f)$.

Statistical dependency is concerned with scenarios where random variables are involved. For example, when $f(k, r) = k \oplus r$, the probability of f being logical 1 (always 50%) is not dependent on k. However, when $f(k, r) = k \vee r$, where r is uniformly distributed in $[0, 1]$, the probability of f being logical 1 is 100% when k is 1, but 50% when k is 0. In the latter case, we say that f is statistically dependent on k, denoted $\mathcal{D}_{sta}(f, k)$.

The relative strengths of the dependency relations are as follows: $\neg\mathcal{D}_{syn}(f, k) \implies \neg\mathcal{D}_{sta}(f, k)$, i.e., if f is syntactically independent of k, it is statistically independent of k. In this work, we rely on $\mathcal{D}_{syn}$ to infer $\mathcal{D}_{sta}$ during type inference, since the detection of HD leaks must be both fast and sound.

4 TYPE-BASED STATIC LEAK DETECTION

We use a type system that starts from the input annotation (IN_{PUBLIC}, IN_{SECRET} and IN_{RANDOM}) and computes a *distribution type* for all variables. The type indicates whether a variable may statistically depend on the secret input.

4.1 The Type Hierarchy

The distribution type of variable v, denoted $\mathsf{TYPE}(v)$, may be one of the following kinds:

- RUD, which stands for *random uniform distribution*, means v is either a random input $m \in IN_{RANDOM}$ or perfectly randomized [18] by m, e.g., $v = k \oplus m$.
- SID, which stands for *secret independent distribution*, means that, while not RUD, v is statistically independent of the secret variable in IN_{SECRET}.
- UKD, which stands for *unknown distribution*, indicates that we are not able to prove that v is RUD or SID and thus have to assume that v may have a leak.

The three types form a hierarchy: UKD is the least desired because it means that a leak may exist. SID is better: although it may not be RUD, we can still prove that it is statistically independent of the secret, i.e., no leak. RUD is the most desired because the variable not only is statistically independent of the secret (same as in SID), but also can be used like a random input, e.g., to mask other (UKD) variables. For leak mitigation purposes, it is always sound to treat an RUD variable as SID, or an SID variable as UKD, although it may force instructions to be unnecessarily mitigated.

In practice, we want to infer as many SID and RUD variables as possible. For example, if $k \in IN_{SECRET}$, $m \in IN_{RANDOM}$ and $k_m = k \oplus m$, then $\mathsf{TYPE}(k) = \mathsf{UKD}$ and $\mathsf{TYPE}(k_m) = \mathsf{RUD}$. If $x \in IN_{PUBLIC}$ and $xk_m = x \wedge k_m$, then $\mathsf{TYPE}(xk_m) = \mathsf{SID}$ because, although x may have any distribution, since k_m is RUD, xk_m is statistically independent of the secret.

We prefer RUD over SID, when both are applicable to a variable x_1, because if x_1 is XOR-ed with a UKD variable x_2, we can easily prove that $x = x_1 \oplus x_2$ is RUD using local inference, as long as x_1 is RUD and x_2 is not randomized by the same input variable. However, if x_1 is labeled not as RUD but as SID, local inference rules may not be powerful enough to prove that x is RUD or even SID; as a result, we have to treat x as UKD (leak), which is less accurate.

4.2 Datalog based Analysis

In the remainder of this section, we present type inference for individual variables first, and then for HD-sensitive pairs.

We use Datalog to implement the type inference. Here, program information is captured by a set of relations called the *facts*, which include the annotation of input in IN_{PUBLIC} (SID), IN_{SECRET} (UKD) and IN_{RANDOM} (RUD). The inference algorithm is codified in a set of relations called the *rules*, which are steps for deducing types. For example, when $z = x \oplus m$ and m is RUD, z is also RUD regardless of the actual expression that defines x, as long as $m \notin supp(x)$. This can be expressed as an inference rule.

After generating both the facts and the rules, we combine them to form a Datalog program, and solve it using an off-the-shelf Datalog engine. Inside the engine, the rules are applied to the facts to generate new facts (types); the iterative procedure continues until the set of facts reaches a fixed point.

Since our type inference is performed on the LLVM IR, there are only a few instruction types to consider. For ease of presentation, we assume that a variable v is defined by either a unary operator or a binary operator (n-ary operator may be handled similarly).

- $v \leftarrow Uop(v_1)$, where Uop is a unary operator such as the Boolean (or bit-wise) negation.
- $v \leftarrow Bop(v_1, v_2)$, where Bop is a binary operator such as Boolean (or bit-wise) $\oplus$, $\wedge$, $\vee$ and $*$ (finite-field multiplication).

For $v \leftarrow Uop(v_1)$, we have $\mathsf{TYPE}(v) = \mathsf{TYPE}(v_1)$, meaning v and v_1 have the same type. For $v \leftarrow Bop(v_1, v_2)$, the type depends on (1) if Bop is Xor, (2) if $\mathsf{TYPE}(v_1)$ and $\mathsf{TYPE}(v_2)$ are SID or RUD, and (3) the sets of input variables upon which v_1 and v_2 depend.

4.3 Basic Type Inference Rules

Prior to defining the rules for Bop, we define two related functions, unq and dom, in addition to $\mathsf{supp}(v)$, which is the set of input variables upon which v depends syntactically.

Definition 4.1. $\mathsf{unq} : V \rightarrow IN_{RANDOM}$ *is a function that returns, for each variable* $v \in V$*, a subset of mask variables defined as follows: if* $v \in IN_{RANDOM}$, $\mathsf{unq}(v) = \{v\}$*; but if* $v \in IN \setminus IN_{RANDOM}$, $\mathsf{unq}(v) = \{\}$*;*

- *if* $v \leftarrow Uop(v_1)$, $\mathsf{unq}(v) = \mathsf{unq}(v_1)$*; and*
- *if* $v \leftarrow Bop(v_1, v_2)$, $\mathsf{unq}(v) = (\mathsf{unq}(v_1) \cup \mathsf{unq}(v_2)) \backslash (\mathsf{supp}(v_1) \cap \mathsf{supp}(v_2))$.

Given the data-flow graph of all instructions involved in computing v and an input variable $m \in \mathsf{unq}(v)$, there must exist a unique path from m to v in the graph. If there are more paths (or no path), m would not have appeared in $\mathsf{unq}(v)$.

Definition 4.2. $\mathsf{dom} : V \rightarrow IN_{RANDOM}$ *is a function that returns, for each variable* $v \in V$*, a subset of mask variables defined as follows: if* $v \in IN_{RANDOM}$, $\mathsf{dom}(v) = \{v\}$, *but if* $v \in IN \setminus IN_{RANDOM}$, *then* $\mathsf{dom}(v) = \{\ \}$;

- *if* $v \leftarrow \mathsf{Uop}(v_1)$, $\mathsf{dom}(v) = \mathsf{dom}(v_1)$; *and*
- *if* $v \leftarrow \mathsf{Bop}(v_1, v_2)$, *where operator* $\mathsf{Bop} = \mathsf{Xor}$, *then* $\mathsf{dom}(v) = (\mathsf{dom}(v_1) \cup \mathsf{dom}(v_2)) \cap \mathsf{unq}(v)$; *else* $\mathsf{dom}(v) = \{\ \}$.

Given the data-flow graph of all instructions involved in computing v and an input $m \in \mathsf{dom}(v)$, there must exist a unique path from m to v, along which all binary operators are Xor; if there are more such paths (or no path), m would not have appeared in $\mathsf{dom}(v)$.

