Abacus: Precise Side-Channel Analysis

Anonymous

***Abstract*—Side-channel attacks allow adversaries to infer sensitive information based on non-functional characteristics. Existing works on side-channel detections identify numerous potential vulnerabilities. However, in practice, many such vulnerabilities leak a negligible amount of sensitive information, and thus developers are often reluctant to address them. However, no existing tools can precisely report the number of leaked bits for each leakage site in production systems.**

We propose a new program analysis method to precisely quantify the leaked information in a single-trace attack through side-channels to overcome this limitation. It can identify covert information flows in programs that can expose confidential information and reason about security flaws that would otherwise be difficult, if not impossible, for a developer to find. We model an attacker's observation of each leakage site as a constraint. We scale symbolic execution to production software to generate the constraints and then run Monte Carlo sampling to estimate the number of leaked bits for each leakage site. By using the Central Limit Theorem, we can also give the error bound for estimation.

We have implemented the above technique in a tool called Abacus, which can not only find the side-channel vulnerabilities but also estimate how many bits are leaked. Abacus outperforms existing dynamic side-channel detection tools in terms of performance and accuracy. Also, Abacus can report a very fine-grained vulnerability leakage information. We evaluate Abacus on OpenSSL, mbedTLS, Libgcrypt, and Monocypher. Our results show that most of the reported vulnerabilities are hard to exploit in practice, which can dramatically improve the process of vulnerability report triage for the developers. We also find several sensitive vulnerabilities that are missed by the existing tools. We confirm those vulnerabilities with manual checks and by the developers.

I. Introduction

Side channels are inevitable in modern computer systems as the sensitive information may be leaked through many kinds of inadvertent behaviors, such as power, electromagnetic radiation, and even sound [1]–[5]. Among them, software-based side-channel attacks, such as cache attacks, memory page attacks, and controlled-channel attacks, are especially common and have been studied for years [6]–[11]. These attacks stem from vulnerable software and shared hardware components. By observing those inadvertent behaviors, attackers can infer program execution flows that manipulate secrets and guess secrets such as encryption keys [12]–[15].

Regarding the root cause of side-channel attacks, many of them originate from two code patterns: data flow from secrets to load addresses and data flow from secrets to branch conditions. We refer to them as secret-dependent memory-accesses and control-flows, respectively. Recent works [14], [16]–[20] can detect plenty of side-channel vulnerabilities. For example, DATA [16] reports 2,248 potential leakage sites for the RSA implementation in OpenSSL. After some inspections, 1,510 leakages are dismissed. But it still leaves 460 data-access leakages and 278 control-flow leakages. However, many of the reported vulnerabilities are not fixed by the developers. First, some vulnerable implementations have better performance. For example, RSA implementations usually adopt the CRT optimization, which is faster but vulnerable to fault attacks [21]. Second, fixing old vulnerabilities can introduce new weaknesses, let alone the majority of them are negligible. That is, some vulnerabilities can result in the key being entirely compromised [21], [22], but many other vulnerabilities are proven to be less severe in reality. Therefore, we need a proper quantification metric to assess the sensitive level of side-channel vulnerabilities, so the developers can efficiently triage the reported vulnerabilities.

Previous work like static methods [20], [23], usually with abstract interpretations, can give a leakage upper bound, which is useful to justify the implementation is secure if they report zero leakage. However, they cannot indicate how severe the leakage is because of the over-approximation method they apply. For example, CacheAudit [20] reports that the upper bound leakage of AES-128 exceeds the original key size. The dynamic methods take another approach with a concrete input and run the program in a real environment. Although they are very precise in terms of true leakages, no existing tool can precisely assess the severity of production software vulnerabilities.

To overcome these limitations, we propose a novel method to quantify information leakages precisely. We quantify the number of leaked bits during one real execution and define the amount of leaked information as the cardinality of possible secrets based on an attacker's observation. Before an attack, an adversary has a large but finite input space. Whenever the adversary observes a leakage site, he can eliminate some impossible inputs and reduce the input space's size. In an extreme case, if the input space's size reduces to one, an adversary can determine the input, which means all the secret information (e.g., the whole secret key) is leaked. By counting the number of inputs [24], we can quantify the information leakage precisely. We use symbolic execution to generate constraints to model the relation between the original sensitive input and an attacker's observations. Symbolic execution can provide fine-grained information but is an expensive operation. Therefore, existing dynamic symbolic execution works [14], [18], [19] either only analyze small programs or apply some domain knowledge [14] to simplify the analysis. We examine the bottleneck of the trace-oriented symbolic execution and optimize it to be scalable to real-world cryptosystems.

We have implemented the proposed technique in a tool called Abacus and show their merits in real-world crypto libraries, including OpenSSL, mbedTLS, and Monocypher. We

```
unsigned long long r;
int secret[32];
...
while(i>0){
    r = (r * r) % n;
    if(secret[--i] == 1)
        r = (r * x) % n;
}
```

Figure 1: Secret-dependent control-flow transfers

```
static char Fsb[256] = {...}
...
uint32_t a = *RK++ ^ \
(FSb[(secret)) ^
(FSb[(secret >> 8)] << 8 ) ^
(FSb[(secret >>16)] << 16 ) ^
(FSb[(secret >>24)] << 24 );
...
```

Figure 2: Secret-dependent memory accesses

collect execution traces of those libraries and run symbolic execution on each instruction. We model each side-channel leakage as a logic formula. Those formulas can precisely model side-channel vulnerabilities. Then we use the conjunction of those formulas to model the same leaks in the source code but appears in the different location of the execution trace file (e.g., leakages inside a loop). Finally, we introduce a Monte Carlo sampling method to estimate the information leakage. The experimental result confirms that Abacus can precisely identify previously known vulnerabilities and report how much information is leaked and which byte in the original sensitive buffer is leaked. We also test Abacus on side-channel free algorithms. Abacus has no false positives. The result also shows the newer version of crypto libraries leaks less amount of information than previous versions. Abacus also discovers new vulnerabilities. With the help of Abacus, we confirm that those vulnerabilities are severe.

In summary, we make the following contributions:

- We propose a novel method that can quantify fine-grained leaked information from side-channel vulnerabilities to match actual attack scenarios. Our approach is different from previous ones in that we model real attack scenarios under one real execution. We transfer the information quantification problem into a counting problem and use the Monte Carlo sampling method to estimate the information leakage.
- We implement the proposed method into a tool and apply it to several real-world software. Abacus successfully identifies previous unknown and known side-channel vulnerabilities and calculates the corresponding information leakage. Our results are surprisingly different, much more useful in practice. The information leakage results provide detailed information that can help developers to triage and fix the vulnerabilities.

II. Background and Threat Model

A. Address-based Side-channels

Side channels leak sensitive information unconsciously through different execution behaviors caused by shared hardware components (e.g., CPU cache, TLB, and DRAM) in modern computer systems [10]–[12], [22], [22], [25], [26].

The key intuition is that many of those side-channel attacks happen when a program accesses different memory addresses if the program has different sensitive inputs. As shown in Figure 1 and Figure 2, if a program shows different patterns in control transfers or data accesses when the program processes different sensitive inputs, the program is vulnerable to side-channel attacks. Different kinds of side-channels can be exploited to retrieve information at various granularities. For example, cache channels can observe cache accesses at the level of cache sets [11], cache lines [22], or other granularities. Other kinds of side-channels, like controlled-channel attack [6], can observe the memory access at the level of memory pages.

B. Threat Model

We consider an attacker shares the same hardware resource with the victim. The attacker attempts to retrieve sensitive information via address-based side-channel attacks. The attacker has no direct access to the memory or cache but can probe the memory or cache at each program point. In reality, the attacker will face many possible obstacles like the noisy observations on the memory or cache. However, for this project, we assume the attacker can have noise-free observations like previous works [14], [18], [20]. The threat model captures most of the cache-based and memory-based side-channel attacks. We only consider the deterministic program for the project and assume an attacker can access the source code of the target program.

III. Abacus: Precise Side-Channel Analysis

In this section, we discuss how Abacus quantifies the amount of leaked information. We first present the limitation of existing quantification metrics. After that, we introduce the abstract of our model, math notations for the rest of the paper, and propose our method.

A. Problem Setting

Existing static side-channel quantification works [17], [20], [27] define information leakage using max entropy or Shannon entropy. If zero bit of information leakage is reported, the program is secure. However, it is not useful in practice if their tools report the program leaks some information. Because their reported result is the "average" leakage, while in a real attack scenario, the leakage could be dramatically different.

```
1 char key[9] = input();
2 if(strcmp(key, "password"))   // leakage site C
3     pass();                   // branch 1
4 else
5     fail();                   // branch 2
```

Figure 3: A dummy password checker

We consider a dummy password checker shown in Figure 3. The password checker will take an 8-byte char array (exclude NULL character) and check if the input is the correct password. If an attacker knows the code executes branch $\{1\}$ by side-channel attacks, he can infer the password equals to "password", in which case the attacker can entirely retrieve the password. Therefore, the total leaked information should be 64 bits, which equals to the size of the original sensitive input if the code executes branch 1.

However, previous static-based approaches cannot precisely reflect the amount of the leakage. According to the definition of Shannon entropy, the leakage will be $\frac{1}{2^{64}} * \log_2 \frac{1}{2^{64}} + \frac{2^{64}-1}{2^{64}} * \log_2 \frac{2^{64}-1}{2^{64}} \approx 0$ bits. Max-entropy is defined on the number of

possible observations. Because the program has two branches, tools based on max-entropy will report the code has $\log_2 2 = 1$ bit leakage.

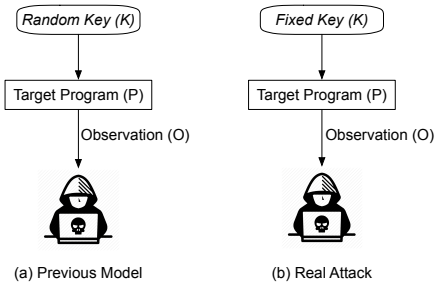


Figure 4: The gap between the real attack and previous models

Both approaches fail to tell how much information is leaked for one real execution. The problem [28] with existing methods is that their approaches neglect input values and real runtime information. They assume an attacker runs the program multiple times with many different or random sensitive inputs. As shown in Figure 4(a), previous models, both Shannon entropy and max-entropy, give an "average" estimate of the information leakage. However, it is not the typical scenario for an adversary to launch a side-channel attack. When a side-channel attack happens, the adversary wants to retrieve the sensitive information, in which case the sensitive information is fixed (e.g., AES keys). The adversary can perform the attack over and over again with fixed input and guess the value bit by bit (e.g., Kocher's timing attacks [29]), as in Figure 4(b). We want to have a theory for dynamic analysis that if the theory says an attack leaks x bits of secret information, then x should be useful in estimating the sensitive level of the vulnerability. However, the above methods all fail in real attack models. This is the first challenge we face **(Challenge C1)**.

B. Notations

In the section, we give necessary definitions and notations for dealing with programs and side-channels. We use capital letters (e.g., A) to represent a set. $|A|$ represents the cardinality of the set A. We use corresponding lower case letters to represent one element in the set (e.g., $a \in A$).

We assume a program (β) has K as the sensitive input. K should be a finite set of keys. The program also takes known messages M as the input. During an AES encryption, for example, β is the encryption function. K is the set of all possible AES keys, and M is encrypted messages. In a real execution, an adversary may have some observations (O) from the program. Examples of those observations include timing, CPU usages, and Electromagnetic signals (EM). In this paper, we consider secret-dependent control-flows and secret-dependent memory accesses as observations.

