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:


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.

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 de-sensitized 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 (v1, v2) 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 v1 and v2 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

![Figure 1: Overview of our secure compilation method](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 such as high-order masking: our mitigated code not only is more secure and compact but also runs faster than code mitigated by existing techniques. In terms of runtime overhead, our method also is more secure and compact but also runs faster than code mitigated by existing techniques.

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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Approach</th>
<th>HW-Sensitive</th>
<th>HD-Sensitive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xor</td>
<td>No Masking</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SecXor</td>
<td>First Order Masking</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>SecXor2</td>
<td>Specialized Hardware &amp; Masking</td>
<td>×</td>
<td>✓</td>
</tr>
</tbody>
</table>

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.

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 \(mask0@mk=\text{key}\). 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]: \(HD(mask1, mask1@key) = HW(mask1 @ key) = HW(\text{key})\), which reveals key. Consider, for example, where \(key = 0001_2\) and \(mask1 = 1111_2\) in binary. If a register stores \(mask1\) (=1111), first and updates its value as \(mask1@key(=1110)\), the transition of the register (bit-flip) is 0001, 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...
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 isHW-sensitive if, statistically, it depends on the secret. For example, in Figure 2, key is HW-sensitive whereas mk=mask1@key is masked. If there were nk=mask1vkey, it would be HW-sensitive because the masking is not perfect.

Second, we check if there is any pair of values (v1, v2) that, when stored in the same register, may cause an HD leak. That is, HD(v1, v2) = HW(v1 ⊕ v2) 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 \(f_{t1t2}(k, m1, m2, m3):\) 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 ← m1 ⊕ m2, where m1 and m2 are random (RUD), it is easy to see that t1 is also random (RUD). For t3 ← t2, 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.

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@key, it means that either mk and mask1 share a register, which causes a leak in HD(mk,
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 %edx1 and %edx2, we have HD(%edx1, %edx2) = 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.

Figure 3: The assembly code before and after mitigation.

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 $(x, k, r)$ be the input: sets $x$, $k$ and $r$ consist of public, secret, and random (mask) variables, respectively. Let $x, k_1, k_2$, and $r$ be valuations of these input variables. Then, $\sigma_1(P, x, k_1, r)$ denotes, at time step $t$, the power consumption of a device executing $P$ under input $x, k_1$ and $r$. Similarly, $\sigma_1(P, x, k_2, r)$ 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 $\sigma_1(P, x, k_1, r)$ differs from that of $\sigma_1(P, x, k_2, r)$. Let random variables in $r$ be uniformly distributed in the domain $R$, and let the probability of each $r \in R$ be $Pr(r)$, we expect $\forall t, x, k_1, k_2$:

$$\sum_{r \in R} \sigma_1(P, x, k_1, r) Pr(r) = \sum_{r \in R} \sigma_1(P, x, k_2, r) Pr(r)$$

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 $\sigma_1$: if we remove all individual leaks, the leak-free property over the sum $\sigma_1(P, x, k, r)$ 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 $0 \leq i < n$, 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
We use a type system that starts from the input annotation (\(\text{IN}\)). Another interpretation is to regard \(\text{HW}(D @ D')\), the Hamming Weight of the bit-wise XOR of \(D\) and \(D'\). Another interpretation is to regard \(\text{HW}(D)\) as a special case of \(\text{HD}(D, D')\), 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 \(D_{\text{syn}}(f, k)\), if \(k\) appears in the expression of \(f\); that is, \(k\) is in the support of \(f\), denoted \(k \in \text{supp}(f)\).

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 \(f(k, r) = k \land r\), 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 \(D_{\text{stat}}(f, k)\).