Following the definitions of supp, unq and dom, it is straightforward to arrive at the basic inference rules [9, 61, 83]:

$$Rule_1 \ \frac{\mathsf{dom}(v) \neq \emptyset}{\mathsf{TYPE}(v) = \mathsf{RUD}}$$

$$Rule_2 \ \frac{\mathsf{supp}(v) \cap IN_{SECRET} = \emptyset \wedge \mathsf{TYPE}(v) \neq \mathsf{RUD}}{\mathsf{TYPE}(v) = \mathsf{SID}}$$

Here, *Rule*$_1$ says if $v = m \oplus expr$, where m is a random input and $expr$ is not masked by m, then v has random uniform distribution. This is due to the property of XOR. *Rule*$_2$ says if v is syntactically independent of variables in IN_{SECRET}, it has a secret independent distribution, provided that it is not RUD.

4.4 Inference Rules to Improve Accuracy

With the two basic rules only, any variable not assigned RUD or SID will be treated as UKD, which is too conservative. For example, $v = (k \oplus m) \wedge x$ where $k \in IN_{SECRET}$, $m \in IN_{RANDOM}$ and $x \in IN_{PUBLIC}$, is actually SID. This is because $k \oplus m$ is random and the other component, x, is secret independent. Unfortunately, the two basic rules cannot infer that v is SID. The following rules are added to solve this problem.

$$Rule_{3a} \ \frac{\begin{array}{c} v \leftarrow Bop(v_1, v_2) \wedge \mathsf{supp}(v_1) \cap \mathsf{supp}(v_2) = \emptyset \\ \wedge Bop \notin \{Xor, GMul\} \wedge \mathsf{TYPE}(v_1) = \mathsf{RUD} \wedge \mathsf{TYPE}(v_2) = \mathsf{SID} \end{array}}{\mathsf{TYPE}(v) = \mathsf{SID}}$$

$$Rule_{3b} \ \frac{\begin{array}{c} v \leftarrow Bop(v_1, v_2) \wedge \mathsf{supp}(v_1) \cap \mathsf{supp}(v_2) = \emptyset \\ \wedge Bop \notin \{Xor, GMul\} \wedge \mathsf{TYPE}(v_1) = \mathsf{SID} \wedge \mathsf{TYPE}(v_2) = \mathsf{RUD} \end{array}}{\mathsf{TYPE}(v) = \mathsf{SID}}$$

These rules mean that, for any $Bop = \{\wedge, \vee\}$, if one operand is RUD, the other operand is SID, and they share no input, then v has a secret independent distribution (SID). *GMul* denotes multiplication in a finite field. Here, $\mathsf{supp}(v_1) \cap \mathsf{supp}(v_2) = \emptyset$ is need; otherwise, the common input may cause problem. For example, if $v_1 \leftarrow m \oplus k$ and $v_2 \leftarrow m \wedge x$, then $v = (v_1 \wedge v_2) = (m \wedge \neg k) \wedge x$ has a leak because if $k = 1$, $v = 0$; but if $k = 0$, $v = m \wedge x$.

$$Rule_4 \ \frac{\begin{array}{c} v \leftarrow Bop(v_1, v_2) \wedge \mathsf{supp}(v_1) \cap \mathsf{supp}(v_2) = \emptyset \\ \wedge \mathsf{TYPE}(v_1) = \mathsf{SID} \wedge \mathsf{TYPE}(v_2) = \mathsf{SID} \end{array}}{\mathsf{TYPE}(v) = \mathsf{SID}}$$

Similarly, *Rule*$_4$ may elevate a variable v from UKD to SID, e.g., as in $v \leftarrow ((k \oplus m) \wedge x_1) \wedge (x_2)$ where x_1 and x_2 are both SID. Again, the condition $\mathsf{supp}(v_1) \cap \mathsf{supp}(v_2) = \emptyset$ in *Rule*$_4$ is needed because, otherwise, there may be cases such as $v \leftarrow ((k \oplus m) \wedge x_1) \wedge (x_2 \wedge m)$, which is equivalent to $v \leftarrow \neg k \wedge (m \wedge x_1 \wedge x_2)$ and thus has a leak.

Figure 5 shows the other inference rules used in our system. Since these rules are self-explanatory, we omit the proofs.

4.5 Detecting HD-sensitive Pairs

Based on the variable types, we compute HD-sensitive pairs. For each pair (v_1, v_2), we check if $HD(v_1, v_2)$ results in a leak when v_1 and v_2 share a register. There are two scenarios:

- $v_1 \leftarrow expr_1; v_2 \leftarrow expr_2$, meaning v_1 and v_2 are defined in two instructions.
- $v_1 \leftarrow Bop(v_2, v_3)$, where the result v_1 and one operand (v_2) are stored in the same register.

In the *two-instruction* case, we check $HW(expr_1 \oplus expr_2)$ using *Xor*-related inference rules. For example, if $v_1 \leftarrow k \oplus m$ and $v_2 \leftarrow m$, since m appears in the supports of both expressions, $(k \oplus m) \oplus m$ is UKD. Such leak will be denoted $\mathsf{SEN\_HD}_D(v_1, v_2)$, where D stands for "Double".

In the *single-instruction* case, we check $HW(Bop(v_2, v_3) \oplus v_2)$ based on the operator type. When $Bop = \wedge$, we have $(v_2 \wedge v_3) \oplus v_2 = v_2 \wedge \neg v_3$; when $Bop = \vee$, we have $(v_2 \vee v_3) \oplus v_2 = (\neg v_2 \wedge v_3)$; when $Bop = \oplus$ (*Xor*), we have $(v_2 \oplus v_3) \oplus v_2 = v_3$; and when $Bop = *$ (*GMul*), the result of $(v_2 * v_3) \oplus v_2$ is $\{v_2, v_3\}$ if $v_2 * v_3 \neq 0x01$ and is $(v_2 \oplus 0x01)$ otherwise. Since the type inference procedure is agnostic to the result of $(v_2 * v_3)$, the type of $(v_2 * v_3) \oplus v_2$ depends on the types of v_3 and v_2; that is, $\mathsf{TYPE}(v_2) = \mathsf{UKD} \vee \mathsf{TYPE}(v_3) = \mathsf{UKD} \implies \mathsf{TYPE}((v_2 * v_3) \oplus v_2) = \mathsf{UKD}$. If there is a leak, it will be denoted $\mathsf{SEN\_HD}_S(v_1, v_2)$.

The reason why HD leaks are divided to $\mathsf{SEN\_HD}_D$ and $\mathsf{SEN\_HD}_S$ is because they have to be mitigated differently. When the leak involves two instructions, it may be mitigated by constraining the register allocation algorithm such that v_1 and v_2 no longer can share a register. In contrast, when the leak involves a single instruction, it cannot be mitigated in this manner because in x86, for example, all binary instructions require the result to share the same register or memory location with one of the operands. Thus, mitigating the $\mathsf{SEN\_HD}_S$ requires that we rewrite the instruction itself.

We also define a relation $Share(v_1, v_2)$, meaning v_1 and v_2 indeed may share a register, and use it to filter the HD-sensitive pairs, as shown in the two rules below.

$$\frac{Share(v_1, v_2) \wedge \mathsf{TYPE}(v_1 \oplus v_2) = \mathsf{UKD} \wedge v_1 \leftarrow expr_1 \wedge v_2 \leftarrow expr_2}{\mathsf{SEN\_HD}_D(v_1, v_2)}$$

$$\frac{Share(v_1, v_2) \wedge \mathsf{TYPE}(v_1 \oplus v_2) = \mathsf{UKD} \wedge v_1 \leftarrow Bop(v_2, v_3)}{\mathsf{SEN\_HD}_S(v_1, v_2)}$$

Backend information (Section 6.1) is required to define the relation; for now, we assume $\forall v_1, v_2 : Share(v_1, v_2) = true$.

5 MITIGATION DURING CODE GENERATION

We mitigate leaks by using the two types of HD-sensitive pairs as constraints during register allocation.

