Abacus: Precise Side-Channel Analysis

Anonymous

***Abstract*—Side-channel attacks allow adversaries to infer sensitive information from non-functional characteristics. Prior side-channel detection work is able to 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. Existing tools do not provide information to evaluate a leak's severity, such as the number of leaked bits.**

To address this issue, we propose a new program analysis method to precisely quantify the leaked information in a single-trace attack through side-channels. It can identify covert information flows in programs that expose confidential information and can 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 use symbolic execution to generate these constraints and then run Monte Carlo sampling to estimate the number of leaked bits for each leakage site. By applying the Central Limit Theorem, we provide an error bound for these estimations.

We have implemented the technique in a tool called Abacus, which not only finds very fine-grained side-channel vulnerabilities but also estimates how many bits are leaked. Abacus outperforms existing dynamic side-channel detection tools in performance and accuracy. We evaluate Abacus on OpenSSL, mbedTLS, Libgcrypt, and Monocypher. Our results demonstrate that most reported vulnerabilities are difficult to exploit in practice and should be de-prioritized by developers. We also find several sensitive vulnerabilities that are missed by the existing tools. We confirm those vulnerabilities with manual checks and by contacting the developers.

I. Introduction

Side channels are ubiquitous in modern computer systems as sensitive information can leak through many mechanisms such as power, electromagnetic radiation, and even sound [1]–[5]. Among them, software side-channel attacks, such as cache attacks, memory page attacks, and controlled-channel attacks, are especially problematic as they do not require physical proximity [6]–[11]. These attacks arises from shared micro-architectural components in a computer processor. By observing inadvertent interactions between two programs, attackers can infer program execution flows that manipulate secrets and guess secrets such as encryption keys [12]–[15].

Many side-channel attacks originate in 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 data accesses and control flows, respectively.

Recent work [14], [16]–[20] can detect many side-channel vulnerabilities. For example, DATA [16] reports 2,248 potential leakage sites for the RSA implementation in OpenSSL 1.1.0f. After inspection, 1,510 leaks can be dismissed. But that leaves 460 data-access leaks and 278 control-flow leaks. Many of these vulnerabilities have not been fixed by developers for a variety of reasons. First, some vulnerable implementations perform better. For example, RSA implementations usually adopt the CRT optimization, which is faster but vulnerable to fault attacks [21]. Second, fixing vulnerabilities can introduce new weaknesses. Third, most vulnerabilities pose a negligible risk. Although some vulnerabilities result in the key being entirely compromised [21], [22], many other vulnerabilities are less severe in reality. Therefore, we need a proper quantification metric to assess the sensitivity of each side-channel vulnerability, so a developer can efficiently triage them.

Previous work, such as static methods based on abstract interpretation [20], [23], provides an upper bound on the amount of leakage, which verifies that an implementation is secure if it incurs zero leakage. However, these techniques cannot quantify the severity of a leak because they over approximate the leakage. For example, CacheAudit [20] reports that the upper-bound leakage of AES-128 exceeds the original key size. By contrast, dynamic methods take another approach by running a program in a real environment. Although they are precise in terms of true leakages, no existing tool can precisely assess the severity of production software vulnerabilities. **JL Can we say something more specific about dynamic techniques?**

To overcome these limitations, we propose a novel method to quantify information leakage precisely. We quantify the number of bits that can leak during a 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, they 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 has determined the input, which means all secret information (e.g., the entire 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 provides fine-grained information, but it is expensive to compute. Therefore, prior dynamic symbolic execution work [14], [18], [19] either analyzes only small programs or applies domain knowledge [14] to simplify the analysis. We examine the bottleneck of the trace-oriented symbolic execution and optimize it to work for real-world crypto-systems.