The relative strengths of the dependency relations are as follows: 

\[ D_{\text{syn}}(f, k) \implies D_{\text{stat}}(f, k), \text{ i.e., if } f \text{ is syntactically independent of } k, \text{ it is statistically independent of } k. \]

In this work, we rely on \(D_{\text{syn}}\) to infer \(D_{\text{stat}}\) 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 (\(\text{IN}_{\text{PUBLIC}}, \text{IN}_{\text{SECRET}}\); and \(\text{IN}_{\text{RANDOM}}\)) 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 \(\text{TYPE}(v)\), may be one of the following kinds:

- **RUD**, which stands for random uniform distribution, means \(v\) is either a random input \(m \in \text{IN}_{\text{RANDOM}}\) 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 \(\text{IN}_{\text{SECRET}}\).
- **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 \(k \in \text{IN}_{\text{SECRET}}\), \(m \in \text{IN}_{\text{RANDOM}}\) and \(k_m = k \oplus m\), then \(\text{TYPE}(k) = \text{UKD}\) and \(\text{TYPE}(k_m) = \text{RUD}\). If \(x \in \text{IN}_{\text{PUBLIC}}\) and \(x = x_m \oplus k_m\), then \(\text{TYPE}(x) = \text{SID}\) because, although \(x\) may have any distribution, since \(k_m\) is RUD, \(x_m\) is statistically independent of the secret.

We prefer RUD over SID, when both are applicable to a variable \(x_k\), because if \(x_k\) is \(\text{XOR}\)-ed with a UKD variable \(x_2\), we can easily prove that \(x = x_k \oplus x_2\) is RUD using local inference, as long as \(x_k\) is RUD and \(x_2\) is not randomized by the same input variable. However, if \(x_k\) 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 \(\text{IN}_{\text{PUBLIC}}\) (SID), \(\text{IN}_{\text{SECRET}}\) (UKD) and \(\text{IN}_{\text{RANDOM}}\) (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 \(z\), as long as \(m \notin \text{supp}(x)\). 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).

- \(v \gets \text{Uop}(v_1)\), where \(\text{Uop}\) is a unary operator such as the Boolean (or bit-wise) negation.
- \(v \gets \text{Bop}(v_1, v_2)\), where \(\text{Bop}\) is a binary operator such as Boolean (or bit-wise) \(\&\), \(\lor\) and \(\oplus\) (finite-field multiplication).

For \(v \gets \text{Uop}(v_1)\), we have \(\text{TYPE}(v) = \text{TYPE}(v_1)\), meaning \(v\) and \(v_1\) have the same type. For \(v \gets \text{Bop}(v_1, v_2)\), the type depends on (1) if \(\text{Bop}\) is \(\text{Xor}\), (2) if \(\text{TYPE}(v_1)\) and \(\text{TYPE}(v_2)\) 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 \(\text{Bop}\), we define two related functions, \(\text{unq}\) and domain, in addition to \(\text{supp}(v)\), which is the set of input variables upon which \(v\) depends syntactically.

**Definition 4.1.** \(\text{unq} : V \rightarrow \text{IN}_{\text{RANDOM}}\) is a function that returns, for each variable \(v \in V\), a subset of mask variables defined as follows: \(\text{unq}(v) = \{v\}; \text{but if } v \notin \text{IN} \setminus \text{IN}_{\text{RANDOM}}\), \(\text{unq}(v) = \{}\); \(\text{if} v \leftarrow \text{Uop}(v_1), \text{unq}(v) = \text{unq}(v_1)\); and \(\text{if} v \leftarrow \text{Bop}(v_1, v_2), \text{unq}(v) = (\text{unq}(v_1) \cup \text{unq}(v_2)) \setminus (\text{supp}(v_1) \cap \text{supp}(v_2))\).
Given the data-flow graph of all instructions involved in computing $v$ and an input variable $m \in \text{unq}(v)$, 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 $\text{unq}(v)$.

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

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

Given the data-flow graph of all instructions involved in computing $v$ and an input $m \in \text{dom}(v)$, 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 $\text{dom}(v)$.

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

$$
\text{Rule}_{1.1}: \begin{cases}
\text{dom}(v) \neq \emptyset \quad & \text{TYPE}(v) = \mathit{RUD} \\
\text{supp}(v) \cap \mathcal{IN}_{\mathit{SECRET}} \neq \emptyset \land \text{TYPE}(v) \neq \mathit{RUD} \quad & \text{TYPE}(v) = \mathit{SID}
\end{cases}
$$