Register Allocation. The classic approach, especially for static compilation, is based on *graph coloring* [24, 38], whereas dynamic compilation may use faster algorithms such as *lossy graph coloring* [28] or *linear scan* [65]. We apply mitigation on both graph coloring and LLVM's basic register allocation algorithms. For ease of comprehension, we use graph coloring to illustrate our constraints.

$$Rule_{5a}\ \frac{v \leftarrow Bop(v_1, v_2) \wedge \mathrm{dom}(v_1) \setminus \mathrm{supp}(v_2) = \emptyset \wedge \mathrm{TYPE}(v_1) = \mathrm{RUD} \wedge \mathrm{dom}(v_1) = \mathrm{dom}(v_2) \wedge \mathrm{supp}(v_1) = \mathrm{supp}(v_2)}{\mathrm{TYPE}(v) = \mathrm{SID}}$$

$$Rule_{5b}\ \frac{v \leftarrow Bop(v_1, v_2) \wedge \mathrm{dom}(v_2) \setminus \mathrm{supp}(v_1) = \emptyset \wedge \mathrm{TYPE}(v_2) = \mathrm{RUD} \wedge \mathrm{dom}(v_1) = \mathrm{dom}(v_2) \wedge \mathrm{supp}(v_1) = \mathrm{supp}(v_2)}{\mathrm{TYPE}(v) = \mathrm{SID}}$$

$$Rule_{6}\ \frac{v \leftarrow Bop(v_1, v_2) \wedge Bop \notin \{Xor, GMul\} \wedge (\mathrm{dom}(v_1) \setminus \mathrm{supp}(v_2) \neq \emptyset \vee \mathrm{dom}(v_2) \setminus \mathrm{supp}(v_1) \neq \emptyset) \wedge \mathrm{TYPE}(v_1) = \mathrm{RUD} \wedge \mathrm{TYPE}(v_2) = \mathrm{RUD}}{\mathrm{TYPE}(v) = \mathrm{SID}}$$

$$Rule_{7a}\ \frac{v \leftarrow Bop(v_1, v_2) \wedge Bop = GMul \wedge \mathrm{TYPE}(v_1) = \mathrm{RUD} \wedge \mathrm{TYPE}(v_2) = \mathrm{SID} \wedge \mathrm{dom}(v_1) \setminus supp(v_2) \neq \emptyset}{\mathrm{TYPE}(v) = \mathrm{SID}}$$

$$Rule_{7b}\ \frac{v \leftarrow Bop(v_1, v_2) \wedge Bop = GMul \wedge \mathrm{TYPE}(v_1) = \mathrm{SID} \wedge \mathrm{TYPE}(v_2) = \mathrm{RUD} \wedge \mathrm{dom}(v_2) \setminus supp(v_1) \neq \emptyset}{\mathrm{TYPE}(v) = \mathrm{SID}}$$

$$Rule_{8}\ \frac{v \leftarrow Bop(v_1, v_2) \wedge Bop = GMul \wedge (\mathrm{dom}(v_1) \setminus dom(v_2) \neq \emptyset \vee \mathrm{dom}(v_2) \setminus dom(v_1) \neq \emptyset) \wedge \mathrm{TYPE}(v_1) = \mathrm{RUD} \wedge \mathrm{TYPE}(v_2) = \mathrm{RUD}}{\mathrm{TYPE}(v) = \mathrm{SID}}$$

Figure 5: The remaining inference rules used in our type system (in addition to $Rule_{1-4}$).

In graph coloring, each variable corresponds to a node and each edge corresponds to an *interference* between two variables, i.e., they may be in use at the same time and thus cannot occupy the same register. Assigning variables to k registers is similar to coloring the graph with k colors. To be efficient, variables may be grouped to clusters, or *virtual registers*, before they are assigned to physical registers (colors). In this case, each virtual register (*vreg*), as opposed to each variable, corresponds to a node in the graph, and multiple virtual registers may be mapped to one physical register.

5.1 Handling SEN_HD$_D$ Pairs

For each $\mathrm{SEN\_HD}_D(v_1, v_2)$, where v_1 and v_2 are defined in two instructions, we add the following constraints. First, v_1 and v_2 are not to be mapped to the same virtual register. Second, virtual registers $vreg_1$ and $vreg_2$ (for v_1 and v_2) are not to be mapped to the same physical register. Toward this end, we constrain the behavior of two backend modules: *Register Coalescer* and *Register Allocator*.

Our constraint on *Register Coalescer* states that $vreg_1$ and $vreg_2$, which correspond to v_1 and v_2, must never coalesce, although each of them may still coalesce with other virtual registers. As for *Register Allocator*, our constraint is on the formulation of the graph. For each HD-sensitive pair, we add a new *interference* edge to indicate that $vreg_1$ and $vreg_2$ must be assigned different colors.

During graph coloring, these new edges are treated the same as all other edges. Therefore, our constraints are added to the register allocator and its impact is propagated automatically to all subsequent modules, regardless of the architecture (x86, MIPS or ARM). When variables cannot fit in the registers, some will be *spilled* to memory, and all reference to them will be directed to memory. Due to the constraints we added, there may be more spilled variables, but spilling is handled transparently by the existing algorithms in LLVM. This is an advantage of our approach: it identifies a way to constrain the behavior of existing modules in LLVM, without the need to reimplement any module from scratch.

5.2 Handling SEN_HD$_S$ Pairs

For each $\mathrm{SEN\_HD}_S(v_1, v_2)$ pair, where v_1 and v_2 appear in the same instruction, we additionally constrain the *DAG Combiner* module to rewrite the instruction before constraining the register allocation modules. To see why, consider $mk = (m \oplus k)$, which compiles to

```
MOVL -4(%rbp), %ecx //-4(%rbp)= m (random)
XORL -8(%rbp), %ecx //-8(%rbp)= k (secret)
```

Here, -4(%rbp) and -8(%rbp) are memory locations for m and k, respectively. Although m and mk are RUD (no leak) when stored in %ecx, the transition from m to mk, $HW(m \oplus mk) = k$, has a leak.

```
1  void remask(uint8_t s[16], uint8_t m1, uint8_t m2, uint8_t m3, uint8_t m4,
       uint8_t m5, uint8_t m6, uint8_t m7, uint8_t m8){
2      int i;
3      for(i = 0; i< 4; i++){
4          s[0+i*4] = s[0+i*4] ^ (m1^m5);
5          s[1+i*4] = s[1+i*4] ^ (m2^m6);
6          s[2+i*4] = s[2+i*4] ^ (m3^m7);
7          s[3+i*4] = s[3+i*4] ^ (m4^m8);
8      }
9  }
```

```
1  //Before Mitigation
2  movslq -28(%rbp), %rdx
3  movq -16(%rbp), %rcx
4  movzbl (%rcx,%rdx,4), %edi
5  movzbl -17(%rbp), %esi
6  movzbl -21(%rbp), %eax
7  xorl %esi, %eax
8  xorl %edi, %eax
9  movb %al, (%rcx,%rdx,4)
```

```
1  //After mitigation
2  movslq -28(%rbp), %rdx
3  movq -16(%rbp), %rcx
4
5  movzbl -17(%rbp), %esi
6  movzbl -21(%rbp), %eax
7  xorl %esi, %eax
8
9  xorb %al, (%rcx,%rdx,4)
```

Figure 6: Code snippet from the Byte Masked AES [82].

To remove the leak, we must rewrite the instruction:

```
MOVL -4(%rbp), %ecx //-4(%rbp)= m
XORL %ecx, -8(%rbp)//-8(%rbp)= k, and then mk
```

While m still resides in %ecx, both k and mk reside in the memory -8(%rbp). There is no leak because %ecx only stores m (RUD) and $HW(m \oplus m) = 0$. Furthermore, the solution is efficient in that no additional memory is needed. If k were to be used subsequently, we would copy k to another memory location and re-directed uses of k to that location.

Example. Figure 6 shows a real program [82], where s is an array storing sensitive data while $m1$-$m8$ are random masks. The compiled code (left) has leaks, whereas the mitigated code (right) is leak free. The reason why the original code (left) has leaks is because, prior to Line 8, %eax stores $m1 \oplus m5$, whereas after Line 8, %eax stores $s[0 + i * 4] \oplus m1 \oplus m5$; thus, bit-flips in %eax is reflected in $HW(\%eax_1 \oplus \%eax_2) = s[0 + i * 4]$, which is the sensitive data.