With the above definition, we have the following mapping between β, K, M, and O:

$$\beta(K, M) \to O$$

We model a side-channel in the following way. An adversary does not have access to K, but he knows β, M, and O. For one execution of a deterministic program, once $k \in K$ and $m \in M$ are fixed, the observation ($o \in O$) should also be determined. As an attacker, he knows β, o, and m. The attacker wants to infer the value of k. We use K^o to denote the set of possible k values that produce the same observation: $K^o = \{k \in K \mid \beta(k, m) \to o\}$

Then the problem of quantifying the amount of leaked information can be transferred into the following question.

How much uncertainty of K can be reduced if an attacker knows β, m, and o?

C. Theoretical Analysis (**Solution to Challege C1**)

In information theory, the mutual information (I) is a measure of the mutual dependence between two variables. We use I to describe the dependence between original sensitive keys (K) and attackers' observations (O).

$$I(K; O) = H(K) - H(K|O) \tag{1}$$

$H(K|O)$ is the entropy of K under the condition O. It quantifies the uncertainty of K, given the value of O. In other words, the conditional entropy $H(K|O)$ marks the uncertainty about K after an adversary has gained some observations (O).

$$H(K|O) = -\sum_{o \in O} p(o) \sum_{k \in K} p(k|o) \log_2 p(k|o) \tag{2}$$

In this project, we hope for a very precise definition of information leakages. Suppose an attacker runs the target program with one input, we want to know how much information he can infer by observing the memory access patterns (o). We come to the simple **~~slogan~~ formulation** [30], [31] that

$$\begin{aligned} &\mathit{Information\ leakage} = \\ &\quad \mathit{Initial\ uncertainty} - \mathit{Remaining\ uncertainty}. \end{aligned}$$

Next we compare the Eq. (1) with the above slogan, we find $H(K)$ is the $\mathit{Initial\ uncertainty}$ and $H(K|O)$ is $\mathit{Remaining\ uncertainty}$. During a real attack, the observation (o) is known. Thus we have $H(K|O) = H(K|o)$. Therefore, we define the amount of leaked information as

$$\mathit{Leakage} = H(K; o) = H(K) - H(K|o)$$

For a program (β) without any domain information, all possible sensitive inputs should appear equally. Therefore, for any $k \in K$, $p(k) = \frac{1}{|K|}$. We have

$$H(K) = \sum_{k \in K} \frac{1}{|K|} \log_2 |K| = \log_2 |K|$$

For any $k' \in K \setminus K^o$, $p(k'|o) = 0$. We get

$$\begin{aligned} H(K; o) &= -\sum_{k \in K^o} p(k|o) \log_2 p(k|o) \\ &\quad - \sum_{k` \in (K \setminus K^o)} p(k'|o) \log_2 p(k'|o) \\ &= \log_2 |K^o| \end{aligned}$$

Definition 1. *Given a program β with the input set K, an adversary has the observation o when the input $k \in K^o$. We denote it as*

$$\beta(K^o, m) \rightarrow o$$

The amount of leaked information $L_{\beta(k)\rightarrow o}$ based on the observation (o) is

$$L_{\beta(k)\rightarrow o} = \log_2 |K| - \log_2 |K^o|$$

The above definition can be understood in an intuitive way. Suppose an attacker guesses a 128-bit encryption key. Without any domain knowledge, he can find the key by performing an exhaustive search over 2^{128} possible keys. However, the program has a side-channel leakage site. After the program finishes execution, the attacker gets some observations and only needs to find the key by performing an exhaustive search over 2^{120} possible keys. Then we can say that 8 bits of the information is leaked. In this example, 2^{128} is the size of K and 2^{120} is the size of K^o.

With the definition, if an attacker observes that the code in Figure 3 runs the branch 1, then the $K^{o^1} = \{\text{“password”}\}$. Therefore, the information leakage $L_{P(k)=o^1} = \log_2 2^{64} - \log_2 1 = 64$ bits, which means the key is totally leaked. If the attacker observes the code hits branch 2, the leaked information is $L_{P(k)=o^2} = \log_2 2^{64} - \log_2 (2^{64} - 1) \approx 0$ bit.

As the size of input sensitive information is usually public, the problem of quantifying the leaked information has been transferred into the problem of estimating the size of input key $|K^o|$ under the condition $o \in O$.

D. Our Conceptual Framework

We now discuss how to model observations (O), which are the direct information that an adversary can get during the side-channel attack.

During an execution, a program (β) have many temporary values ($t_i \in T$). Once β (program), k (secret), and m (message, public) are determined, t_i is also fixed. Therefore, $t_i = f_i(\beta, k, m)$, where f_i is a function that maps between t_i and (β, k, m).

In the paper, we consider two code patterns that can be exploited to infer sensitive information by an attacker, *secret-dependent control transfers* and *secret-dependent data accesses*.

1) Secret-dependent Control Transfers: We think a control-flow is secret-dependent if different input sensitive keys (K) can lead to different branch conditions. We define a branch is secret-dependent if:

$$\exists k_{i1}, k_{i2} \in K,\ f_i(\beta, k_{i1}, m) \neq f_i(\beta, k_{i2}, m)$$

An adversary can observe which branch the code executes, if the branch condition equals to t_b. We use the constraint $c_i : f_i(\beta, k, m) = t_b$ to model the observation (o) on secret-dependent control-transfers.

2) Secret-dependent Data Accesses: Similar to secret-dependent control-flow transfers, a data access operation is secret-dependent if different input sensitive keys (K) can lead to different memory addresses. We use the model from CacheD [14]. The low L bits of the address are irrelevant in side-channels.

We consider a data access is secret-dependent if:

$$\exists k_{i1}, k_{i2} \in K,\ f_i(\beta, k_{i1}, m) >> L \neq f_i(\beta, k_{i2}, m) >> L$$

If the memory access equals to t_b, we can use the constraint $c_i : f_i(\beta, k, m) >> L = t_b >> L$ to model the observation on secret-dependent data accesses.

IV. Scalable to Real-world Crypto Systems

In §III, we propose an advanced information leakage definition for realistic attack scenarios to model two types of address-based side-channel leakages as math formulas, and quantify them by calculating the number of input keys (K^o) that satisfy those math formulas. Intuitively, we can use traditional symbolic execution to capture math formulas and model counting to get the number of satisfying input keys (K^o). However, some preliminary experiments show that the above approach suffers from unbearable costs, which impede its usage to detect and quantify side-channel leakages in real-world applications. In this section, we begin by discussing the bottlenecks of applying the above approaches in real-world software. After that, we propose our methods.

In general, Abacus faces the following performance challenges in order to *scale to production crypto system analysis*.

- Symbolic execution (**Challenge C2**)
- Counting the number of items in K^o (**Challenge C3**)

A. Trace-oriented Symbolic Execution

Symbolic execution is notorious for its unbearable performance cost. Previous trace-oriented symbolic execution based works [14], [28] have large performance bottlenecks. As a result, those approaches either only apply to small-size programs [28] or apply some domain knowledge [32] to simplify the analysis. We implement the approach presented in §III and model the side-channels as formulas. While the tool can finish analyzing some simple cases like AES, it can not handle complicated cases like RSA. We observe that finding side-channels using symbolic execution is different from traditional general symbolic execution and can be optimized to be as efficient as other methods with approaches below.

1) Interprete Instructions Symbolically: Existing binary analysis tools [33], [34] translate machine instructions into intermediate languages (IR) to simplify the analysis. The reason is that the number of machine instructions is enormous, and the semantics of each instruction is complex. Intel Developer Manual [35] introduces more than 1000 different x86 instructions. However, the IR layer, which predigests the implementation and reduces the workload of those tools, is not suitable for side-channels analysis. In general, IR-based or source code side-channels analyses are not accurate enough. In many cases, compilers use conditional moves or bitwise

Table I: The number of x86, REIL IR, and VEX IR instructions on the traces of crypto programs.

	Number of x86 Instructions	Number of VEX IR	Number of REIL IR
AES OpenSSL 0.9.7	$1,704$	$23,938$ (15x)	$62,045$ (36x)
DES OpenSSL 0.9.7	$2,976$	$41,897$ (15x)	$100,365$ (33x)
RSA OpenSSL 0.9.7	$1.6 * 10^7$	$2.4 * 10^8$ (15x)	$5.9 * 10^8$ (37x)
RSA mbedTLS 2.5	$2.2 * 10^7$	$3.1 * 10^8$ (15x)	$8.6 * 10^8$ (39x)

operations to eliminate branches. Also, as some IRs are not a superset or a subset of ISA, it is hard to rule out conditional jumps introduced by IR and add real branches eliminated by IR transformations.

Moreover, the IR design causes significant overhead [36]. Transferring machine instructions into IR is time-consuming. For example, REIL IR [37], adopted in CacheS [19], has multiple transform processes, from binary to VEX IR, BAP IR, and finally REIL IR. Also, IR increases the total number of instructions. For example, x86 instruction *test eax, eax* transfers into 18 REIL IR instructions.

Our Solution: We abandon the IR design and take the engineer efforts to implement the symbolic execution directly on the top of x86 instructions from scratch. Table I shows that eliminating the IR layer can reduce the number of instructions executed during the analysis. Previous works [36] also has a similar approach to speed up the fuzzing. Our implementation is different from their works in two aspects: 1) We use complete constraints. 2) We run the symbolic execution on one execution path each time. Our result indicate the design is around 30 times faster than the IR design (transferring ISA into IR and symbolically executing instructions).

2) Constraint Solving: As discussed in §III-D, the problem of identifying side-channels can be reduced to the question below.

> *Can we find two different input variables $k_1, k_2 \in K$ that satisfy the formula $f_a(k_1) \neq f_a(k_2)$?*

Existing ~~**approach relies**~~ **approaches rely** on satisfiability modulo theories (SMT) solvers (e.g, Z3 [38]) to find satisfying k_1 and k_2. We argue that while it is a universal approach to solving constraints with SMT solvers, for constraints with the above formats, using custom heuristics and testing is much more efficient in practice. Constraint solving is a decision problem expressed in logic formulas. SMT solvers transfer the inputted SMT formula into the boolean conjunctive normal form (CNF) and feed it into the internal boolean satisfiability problem (SAT) solver. The translation process, called "bit blasting", is time-consuming. Also, as the SAT problem is a well-known NP-complete problem, it is hard to deal when it comes to practical uses with huge formulas. Despite the rapid development of SMT solvers in recent years, constraint solving remains one of the obstacles to achieving the scalability of analyzing real-world cryptosystems.

Our Solution: Instead of feeding the formula $f_a(k_1) \neq f_a(k_2)$ into a SMT solver, we randomly pick up $k_1, k_2 \in K$ and test them if they can satisfy the formula. Our solution is based on the following intuition. For most combination of (k_1, k_2), the formula $f_a(k_1) \neq f_a(k_2)$ holds. As long as f_a is not a constant function, such k_1, k_2 must exist. For example, suppose each time we only have 5% chance to find such k_1, k_2, then after we test with different input combination with 100 times, we have $1 - (1 - 0.05)^{100} = 99.6\%$ chance find such k_1, k_2. Such random algorithms work well for our problem.

B. Counting the Number

In this section, we present the algorithm to calculate the information leakage based on Definition 1 (§III), answering to **Challenge C3**.