Here, $\text{Rule}_{1.2}$ says if $v = m \oplus \text{expr}$, where $m$ is a random input and $\text{expr}$ is not masked by $m$, then $v$ has random uniform distribution. This is due to the property of XOR. $\text{Rule}_{2.2}$ says if $v$ is syntactically independent of variables in $\mathcal{IN}_{\mathit{SECRET}}$, 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, $v = (k \oplus m) \land x$ where $k \in \mathcal{IN}_{\mathit{SECRET}}$, $m \in \mathcal{IN}_{\mathit{RANDOM}}$ and $x \in \mathcal{IN}_{\mathit{PUBLIC}}$; 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.

$$
\begin{align*}
\text{Rule}_{1.3a}: & \quad \text{ TYPE}(v) = \mathit{SID} \\
& \quad v \leftarrow \text{Bop}(v_1, v_2) \land \text{supp}(v_1) \cap \text{supp}(v_2) = \emptyset \\
& \quad \text{ TYPE}(v) = \mathit{SID} \\
\text{Rule}_{1.3b}: & \quad \text{ TYPE}(v) = \mathit{SID} \\
& \quad v \leftarrow \text{Bop}(v_1, v_2) \land \text{supp}(v_1) \cap \text{supp}(v_2) = \emptyset \\
& \quad \text{ TYPE}(v) = \mathit{RUD} \\
\end{align*}
$$

These rules mean that, for any Bop \{\land, \lor\}, 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, supp($v_1$) \cap supp($v_2$) = \emptyset is need; otherwise, the common input may cause problem. For example, if v_1 \leftarrow m \circ k and v_2 \leftarrow m \land x, then v = (v_1 \lor v_2) = (m \land k) \land x has a leak because if k = 1, v = 0; but if k = 0, v = m \land x.

$$
\begin{align*}
\text{Rule}_{1.4}: & \quad \text{ TYPE}(v) = \mathit{SID} \\
& \quad v \leftarrow \text{Bop}(v_1, v_2) \land \text{supp}(v_1) \cap \text{supp}(v_2) = \emptyset \\
& \quad \text{ TYPE}(v) = \mathit{SID} \\
\end{align*}
$$

Similarly, $\text{Rule}_{4}$ may elevate a variable $v$ from UKD to SID, e.g., as in $v \leftarrow ((k \circ m) \land x_1) \land (x_2)$ where $x_1$ and $x_2$ are both SID. Again, the condition supp($v_1$) \cap supp($v_2$) = \emptyset is needed because, otherwise, there may be cases such as $v \leftarrow ((k \circ m) \land x_1) \land (x_2 \circ m)$, which is equivalent to $v \leftarrow \neg k \land (m \land x_1 \land x_2)$ 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 $\text{HD}(v_1, v_2)$ results in a leak when $v_1$ and $v_2$ share a register. There are two scenarios:

- $v_1 \leftarrow \text{expr}_1; v_2 \leftarrow \text{expr}_2$, meaning $v_1$ and $v_2$ are defined in two instructions.
- $v_1 \leftarrow \text{Bop}(v_2, v_3)$, where the result $v_1$ and one operand ($v_2$) are stored in the same register.

In the two-instruction case, we check $\text{HW}(\text{expr}_1 \circ \text{expr}_2)$ using Xor-related inference rules. For example, if $v_1 \leftarrow k \circ m$ and $v_2 \leftarrow m$, since $m$ appears in the supports of both expressions, $(k \circ m) \circ m$ is UKD. Such leak will be denoted $\text{SEN}_{\mathit{HD}}(v_1, v_2)$, where $D$ stands for “Duplicate”.

In the single-instruction case, we check $\text{HW}(\text{Bop}(v_2, v_3) \circ v_2)$ based on the operator type. When Bop = $\land$, we have $(v_2 \land v_3) \circ v_2 = v_2 \land v_3$; when Bop = $\lor$, we have $(v_2 \lor v_3) \circ v_2 = \neg v_2 \lor v_3$; when Bop = $\oplus$ (Xor), we have $(v_2 \oplus v_3) \circ v_2 = v_2; v_3$ if $v_2 \neq v_3 \neq 0x01$ and is $(0 \lor 0x01)$ otherwise. Since the type inference procedure is agnostic to the result of $v_2 \circ v_3$, the type of $(v_2 \circ v_3) \circ v_2$ depends on the types of $v_2$ and $v_3$; that is, $\text{TYPE}(v_2) = \mathit{UKD} \lor \mathit{TYPE}(v_3) = \mathit{UKD} \Rightarrow \mathit{TYPE}(v_2 \circ v_3) = \mathit{UKD}$. If there is a leak, it will be denoted $\text{SEN}_{\mathit{HD}}(v_1, v_2)$. The reason why HD leaks are divided to $\text{SEN}_{\mathit{HD}}$ and $\text{SEN}_{\mathit{D}}$ 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 $\mathit{SEN}_{\mathit{D}}$ requires that we rewrite the instruction itself.