We have implemented the proposed technique in a tool called Abacus and demonstrated it on real-world crypto libraries, including OpenSSL, mbedTLS, and Monocypher. We collect execution traces of these libraries and apply symbolic execution to each instruction. We model each side-channel leak as a logic formula. These formulas precisely model side-

```
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

channel vulnerabilities. Then we use the conjunction of those formulas to model the leaks at a statement that appears in different location in the execution trace file (e.g., leaks inside a loop). Finally, we introduce a Monte Carlo sampling method to estimate the information leakage. The experimental results confirm that Abacus precisely identifies previously known vulnerabilities and reports 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 produces no false positives. The result also shows the newer version of crypto libraries leak less information than earlier versions. Abacus also discovers new vulnerabilities. With the help of Abacus, we confirm that some of these 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 that result from actual attack scenarios. Our approach differs from previous ones in that we model real attack scenarios for one execution. We model the information quantification problem as a counting problem and use a Monte Carlo sampling method to estimate the information leakage.
- We implement the method into a tool and apply it to several pieces of real software. Abacus successfully identifies previous unknown and known side-channel vulnerabilities and calculates the corresponding information leakage. Our results are useful in practice. The leakage estimates and the corresponding trigger inputs can help developers to triage and fix the vulnerabilities.

+ Abacus is publicly available at https://github.com/1c0e/Abacus. The repository also contains benchmarks, metadata, and raw results of our experiments.

II. Background and Threat Model

A. Address-based Side-channels

Side channels leak sensitive information unintentionally 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 depending on the values in *sensitive inputs*. As shown in Figure 1 and Figure 2, if a program executes different patterns of control transfers or data accesses when it processes different sensitive inputs, the program may be vulnerable to side-channel attacks. Different side-channels can be exploited to retrieve information at various granularities. For example, cache side channels observe cache accesses at the level of cache sets [11], cache lines [22], or finer granularities [27], [28]. Other types of side-channels, such as controlled-channel attack [6], can observe the memory accesses at the granularity of memory pages.

B. Threat Model

We assume that an attacker shares the same hardware platform with the target. The attacker attempts to retrieve sensitive information through address-based side-channel attacks. The attacker has no direct access to the target's memory or cache, but it can probe its memory or cache at each program point. In reality, the attacker will face many possible obstacles such as the noisy observations of the memory or cache. However, for this project, we assume the attacker has noise-free observations as in previous work [14], [18], [20]. The threat model captures most cache-based and memory-based side-channel attacks. We only consider deterministic programs and assume an attacker has access to the source code + or binary executable 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. Then, we introduce our model, the mathematical notation used in the paper, and our method.

A. Problem Setting

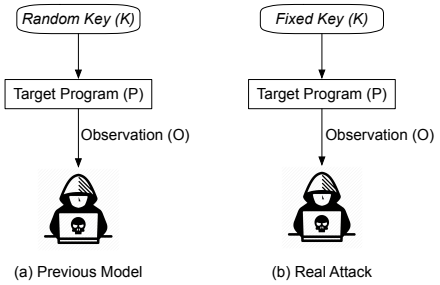
Existing static side-channel quantification works [17], [20], [29] define information leakage using max entropy or Shannon entropy. If zero bits of information leakage is reported, a program is secure. However, when a tool using these metrics reports leakage, it is the "average" leakage. In a real attack, 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

Consider a password checker sketched in Figure 3. The password checker takes an 8-byte char array (exclude NULL character) and checks if the input is the correct password. If an attacker uses a side-channel attack to determine that the code executes branch $\{1\}$, they can infer the password equals to "password", in which case the attacker retrieves the full password. Therefore, the total leaked information should be 64 bits, which equals to the size of the original sensitive input when the code executes branch 1.

However, prior static approaches cannot precisely capture the amount of 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 from the number of possible observations. Because the program has two



(a) Previous Model (b) Real Attack

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

branches, tools based on max-entropy will report the code has a $\log_2 2 = 1$ bit leakage.