1) Problem Statement: For each leakage site, we model it with a math formula constraint with the method presented in §III-D. Suppose the address of the leakage site is ξ_i, we use c_{ξ_i} to denote the constraint that models the side-channel leakage.

According to the Definition 1, to calculate the amount of leaked information, the key is to calculate the cardinality of K^o. Suppose an attacker can observe n leakage sites, and each leakage site has the following constraints: $c_{\xi_1}, c_{\xi_2}, \ldots, c_{\xi_n}$ respectively. The total leakage can be calculated from the constraint $c_t(\xi_1, \xi_2, \ldots, \xi_n) = c_{\xi_1} \wedge c_{\xi_2} \wedge \ldots \wedge c_{\xi_n}$. A simple method is to pick elements k from K and check if the element also contained in K^o. Assume q elements satisfy this condition. In expectation, we can use $\frac{k}{q}$ to approximate the value of $\frac{|K|}{|K^o|}$.

However, the above sampling method fails in practice due to the following two problems:

1) The curse of dimensionality problem. $c_t(\xi_1, \ldots, \xi_n)$ is the conjunction of many constraints. Therefore, the input variables of each constraints will also be the input variables of the $c_t(\xi_1, \ldots, \xi_n)$. The sampling method will fail as n increases.
2) The number of satisfying assignments could be exponentially small. According to Chernoff bound, we need exponentially many samples to get a tight bound.

However, despite above two problems, we also observe two characteristics of the problem:

1) $c_t(\xi_1, \xi_2, \ldots, \xi_n)$ is the conjunction of several short constraints c_{ξ_i}. The set containing the input variables of c_{ξ_i} is the subset of the input variables of $c_t(\xi_1, \xi_2, \ldots, \xi_n)$. Some constraints have completely different input variables from other constraints.
2) Each time when we sample $c_t(\xi_1, \xi_2, \ldots, \xi_n)$ with a point, the sampling result is *Satisfied* or not *Not Satisfied*. The result is randomly generated in a way that does not depend on the result in previous experiments. Also, as the amount of leaked information is calculated by $\log$ function, we do not need to count the number of solutions for a given constraint precisely.

In regard to the above problems, we present our methods. First, we split $c_t(\xi_1, \xi_2, \ldots, \xi_n)$ into several independent constraint groups. After that, we run a multi-step sampling method for each constraint.

2) Maximum Independent Partition: For a constraint c_{ξ_i}, we define function π, which maps the constraint into a set of

different input symbols. For example, $\pi(k1 + k2 > 128) = \{k1, k2\}$.

Definition 2. *Given two constraints c_m and c_n, we call them independent iff*

$$\pi(c_m) \cap \pi(c_n) = \varnothing$$

Based on the Definition 2, we can split the constraint $c_t(\xi_1, \xi_2, \ldots, \xi_n)$ into several independent constraints. There are many partitions. For our project, we are interested in the following one.

Definition 3. *For the constraint $c_t(\xi_1, \xi_2, \ldots, \xi_n)$, we call the constraint group $g_1, g_2, \ldots, g_m$ the maximum independent partition of $c_t(\xi_1, \xi_2, \ldots, \xi_n)$ iff*

1) $g_1 \wedge g_2 \wedge \ldots \wedge g_m = c_t(\xi_1, \xi_2, \ldots, \xi_n)$
2) $\forall i, j \in \{1, \ldots, m\} \quad and \quad i \neq j, \quad \pi(g_i) \cap \pi(g_j) = \varnothing$
3) *For any other partitions $h_1, h_2, \ldots, h_{m'}$ satisfy 1) and 2), $m \geq m'$*

The reason we want a good partition of constraints is we want to reduce the dimensions. For example, A good partition of $F : (k_1 = 1) \wedge (k_2 = 2) \wedge (k_3 > 4) \wedge (k_3 - k_4 > 10)$ would be $g_1 : (k_1 = 1) \quad g_2 : (k_2 = 2) \quad g_3 : (k_3 > 4) \wedge (k_3 - k_4 > 10)$ We can sample each constraint independently and combine them together with Theorem 1.

Theorem 1. *Let $g_1, g_2, \ldots, g_m$ be a maximum independent partition of $c_t(\xi_1, \xi_2, \ldots, \xi_n)$. K_c is the input set that satisfies constraint c. We can have the following equation in regard to the size of K_c*

$$|K_{c_t(\xi_1, \xi_2, \ldots, \xi_n)}| = |K_{g_1}| * |K_{g_2}| * \ldots * |K_{g_n}|$$

With Theorem 1, we transfer the problem of counting the number of solutions to a complicated constraint in high-dimension space into counting solutions to several small constraints. We compute the maximum independent partition by iterating each ξ_i and applying the function π over the constraint ξ_i.

3) Multiple Step Monte Carlo Sampling: After we split those constraints into several small constraints, we count the number of solutions for each constraint. Even though the dimension has been significantly reduced after the previous step, this is still a #P problem.

We apply the "counting by sampling" method. For the constraint $g_i = c_{i_1} \wedge c_{i_2} \wedge, \ldots, \wedge c_{i_j} \wedge, \ldots, \wedge c_{i_m}$, if the solution satisfies g_i, it should also satisfy any constraint from c_{i_1} to c_{i_m}. In other words, $K_{c_g i}$ should be the subset of K_{c_1}, K_{c_2}, $\ldots$, K_{c_m}. We notice that c_i usually has less numbers of input compared to g_i. For example, if c_{i_j} has only one 8-bit input variable, we can find the exact solution set $K_{c_{i_j}}$ of c_{i_j} by trying every possible 256 solutions. After that, we can only generate random input numbers for the rest input variables in constraint g_i. With this simple yet effective trick, we can reduce the number of input while still ensure the accuracy.

4) Error Estimation: Our result has the probabilistic guarantee that the error of the estimated amount of leaked information is less than 1 bit under the Central Limit Theorem (CLT) and uncertainty propagation theorem.

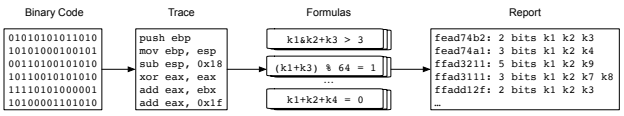


Figure 5: The workflow of Abacus.

Let n be the number of samples and n_s be the number of samples that satisfy the constraint C. Then we can get $\hat{p} = \frac{n_s}{n}$. If we repeat the experiment multiple times, each time we can get a $\hat{p}$. As each $\hat{p}$ is independent and identically distributed, according to the central limit theorem, the mean value should follow normal distribution $\frac{\bar{p} - E(p)}{\sigma\sqrt{n}} \rightarrow N(0, 1)$. Here $E(p)$ is the mean value of p, and σ is the standard variance of p. If we use the observed value $\hat{p}$ to the calculate the standard deviation, we can claim that we have 95%[1] confidence that the error $\Delta p = \bar{p} - E(p)$ falls in the interval:

$$|\Delta p| \leq 1.96\sqrt{\frac{\hat{p}(1 - \hat{p})}{n}}$$

Since we use $L = \log_2 p$ to estimate the amount of leaked information, we can have the following error propagation formula $\Delta L = \frac{\Delta p}{p \ln 2}$ by taking the derivative from Definition 1. For Abacus, we want the error of estimated leaked information (ΔL) to be less than 1 bit. So we get $\frac{\Delta p}{p \ln 2} \leq 1$. Therefore, as long as $n \geq \frac{1.96^2(1-p)}{p(\ln 2)^2}$, we can have 95% confidence that the error of estimated leaked information is less than 1 bit. During the simulation, if n and p satisfy the above inequation, Monte Carlo simulation will terminate.

V. Design and Implementation

A. Design

Abacus has three steps, as shown in Figure 5. First, we run the target program with a concrete input (sensitive information) under the dynamic binary instrumentation (DBI) frameworks to collect execution traces. After that, we run the symbolic execution to capture the fine-grained semantic information of each secret-dependent control-flow transfers and data-accesses. Finally, we run Monte Carlo (MC) simulations to estimate the amount of leaked information.

1) *Execution trace generation.* The design goal of Abacus is to estimate the information leakage as precisely as possible. We run the target binary under dynamic binary instrumentations (DBI) to record execution traces and the runtime information. Once sensitive information is loaded into memory, we start to collect the trace.
2) *Instruction level symbolic execution.* We model attackers' observations from side-channel vulnerabilities with logic formulas. Each formula captures the fine-grained information between input secrets and leakage sites. The engine only symbolically executes the instruction that might be affected by the input key.
3) *Leakage estimation.* We transfer the information leakage quantification problem into the counting problem. We

[1]For a normal distribution, 95% of variable Δp fall within two sigmas of the mean.

propose a Monte Carlo method to estimate the number of satisfying solutions. With the help of the Central Limit Theorem (CLT), we also give an error estimate with the probability, which provides us with the *precision guarantee*.

B. Implementation

We implement Abacus with 16,729 lines of code in C++17 and Python from scratch. It has three components, Intel Pin tool that can collect the execution trace, the instruction-level symbolic execution engine, and the backend that can estimate the information leakage.

Our current implementation supports most Intel 32-bit instructions, including bitwise operations, control transfer, data movement, and logic instructions, which are essential in finding memory-based side-channel vulnerabilities. The tool will use the real values to update the registers and memory cells for other instructions that the current implementation does not support. Therefore, the tool may miss some leakages but will not give us any new false positives. We will release Abacus for public dissemination after the paper is published.

VI. Evaluation

We evaluate Abacus on real-world crypto libraries including OpenSSL, mbedTLS, Libgcrypt and Monocypher. We mark variables and buffers that store the secret. The operation can be done either by adding an annotation on the source code or telling the Pin tool the address and the length of secrets. For DES and AES, we mark symmetric keys as secrets. For RSA, we mark private keys as secrets. For ECDSA, we mark nonces and private keys as secrets.

We build the source code into 32-bit x86 Linux executables with GCC 8.0 under Ubuntu 16.04. We run our experiments on a 2.90GHz Intel Xeon(R) E5-2690 CPU with 128GB RAM. The execution time is calculated on a single-core. During our evaluation process, we are interested in the following aspects:

1) **Identifying side-channels leakages.** Is Abacus effective to detect side-channels in real-world crypto systems? (§VI-A and §VI-B)
2) **Quantifying side-channel leakages.** Can Abacus precisely report the number of leaked bits in crypto libraries? Is the number of leaked bits reported by Abacus useful to justify the severity levels of each side-channel vulnerability? (§VI-C1, §VI-C2, §VI-C3)

A. Evaluation Result Overview

Table II shows the overview of evaluation results. Abacus finds 904 leakages in total from real-world crypto libraries. Among those 904 leak points, 241 of them are leaked due to secret-dependent control-flow transfers, and 663 of them are leaked due to secret-dependent memory accesses.

Abacus also identifies that most side-channel vulnerabilities leak very little information in practice, which confirms our initial assumption. However, we do find some vulnerabilities that Abacus reports with severe leakages. Some of them have been confirmed by existing research that those vulnerabilities can be exploited to realize real attacks. With our tool, developers will be able to distinguish those "vulnerabilities" from severe ones and ignore others easily.

Table II: Evaluation results overview: Name, Side-channel Leaks (Leaks), Secret-dependent Control-flows (CF), Secret-dependent Data-flows (DA), The number of instructions (# Instructions), Symbolic Execution (SE), and Monte Carlo (MC).