We also define a relation $\text{Share}(v_1, v_2)$, 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:

$$
\begin{align*}
\text{Share}(v_1, v_2) \land \text{TYPE}(v_1) = \mathit{UKD} \land v_1 \leftarrow \text{expr}_1; v_2 \leftarrow \text{expr}_2 \\
& \quad \Rightarrow \text{SEN}_{\mathit{HD}}(v_1, v_2) \\
\text{Share}(v_1, v_2) \land \text{TYPE}(v_1) = \mathit{UKD} \land v_1 \leftarrow \text{Bop}(v_2, v_3) \\
& \quad \Rightarrow \text{SEN}_{\mathit{HD}}(v_1, v_2)
\end{align*}
$$

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

### 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.
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\(_{HD}\) Pair

For each SEN\(_{HD}\) pair, 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, 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\(_{HD}\) Pair

For each SEN\(_{HD}\) 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

\[
\text{MOVL} -4(\text{Regb}), \text{Xecx} /-4(\text{Regb}) = m \text{ (random)}
\]
\[
\text{XORL} -8(\text{Regb}), \text{Xecx} /-8(\text{Regb}) = k \text{ (secret)}
\]

Here, \(-4(\text{Regb})\) and \(-8(\text{Regb})\) are memory locations for \( m \) and \( k \), respectively. Although \( m \) and \( k \) are RUD (no leak) when stored in %ecx, the transition from \( m \) to \( mk \), \( HW(m \oplus k) = k \), has a leak.
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 \( \text{Share}(v_1, v_2) \), meaning \( v_1 \) and \( v_2 \) may share a register. After inferring the variable types as RUD, S1D, or UKD, we use \( \text{Share}(v_1, v_2) \) to filter the variable pairs subjected to checking for SEN HD3 and SEN HD5 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 \( \text{supp}(v) \), \( \text{unq}(v) \) and \( \text{dom}(v) \), or performing set operations such as \( m \in \text{supp}(v) \).

In contrast, it is linear time [61] to compute sets such as \( \text{supp}(v) \), \( \text{unq}(v) \) and \( \text{dom}(v) \) 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 \( \text{supp}(v_1) = \{i_1, i_2\} \) and \( \text{supp}(v_2) = \{i_0, i_1, i_3\} \). One way is to encode the sets is using a relation \( \text{Supp} : V \times IN \), where \( V \) are variables and \( IN \) are supporting inputs:

\[
\begin{align*}
\text{Supp}(v_1, i_1) \land \text{Supp}(v_1, i_2) &= \text{supp}(v_1) \\
\text{Supp}(v_2, i_0) \land \text{Supp}(v_2, i_1) \land \text{Supp}(v_2, i_3) &= \text{supp}(v_2)
\end{align*}
\]

While the size of \( \text{Supp} \) 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 \( \text{Supp} : V \times 2^{IN} \), where \( 2|IN| \) is the power-set (set of all subsets of \( IN \)):

\[
\begin{align*}
\text{Supp}(v_1, b_{0110}) &= \text{supp}(v_1) \\
\text{Supp}(v_2, b_{1011}) &= \text{supp}(v_2)
\end{align*}
\]

While the size of \( \text{Supp} \) is \( |V|2^{2|IN|} \), 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 \( \text{Supp} \) increases exponentially.

Therefore, we propose an encoding, called segmented bitset representation (\( \text{idx} \times \text{bitset} \)), where \( \text{idx} \) refers to the \( i \)-th segment and \( \text{bitset} \) denotes the bits in the \( i \)-th segment:

\[
\begin{align*}
\text{Supp}(v_1, 1, b_{01}) \land \text{Supp}(v_1, 0, b_{10}) &= \text{supp}(v_1) \\
\text{Supp}(v_2, 1, b_{10}) \land \text{Supp}(v_2, 0, b_{11}) &= \text{supp}(v_2)
\end{align*}
\]

In practice, when the \( \text{bitset} \) 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 \( \text{bitset} \) decreases to 1 and the number of segments increases to \( |IN| \), it degenerates to the first approach.