During register allocation, a virtual register $vreg_1$ would correspond to $m1 \oplus m5$ while $vreg_2$ would correspond to $s[0 + i * 4] \oplus m1 \oplus m5$. Due to a constraint from this $\mathrm{SEN\_HD}_S$ pair, our method would prevent $vreg_1$ and $vreg_2$ from coalescing, or sharing a physical register. After rewriting, $vreg_2$ shares the same memory location as $s[0 + i * 4]$) while $vreg_1$ remains unchanged. Thus, $m1 \oplus m5$ is stored in %al and $s[0 + i * 4] \oplus m1 \oplus m5$ is spilled to memory, which removes the leak.

6 DOMAIN-SPECIFIC OPTIMIZATIONS

While the method presented so far has all the functionality, it can be made faster by domain-specific optimizations.

6.1 Leveraging the Backend Information

To detect HD leaks that likely occur, we focus on pairs of variables that may share a register as opposed to arbitrary pairs of variables. For example, if the live ranges of two variables overlap, they will never share a register, and we should not check them for HD leaks. Such information is readily available in the compiler's backend modules, e.g., in graph coloring based register allocation, variables associated with any *interference* edge cannot share a register.

Thus, we define $Share(v_1, v_2)$, meaning v_1 and v_2 may share a register. After inferring the variable types as RUD, SID, or UKD, we use $Share(v_1, v_2)$ to filter the variable pairs subjected to checking for $\texttt{SEN\_HD}_D$ and $\texttt{SEN\_HD}_S$ leaks (see Section 4.5). We will show in experiments that such backend information allows us to dramatically reduce the number of HD-sensitive pairs.

6.2 Pre-computing Datalog Facts

By default, only input annotation and basic data-flow (def-use) are encoded as Datalog facts, whereas the rest has to be deduced by inference rules. However, Datalog is not the most efficient way of computing sets, such as $supp(v)$, $unq(v)$ and $dom(v)$, or performing set operations such as $m_1 \in supp(v)$.

In contrast, it is linear time [61] to compute sets such as $supp(v)$, $unq(v)$ and $dom(v)$ explicitly. Thus, we choose to precompute them in advance and encode the results as Datalog facts. In this case, precomputation results are used to jump start Datalog based type inference. We will show, through experiments, that the optimization can lead to faster type inference than the default implementation.

6.3 Efficient Encoding of Datalog Relations

There are different encoding schemes for Datalog. For example, if $IN = \{i_0, \ldots, i_3\}$ and $supp(v_1) = \{i_1, i_2\}$ and $supp(v_2) = \{i_0, i_1, i_3\}$. One way is to encode the sets is using a relation $Supp : V \times IN$, where V are variables and IN are supporting inputs:

$$\begin{aligned} &Supp(v_1, i_1) \wedge Supp(v_1, i_2) &&= supp(v_1) \\ &Supp(v_2, i_0) \wedge Supp(v_2, i_1) \wedge Supp(v_2, i_3) &&= supp(v_2) \end{aligned}$$

While the size of $Supp$ is $|V||IN|$, each set needs up to $|IN|$ predicates, and set operation needs $|IN|^2$ predicates.

Another way is to encode the sets is using a relation $Supp : V \times 2^{IN}$, where 2^{IN} is the power-set (set of all subsets of IN):

$$\begin{aligned} Supp(v_1, b0110) &= supp(v_1) \\ Supp(v_2, b1011) &= supp(v_2) \end{aligned}$$

While the size of $Supp$ is $|V|\ 2^{|IN|}$, each set needs one predicate, and set operation needs 2 predicates (a bit-wise operation). When $|IN|$ is small, the second approach is more compact; but as $|IN|$ increases, the table size of $Supp$ increases exponentially.

Therefore, we propose an encoding, called segmented bitset representation *(idx,bitset)*, where *idx=i* refers to the i-th segment and $bitset_i$ denotes the bits in the i-th segment.

$$\begin{aligned} &Supp(v_1, 1, b01) \wedge Supp(v_1, 0, b10) &&= supp(v_1) \\ &Supp(v_2, 1, b10) \wedge Supp(v_2, 0, b11) &&= supp(v_2) \end{aligned}$$

In practice, when the *bitset* size is bounded, e.g., to 4, the table size remains small while the number of predicates increases moderately. This encoding scheme is actually a generalization of the previous two. When the size of *bitset* decreases to 1 and the number of segments increases to $|IN|$, it degenerates to the first approach. When the size of *bitset* increases to $|IN|$ and the number of segments decrease to 1, it degenerates to the second approach.

Table 2: Statistics of the benchmark programs.

Name	Description	LoC	Program Variables			
			IN_{PUBLIC}	IN_{SECRET}	IN_{RANDOM}	Internal
P1	AES Shift Rows [14]	11	0	2	2	22
P2	Messerges Boolean [14]	12	0	2	2	23
P3	Goubin Boolean [14]	12	0	1	2	32
P4	SecMultOpt_wires_1 [69]	25	1	1	3	44
P5	SecMult_wires_1 [69]	25	1	1	3	35
P6	SecMultLinear_wires_1 [69]	32	1	1	3	59
P7	CPRR13-lut_wires_1 [30]	81	1	1	7	169
P8	CPRR13-OptLUT_wires_1 [30]	84	1	1	7	286
P9	CPRR13-1_wires_1 [30]	104	1	1	7	207
P10	KS_transitions_1 [8]	964	1	16	32	2,329
P11	KS_wires [8]	1,130	1	16	32	2,316
P12	keccakf_1turn [8]	1,256	0	25	75	2,314
P13	keccakf_2turn [8]	2,506	0	25	125	4,529
P14	keccakf_3turn [8]	3,764	0	25	175	6,744
P15	keccakf_7turn [8]	8,810	0	25	349	15,636
P16	keccakf_11turn [8]	13,810	0	25	575	24,472
P17	keccakf_15turn [8]	18,858	0	25	775	33,336
P18	keccakf_19turn [8]	23,912	0	25	975	42,196
P19	keccakf_24turn [8]	30,228	0	25	1,225	53,279
P20	AES_wires_1 [30]	34,358	16	16	1,232	63,263

7 EXPERIMENTS

We have implemented our method in LLVM 3.6 [49]. We used the μZ [42] Datalog engine in Z3 [31] to infer types. While the mitigation part targeted x86, it may be extended to other platforms. We conducted experiments on a number of cryptographic programs. Table 2 shows the statistics, including the name, a description, the number of lines of code, and the number of variables, which are divided further to input and internal variables. All benchmarks are masked. P1-P3, in particular, are protected by Boolean masking that was previously verified [14, 35, 83]. The other programs, from [8], are masked multiplication [69], masked S-box [30], masked AES [30] and various masked MAC-Keccak functions [8].

Our experiments were designed to answer three questions: (1) Is our type system effective in detecting HD leaks? (2) Are the domain-specific optimizations effective in reducing the computational overhead? (3) Does the mitigated code have good performance after compilation, in terms of both the code size and the execution speed?

In all the experiments, we used a computer with 2.9 GHz CPU and 8GB RAM, and set the timeout (T/O) to 120 minutes.

7.1 Leak Detection Results

Table 3 shows the results, where Columns 1-2 show the benchmark name and detection time and Columns 3-4 show the number of HD leaks detected. The leaks are further divided into $\texttt{SEN\_HD}_D$ (two-instruction) and $\texttt{SEN\_HD}_S$ (single-instruction). Columns 5-7 show more details of the type inference, including the number of RUD, SID and UKD variables, respectively. While the time taken to complete type inference is not negligible, e.g., minutes for the larger programs, it is reasonable because we perform a much deeper program analysis than mere compilation. To put it into perspective, the heavy-weight formal verification approaches often take hours [35, 83].