Both approaches fail to tell how much information is leaked an actual execution. The problem [30] with existing methods is that their approaches do not consider 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), both Shannon entropy and max-entropy give an "average" estimate of the information leakage. However, generating random inputs is not the typical scenario for an adversary launching a side-channel attack. In a typical attack, the adversary wants to retrieve sensitive information, which is typically fixed (e.g., AES keys). The adversary performs their attack over and over again with fixed input and guess the value bit by bit (e.g., Kocher's timing attacks [31]), as in Figure 4 (b). We need a theory for dynamic analysis that says an attack leaks x bits of secret information, where x is useful in estimating the sensitive level of the vulnerability. However, all previous methods fail for real attack models. This is the first challenge we face **(Challenge C1)**. **JL Tis explaination of why previous techniques fail should also be in the introduction.**

B. Notation

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

We assume a program (β) has K as its sensitive input. K should be a finite set of keys. The program also takes known messages M as its input. During an AES encryption, for example, β is the encryption function. K is the set of all possible AES keys, and M represents the set consisting of all plaintext messages to be encrypted. In a real execution, an adversary may have some observations (O) of the program. Examples of those observations include timing, CPU usages, and electromagnetic signals (EM). This paper only uses 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. An attacker 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 restated as the following question:

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

C. Theoretical Analysis (**Solution to Challenge 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), which is defined as:

$$I(K;O) = \sum_{k \in K} \sum_{o \in O} p(k,o) \log_2 \frac{p(k,o)}{p(k)p(o)} \tag{1}$$

where $p(k, o)$ is the joint probability mass function of K and O. Alternatively, the mutual information can also be equivalently expressed as:

$$I(K;O) = H(K) - H(K|O) \tag{2}$$

$H(K|O)$ 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 made observations O.

$$H(K|O) = -\sum_{o \in O} p(o) \sum_{k \in K} p(k|o) \log_2 p(k|o) \tag{3}$$

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 they can infer by observing the memory access patterns (o). We come to the simple formulation [32], [33] that

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

Next we compare the Eq. (2) with the above formulation, we find $H(K)$ is the *Initial uncertainty* and $H(K|O)$ is *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

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

For a program (β) without any domain information, all possible sensitive inputs appear equally likely. 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

$$\begin{aligned} H(K;o) &= -\sum_{k\in K^o} p(k|o)\log_2 p(k|o) \\ &\quad -\sum_{k`\in(K\setminus K^o)} p(k'|o)\log_2 p(k'|o) \\ &= \sum_{k\in K^o} \frac{1}{|K^o|}\log_2 |K^o| \\ &= \log_2 |K^o| \end{aligned}$$

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*

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

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

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

The above definition can be understood in an intuitive way. Suppose an attacker guesses a 128-bit encryption key. Without any domain knowledge, they can find the key by performing an exhaustive search over 2^{128} possible keys. However, assume the program has a side-channel leakage site. After the program finishes execution, the attacker has some observations and only needs to find the key by performing an exhaustive search over 2^{120} possible keys. Then, we 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 this definition, if an attacker observes that the code in Figure 3 executes branch 1, then $K^{o^1} = \{\text{“password”}\}$. Therefore, the information leakage $L_{P(k)=o^1} = \log_2 2^{64} - \log_2 1 = 64$ bits, which means the key is entirely leaked. If the attacker observes the code hits branch 2, the leaked information is $L_{P(k)=o^2} = \log_2 2^{64} - \log_2 (2^{64} - 1) \approx 0$ bits.

As the size of input-sensitive information is usually public, the problem of quantifying the leaked information is equivalent to 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 obtain during a side-channel attack.

During an execution, a program (β) has many temporary values ($t_i \in T$). Once β (program), k (secret), and m (message, public) are determined, t_i is also fixed (for deterministic programs). 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: A control-flow path is secret-dependent if different input-sensitive keys (K) can lead to different branch conditions. We define a branch to be 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 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) cause access to different memory addresses. We use the model from CacheD [14]. The low L bits of the address are generally unimportant in side-channels.

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 of a secret-dependent data access.

IV. Scaling to Real-world Crypto Systems

In the previous section, we propose an information leakage definition for realistic attack scenarios to model two types of address-based side-channel leakages and showed how to quantify them by calculating the number of input keys (K^o) that satisfy the formulas. Intuitively, we can use symbolic execution to capture math formulas and model counting to obtain the number of satisfying input keys (K^o). However, preliminary experiments showed that this approach was far too expensive to use with real-world applications. In this section, we discuss the bottlenecks in this approach and propose a practical solution.