Name	# Leaks	# CF	# DF	# Instructions	SE	MC
					ms	ms
AES[1]	68	0	68	39,855	512	1,052
AES[2]	68	0	68	39,855	520	1,057
AES[4]	75	0	75	1,704	231	9,199
AES[5]	88	0	88	1,350	36	1,924
AES[6]	88	0	88	1,350	35	1,961
AES[7]	88	0	88	1,420	36	2,161
AES[8]	88	0	88	1,586	43	1,631
DES[1]	15	0	15	4,596	58	162
DES[2]	15	0	15	4,596	57	162
DES[4]	6	0	6	2,976	163	4,677
DES[5]	8	0	8	2,593	166	6,509
DES[6]	8	0	8	2,593	165	5,975
DES[7]	8	0	8	4,260	182	5,292
DES[8]	6	0	6	8,272	229	5,152
					seconds	seconds
Chacha20[3]	0	0	0	149,353	2	0
Poly1305[3]	0	0	0	1,213,937	15	0
Argon2i[3]	0	0	0	4,595,142	37	0
Ed25519[3]	0	0	0	5,713,619	271	0
ECDSA[2]	6	6	0	4,214,946	48	31
ECDSA[3]	4	4	0	4,192,558	102	1639
ECDSA[4]	5	4	1	8,248,322	101	62
ECDSA[5]	5	4	1	8,263,599	100	58
ECDSA[6]	5	4	1	6,100,465	76	42
ECDSA[7]	0	0	0	10,244,076	121	0
ECDSA[8]	0	0	0	9,266,191	102	59
					minutes	minutes
RSA[1]	6	6	0	22,109,246	39	41
RSA[2]	12	12	0	24,484,441	44	251
RSA[4]	107	105	2	17,002,523	23	428
RSA[5]	38	27	11	14,468,307	29	436
RSA[6]	36	27	9	15,285,210	40	714
RSA[7]	31	22	9	16,390,750	34	490
RSA[8]	4	4	0	18,207,016	8	53
RSA[9]	8	8	0	18,536,796	5	780
RSA[10]	8	8	0	27,407,986	113	6560
Total	904	241	663	167,141,947	341m	10,232m

[1] mbedTLS 2.5 [2] mbedTLS 2.15 [3] Monocypher 3.0
[4] OpenSSL 0.9.7 [5] OpenSSL 1.0.2f [6] OpenSSL 1.0.2k
[7] OpenSSL 1.1.0f [8] OpenSSL 1.1.1 [9] OpenSSL 1.1.1g [10] Libgcrypt 1.8.5

Symmetric encryption implementations in OpenSSL and mbedTLS have significant leakages due to the lookup table implementation. Abacus confirms that all those leakages come from table lookups. We find the new implementation of OpenSSL instead uses typical four 1K tables. It only uses one 1K table. This implementation is rather easy but does somehow decrease the total amount of leaked information as the quantification result shown in the next section.

We also evaluate our tool on the RSA implementation. With the optimization introduced in §IV, we do not apply any domain knowledge to simplify the analysis. Therefore, our tool can identify all the leakage sites reported by CacheD [14] and find new leakages in a shorter time. We also find newer versions of RSA in OpenSSL tend to have fewer leakages detected by Abacus. We will discuss the version changes and

Table III: Comparison with CacheD

	Number of Instructions		Time (s)		Number of Leakages	
	CacheD	Abacus	CacheD	Abacus	CacheD	Abacus
AES 0.9.7	791	1,704	43.4	0.30	48	75
AES 1.0.2f	2,410	1,350	48.5	0.08	32	88
RSA 0.9.7	674,797	16,980,109	199.3	1681	2	105
RSA 1.0.2f	473,392	14,468,307	165.6	1692	2	38
Total	1,151,390	31,451,470	456.8	3373.4	84	317
# of Instructions per second		CacheD: 2,519		Abacus: 9,324		

corresponding leakages in §VI-C2.

Abacus can estimate how much information is leaked from each vulnerability. During the evaluation, for each leakage site, Abacus will stop once 1) it has 95% confidence possibility that the error of estimated leaked information is less than 1 bit, which gives us confidence on the leakage quantification with the *precision guarantee*, or 2) it cannot reach the termination condition after 10 minutes. In the latter case, it means Abacus cannot estimate the amount of leakage with a probabilistic guarantee. We manually check those leakage sites and find most of them are quite severe. We will present the details in the subsequent sections.

B. Comparison with the Existing Tools

In this section, we compare Abacus with the existing trace-based side-channel detection tools.

As shown in Table III, Abacus not only discovers all the leakage sites reported by CacheD [14], but also finds many new ones. CacheD fails to detect many vulnerabilities for two reasons. First, CacheD can only detect secret-dependent memory access vulnerabilities. Abacus can detect secret-dependent control-flows as well. Second, CacheD uses some domain knowledge to simplify symbolic execution to trim the traces before processing, which does not introduce false positives, but may miss some vulnerabilities. The table III shows that Abacus is three times faster than CacheD. As the time of symbolic execution grows quadratically, Abacus is much faster than CacheD when analyzing the same number of instructions. For example, when we test Abacus on AES from OpenSSL 0.9.7, Abacus is over 100x faster than CacheD.

Since DATA [16] needs to compare several execution traces to identify side-channel leakages, Abacus also outperforms DATA in terms of performance. For example, it takes 234 minutes for DATA to analyze the RSA of implementation in OpenSSL 1.1.0f. Abacus only spends 34 minutes according to Table II. Also, DATA reports report 278 control-flow and 460 memory-access leaks. Among those leakages, they find one new vulnerability in RSA after some manual analysis. Abacus finds the vulnerability and reports the vulnerability is severe (int_bn_mod_inverse leaks more than 14.9 bits and BN_div leaks more than 17.2 bits), which eases the pain to identify real sensitive leaks.

C. Case Studies

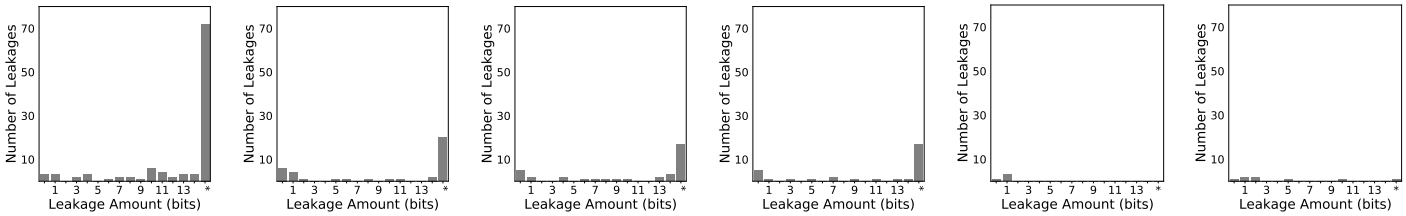
1) Symmetric Ciphers: DES and AES: We test both DES and AES ciphers from mbedTLS and OpenSSL. Both cipher implementations apply lookup tables, which can speed up the performance, but can also introduce additional side-channels as well. During our evaluation, we find mbedTLS 2.5 and 2.15.1 have the same implementation of AES and DES. Therefore, our tool also provides the same leakage report for both versions.

According to Abacus, we find the DES implementations in both mbedTLS and OpenSSL have several severe information leakages in the key schedule function. We do not see any mitigation in the new version. We think it is not seen as worth the engineering efforts given the life cycles of DES.

Abacus shows that the AES in OpenSSL 1.1.1 has less amount of leakages compared to other versions by our tools. We find that OpenSSL 1.1.1 instead uses the 1KB lookup tables with 32-bit entries like older versions, it uses a table with 8-bit entries. Our tool suggests a smaller lookup table can mitigate side-channel vulnerabilities.

2) Asymmetric Ciphers: RSA: We also evaluate Abacus on RSA. Due to the page limit, we do not list the detailed leakage report. In general, developers are more interested in fixing side-channel vulnerabilities for RSA implementations. As shown in Figure 6, the result indicates that the newer version of OpenSSL leaks less amount of information than previous versions. After version 0.9.7g, OpenSSL adopts a fixed-window mod_exp_mont implementation for RSA. With the new design, the sequence of squares and multiples and the memory access patterns are independent of the secret key. Abacus's result confirms the new exponentiation implementation has mitigated most of the leakages effectively because the other four versions have fewer leakages compared with version 0.9.7. OpenSSL version 1.0.2f, 1.0.2k, and 1.1.0f almost have the same amount of leakage. We check the changelog and find only one change for patching the vulnerability CVE-2016-0702. Abacus finds OpenSSL 1.1.1 and 1.1.1g have significantly less amount of leaked information compared to other versions. We check the changelog of those two versions and find it claims the new RSA implementation adopts "numerous side-channel attack mitigation", which proves the effectiveness of our quantifying method. We also observe the latest version (1.1.1g) has some new leakages. We have contacted the developers and are waiting for their responses.

Our quantification result shows vulnerabilities that leak more information by Abacus are more likely to be fixed in the updated version. As presented in Figure 6, OpenSSL 0.9.7 has several severe leaks from function bn_sqr_comba8, which is a main component of the OpenSSL big number implementation. Shown in Figure 7, it has a secret-dependent control flow at line 8. The value of the function parameter **a** is derived from the secret key. As function bn_sqr_comba8 calls the macro (sqr_add_c2) multiple times, and the code can leak some information each time. Abacus thinks the vulnerability is quite serious. The vulnerability has been patched in OpenSSL 1.1.1. Seen in Figure 8, control-flows transfers are replaced. So there is no leaks in the function sqr_add_c2 in OpenSSL 1.1.1. We mention that line 4 and 9 in Figure 7 both have if branches. However, it is not a leak site because most compilers will use *add with carry* instruction to remove the branch. Besides, branches can also be compiled into non-branch machine instructions like conditional moves. We notice a bitwise operation in Libgcrypt 1.8.5 is compiled



(a) OpenSSL 0.9.7 (b) OpenSSL 1.0.2f (c) OpenSSL 1.0.2k (d) OpenSSL 1.1.0f (e) OpenSSL 1.1.1 (f) OpenSSL 1.1.1g

Figure 6: Side-channel leakages in different implementations of RSA in OpenSSL. We round the number of leaked information into the nearest integer. The mark $*$ means timeout (see §VI-A).

```
# define mul_add_c2(a,b,c0,c1,c2)                    \
    t=(BN_ULLONG)a*b;                                \
    tt=(t+t)&BN_MASK;                                \
    if (tt < t) c2++;                                \
    t1=(BN_ULONG)Lw(tt);                             \
    t2=(BN_ULONG)Hw(tt);                             \
    c0=(c0+t1)&BN_MASK2;                             \
    if ((c0 < t1) && (((++t2)&BN_MASK2) == 0)) c2++; \
    c1=(c1+t2)&BN_MASK2; if ((c1) < t2) c2++;
```

Figure 7: Macro sqr_add_c2 in OpenSSL 0.9.7

```
# define mul_add_c2(a,b,c0,c1,c2)      do { \
    BN_ULONG ta = (a), tb = (b);            \
    BN_ULONG lo, hi, tt;                    \
    BN_UMULT_LOHI(lo,hi,ta,tb);             \
    c0 += lo; tt = hi+((c0<lo)?1:0);        \
    c1 += tt; c2 += (c1<tt)?1:0;            \
    c0 += lo; hi += (c0<lo)?1:0;            \
    c1 += hi; c2 += (c1<hi)?1:0;            \
    } while(0)
```

Figure 8: Macro sqr_add_c2 in OpenSSL 1.1.1

to a conditional jump, which leads to a side-channel leakage. Therefore, simple code reviews are not accurate enough to detect side-channels.