As for the number of leaks detected, although the benchmark programs are all masked, during normal compilation, new HD leaks were still introduced as a result of register reuse. For example, in P20, which is a masked AES [8], we detected 33 $\texttt{SEN\_HD}_S$ leaks after analyzing more than 60K intermediate variables. Overall, we detected HD leaks in 17 out of the 20 programs. Furthermore, 6 of

Table 3: Results of type-based HD leak detection.

Name	Detection Time	HD Leaks Detected		Details of the Inferred Types			UKD[35]
		SEN_HD_D	SEN_HD_S	RUD	SID	UKD	
P1	0.061s	NONE	NONE	22	0	4	4
P2	0.105s	NONE	NONE	20	0	7	6
P3	0.099s	NONE	2	31	3	1	1
P4	0.208s	NONE	2	31	12	6	5
P5	0.216s	NONE	2	29	10	1	1
P6	0.276s	4	2	48	15	1	1
P7	0.213s	10	2	151	25	2	2
P8	0.147s	12	2	249	42	4	4
P9	0.266s	6	2	153	61	2	2
P10	0.550s	NONE	NONE	2,334	12	31	-*
P11	0.447s	4	16	2,334	0	31	-
P12	0.619s	NONE	7	2,062	300	52	-
P13	1.102s	NONE	5	4,030	600	49	-
P14	1.998s	NONE	5	5,995	900	49	-
P15	16.999s	NONE	25	13,861	2,100	49	-
P16	24.801s	NONE	5	21,723	3,300	49	-
P17	59.120s	NONE	5	29,587	4,500	49	-
P18	2m1.540s	NONE	4	37,449	5,700	47	-
P19	3m22.415s	NONE	5	47,280	7,200	49	-
P20	16m12.320s	29	33	38,070	26,330	127	-

-*Model counting can not finish on P10-P20 due to the limited scalability

these 17 programs have both SEN_HD_D and SEN_HD_S leaks, while the remaining 11 have only SEN_HD_S leaks.

Results in Columns 5-7 of Table 3 indicate the inferred types of program variables. Despite the large number of variables in a program, our type inference method does a good job in proving that the vast majority of them are RUD or SID (no leak); even for the few UKD variables, after the backend information is used, the number of actual HD leaks detected by our method is small. The last column of Table 3 shows the UKD variables detected by model counting [35, 83]. In comparison, our type system reports only 5% false postives (i.e., our inference rules are conservative).

7.2 Effectiveness of Optimizations

To quantify the impact of our optimizations, we measured the performance of our method with and without them. Table 4 shows the significant differences in analysis time (Columns 2-3) and detected HD leaks (Columns 4-7). Overall, the optimized version completed all benchmarks whereas the unoptimized only completed half. For P12, in particular, the optimized version was 11,631X faster because the unoptimized version ran out of memory and started using virtual memory, which resulted in the slow-down.

Leveraging the backend information also drastically reduced the number of detected leaks. This is because, otherwise, we have to be conservative and assume any two variables may share a register, which results in many false leaks in x86. In P12, for example, using the backend information resulted in 260X fewer leaks.

7.3 Leak Mitigation Results

We compared the size and execution speed of the LLVM compiled code, with and without our mitigation. The results are shown in Table 5, including the number of bytes in the assembly code and the execution time. Columns 8-9 show more details: the number of virtual registers marked as sensitive and non-sensitive, respectively.

The results show that our mitigation has little performance overhead. First, the code sizes are almost the same. For P8, the mitigated code is even smaller because, while switching the storage from register to memory during our handling of the SEN_HD_S pairs, subsequent memory stores may be avoided. Second, the execution speeds are also similar. Overall, the mitigated code is 8%-11% slower, but

Table 4: Results of quantifying impact of optimizations.

Name	Detection Time		Without Backend-Info		With Backend-Info	
	w/o optimization	w/ optimization	SEN_HD_D	SEN_HD_S	SEN_HD_D	SEN_HD_S
P1	0.865s	0.061s	0	18	0	0
P2	0.782s	0.105s	0	9	0	0
P3	0.721s	0.099s	0	15	0	2
P4	1.102s	0.208s	0	32	0	2
P5	1.206s	0.216s	0	32	0	2
P6	1.113s	0.276s	8	40	4	2
P7	5.832s	0.213s	44	144	10	2
P8	4.306s	0.147s	68	323	12	2
P9	5.053s	0.266s	43	160	6	2
P10	10m1.513s	0.550s	12	180	0	0
P11	15m51.969s	0.447s	12	180	4	16
P12	T/O	0.619s	473	1,820	0	7
P13	T/O	1.102s	492	1,884	0	5
P14	T/O	1.998s	492	1,884	0	5
P15	T/O	16.999s	492	1,884	0	25
P16	T/O	24.801s	492	1,884	0	5
P17	T/O	59.120s	492	1,884	0	5
P18	T/O	2m1s	468	1,800	0	4
P19	T/O	3m22s	492	1,884	0	5
P20	T/O	16m13s	620	1,944	29	33

Table 5: Results of our HD leak mitigation.

Name	Code-size Overhead (byte)			Runtime Overhead (us)			Virtual Register	
	original	mitigated	%	original	mitigated	%	sensitive	non-sensitive
P3	858	855	0.3	-	-	-	2	4
P4	1,198	1,174	2	0.23	0.20	-13	2	13
P5	1,132	1,108	2.12	0.30	0.37	2.3	2	9
P6	1,346	1,339	0.52	0.30	0.27	-10	5	8
P7	3,277	3,223	1.64	0.29	0.30	3.4	10	27
P8	3,295	3,267	0.85	0.20	0.22	10	11	83
P9	3,725	3,699	0.69	0.7	0.78	11	10	29
P11	44,829	44,735	0.21	5.60	6.00	7.1	18	680
P12	46,805	46,787	0.03	6.20	6.50	4.83	7	726
P13	90,417	90,288	0.14	13.60	13.00	-4.41	5	1,384
P14	134,060	133,931	0.09	23.00	21.00	-8.69	5	2,040
P15	313,454	312,930	0.16	52.00	58.00	11.5	25	4,637
P16	496,087	495,943	0.03	91.00	96.00	5.49	5	7,288
P17	677,594	677,450	0.02	129.00	136.00	5.42	5	9,912
P18	859,150	859,070	0.009	178.00	183.00	2.80	4	12,537
P19	1,086,041	1,085,897	0.047	237.000	250.000	5.48	5	15,816
P20	957,372	957,319	0.005	228.600	248.300	8.75	56	9,035

in some cases, e.g., P4 and P6, the mitigated code is faster because of our memory related rewriting.

The main reason why our mitigation has little performance overhead is because, as shown in the last two columns of Table 5, compared to the total number of virtual registers, the number of sensitive ones is extremely small. P17 (keccakf_15turn), for example, has only 5 sensitive virtual registers out of the 9,917 in total. Thus, our mitigation only has to modify a small percentage of the instructions, which does not lead to significant overhead.

7.4 Comparison to High-Order Masking

On the surface, HD leaks seem to be a type of second-order leaks, which involves two values. For people familiar with high-order masking [8], a natural question is whether the HD leaks can be mitigated using high-order masking techniques. To answer the question, we conducted two experiments. First, we checked if HD leaks exist in programs equipped with high-order masking. Second, we compared the size and execution speed of the code protected by either high-order masking or our mitigation.

Table 6 shows the results on P4-P9, which come from [8] and have versions protected by d-order masking, where $d = 2$ to 5. While initially we also expected to see no HD leaks in these versions, the results surprised us. As shown in the last two columns, HD leaks were detected in all these high-order masking protected programs. A closer look shows that these leaks are all of the SEN_HD_S type, meaning they are due to restriction of the x86 ISA: any binary

Table 6: Comparison with order-d masking techniques [8].