In general, Abacus faces the following performance challenges in *scaling to production-system crypto 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 high performance cost. Previous trace-oriented symbolic execution work [14], [30] has serious performance bottlenecks. As a result, these approaches either apply only to small programs [30] or require domain knowledge [34] to simplify the analysis. We implement the approach presented in §III and model the side-channels as formulas. While the tools can analyze some simple cases such as AES, it cannot handle complicated examples such as RSA. We observe that finding side-channels using symbolic execution differs from traditional symbolic execution, and it can be optimized to be as efficient as other methods.

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)

1) Interpret Instructions Symbolically: Existing binary analysis - tools + frameworks [35]–[37] translate machine instructions into intermediate languages (IR) to simplify analysis since the variety of machine instructions is enormous, and their semantics is complex. The Intel Developer Manual [38] documents more than 1000 different x86 instructions. Unfortunately, the IR layer, which reduces the workload of these tools, is not suitable for side-channels analysis because IR-based or source code side-channels analyses do not represent the executed instructions accurate enough to analyze fully their control and memory accesses. For example, a compiler may use conditional moves or bitwise 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 causes significant overhead [39]. Translating machine instructions into IR is time-consuming. For example, REIL IR [40], 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 abandoned IR and expended the effort to implement symbolic execution directly on x86 instructions. Table I shows that eliminating the IR reduces the number of instructions examined during analysis. Previous works [39] also adopted a similar approach to speed up fuzzing. Our implementation differs from that work in two aspects: 1) We use complete constraints. 2) We run the symbolic execution on one execution path each time. Our approach is approximately 30 times faster than using an IR (transferring ISA into IR and symbolically executing it).

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

> *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 approaches rely on satisfiability modulo theories (SMT) solvers (e.g, Z3 [41]) to find satisfying assignments to k_1 and k_2. While this is a universal approach to solving constraints, for constraints of this form, 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 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 improvement in SMT solvers in recent years, constraint solving remains one of the obstacles to scaling the analysis of real-world crypto-systems.

Our Solution: Instead of feeding the formula $f_a(k_1) \neq f_a(k_2)$ into a SMT solver, we randomly pick $k_1, k_2 \in K$ and test them if they satisfy the formula. Our solution is based on the following intuition. For most combination of (k_1, k_2), $f_a(k_1) \neq f_a(k_2)$. 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. This type of random algorithm works well for our problem.

B. Counting the Solutions

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 constraint using 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 its 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 an element is 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 fails as n grows.
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 outcome does not depend on the result of previous experiments. Also, as the amount of leaked information is calculated by a log function, we need not exactly count the number of solutions for a given constraint.

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) $\forall i, j \in \{1, \ldots, m\}$ *and* $i \neq j, \quad \pi(g_i) \cap \pi(g_j) = \varnothing$
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 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 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 change the problem of counting the number of solutions to a complicated constraint in a 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 by 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}, ..., K_{c_m}. We notice that c_i usually has fewer inputs than 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 solution. After that, we only generate random input numbers for the other input variables in constraint g_i. With this simple yet effective trick, we reduce the number of input while still ensuring accuracy.

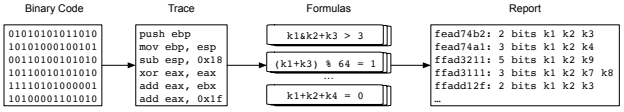


Figure 5: The workflow of Abacus.

4) Error Estimation: Our result has a 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.

Let n be the number of samples and n_s be the number of samples that satisfy the constraint C. Then we get $\hat{p} = \frac{n_s}{n}$. If we repeat the experiment multiple times, each time we 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 $\frac{\bar{p} - E(p)}{\sigma\sqrt{n}} \to N(0, 1)$. 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 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 have 95% confidence that the error of estimated leaked information is less than 1 bit. During the simulation, if n and p satisfy this inequality, the Monte Carlo simulation will terminate.