When the size of \( \text{bitset} \) increases to \( |IN| \) and the number of segments decrease to 1, it degenerates to the second approach.

7 EXPERIMENTS

We have implemented our method in LLVM 3.6 [49]. We used the \( \mu \)C [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/0) 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 SEN HD2 (two-instruction) and SEN HD5 (single-instruction). Columns 5-7 show more details of the type inference, including the number of RUD, S1D 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 SEN HD2 leaks after analyzing more than 60K intermediate variables. Overall, we detected HD leaks in 17 out of the 20 programs. Furthermore, 6 of
these 17 programs have both SEN_HD2 and SEN_HD3 leaks, while the remaining 11 have only SEN_HD3 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 RD 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_HD3 pairs, subsequent memory stores may be avoided. Second, the execution speeds are also similar. Overall, the mitigated code is 8%-11% slower, but 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 (kccalcd_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 5 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_HD2 type, meaning they are due to restriction of the x86 ISA: any binary
Table 6: Comparison with order-d masking techniques [8].

<table>
<thead>
<tr>
<th>Name</th>
<th>Code size (byte)</th>
<th>Run time (us)</th>
<th>HW-leak</th>
<th>HD-leak</th>
<th>HD-leak</th>
<th>HD-leak</th>
<th>HD-leak</th>
</tr>
</thead>
<tbody>
<tr>
<td>P4 (ours)</td>
<td>1,717</td>
<td>0.20</td>
<td>No</td>
<td>No</td>
<td>NONE</td>
<td>NONE</td>
<td>NONE</td>
</tr>
<tr>
<td>P4 (d=2)</td>
<td>4,009</td>
<td>0.28</td>
<td>No</td>
<td>Yes</td>
<td>NONE</td>
<td>NONE</td>
<td>NONE</td>
</tr>
<tr>
<td>P4 (d=4)</td>
<td>3,578</td>
<td>0.75</td>
<td>No</td>
<td>Yes</td>
<td>NONE</td>
<td>NONE</td>
<td>NONE</td>
</tr>
<tr>
<td>P4 (d=5)</td>
<td>7,926</td>
<td>1.00</td>
<td>No</td>
<td>Yes</td>
<td>NONE</td>
<td>NONE</td>
<td>NONE</td>
</tr>
<tr>
<td>P5 (ours)</td>
<td>2,108</td>
<td>0.37</td>
<td>No</td>
<td>No</td>
<td>NONE</td>
<td>NONE</td>
<td>NONE</td>
</tr>
<tr>
<td>P5 (d=2)</td>
<td>7,274</td>
<td>0.79</td>
<td>No</td>
<td>Yes</td>
<td>NONE</td>
<td>NONE</td>
<td>NONE</td>
</tr>
<tr>
<td>P5 (d=3)</td>
<td>3,733</td>
<td>0.60</td>
<td>No</td>
<td>Yes</td>
<td>NONE</td>
<td>NONE</td>
<td>NONE</td>
</tr>
<tr>
<td>P5 (d=4)</td>
<td>5,120</td>
<td>0.75</td>
<td>No</td>
<td>Yes</td>
<td>NONE</td>
<td>NONE</td>
<td>NONE</td>
</tr>
<tr>
<td>P5 (d=5)</td>
<td>7,197</td>
<td>0.67</td>
<td>No</td>
<td>Yes</td>
<td>NONE</td>
<td>NONE</td>
<td>NONE</td>