Name	Code size (byte)	Run time (us)	HW-leak	HD-leak	SEN_HD$_D$	SEN_HD$_S$
P4 (ours)	1,171	0.20	No	No	NONE	NONE
P4 (d=2)	2,207	0.75	No	Yes	NONE	2
P4 (d=3)	4,009	0.28	No	Yes	NONE	2
P4 (d=4)	5,578	0.75	No	Yes	NONE	2
P4 (d=5)	7,950	1.00	No	Yes	NONE	2
P5 (ours)	1,108	0.37	No	No	NONE	NONE
P5 (d=2)	2,074	0.70	No	Yes	NONE	2
P5 (d=3)	3,733	0.60	No	Yes	NONE	2
P5 (d=4)	5,120	0.75	No	Yes	NONE	2
P5 (d=5)	7,197	0.67	No	Yes	NONE	2
P6 (ours)	1,339	0.27	No	No	NONE	NONE
P6 (d=2)	3,404	0.83	No	Yes	NONE	2
P6 (d=3)	6,089	0.57	No	Yes	NONE	2
P6 (d=4)	9,640	0.80	No	Yes	NONE	2
P6 (d=5)	14,092	1.60	No	Yes	NONE	2
P7 (ours)	3,223	0.30	No	No	NONE	NONE
P7 (d=2)	8,456	1.41	No	Yes	NONE	2
P7 (d=3)	15,881	3.20	No	Yes	NONE	2
P7 (d=4)	25,521	4.20	No	Yes	NONE	2
P7 (d=5)	37,578	7.80	No	Yes	NONE	2
P8 (ours)	3,267	0.25	No	No	NONE	NONE
P8 (d=2)	8,782	1.30	No	Yes	NONE	2
P8 (d=3)	16,420	2.00	No	Yes	NONE	2
P8 (d=4)	26,431	4.00	No	Yes	NONE	2
P8 (d=5)	38,996	8.00	No	Yes	NONE	2
P9 (ours)	3,699	0.45	No	No	NONE	NONE
P9 (d=2)	9,258	1.15	No	Yes	NONE	2
P9 (d=3)	17,565	3.00	No	Yes	NONE	2
P9 (d=4)	28,189	5.11	No	Yes	NONE	2
P9 (d=5)	41,383	8.40	No	Yes	NONE	2

operation has to store the result and one of the operands in the same place, and by default, that place is a general-purpose register.

Measured by the code size and speed, our method is more efficient. In P9, for example, our mitigated code has 3K bytes in size and runs in 0.45us, whereas the high-order masking protected code has 9K to 41K bytes (for $d = 2$ to 5) and runs in 1.15us to 8.40us.

7.5 Threat to Validity

We rely on the HW/HD models [52, 53] and thus our results are valid only when these models are valid. We assume the attacker can only measure the power consumption but not other information such as data-bus or timing. If such information becomes available, our mitigation may no longer be secure. Since we focus on cryptographic software, which has simple program structure and language constructs, there is no need for more sophisticated analysis than what is already available in LLVM. Our analysis is intra-procedural: for cryptographic benchmarks, we can actually inline all functions before conducting the analysis. Nevertheless, some of these issues need to be addressed to broaden the scope of our tool.

8 RELATED WORK

Existing methods for detecting power side channels fall into three categories: static analysis, formal verification, and hybrid approach. Static analysis relies on compile-time information to check if masking is implemented correctly [8, 9, 14, 16, 61]. They are faster than formal verification, which often relies on model counting [35–37]. However, formal verification is more accurate than static analysis. The hybrid approach [83] aims to combine the two types of techniques to obtain the best of both worlds. However, none of these methods focused on the leaks caused by register reuse inside a compiler, which is our main contribution.

Specifically, although our type based method for detecting side-channel leaks is inspired by several prior works [8, 16, 61, 83], it is significantly different from theirs. For example, the most recent method, proposed by Zhang et al. [83], interleaves type inference with a model-counting procedure, with the goal of detecting HW leaks caused by errors in masking implementations; however, it does not detect HD leaks caused by register reuse nor remove these leaks, and does not use Datalog or any of the domain-specific optimizations we have proposed.

Barthe et al. [8] proposed a relational analysis technique to check the correctness of high-order masking. When applied to a pair of variables, however, it has to consider all possible ways in which second-order leaks may occur, as opposed to the specific type involved in register reuse. Thus, mitigation has to be more expensive in terms of the code size and the execution speed. Furthermore, as we have shown in experiments, it is not effective in preventing leaks caused by register reuse.

Another difference between our method and existing methods is our focus on analyzing the word-level representation of a program, as opposed to a bit-level representation. While turning a program into a purely Boolean, circuit-like, representation is feasible [3, 15, 35, 83], it does not fit into the standard flow of compilers. As such, implementing the approach in compilers is not straightforward.

The practical security against side-channel leakages via masking can be evaluated using the ISW model [45] and subsequent extensions [6, 29] with transitions. However, they do not consider leaks that are specific to register use in modern compilers. They do not consider constraints imposed by the instruction set architecture either. Furthermore, they need to double the masking order [6] to deal with leaks with transitions, but still do not prevent leaks introduced by compilation.

It is known that security guarantees of software countermeasures may become invalid after compilation [11, 40, 54, 62]. In this context, Barthe et al. [11] showed that the compilation process could maintain the constant-time property for timing side-channel leaks, while our work addresses potential leaks through power side channels. Marc [40] also investigated potential vulnerabilities in power side-channel countermeasures during compiler optimizations, but did not provide a systematic method for mitigating them.

Beyond power side channels, there are techniques for analyzing other side channels using logical reasoning [5, 26, 72, 74], abstract interpretation [12, 32, 76, 80, 81], symbolic execution [7, 21, 41, 51, 63, 64] and dynamic analysis [60, 77]. As for mitigation, there are techniques based on compilers [1, 13, 59, 80] or program synthesis tools [19, 34, 75]. However, these techniques focus on side-channel leaks in the input program. None of them focuses on leaks introduced by register reuse during the compilation.

9 CONCLUSIONS

We have presented a method for mitigating power side-channel leaks caused by register reuse. The method relies on type inference to detect leaks, and leverages the type information to constrain the compiler's backend to guarantee that register allocation is secure. We have implemented the method in LLVM for x86 and evaluated it on cryptographic software. Our experiments demonstrate that the method is effective in mitigating leaks and the mitigated program has low runtime overhead. Specifically, it outperforms state-of-the-art high-order masking techniques in terms of both the code size and the execution speed.

ACKNOWLEDGMENTS

This work was partially funded by the U.S. National Science Foundation (NSF) under grants CNS-1617203 and CNS-1702824 and Office of Naval Research (ONR) under the grant N00014-17-1-2896.

REFERENCES

[1] Giovanni Agosta, Alessandro Barenghi, and Gerardo Pelosi. 2012. A code morphing methodology to automate power analysis countermeasures. In *Proceedings of the The 49th Annual Design Automation Conference 2012 (DAC)*. 77–82.

[2] Mehdi-Laurent Akkar and Louis Goubin. 2003. A generic protection against high-order differential power analysis. In *International Workshop on Fast Software Encryption*. 192–205.

[3] José Bacelar Almeida, Manuel Barbosa, Jorge Sousa Pinto, and Bárbara Vieira. 2013. Formal verification of side-channel countermeasures using self-composition. (2013).

[4] Nikolaos Athanasios Anagnostopoulos, Stefan Katzenbeisser, John A. Chandy, and Fatemeh Tehranipoor. 2018. An overview of DRAM-based security primitives. *Cryptography* 2, 2 (2018), 7.

[5] Timos Antonopoulos, Paul Gazzillo, Michael Hicks, Eric Koskinen, Tachio Terauchi, and Shiyi Wei. 2017. Decomposition instead of self-composition for proving the absence of timing channels. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. 362–375.