V. Design and Implementation

A. Design

Figure 5 shows the three steps of Abacus. First, we run the target program with a concrete input (sensitive information) under the dynamic binary instrumentation (DBI) framework to collect an execution trace. After that, we run the symbolic execution to capture fine-grained semantic information for each secret-dependent control-flow transfer and data access. Finally, we run Monte Carlo (MC) simulation 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 a dynamic binary instrumentation (DBI) tool to record execution traces and runtime information. Once the sensitive information is loaded into memory, we start to collect the trace. In this step, we mark variables and buffers that hold the sensitive data by

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

either annotating the source code (make_abacus_symbolic) or telling the DBI tool of the memory address and the length of secrets.

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 instruction that might be affected by the input sensitive data. Abacus works on one path at a time. The memory model is conceptually similar to other offline executors (e.g., SAGE [42] and the trace-based executor of BitBlaze [37]). That is, we use symbolic execution to track secrets. When secrets are loaded into the memory, Abacus starts to interpret instructions symbolically. We treat secrets as symbols (S). For other variables, we use concrete values (C) from the execution. We do not know which instruction may manipulate a secret until we execute it. For each instruction, if all its operands and implicit memory accesses are concrete values, we perform concrete calculation and update the destination with the concrete value according to the instruction semantics. Otherwise, we symbolically interpret the instruction and update the destination with a formula.

3) Leakage estimation: We change the information leakage quantification problem into the counting problem. We 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

Abacus consists of 16,729 lines of code in C++17 and Python. It has three components: an Intel Pin tool that collects the execution trace, the instruction-level symbolic execution engine, and the backend that estimates the information leakage.

Our current implementation supports most Intel 32-bit instructions that are essential to find memory-based side-channel vulnerabilities, including bitwise operations, control transfer, data movement, and logic instructions. The tool uses the real values to update the registers and memory for instructions that the implementation does not support. Therefore, the tool may miss some leakages but will not raise false positives.

VI. Evaluation

We evaluate Abacus on widely used crypto libraries including OpenSSL, mbedTLS, Libgcrypt and Monocypher. We mark variables and buffers that store a secret. 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 program into 32-bit x86 Linux executables with GCC 8.0 running on 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:

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

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

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 summarizes the results. Abacus finds 904 leaks in the crypto libraries. Among these 904 leaks, 241 are due to secret-dependent control-flow transfers and 663 are due to secret-dependent memory accesses.

Abacus also finds that most side-channel vulnerabilities leak very little information in practice, which confirms our initial assumption. However, Abacus finds some vulnera-

bilities with severe leakages. Prior research has confirmed that some of these vulnerabilities can be exploited in real attacks. With our tool, developers can distinguish non-critical "vulnerabilities" from severe ones.

Symmetric encryption implementations in OpenSSL and mbedTLS have significant leakage due to their lookup table implementations. Abacus confirms that all leakage comes from table lookups. The new implementation of OpenSSL uses four 1K tables with smaller entries instead of the previous 1K table. **JL Is this right? DW: 4K in the previous sentences?** The change is rather easy but significantly decreases 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 need not apply domain knowledge to simplify the analysis. Our tool identifies all leakage sites reported by CacheD [14] and find new leaks in less time. We also find newer versions of RSA in OpenSSL have fewer leaks. We will discuss the version changes and 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 that the error of the estimated leaked information is less than 1 bit, which gives the leakage quantification a *precision guarantee*, or 2) it cannot reach the termination condition after 10 minutes. **JL Doesn't this belong in the implementation section?** In the latter case, it means Abacus cannot estimate the amount of leakage with a probabilistic guarantee. We manually check these 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. + DATA [16] identifies side-channel leakages by finding differences in execution traces of the test program under *various secret inputs*. According to the original DATA paper, it uses 443 different traces to detect the side-channel vulnerabilities in symmetric cyphers and 450 different traces to detect the side-channel vulnerabilities in asymmetric cyphers. On the other hand, Abacus detects side-channel vulnerabilities from one execution trace. Abacus uses symbolic analysis to extract formulas that model each side-channel leakages. After that, we sample the formula with *various secret inputs* to detect and quantify each leakage site. In theory, DATA might have better code coverage than Abacus because it uses more execution traces, but Abacus has the following advantages. a) Abacus is faster than DATA. For example, it takes 234 minutes for DATA to analyze the RSA implementation in OpenSSL 1.1.0f. Abacus only spends 34 minutes, as shown in Table II. It takes 13 minutes and 20 minutes for DATA to detect the side-channel leakages in AES and DES, respectively. On the other hand, Abacus finishes its analysis in less than ten seconds while finding all the leakages reported by DATA. b) Because Abacus does not run the test program again when we have a new *secret input*, Abacus can test more input secrets on those formulas to achieve better precision. DATA reports 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 helps identify real sensitive leaks. For each leakage site, Abacus can provide concrete examples to trigger the issue and give an estimation to assess the severity level of the vulnerability.