</tr>
<tr>
<td>P6 (ours)</td>
<td>1,389</td>
<td>0.27</td>
<td>No</td>
<td>No</td>
<td>NONE</td>
<td>NONE</td>
<td>NONE</td>
</tr>
<tr>
<td>P6 (d=2)</td>
<td>3,404</td>
<td>0.83</td>
<td>No</td>
<td>No</td>
<td>NONE</td>
<td>NONE</td>
<td>NONE</td>
</tr>
<tr>
<td>P6 (d=3)</td>
<td>6,089</td>
<td>0.57</td>
<td>No</td>
<td>No</td>
<td>NONE</td>
<td>NONE</td>
<td>NONE</td>
</tr>
<tr>
<td>P6 (d=4)</td>
<td>9,640</td>
<td>0.80</td>
<td>No</td>
<td>No</td>
<td>NONE</td>
<td>NONE</td>
<td>NONE</td>
</tr>
<tr>
<td>P6 (d=5)</td>
<td>14,092</td>
<td>1.60</td>
<td>No</td>
<td>No</td>
<td>NONE</td>
<td>NONE</td>
<td>NONE</td>
</tr>
<tr>
<td>P7 (ours)</td>
<td>3,223</td>
<td>0.30</td>
<td>No</td>
<td>No</td>
<td>NONE</td>
<td>NONE</td>
<td>NONE</td>
</tr>
<tr>
<td>P7 (d=2)</td>
<td>8,456</td>
<td>1.41</td>
<td>No</td>
<td>Yes</td>
<td>NONE</td>
<td>NONE</td>
<td>NONE</td>
</tr>
<tr>
<td>P7 (d=3)</td>
<td>15,831</td>
<td>3.20</td>
<td>No</td>
<td>Yes</td>
<td>NONE</td>
<td>NONE</td>
<td>NONE</td>
</tr>
<tr>
<td>P7 (d=4)</td>
<td>25,521</td>
<td>4.20</td>
<td>No</td>
<td>Yes</td>
<td>NONE</td>
<td>NONE</td>
<td>NONE</td>
</tr>
<tr>
<td>P7 (d=5)</td>
<td>37,378</td>
<td>7.80</td>
<td>No</td>
<td>Yes</td>
<td>NONE</td>
<td>NONE</td>
<td>NONE</td>
</tr>
<tr>
<td>P8 (ours)</td>
<td>3,267</td>
<td>0.30</td>
<td>No</td>
<td>No</td>
<td>NONE</td>
<td>NONE</td>
<td>NONE</td>
</tr>
<tr>
<td>P8 (d=2)</td>
<td>8,742</td>
<td>1.40</td>
<td>No</td>
<td>Yes</td>
<td>NONE</td>
<td>NONE</td>
<td>NONE</td>
</tr>
<tr>
<td>P8 (d=3)</td>
<td>16,420</td>
<td>2.00</td>
<td>No</td>
<td>Yes</td>
<td>NONE</td>
<td>NONE</td>
<td>NONE</td>
</tr>
<tr>
<td>P8 (d=4)</td>
<td>28,450</td>
<td>4.00</td>
<td>No</td>
<td>Yes</td>
<td>NONE</td>
<td>NONE</td>
<td>NONE</td>
</tr>
<tr>
<td>P8 (d=5)</td>
<td>38,996</td>
<td>8.00</td>
<td>No</td>
<td>Yes</td>
<td>NONE</td>
<td>NONE</td>
<td>NONE</td>
</tr>
<tr>
<td>P9 (ours)</td>
<td>5,999</td>
<td>0.45</td>
<td>No</td>
<td>No</td>
<td>NONE</td>
<td>NONE</td>
<td>NONE</td>
</tr>
<tr>
<td>P9 (d=2)</td>
<td>7,258</td>
<td>1.15</td>
<td>No</td>
<td>Yes</td>
<td>NONE</td>
<td>NONE</td>
<td>NONE</td>
</tr>
<tr>
<td>P9 (d=3)</td>
<td>17,565</td>
<td>3.00</td>
<td>No</td>
<td>Yes</td>
<td>NONE</td>
<td>NONE</td>
<td>NONE</td>
</tr>
<tr>
<td>P9 (d=4)</td>
<td>28,189</td>
<td>5.11</td>
<td>No</td>
<td>Yes</td>
<td>NONE</td>
<td>NONE</td>
<td>NONE</td>
</tr>
<tr>
<td>P9 (d=5)</td>
<td>41,383</td>
<td>8.40</td>
<td>No</td>
<td>Yes</td>
<td>NONE</td>
<td>NONE</td>
<td>NONE</td>
</tr>
</tbody>
</table>

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 [55–57]. 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.