For vulnerabilities that leak less amount of information, developers are more reluctant to fix them. For example, OpenSSL 0.9.7 adopts a fixed windows version of function BN_mod_exp_mont_consttime to replace original function BN_mod_exp_mont. Abacus detects a minor vulnerability in the original function that can leak the last bit of the big number **m**. In the updated version, developers make the fixed windows become the default option and rewrite most parts of the function. However, the leakage site still exists in OpenSSL 1.1.1, despite it is quite blunt.

3) Monocypher: Monocypher is a small, easy to use cryptographic library with a comparable performance as LibSodium [39] and NaCl [40]. We choose four ciphers that are designed to be side-channel resistant from the library. Because those ciphers have no data flow from secrets to branch conditions and load addresses. Monocypher should be safe under our threat models. We analyze those ciphers with Abacus, and it reports no leaks. This indicates that Abacus can be used for countermeasure confirmation.

VII. Discussions and Limitations

While recent works reported lots of side-channel vulnerabilities, we find many of them are not patched by developers. However, side-channels are inevitable in software and it is hard to fix all of them. Addressing old vulnerabilities may also introduce new leakage sites. We need a tool that can automatically estimate the sensitivity level of each vulnerability. So software engineers can focus on "severe" leakages. For example, our tool will report that the modular exponentiation using square and multiply algorithms can leak more information than a key validation function.

Abacus can be used by software developers to find severe vulnerabilities and reason about countermeasures. Abacus estimates the amount of leaked information for each side-channel leakage in one execution trace. Abacus is useful for software engineers to test programs and fix vulnerabilities. The design, which is very precise in terms of true leakages compared to other static source code method [41], [42], can omit some leakages on other traces. The amount of leaked information also depends on the secret key. However, as the tool is intended for debugging and testing, we think it is software engineers' responsibility to select the input key and trigger the path in which they are interested. It is not a problem for crypto software since virtually all keys follow similar computational paths.

We use the amount of leaked information to represent the sensitivity level of each side-channel vulnerability. Although imperfect, Abacus produces a reasonable measurement for each leak. For example, the simple modular exponentiation is notoriously famous for various side-channel attacks [29]. During the execution, the single leak points will be executed multiple times, and each time it leaks one different bit. Therefore, Abacus reports that the vulnerability can leak the whole key. However, not every leak point inside a loop is severe. Because very often the leakages may leak the same bit in the original key, and those leaks are not independent. Abacus can capture the most fine-grained information by modeling each leak during the execution as a math formula and use the conjunction of those formulas to describe the total effect. Some leakage sites (e.g., square and multiply) can leak one particular bit of the original key, but some leakage sites leak one bit from several bytes in the original key.

Abacus reaches full precision if the number of estimated leaked bits equals to Definition 1. Abacus may lose precision from the memory model it uses in theory. However, we do not find false positives caused by the imprecise memory model during the evaluation. Sampling can also introduce imprecision but with a probabilistic guarantee. However, during the evaluation, we find that Abacus cannot estimate the amount of leak-

age for some leakage sites in a reasonable time, which means the number of K^o is very small. According to Definition 1, it means the leakage is very severe. The sampling method in §IV seems simple and may miss some leakages (e.g., chosen ciphertext attacks) in theory. However, the evaluation result shows Abacus can identify all the leakages that can be found by the previous work [14], [18], [19] when we evaluate the same victim program.

VIII. Related Work

There is a vast amount of work on side channel detection [14], [16]–[20], [43], mitigation [32], [44]–[51], information quantification [23], [27], [28], [52]–[56], and model counting [28], [57]–[60]. Here we only present the closely related work to ours. Due to space limit, we do not include related work on side-channel attacks.

A. Detection and Mitigation

CacheAudit [20] uses abstract domains to compute the over approximation of cache-based side-channel information leakage upper bound. However, it is hard to judge the sensitive level of the side-channel leakage based on the leakage provided by CacheAudit. CacheS [19] improves the work of CacheAudit by proposing the novel abstract domains, which only track secret-related code. Like CacheAudit, CacheS cannot provide the information to indicate the sensitive level of side-channel vulnerabilities. CacheSym [18] introduces a static cache-aware symbolic reasoning technique to cover multiple paths for the target program. Still, their approaches cannot assess the sensitive level for each side-channel vulnerability. And their approaches only work on small code snippets.

The dynamic approach, usually with taint analysis and symbolic execution, can perform a very precise analysis. CacheD [14] takes a concrete execution trace and run the symbolic execution on the top of the trace to get the formula of each memory address. Therefore, CacheD is quite precise in term of false positives. We adopted a similar idea to model the secret-dependent memory accesses. DATA [16] detects address-based side-channel vulnerabilities by comparing different execution traces under various test inputs. MicroWalk [17] uses mutual information (MI) between sensitive input and execution state to detect side-channels.

Both hardware [32], [44]–[47], [61] and software [41], [48]–[51] side-channels mitigation methods have been proposed recently. Hardware countermeasures, including parting the hardware computing resource [44], randomizing cache accesses [32], [47], and designing new architecture [62], which need to change the hardware and is usually hard to adopt in reality. On the contrary, software approaches are usually easy to implement. Coppens et al. [49] introduced a compiler-based approach to eliminate key-dependent control-flow transfers. Crane et al. [51] mitigated side-channels by randomizing software. As for crypto libraries, the basic idea is to eliminate key-dependent control-flow transfers and data accesses. Common approaches include bit-slicing [63], [64] and unifying control-flows [49].

B. Quantification

Proposed by Denning [65] and Gray [66], Quantitative Information Flow (QIF) aims at providing an estimation of the amount of leaked information from the sensitive information given the public output. If zero bit of the information is leaked, the program is called non-interference. McCamant and Ernst [54] quantify the information leakage as the network flow capacity. Backes et al. [23] propose an automated method for QIF by computing an equivalence relation on the set of input keys. But the approach cannot handle real-world programs with bitwise operations. Phan et al. [55] propose symbolic QIF. The goal of their work is to ensure the program is non-interference. They adopt an over approximation way of estimating the total information leakage and their method does not work for secret-dependent memory access side-channels. CHALICE [28] quantifies the leaked information for a given cache behavior. CHALICE also mentions the same issue of max-entropy. It symbolically reason about cache behavior and estimate the amount of leaked information based on cache miss/hit. Their approach can only scale to small programs, which limits its usage in real-world applications. On the contrary, Abacus can assess the sensitive level of side-channels with different granularities. It can also analyze side-channels in real-world crypto libraries.

C. Model Counting

Model counting usually refers to the problem of computing the number of models for a propositional formula (#SAT). There are two directions solving the problem, exact model counting and approximate model counting. We focus on approximate model counting since it shares similar idea as our approach. Wei and Selman [57] introduce ApproxCount, a local search based method using Markov Chain Monte Carlo (MCMC). ApproxCount has the better scalability compared to exact model counters. Other approximate model counter includes SampleCount [58], Mbound [59], and MiniCount [60]. Compared to ApproxCount, those model counters can give lower or upper bounds with guarantees. Despite the rapid development of model counters for SAT and some research [67], [68] on Modulo Theories model counting (#SMT). They cannot be directly applied to side channel leakage quantification. ApproxFlow [52] uses ApproxMC [69] for information flow quantification, but it's only tested with small programs while Abacus can scale to production crypto libraries.

IX. Conclusion

In this paper, we present a novel method to quantify memory-based side-channel leakages. We implement the method in a prototype called Abacus and show its effectiveness in finding and quantifying side-channel leakages. With the new definition of information leakage that imitates real side-channel attackers, the number of leaked bits is useful in practice to justify and understand the severity level of side-channel vulnerabilities. The evaluation results confirm our design goal and show Abacus is useful in estimating the amount of leaked information in real-world applications.

REFERENCES

[1] D. Agrawal, B. Archambeault, J. R. Rao, and P. Rohatgi, "The EM side-channel(s)," in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2002.

[2] M. Kar, A. Singh, S. Mathew, A. Rajan, V. De, and S. Mukhopadhyay, "Improved power-side-channel-attack resistance of an aes-128 core via a security-aware integrated buck voltage regulator," in *ISSCC 2017*.

[3] S. Chari, C. S. Jutla, J. R. Rao, and P. Rohatgi, "Towards sound approaches to counteract power-analysis attacks," in *Annual International Cryptology Conference*. Springer, 1999.

[4] M. Alam, H. A. Khan, M. Dey, N. Sinha, R. Callan, A. Zajic, and M. Prvulovic, "One&done: A single-decryption em-based attack on openssl's constant-time blinded RSA," in *USENIX Security 18*.

[5] D. Genkin, A. Shamir, and E. Tromer, "RSA key extraction via low-bandwidth acoustic cryptanalysis," in *Annual Cryptology Conference*. Springer, 2014.

[6] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *IEEE Symposium on Security and Privacy 2015*.

[7] J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution," in *USENIX Security 18*.

[8] S. van Schaik, C. Giuffrida, H. Bos, and K. Razavi, "Malicious management unit: Why stopping cache attacks in software is harder than you think," in *USENIX Security 18*.

[9] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring fine-grained control flow inside SGX enclaves with branch shadowing," in *USENIX Security 17*.

[10] D. Gruss, R. Spreitzer, and S. Mangard, "Cache template attacks: Automating attacks on inclusive last-level caches," in *USENIX Security 15*.

[11] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *IEEE Symposium on Security and Privacy 2015*.

[12] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of AES," in *Proceedings of the 2006 The Cryptographers' Track at the RSA Conference on Topics in Cryptology*, ser. CT-RSA'06. Springer-Verlag, 2006.

[13] D. Gullasch, E. Bangerter, and S. Krenn, "Cache games – bringing access-based cache attacks on aes to practice," in *IEEE Symposium on Security and Privacy 2011*.

[14] S. Wang, P. Wang, X. Liu, D. Zhang, and D. Wu, "CacheD: Identifying cache-based timing channels in production software," in *USENIX Security 17*.

[15] Y. Tsunoo, T. Saito, T. Suzaki, M. Shigeri, and H. Miyauchi, "Cryptanalysis of des implemented on computers with cache," in *Cryptographic Hardware and Embedded Systems - CHES 2003*, C. D. Walter, Ç. K. Koç, and C. Paar, Eds. Springer Berlin Heidelberg, 2003.

[16] S. Weiser, A. Zankl, R. Spreitzer, K. Miller, S. Mangard, and G. Sigl, "DATA – differential address trace analysis: Finding address-based side-channels in binaries," in *USENIX Security 18*.

[17] J. Wichelmann, A. Moghimi, T. Eisenbarth, and B. Sunar, "Microwalk: A framework for finding side channels in binaries," in *ACSAC '18*, 2018.

[18] R. Brotzman, S. Liu, D. Zhang, G. Tan, and M. Kandemir, "CaSym: Cache aware symbolic execution for side channel detection and mitigation," in *IEEE Symposium on Security and Privacy 2019*.

[19] S. Wang, Y. Bao, X. Liu, P. Wang, D. Zhang, and D. Wu, "Identifying cache-based side channels through secret-augmented abstract interpretation," in *USENIX Security 19*.