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		

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 improve performance but can also introduce 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 reports the same leakages for both versions.

We find that 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 leakage than other versions. OpenSSL 1.1.1 uses 1KB lookup tables with 8-bit entries, unlike older versions that use a table with 32-bit entries. Our tool suggests a smaller lookup table might mitigate side-channel vulnerabilities. **JL Is my change right?**

2) Asymmetric Ciphers: RSA: We also evaluate Abacus on RSA. Due to the page limit, we do not present the detailed leakage report. As shown in Figure 6, the result indicates that the newer versions of OpenSSL leak less information than earlier versions. After version 0.9.7g, OpenSSL adopts a fixed-window mod_exp_mont implementation for RSA. With this 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 leakages effectively because the four newer versions have fewer leakages than version 0.9.7, which introduced this change. 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 to patch vulnerability CVE-2016-0702. Abacus finds OpenSSL 1.1.1 and 1.1.1g have significantly less leaked information than other versions. We check the ChangeLog of these two versions and find a claim that 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) contains some new leakages. We have contacted the developers and they have confirmed our findings.

Our quantification result shows vulnerabilities that leak significant amounts of information 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 indicates the vulnerability is quite serious. It was patched in OpenSSL 1.1.1. In Figure 8, control-flows transfers are replaced so there are no leaks in the function sqr_add_c2 in OpenSSL 1.1.1. We note that line 4 and 9 in Figure 7 both contain if branches. However, they are not leaks because most compilers use *add with carry* instruction to eliminate the branch. In addition, branches can be compiled into non-branch machine instructions with conditional moves. We notice a bitwise operation in Libgcrypt 1.8.5 is compiled to a conditional jump, which leads to a side-channel leakage. Therefore, source-level code reviews are not accurate enough to detect side-channels.

For vulnerabilities that leak less 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 the default option and rewrite most of the function. However, the leakage site still exists in OpenSSL 1.1.1, despite it is quite blunt. **JL What do you mean by blunt?**

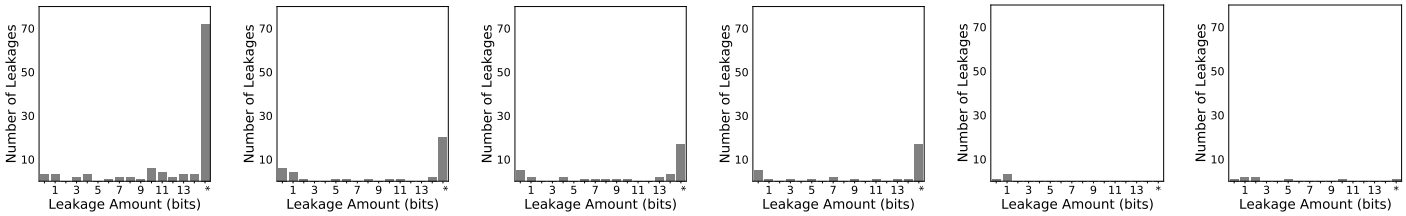
3) Monocypher: Monocypher is a small, easy to use cryptographic library with performance comparable to LibSodium [43] and NaCl [44]. We choose four ciphers that are designed to be side-channel resistant from the library. Because those ciphers have no data flow from secrets to branch conditions and load addresses. Monocypher should be safe under our threat models. We analyze those ciphers with Abacus, and it reports no leaks. This indicates that Abacus is effective for validating countermeasures.