[6] Josep Balasch, Benedikt Gierlichs, Vincent Grosso, Oscar Reparaz, and François-Xavier Standaert. 2014. On the cost of lazy engineering for masked software implementations. In *International Conference on Smart Card Research and Advanced Applications*. Springer, 64–81.

[7] Lucas Bang, Abdulbaki Aydin, Quoc-Sang Phan, Corina S. Pasareanu, and Tevfik Bultan. 2016. String analysis for side channels with segmented oracles. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*. 193–204.

[8] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, and Pierre-Yves Strub. 2015. Verified proofs of higher-order masking. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. 457–485.

[9] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, Pierre-Yves Strub, and Rébecca Zucchini. 2016. Strong non-interference and type-directed higher-order masking. In *ACM SIGSAC Conference on Computer and Communications Security*. 116–129.

[10] Gilles Barthe, François Dupressoir, Sebastian Faust, Benjamin Grégoire, François-Xavier Standaert, and Pierre-Yves Strub. 2017. Parallel implementations of masking schemes and the bounded moment leakage model. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. 535–566.

[11] Gilles Barthe, Benjamin Grégoire, and Vincent Laporte. 2018. Secure compilation of side-channel countermeasures: the case of cryptographic âĂIJconstant-timeâĂİ. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. IEEE, 328–343.

[12] Gilles Barthe, Boris Köpf, Laurent Mauborgne, and Martín Ochoa. 2014. Leakage Resilience against Concurrent Cache Attacks. In *International Conference on Principles of Security and Trust*. 140–158.

[13] Ali Galip Bayrak, Francesco Regazzoni, Philip Brisk, François-Xavier Standaert, and Paolo Ienne. 2011. A first step towards automatic application of power analysis countermeasures. In *ACM/IEEE Design Automation Conference*. 230–235.

[14] Ali Galip Bayrak, Francesco Regazzoni, David Novo, and Paolo Ienne. 2013. Sleuth: Automated verification of software power analysis countermeasures. In *International Workshop on Cryptographic Hardware and Embedded Systems*. 293–310.

[15] Swarup Bhunia, Michael S Hsiao, Mainak Banga, and Seetharam Narasimhan. 2014. Hardware Trojan attacks: threat analysis and countermeasures. *Proc. IEEE* 102, 8 (2014), 1229–1247.

[16] Elia Bisi, Filippo Melzani, and Vittorio Zaccaria. 2017. Symbolic analysis of higher-order side channel countermeasures. *IEEE Trans. Computers* 66, 6 (2017), 1099–1105.

[17] Roderick Bloem, Hannes Gross, Rinat Iusupov, Bettina Könighofer, Stefan Mangard, and Johannes Winter. 2018. Formal verification of masked hardware implementations in the presence of glitches. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 321–353.

[18] Johannes Blömer, Jorge Guajardo, and Volker Krummel. 2004. Provably secure masking of AES. In *International workshop on selected areas in cryptography*. Springer, 69–83.

[19] Arthur Blot, Masaki Yamamoto, and Tachio Terauchi. 2017. Compositional Synthesis of Leakage Resilient Programs. In *International Conference on Principles of Security and Trust*. 277–297.

[20] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly declarative specification of sophisticated points-to analyses. *ACM SIGPLAN Notices* 44, 10 (2009), 243–262.

[21] Tegan Brennan, Seemanta Saha, and Tevfik Bultan. 2018. Symbolic path cost analysis for side-channel detection. In *International Conference on Software Engineering*. 424–425.

[22] Eric Brier, Christophe Clavier, and Francis Olivier. 2004. Correlation power analysis with a leakage model. In *International workshop on cryptographic hardware and embedded systems*. 16–29.

[23] David Canright and Lejla Batina. 2008. A very compact "perfectly masked" S-box for AES. In *International Conference on Applied Cryptography and Network Security*. Springer, 446–459.

[24] Gregory Chaitin. 2004. Register allocation and spilling via graph coloring. *ACM SIGPLAN notices* 39, 4 (2004), 66–74.

[25] Suresh Chari, Charanjit S Jutla, Josyula R Rao, and Pankaj Rohatgi. 1999. Towards sound approaches to counteract power-analysis attacks. In *Annual International Cryptology Conference*. 398–412.

[26] Jia Chen, Yu Feng, and Isil Dillig. 2017. Precise detection of side-channel vulnerabilities using quantitative cartesian hoare logic. In *ACM SIGSAC Conference on Computer and Communications Security*. 875–890.

[27] Christophe Clavier, Jean-Sébastien Coron, and Nora Dabbous. 2000. Differential power analysis in the presence of hardware countermeasures. In *International Workshop on Cryptographic Hardware and Embedded Systems*. 252–263.

[28] Keith D. Cooper and Anshuman Dasgupta. 2006. Tailoring graph-coloring register allocation for runtime compilation. In *Fourth IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2006), 26-29 March 2006, New York, New York, USA*. 39–49.

[29] Jean-Sébastien Coron, Christophe Giraud, Emmanuel Prouff, Soline Renner, Matthieu Rivain, and Praveen Kumar Vadnala. 2012. Conversion of security proofs from one leakage model to another: A new issue. In *International Workshop on Constructive Side-Channel Analysis and Secure Design*. Springer, 69–81.

[30] Jean-Sébastien Coron, Emmanuel Prouff, Matthieu Rivain, and Thomas Roche. 2013. Higher-order side channel security and mask refreshing. In *International Workshop on Fast Software Encryption*. 410–424.

[31] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: an efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer-Verlag, Berlin, Heidelberg, 337–340.

[32] Goran Doychev, Dominik Feld, Boris Köpf, Laurent Mauborgne, and Jan Reineke. 2013. CacheAudit: A tool for the static analysis of cache side channels. In *Proceedings of the 22th USENIX Security Symposium*. 431–446.

[33] Alexandre Duc, Sebastian Faust, and François-Xavier Standaert. 2015. Making masking security proofs concrete. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. 401–429.

[34] Hassan Eldib and Chao Wang. 2014. Synthesis of masking countermeasures against side channel attacks. In *International Conference on Computer Aided Verification*. 114–130.

[35] Hassan Eldib, Chao Wang, and Patrick Schaumont. 2014. Formal verification of software countermeasures against side-channel attacks. *ACM Transactions on Software Engineering and Methodology* 24, 2 (2014), 11.

[36] Hassan Eldib, Chao Wang, Mostafa Taha, and Patrick Schaumont. 2014. QMS: Evaluating the side-channel resistance of masked software from source code. In *ACM/IEEE Design Automation Conference*. 1–6.

[37] Pengfei Gao, Hongyi Xie, Jun Zhang, Fu Song, and Taolue Chen. 2019. Quantitative Verification of Masked Arithmetic Programs Against Side-Channel Attacks. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*. 155–173.

[38] Lal George and Andrew W. Appel. 1996. Iterated register coalescing. In *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*. 208–218.

[39] Louis Goubin. 2001. A sound method for switching between boolean and arithmetic masking. In *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 3–15.

[40] Marc Gourjon. 2019. Towards Secure Compilation of Power Side-Channel Countermeasures.

[41] Shengjian Guo, Meng Wu, and Chao Wang. 2018. Adversarial symbolic execution for detecting concurrency-related cache timing leaks. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 377–388.

[42] Krystof Hoder, Nikolaj Bjørner, and Leonardo de Moura. 2011. muZ - An efficient engine for fixed points with constraints. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings (Lecture Notes in Computer Science)*, Vol. 6806. 457–462.

[43] Shourong Hou, Yujie Zhou, Hongming Liu, and Nianhao Zhu. 2017. Improved DPA attack on rotating S-boxes masking scheme. In *Communication Software and Networks (ICCSN), 2017 IEEE 9th International Conference on*. 1111–1116.

[44] Ralf Hund, Carsten Willems, and Thorsten Holz. 2013. Practical timing side channel attacks against kernel space ASLR. In *IEEE Symposium on Security and Privacy*. 191–205.