[20] G. Doychev, D. Feld, B. Kopf, L. Mauborgne, and J. Reineke, "CacheAudit: A tool for the static analysis of cache side channels," in *USENIX Security 13*.

[21] C. Aumüller, P. Bier, W. Fischer, P. Hofreiter, and J.-P. Seifert, "Fault attacks on rsa with CRT: Concrete results and practical countermeasures," in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2002, pp. 260–275.

[22] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack," in *USENIX Security 14*.

[23] M. Backes, B. Köpf, and A. Rybalchenko, "Automatic discovery and quantification of information leaks," in *IEEE Symposium on Security and Privacy 2009*.

[24] W. Wei and B. Selman, "A new approach to model counting," in *Theory and Applications of Satisfiability Testing*, F. Bacchus and T. Walsh, Eds. Springer Berlin Heidelberg, 2005.

[25] Q. Ge, Y. Yarom, D. Cock, and G. Heiser, "A survey of microarchitectural timing attacks and countermeasures on contemporary hardware," *Journal of Cryptographic Engineering*, vol. 8, no. 1, 2018.

[26] J. Szefer, "Survey of microarchitectural side and covert channels, attacks, and defenses," *Journal of Hardware and Systems Security*, vol. 3, no. 3, 2019.

[27] K. Zhang, Z. Li, R. Wang, X. Wang, and S. Chen, "Sidebuster: automated detection and quantification of side-channel leaks in web application development," in *CCS 2010*.

[28] S. Chattopadhyay, M. Beck, A. Rezine, and A. Zeller, "Quantifying the information leak in cache attacks via symbolic execution," in *MEMOCODE '17*.

[29] P. C. Kocher, "Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems," in *Annual International Cryptology Conference*. Springer, 1996, pp. 104–113.

[30] G. Smith, "On the foundations of quantitative information flow," in *Foundations of Software Science and Computational Structures*, L. de Alfaro, Ed. Springer Berlin Heidelberg, 2009.

[31] A. Askarov and S. Chong, "Learning is change in knowledge: Knowledge-based security for dynamic policies," in *CSF 2012*. Piscataway, NJ, USA: IEEE Press, Jun., pp. 308–322.

[32] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," in *ISCA '07*.

[33] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SoK: (state of) the art of war: Offensive techniques in binary analysis," in *IEEE Symposium on Security and Privacy*, 2016.

[34] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "Bap: A binary analysis platform," in *Computer Aided Verification*, G. Gopalakrishnan and S. Qadeer, Eds., 2011.

[35] Intel Corporation, *Intel® 64 and IA-32 Architectures Software Developer's Manual*, 2019.

[36] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM: A practical concolic execution engine tailored for hybrid fuzzing," in *USENIX Security 18*.

[37] T. Dullien and S. Porst, "Reil: A platform-independent intermediate representation of disassembled code for static code analysis," 2009.

[38] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *TACAS'08/ETAPS'08*.

[39] *LibSodium*. [Online]. Available: https://libsodium.org

[40] D. J. Bernstein, T. Lange, and P. Schwabe, "The security impact of a new cryptographic library," in *International Conference on Cryptology and Information Security in Latin America*. Springer, 2012, pp. 159–176.

[41] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, "Verifying constant-time implementations," in *USENIX Security 16*.

[42] J. Bacelar Almeida, M. Barbosa, J. S. Pinto, and B. Vieira, "Formal verification of side-channel countermeasures using self-composition," *Sci. Comput. Program.*, vol. 78, no. 7, 2013.

[43] A. Langley, "ctgrind-checking that functions are constant time with valgrind, 2010," *URL https://github. com/agl/ctgrind*, vol. 84, 2010.

[44] D. Page, "Partitioned cache architecture as a side-channel defence mechanism," *IACR Cryptology ePrint Archive*, vol. 2005, 2005.

[45] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers, "A hardware design language for timing-sensitive information-flow security."

[46] X. Li, V. Kashyap, J. K. Oberg, M. Tiwari, V. R. Rajarathinam, R. Kastner, T. Sherwood, B. Hardekopf, and F. T. Chong, "Sapper: A language for hardware-level security policy enforcement," in *ASPLOS '14*.

[47] M. Werner, T. Unterluggauer, L. Giner, M. Schwarz, D. Gruss, and S. Mangard, "Scattercache: Thwarting cache attacks via cache set randomization," in *USENIX Security 19*.

[48] M.-W. Shih, S. Lee, T. Kim, and M. Peinado, "T-sgx: Eradicating controlled-channel attacks against enclave programs." in *NDSS 2017*.

[49] B. Coppens, I. Verbauwhede, K. D. Bosschere, and B. D. Sutter, "Practical mitigations for timing-based side-channel attacks on modern x86 processors," in *IEEE Symposium on Security and Privacy 2009*.

[50] E. Brickell, G. Graunke, M. Neve, and J.-P. Seifert, "Software mitigations to hedge aes against cache-based software side channel vulnerabilities." *IACR Cryptology ePrint Archive*, vol. 2006, 2006.

[51] S. Crane, A. Homescu, S. Brunthaler, P. Larsen, and M. Franz, "Thwarting cache side-channel attacks through dynamic software diversity." in *NDSS*, 2015.

[52] F. Biondi, M. A. Enescu, A. Heuser, A. Legay, K. S. Meel, and J. Quilbeuf, "Scalable approximation of quantitative information flow in programs," in *International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer, 2018, pp. 71–93.

[53] B. Kopf, L. Mauborgne, and M. Ochoa, "Automatic quantification of cache side-channels," in *Computer Aided Verification*, P. Madhusudan and S. A. Seshia, Eds. Springer Berlin Heidelberg, 2012.

[54] S. McCamant and M. D. Ernst, "Quantitative information flow as network flow capacity," in *PLDI 2008*.

[55] Q.-S. Phan, P. Malacaria, O. Tkachuk, and C. S. Păsăreanu, "Symbolic quantitative information flow," *SIGSOFT Softw. Eng. Notes*, vol. 37, no. 6, 2012.

[56] Z. Zhou, Z. Qian, M. K. Reiter, and Y. Zhang, "Static evaluation of noninterference using approximate model counting," in *IEEE Symposium on Security and Privacy 2018*.

[57] W. Wei and B. Selman, "A new approach to model counting," in *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 2005, pp. 324–339.

[58] C. P. Gomes, J. Hoffmann, A. Sabharwal, and B. Selman, "From sampling to model counting." in *IJCAI 2007*, 2007, pp. 2293–2299.

[59] C. P. Gomes, A. Sabharwal, and B. Selman, "Model counting: A new strategy for obtaining good bounds," in *AAAI 2006*.

[60] L. Kroc, A. Sabharwal, and B. Selman, "Leveraging belief propagation, backtrack search, and statistics for model counting," in *International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming*. Springer, 2008, pp. 127–141.

[61] K. v. Gleissenthall, R. G. Kıcı, D. Stefan, and R. Jhala, "IODINE: Verifying constant-time execution of hardware," in *USENIX Security 19*.

[62] M. Tiwari, J. K. Oberg, X. Li, J. Valamehr, T. Levin, B. Hardekopf, R. Kastner, F. T. Chong, and T. Sherwood, "Crafting a usable microkernel, processor, and i/o system with strict and provable information flow security," in *ACM SIGARCH Computer Architecture News*, vol. 39, no. 3. ACM, 2011.

[63] R. Könighofer, "A fast and cache-timing resistant implementation of the AES," in *Cryptographers' Track at the RSA Conference*. Springer, 2008.

[64] C. Rebeiro, D. Selvakumar, and A. Devi, "Bitslice implementation of aes," in *International Conference on Cryptology and Network Security*. Springer, 2006.

[65] D. E. Robling Denning, *Cryptography and data security*. Addison-Wesley Longman Publishing Co., Inc., 1982.

[66] J. W. Gray III, "Toward a mathematical foundation for information flow security," *Journal of Computer Security*, vol. 1, no. 3-4, pp. 255–294, 1992.

[67] D. Chistikov, R. Dimitrova, and R. Majumdar, "Approximate counting in smt and value estimation for probabilistic programs," *Acta Informatica*, vol. 54, no. 8, pp. 729–764, 2017.

[68] Q.-S. Phan, "Model counting modulo theories," *arXiv preprint arXiv:1504.02796*, 2015.

[69] S. Chakraborty, K. S. Meel, and M. Y. Vardi, "Algorithmic improvements in approximate counting for probabilistic inference: From linear to logarithmic sat calls," Tech. Rep., 2016.

[70] C. E. Shannon, "A mathematical theory of communication," *Bell system technical journal*, vol. 27, no. 3, pp. 379–423, 1948.

[71] D. Clark, S. Hunt, and P. Malacaria, "A static analysis for quantifying information flow in a simple imperative language," *Journal of Computer Security*, vol. 15, no. 3, pp. 321–371, 2007.

[72] G. Doychev and B. Köpf, "Rigorous analysis of software countermeasures against cache attacks," in *PLDI 2017*.

SUPPLEMENTARY MATERIALS TO SUBMISSION #63

I. EXISITING INFORMATION LEAKAGE QUANTIFICATION

Given an event e that occurs with the probability $p(e)$, we receive

$$I = -\log_2 p(e)$$

bits of information by knowing the event e happens according to information theory [70]. Considering a char variable a with one byte size in a C program, its value ranges from 0 to 255. If we observe a equals 1, without any domain knowledge, the probability of this observation is $\frac{1}{256}$. So we get $-\log(\frac{1}{256}) = 8$ bits information, which is exactly the size of a char variable in the C program. Existing works on information leakage quantification typically use Shannon entropy [17], [71], min-entropy [30], and max-entropy [20], [72]. In these frameworks, the input sensitive information K is considered as a random variable.

Let k be one of the possible value of K. The Shannon entropy $H(K)$ is defined as

$$H(K) = -\sum_{k \in K} p(k)\log_2(p(k))$$

Shannon entropy can be used to quantify the initial uncertainty about sensitive information. It measures the amount of information in a system.

Min-entropy describes the information leaks for a program with the most likely input. For example, min-entropy can be used to describe the best chance of success in guessing one's password using the most common password.

$$\textit{min-entropy} = -\log_2(p_{max})$$

Max-entropy is defined solely on the number of possible observations.

$$\textit{max-entropy} = -\log_2 n$$

As it is easy to compute, most recent works [20], [72] use max-entropy as the definition of the amount of leaked information.

To illustrate how these definitions work, we consider the code fragment in Figure 9. It has two possible leakage sites, A and B.

```
1 uint8_t key[2], t1, t2;
2 get_key(key);              // 0 <= key[0], key[1] < 256
3 t1 = key[0] + key[1];
4 t2 = key[0] - key[1];
5 if (t1 < 4)                // leakage site A
6     foo();
7 if (t2 > 0)                // leakage site B
8     bar();
```

Figure 9: Side-channel leakage

In this paper we assume an attacker can observe the secret-dependent control-flows in Figure 9. Therefore, an attacker can have two different observations for each leak site depending on the value of the key: A for function foo is executed, $\neg A$ for function foo is not executed, B for function bar is executed, and $\neg B$ for function bar is not executed.

Table IV: The distribution of observation

Observation (o)	A	$\neg A$	B	$\neg B$
Number of Solutions	65526	10	32768	32768
Possibility (p)	0.9998	0.0002	0.5	0.5

Assuming key is uniformly distributed, we can calculate the corresponding possibility by counting the number of possible inputs. Table IV describes the probability of each observation. We use the above three types of leakage metrics to quantify the leaked information for leak A and leak B.

Min Entropy. As $p_{Amax} = 0.9998$ and $p_{Bmax} = 0.5$, with the definition, $\textit{min-entropy}_A$ equals to 0.000 bits and $\textit{min-entropy}_B$ equals to 1.000 bits.

Max Entropy. Depending on the value of key, the code have two branches for each leakage site. Therefore, with the max entropy definition, both leakage sites leak 1.000 bits.

Shannon Entropy. Based on Shannon entropy, the respective amount of information in A and B equals to

$$\begin{aligned}\textit{Shannon-entropy}_A &= -(0.9998 * \log_2 0.9998 \\ &\quad + 0.0002 * \log_2 0.0002) \\ &= 0.000 \text{ bits} \\ \textit{Shannon-entropy}_B &= -(0.5 * \log_2 0.5 \\ &\quad + 0.5 * \log_2 0.5) \\ &= 1.000 \text{ bits}\end{aligned}$$

II. ABACUS' MAIN COMPONENTS

Table V: Abacus' main components and sizes

Component	Lines of Code (LOC)
Trace Logging	501 lines of C++
Symbolic Execution	14,963 lines of C++
Data Flow	451 lines of C++
Monte Carlo Sampling	603 lines of C++
Others	211 lines of Python
Total	16,729 lines

III. ALGORITHM TO COMPUTE THE MAXIMUM INDEPENDENT PARTITION

Algorithm 1: The Maximum Independent Partition

input : $c_t(\xi_1, \xi_2, \ldots, \xi_n) = c_{\xi_1} \wedge c_{\xi_2} \wedge \ldots \wedge c_{\xi_m}$
output: The Maximum Independent Partition of $G = \{g_1, g_2, \ldots, g_m\}$

Insert c_{ξ_1} to G as a new entry
for $i \leftarrow 2$ **to** n **do**
  $S_{c_{\xi_i}} \leftarrow \pi(c_{\xi_i})$
  for $g_i \in G$ **do**
    $S_{g_j} \leftarrow \pi(g_j)$
    $S \leftarrow S_{c_{\xi_i}} \cap S_{g_j}$
    if $S \neq \varnothing$ **then**
      $g_j \leftarrow g_i \wedge g_{\xi_i}$
      continue
    end
    Insert c_{ξ_i} to G as a new entry
  end
end

IV. Algorithm to Compute the Number of Satisfying Assignments

Algorithm 2: Multiple Step Monte Carlo Sampling

Input: The constraint $g_i = c_{i_1} \wedge c_{i_2} \wedge \ldots \wedge c_{i_m}$
Output: The number of assignments that satisfy g_i $|K_{g_i}|$

n: the number of sampling times
S_{c_i}: the set contains input variables for c_i
n_s: the number of satisfying assignments
N_{c_t}: the set contains all solution for c_t
r: times of reducing g
k: the input variable
R: a function that produces a random point from S_{c_i}
$r \leftarrow 1$, $n \leftarrow 0$
for $t \leftarrow 1$ **to** m **do**
  $S_{c_t} \leftarrow \pi(c_t)$
  if $|S_{c_t}| = 1$ **then**
    $N_{c_t} \leftarrow$ Compute all solutions of c_i
    $N_{c_t} = \{n_1, \ldots, n_m\},\ S_{c_t} = \{k\}$
    $g_i = g_i(k = n_1) \wedge \ldots \wedge g_i(k = n_m)$
    $r \leftarrow r + 1$
  end
end
while $n \leq \frac{8p}{1-p}$ **do**
  $S_{g_i} \leftarrow \pi(g_i)$
  $v \leftarrow R(S_{g_i})$ **if** v *satisfies* g_i **then**
    $n_s \leftarrow n_s + 1$
  end
  $n \leftarrow n + 1,\ p = \frac{n_s}{n}$
end
$|K_{g_i}| \leftarrow n_s|K|/(n * r * range(k))$

V. AES Lookup Tables Leakage

Shown in Figure 10, usually a smaller look up table leaks less amount of information.

```
1  int mbedtls_internal_aes_encrypt( mbedtls_aes_context *ctx,
2  const unsigned char input[16],
3  unsigned char output[16] )
4  {
5  uint32_t *RK, X0, X1, X2, X3, Y0, Y1, Y2, Y3;
6  ...
7  for( i = ( ctx->nr >> 1 ) - 1; i > 0; i-- )
8  {
9    AES_FROUND( Y0, Y1, Y2, Y3, X0, X1, X2, X3 ); // Leakage 1
10   AES_FROUND( X0, X1, X2, X3, Y0, Y1, Y2, Y3 ); // Leakage 2
11 }
12 AES_FROUND( Y0, Y1, Y2, Y3, X0, X1, X2, X3 );   // Leakage 3
13 X0 = *RK++ ^ \                                  // Leakage 4
14     ( (uint32_t) FSb[ ( Y0       ) & 0xFF ]       ) ^
15     ( (uint32_t) FSb[ ( Y1 >>  8 ) & 0xFF ] <<  8 ) ^
16     ( (uint32_t) FSb[ ( Y2 >> 16 ) & 0xFF ] << 16 ) ^
17     ( (uint32_t) FSb[ ( Y3 >> 24 ) & 0xFF ] << 24 );
18 // X1, X2, X3 do the same computation as X0
19 ...                                          // Leakage 5,6,7
20 PUT_UINT32_LE( X0, output,  0 );
21 ...
22 return( 0 );
23 }
```

Figure 10: Function mbedtls_internal_aes_encrypt

VI. Minor side-channels vulnerability

Here we present a side-channel vulnerability that leaks less than one bit information by Abacus. The vulnerability exists from OpenSSL 0.9.7 to OpenSSL 1.1.1. Shown in Figure 11 and Figure 12, **m** is a big number that derives from the private key. At line 6, it can leak the last bit of **m** by observing the branch. As the leak is tiny, we think developers do not have enough motivations to fix the vulnerability.

```
1  int BN_mod_exp_mont_consttime(BIGNUM *rr,
2  const BIGNUM *a, const BIGNUM *p,
3  const BIGNUM *m, BN_CTX *ctx, BN_MONT_CTX *in_mont)
4  {
5  ...
6  if (!(m->d[0] & 1)) {
7      ...
8      return 0;
9  }
10 bits = BN_num_bits(p);
11 if (bits == 0)
12 ...
13 }
```

Figure 11: BN_mod_exp_mont_consttime in OpenSSL 0.9.7

```
1  int BN_mod_exp_mont_consttime(BIGNUM *rr,
2  const BIGNUM *a, const BIGNUM *p,
3  const BIGNUM *m, BN_CTX *ctx, BN_MONT_CTX *in_mont)
4  {
5  ...
6  if (!BN_is_odd(m)) {
7      ...
8      return 0;
9  }
10 bits = BN_num_bits(p);
11 if (bits == 0)
12 ...
13 }
```

Figure 12: BN_mod_exp_mont_consttime in OpenSSL 1.1.1

VII. Unknown Leaks in OpenSSL 1.1.1

Shown in Table XV, Abacus discovers a series of side-channel vulnerabilities in the up-to-date version of OpenSSL library. However, many of them are negligible quantified by Abacus. Here we present a few vulnerabilitites in Figure 13 and Figure 14.

```
1  BN_ULONG bn_sub_words(BN_ULONG *r, const BN_ULONG *a,
2  const BN_ULONG *b, int n)
3  {
4  BN_ULONG t1, t2;
5  int c = 0;
6  ...
7  while (n) {
8      t1 = a[0];
9      t2 = b[0];
10     r[0] = (t1 - t2 - c) & BN_MASK2;
11     if (t1 != t2)         // leakage
12         c = (t1 < t2);
13     a++;
14     b++;
15     r++;
16     n--;
17 }
18 return c;
19 }
```

Figure 13: Unknown sensitive secret-dependent branch leaks from function bn_sub_words in OpenSSL 1.1.1g.

```
1  int bn_div_fixed_top(BIGNUM *dv, BIGNUM *rm,
2  const BIGNUM *num,
3  const BIGNUM *divisor, BN_CTX *ctx)
4  {
5  ...
6  t2 = (BN_ULLONG) d1 *q;
7  for (;;) {
8      if(t2 <= ((((BN_ULLONG) rem) << BN_BITS2) | n2)) //leakage
9          break;
10     q--;
11     rem += d0;
12     if (rem < d0)
13         break;      /* don't let rem overflow */
14     t2 -= d1;
15 }
16 ...
17 }
```

Figure 14: Unknown sensitive secret-dependent branch leaks from function bn_div_fixed_top in OpenSSL 1.1.1g.

VIII. Detailed Experimental Results

Here we present the detailed experimental results. Due to space limitation, we select the representative implementations of AES, DES, RSA, and ECDSA in mbedTLS 2.5, OpenSSL 1.1.0f, and OpenSSL 1.1.1. The results are representative to other versions. All the results will be made available in electronic format online when the paper is published.

In all the tables presented in this appendix, the mark "$*$" means timeout, which indicates more severe leakages. See §VI-A for the details. Also note that we round the calculated numbers of leaked bits to include one digit after the decimal point, so 0.0 really means very small amount of leakage, but not exactly zero. See §IV-B4 for the details of error estimate.

Table VI: Leakages in DES implemented by mbed TLS 2.5

File	Line No.	Function	# Leaked Bits	Type
des.c	441	mbedtls_des_setkey	0.9	DA
des.c	438	mbedtls_des_setkey	1.0	DA
des.c	438	mbedtls_des_setkey	1.0	DA
des.c	439	mbedtls_des_setkey	1.1	DA
des.c	439	mbedtls_des_setkey	1.0	DA
des.c	440	mbedtls_des_setkey	1.0	DA
des.c	446	mbedtls_des_setkey	0.9	DA
des.c	446	mbedtls_des_setkey	1.0	DA
des.c	444	mbedtls_des_setkey	1.0	DA
des.c	444	mbedtls_des_setkey	1.0	DA
des.c	443	mbedtls_des_setkey	1.0	DA
des.c	443	mbedtls_des_setkey	1.0	DA
des.c	444	mbedtls_des_setkey	1.0	DA
des.c	445	mbedtls_des_setkey	1.1	DA
des.c	448	mbedtls_des_setkey	0.9	DA

Table VII: Leakages in DES implemented by OpenSSL 1.1.0f

File	Line No.	Function	# Leaked Bits	Type
set_key.c	351	DES_set_key_unchecked	7.1	DA
set_key.c	353	DES_set_key_unchecked	8.8	DA
set_key.c	361	DES_set_key_unchecked	8.0	DA
set_key.c	362	DES_set_key_unchecked	5.7	DA
set_key.c	362	DES_set_key_unchecked	2.0	DA
set_key.c	364	DES_set_key_unchecked	3.5	DA
set_key.c	364	DES_set_key_unchecked	4.9	DA
set_key.c	365	DES_set_key_unchecked	0.4	DA

Table VIII: Leakages in DES implemented by OpenSSL 1.1.1

File	Line No.	Function	# Leaked Bits	Type
set_key.c	350	DES_set_key_unchecked	5.8	DA
set_key.c	350	DES_set_key_unchecked	6.6	DA
set_key.c	350	DES_set_key_unchecked	7.5	DA
set_key.c	350	DES_set_key_unchecked	6.4	DA
set_key.c	355	DES_set_key_unchecked	1.9	DA
set_key.c	355	DES_set_key_unchecked	3.1	DA