[45] Yuval Ishai, Amit Sahai, and David A. Wagner. 2003. Private Circuits: Securing Hardware against Probing Attacks. In *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*. 463–481.

[46] Paul Kocher, Joshua Jaffe, and Benjamin Jun. 1999. Differential power analysis. In *Annual International Cryptology Conference*. 388–397.

[47] Paul C Kocher. 1996. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Annual International Cryptology Conference*. 104–113.

[48] Monica S Lam, John Whaley, V Benjamin Livshits, Michael C Martin, Dzintars Avots, Michael Carbin, and Christopher Unkel. 2005. Context-sensitive program

analysis as database queries. In *Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM, 1–12.

[49] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization*. 75.

[50] Abhranil Maiti and Patrick Schaumont. 2011. Improved Ring Oscillator PUF: An FPGA-friendly Secure Primitive. *J. Cryptology* 24, 2 (2011), 375–397.

[51] Pasquale Malacaria, MHR Khouzani, Corina S Pasareanu, Quoc-Sang Phan, and Kasper Luckow. 2018. Symbolic side-channel analysis for probabilistic programs. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. IEEE, 313–327.

[52] Stefan Mangard. 2002. A simple power-analysis (SPA) attack on implementations of the AES key expansion. In *International Conference on Information Security and Cryptology*. Springer, 343–358.

[53] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. 2007. *Power analysis attacks - revealing the secrets of smart cards*.

[54] David McCann, Carolyn Whitnall, and Elisabeth Oswald. 2016. ELMO: Emulating Leaks for the ARM Cortex-M0 without Access to a Side Channel Lab. *IACR Cryptology ePrint Archive* 2016 (2016), 517.

[55] Thomas S Messerges. 2000. Using second-order power analysis to attack DPA resistant software. In *International Workshop on Cryptographic Hardware and Embedded Systems*. 238–251.

[56] Thomas S Messerges, Ezzy A Dabbish, and Robert H Sloan. 1999. Investigations of power analysis attacks on smartcards. *Smartcard* 99 (1999), 151–161.

[57] Thomas S Messerges, Ezzat A Dabbish, and Robert H Sloan. 2002. Examining smart-card security under the threat of power analysis attacks. *IEEE transactions on computers* 51, 5 (2002), 541–552.

[58] Amir Moradi. 2014. Side-channel leakage through static power. In *International Workshop on Cryptographic Hardware and Embedded Systems*. 562–579.

[59] Andrew Moss, Elisabeth Oswald, Dan Page, and Michael Tunstall. 2012. Compiler assisted masking. In *International Conference on Cryptographic Hardware and Embedded Systems*. 58–75.

[60] Shirin Nilizadeh, Yannic Noller, and Corina S. Pasareanu. 2019. DiffFuzz: differential fuzzing for side-channel analysis. In *International Conference on Software Engineering*. 176–187.

[61] Inès Ben El Ouahma, Quentin Meunier, Karine Heydemann, and Emmanuelle Encrenaz. 2017. Symbolic approach for Side-Channel resistance analysis of masked assembly codes. In *Security Proofs for Embedded Systems*.

[62] Kostas Papagiannopoulos and Nikita Veshchikov. 2017. Mind the gap: towards secure 1st-order masking in software. In *International Workshop on Constructive Side-Channel Analysis and Secure Design*. Springer, 282–297.

[63] Corina S. Pasareanu, Quoc-Sang Phan, and Pasquale Malacaria. 2016. Multi-run Side-Channel analysis using symbolic execution and Max-SMT. In *IEEE Computer Security Foundations Symposium*. 387–400.

[64] Quoc-Sang Phan, Lucas Bang, Corina S. Pasareanu, Pasquale Malacaria, and Tevfik Bultan. 2017. Synthesis of Adaptive Side-Channel Attacks. In *IEEE Computer Security Foundations Symposium*. 328–342.

[65] Massimiliano Poletto and Vivek Sarkar. 1999. Linear scan register allocation. *ACM Trans. Program. Lang. Syst.* 21, 5 (1999), 895–913.

[66] Emmanuel Prouff and Matthieu Rivain. 2013. Masking against side-channel attacks: A formal security proof. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. 142–159.

[67] Jean-Jacques Quisquater and David Samyde. 2001. Electromagnetic analysis (ema): Measures and counter-measures for smart cards. In *Smart Card Programming and Security*. 200–210.

[68] Oscar Reparaz, Begül Bilgin, Svetla Nikova, Benedikt Gierlichs, and Ingrid Verbauwhede. 2015. Consolidating masking schemes. In *Annual Cryptology Conference*. Springer, 764–783.

[69] Matthieu Rivain and Emmanuel Prouff. 2010. Provably secure higher-order masking of AES. In *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 413–427.

[70] Ulrich Rührmair, Heike Busch, and Stefan Katzenbeisser. 2010. Strong PUFs: models, constructions, and security proofs. In *Towards Hardware-Intrinsic Security - Foundations and Practice*. 79–96.

[71] Kai Schramm and Christof Paar. 2006. Higher order masking of the AES. In *Cryptographers' track at the RSA conference*. Springer, 208–225.

[72] Marcelo Sousa and Isil Dillig. 2016. Cartesian hoare logic for verifying k-safety properties. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. 57–69.

[73] François-Xavier Standaert, Tal G Malkin, and Moti Yung. 2009. A unified framework for the analysis of side-channel key recovery attacks. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. 443–461.

[74] Chungha Sung, Brandon Paulsen, and Chao Wang. 2018. CANAL: A cache timing analysis framework via LLVM transformation. In *IEEE/ACM International Conference On Automated Software Engineering*.

[75] Chao Wang and Patrick Schaumont. 2017. Security by compilation: an automated approach to comprehensive side-channel resistance. *ACM SIGLOG News* 4, 2 (2017), 76–89.

[76] Shuai Wang, Yuyan Bao, Xiao Liu, Pei Wang, Danfeng Zhang, and Dinghao Wu. 2019. Identifying Cache-Based Side Channels through Secret-Augmented Abstract Interpretation. *CoRR* abs/1905.13332 (2019).

[77] Shuai Wang, Pei Wang, Xiao Liu, Danfeng Zhang, and Dinghao Wu. 2017. CacheD: Identifying cache-based timing channels in production software. In *USENIX Security Symposium*. 235–252.

[78] John Whaley, Dzintars Avots, Michael Carbin, and Monica S Lam. 2005. Using datalog with binary decision diagrams for program analysis. In *Asian Symposium on Programming Languages and Systems*. Springer, 97–118.

[79] John Whaley and Monica S Lam. 2004. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *ACM SIGPLAN Notices*, Vol. 39. ACM, 131–144.

[80] Meng Wu, Shengjian Guo, Patrick Schaumont, and Chao Wang. 2018. Eliminating timing side-channel leaks using program repair. In *International Symposium on Software Testing and Analysis*.

[81] Meng Wu and Chao Wang. 2019. Abstract Interpretation under Speculative Execution. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*.

[82] Yuan Yao, Mo Yang, Conor Patrick, Bilgiday Yuce, and Patrick Schaumont. 2018. Fault-assisted side-channel analysis of masked implementations. In *2018 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 57–64.

[83] Jun Zhang, Pengfei Gao, Fu Song, and Chao Wang. 2018. SCInfer: Refinement-based verification of software countermeasures against Side-Channel attacks. In *International Conference on Computer Aided Verification*.

[84] Xin Zhang, Ravi Mangal, Radu Grigore, Mayur Naik, and Hongseok Yang. 2014. On abstraction refinement for program analyses in Datalog. *ACM SIGPLAN Notices* 49, 6 (2014), 239–248.

[85] Yongbin Zhou and Dengguo Feng. 2005. Side-Channel Attacks: Ten years after its publication and the impacts on cryptographic module security testing. *IACR Cryptology ePrint Archive* (2005), 388.