Table IX: Leakages in RSA implemented by mbed TLS 2.5

File	Line No.	Function	# Leaked Bits	Type
bignum.c	1617	mbedtls_mpi_exp_mod	0.9	CF
bignum.c	861	mbedtls_mpi_cmp_mpi	8.5	CF
bignum.c	862	mbedtls_mpi_cmp_mpi	7.7	CF
bignum.c	1167	mpi_mul_hlp	*	CF
bignum.c	828	mbedtls_mpi_cmp_abs	9.6	CF
bignum.c	829	mbedtls_mpi_cmp_abs	9.5	CF

Table X: Leakages in RSA implemented by mbed TLS 2.15.1

File	Line No.	Function	# Leaked Bits	Type
bignum.c	855	mbedtls_mpi_cmp_mpi	*	CF
rsa.c	184	rsa_check_context.isra.0	1.0	CF
bignum.c	825	mbedtls_mpi_cmp_abs	*	CF
bignum.c	197	mbedtls_mpi_copy	*	CF
bignum.c	1629	mbedtls_mpi_exp_mod	0.9	CF
bignum.c	829	mbedtls_mpi_cmp_abs	*	CF
bignum.c	859	mbedtls_mpi_cmp_mpi	*	CF
bignum.c	873	mbedtls_mpi_cmp_mpi	2.8	CF
bignum.c	874	mbedtls_mpi_cmp_mpi	2.7	CF
bignum.c	840	mbedtls_mpi_cmp_abs	8.2	CF
bignum.c	841	mbedtls_mpi_cmp_abs	8.7	CF
bignum.c	1201	mbedtls_mpi_mul_mpi	*	CF

Table XI: Leakages in RSA implemented by OpenSSL 1.0.2f

File	Line No.	Function	# Leaked Bits	Type
bn_lib.c	199	BN_num_bits	*	CF
bn_lib.c	200	BN_num_bits	15.6	CF
bn_lib.c	201	BN_num_bits	16.2	DA
bn_lib.c	673	BN_ucmp	*	CF
bio_asn1.c	482	__udivdi3	8.5	CF
bn_div.c	381	BN_div	*	CF
bn_div.c	456	BN_div	*	CF
bn_gcd.c	279	BN_mod_inverse	1.0	CF
bn_gcd.c	302	BN_mod_inverse	5.0	CF
bn_gcd.c	324	BN_mod_inverse	6.6	CF
bn_add.c	255	BN_usub	*	CF
bn_gcd.c	305	BN_mod_inverse	12.8	CF
bn_gcd.c	327	BN_mod_inverse	15.6	CF
bn_lib.c	203	BN_num_bits	15.2	DA
bn_lib.c	208	BN_num_bits	12.7	CF
bn_lib.c	209	BN_num_bits	*	DA
bn_lib.c	212	BN_num_bits	*	DA
bn_gcd.c	515	BN_mod_inverse	*	CF
bn_div.c	381	BN_div	2.7	CF
bn_div.c	439	BN_div	14.2	CF
bn_div.c	385	BN_div	11.4	CF
bn_div.c	381	BN_div	1.1	CF
bn_div.c	381	BN_div	1.0	CF
bn_div.c	469	BN_div	0.9	CF
bn_exp.c	676	BN_mod_exp_mont_consttime	1.0	CF
bn_exp.c	796	BN_mod_exp_mont_consttime	1.0	CF
bn_mont.c	262	BN_from_montgomery_word	*	DA
bn_mont.c	263	BN_from_montgomery_word	*	DA
bn_mont.c	264	BN_from_montgomery_word	*	DA
bn_mont.c	266	BN_from_montgomery_word	*	DA
bn_mont.c	276	BN_from_montgomery_word	*	DA
bn_mont.c	276	BN_from_montgomery_word	0.2	DA
bn_mont.c	276	BN_from_montgomery_word	0.2	DA
bn_mont.c	275	BN_from_montgomery_word	0.1	CF
bn_mont.c	282	BN_from_montgomery_word	*	CF
bn_asm.c	787	bn_sqr_comba8	*	CF
bn_asm.c	646	bn_mul_comba8	*	CF
bn_mont.c	201	BN_from_montgomery_word	0.0	CF

Table XII: Leakages in RSA implemented by OpenSSL 1.0.2k

File	Line No.	Function	# Leaked Bits	Type
bn_lib.c	199	BN_num_bits	*	CF
bn_lib.c	200	BN_num_bits	9.8	CF
bn_lib.c	201	BN_num_bits	12.1	DA
bn_shift.c	168	BN_lshift	5.3	CF
bn_lib.c	673	BN_ucmp	*	CF
bio_asn1.c	484	__udivdi3	6.4	CF
bn_div.c	381	BN_div	*	CF
bn_div.c	456	BN_div	*	CF
bn_gcd.c	279	BN_mod_inverse	1.0	CF
bn_gcd.c	302	BN_mod_inverse	8.4	CF
bn_gcd.c	324	BN_mod_inverse	8.6	CF
bn_add.c	255	BN_usub	*	CF
bn_gcd.c	327	BN_mod_inverse	14.2	CF
bn_gcd.c	305	BN_mod_inverse	13.8	CF
bn_lib.c	203	BN_num_bits	14.2	DA
bn_lib.c	208	BN_num_bits	13.5	CF
bn_lib.c	209	BN_num_bits	*	DA
bn_lib.c	212	BN_num_bits	*	DA
bn_gcd.c	515	BN_mod_inverse	*	CF
bn_div.c	381	BN_div	8.2	CF
bn_div.c	439	BN_div	*	CF
bn_div.c	385	BN_div	3.9	CF
bn_div.c	381	BN_div	1.0	CF
bn_div.c	381	BN_div	0.8	CF
bn_div.c	469	BN_div	1.6	CF
bn_exp.c	716	BN_mod_exp_mont_consttime	1.0	CF
bn_exp.c	836	BN_mod_exp_mont_consttime	1.1	CF
bn_mont.c	262	BN_from_montgomery_word	*	DA
bn_mont.c	263	BN_from_montgomery_word	*	DA
bn_mont.c	264	BN_from_montgomery_word	*	DA
bn_mont.c	266	BN_from_montgomery_word	*	DA
bn_mont.c	276	BN_from_montgomery_word	*	DA
bn_mont.c	282	BN_from_montgomery_word	*	CF
bn_mont.c	201	BN_from_montgomery_word	0.0	CF
bn_asm.c	646	bn_mul_comba8	*	CF
bn_asm.c	787	bn_sqr_comba8	*	CF

Table XIII: Leakages in RSA implemented by OpenSSL 1.1.0f

File	Line No.	Function	# Leaked Bits	Type
bn_lib.c	143	BN_num_bits_word	*	CF
bn_lib.c	144	BN_num_bits_word	*	CF
bn_lib.c	145	BN_num_bits_word	17.2	DA
bn_lib.c	1029	bn_correct_top	*	CF
bn_lib.c	639	BN_ucmp	*	CF
ct_b64.c	164	__udivdi3	5.9	CF
bn_div.c	330	BN_div	*	CF
bn_gcd.c	192	int_bn_mod_inverse	1.0	CF
bn_gcd.c	215	int_bn_mod_inverse	7.9	CF
bn_gcd.c	237	int_bn_mod_inverse	8.2	CF
bn_gcd.c	218	int_bn_mod_inverse	14.9	CF
bn_gcd.c	240	int_bn_mod_inverse	9.2	CF
bn_lib.c	147	BN_num_bits_word	*	DA
bn_lib.c	152	BN_num_bits_word	12.6	CF
bn_lib.c	153	BN_num_bits_word	*	DA
bn_lib.c	156	BN_num_bits_word	*	DA
bn_div.c	384	BN_div	17.2	CF
bn_div.c	330	BN_div	11.9	CF
bn_div.c	334	BN_div	3.8	CF
bn_exp.c	622	BN_mod_exp_mont_consttime	1.0	CF
bn_exp.c	741	BN_mod_exp_mont_consttime	1.0	CF
bn_mont.c	138	BN_from_montgomery_word	*	DA
bn_mont.c	139	BN_from_montgomery_word	*	DA
bn_mont.c	140	BN_from_montgomery_word	*	DA
bn_mont.c	142	BN_from_montgomery_word	*	DA
bn_mont.c	152	BN_from_montgomery_word	*	DA
bn_asm.c	733	bn_sqr_comba8	*	CF
bn_asm.c	592	bn_mul_comba8	*	CF
bn_mont.c	98	BN_from_montgomery_word	0.0	CF
bn_div.c	330	BN_div	0.3	CF
bn_div.c	330	BN_div	0.3	CF

Table XIV: Leakages in RSA implemented by OpenSSL 1.1.1

File	Line No.	Function	# Leaked Bits	Type
rsa_ossl.c	649	rsa_ossl_mod_exp	1.0	CF
bn_asm.c	592	bn_mul_comba8	0.3	CF
bn_exp.c	613	BN_mod_exp_mont_consttime	1.0	CF
bn_exp.c	745	BN_mod_exp_mont_consttime	1.0	CF

Table XV: Leakages in RSA implemented by OpenSSL 1.1.1g

File	Line No.	Function	# Leaked Bits	Type
	29	__udivdi3	1.3	CF
bn_div.c	374	bn_div_fixed_top	10.0	CF
bn_asm.c	413	bn_sub_words	*	CF
bn_div.c	374	bn_div_fixed_top	5.7	CF
rsa_ossl.c	654	rsa_ossl_mod_exp	1.7	CF
bn_exp.c	625	BN_mod_exp_mont_consttime	2.1	CF
bn_exp.c	745	BN_mod_exp_mont_consttime	2.0	CF
bn_div.c	378	bn_div_fixed_top	0.0	CF

Table XVI: Leakages in ECDSA implemented by OpenSSL 1.1.0f

File	Line No.	Function	# Leaked Bits	Type
bn_lib.c	1029	bn_correct_top	*	CF
bn_div.c	330	BN_div	8.4	CF
bn_div.c	330	BN_div	3.2	CF
bn_lib.c	144	BN_num_bits_word	0.0	CF
bn_lib.c	145	BN_num_bits_word	2.0	DA

Table XVII: Leakages in ECDSA implemented by mbedTLS 2.5

File	Line No.	Function	# Leaked Bits	Type
bignum.c	369	mbedtls_mpi_bitlen	7.5	CF
bignum.c	1167	mpi_mul_hlp	1.8	CF
bignum.c	861	mbedtls_mpi_cmp_mpi	1.1	CF
bignum.c	862	mbedtls_mpi_cmp_mpi	0.8	CF
bignum.c	861	mbedtls_mpi_cmp_mpi	1.8	CF
bignum.c	862	mbedtls_mpi_cmp_mpi	12.0	CF

Table XVIII: Leakages in ECDSA implemented by mbedTLS 2.15.1

File	Line No.	Function	# Leaked Bits	Type
bignum.c	395	mbedtls_mpi_bitlen	11.7	CF
bignum.c	840	mbedtls_mpi_cmp_abs	0.4	CF
bignum.c	1179	mpi_mul_hlp	9.7	CF
bignum.c	841	mbedtls_mpi_cmp_abs	0.5	CF