VII. Discussions and Limitations

While recent work found many side-channel vulnerabilities, we note that many of them have not been patched by developers. Side-channels are inevitable in software and it would be difficult to fix all of them. We need a tool that estimates the sensitivity of each vulnerability so software engineers can focus on "severe" leakages. For example, Abacus reports that the modular exponentiation using square and multiply algorithms can leak more information than a key validation function.

Software developers can use Abacus 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 more precise in reporting true leakages as compared with other static methods [45], [46], obviously misses leakages on unexplored 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 a software engineer's 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 multiple side-channel attacks [31]. During the execution, a single leak point may execute multiple times and each time leak a different bit. In this case, Abacus reports that the vulnerability can leak the whole key. However, not every leak point inside a loop is severe. If a site in the loop leaks the same bit from the original key, and these leaks are not independent. Abacus captures most fine-grained information by modeling each leak during the execution as a formula and the conjunction of the formulas to describe its 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 can



(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

capture the dependency among the leaks and reports more precise leakage information.

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 did not find false positives caused by the imprecise memory model during our evaluation. Sampling introduces imprecision but with a probabilistic guarantee. However, during the evaluation, we find that Abacus cannot estimate the amount of leakage 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 leakages found by the previous work [14], [18], [19].

VIII. Related Work

There is a vast amount of work on side channel detection [14], [16]–[20], [47], mitigation [34], [48]–[55], information quantification [23], [29], [30], [56]–[61], and model counting [30], [62]–[65]. Here we only present only work closely related to ours. Due to space limit, we do not discuss related work on side-channel attacks.

A. Detection and Mitigation

CacheAudit [20] uses abstract domains to compute an over approximation of cache-based side-channel information leakage upper bound. However, it is difficult to judge the sensitive level of the side-channel leakage based on the leakage provided by CacheAudit. CacheS [19] improves on CacheAudit with new abstract domains that only track secret-related code. Like CacheAudit, CacheS cannot indicate the sensitive level of side-channel vulnerabilities. CaSym [18] introduces a static cache-aware symbolic reasoning technique to cover multiple paths in a target program. Again, their approaches cannot evaluate the sensitive level for each side-channel vulnerability, and it only work on small code snippets.

The dynamic approach, usually consists of taint analysis and symbolic execution, can perform a very precise analysis. CacheD [14] takes a concrete execution trace and runs symbolic execution on the trace to get the formula of each memory address. Therefore, CacheD is quite precise in avoiding false positives. However, CacheD is not able to detect secret-dependent control-flows. We adopted a similar approach to model the secret-dependent data accesses, but Abacus also finds secret-dependent control-flows and give a precise quantification of the leakage. DATA [16] detects address-based side-channel vulnerabilities by comparing different execution traces under various test inputs. After collecting execution traces, DATA aligns them and finds the differences. It uses statistical hypothesis testing to find true leakages. However, both imperfect trace alignment and statistical testing result that DATA can produce false positives. MicroWalk [17] uses mutual information (MI) between sensitive input and execution state to detect side-channels.

Both hardware [34], [48]–[51], [66] and software [45], [52]–[55] side-channels mitigation techniques have been proposed recently. Hardware countermeasures, including partitioning hardware resources [48], randomizing cache accesses [34], [51], and designing new architecture [67], require changes to complex processors and are complex to adopt. On the contrary, software approaches are usually easy to implement. Coppens et al. [53] uses a compiler to eliminate key-dependent control-flow transfers. Crane et al. [55] 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 [68], [69] and unifying control-flows [53].

B. Quantification

Proposed by Denning [70] and Gray [71], 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 bits of the information are leaked, the program is called non-interference. McCamant and Ernst [58] 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. [59] propose symbolic QIF. The goal of their work is to ensure a program is non-interference. They adopt an over approximation method to estimate the total information leakage and their method does not work for secret-dependent memory access side-channels. + Pasareanu et al. [61] combine symbolic analysis and Max-SMT solving to synthesize the concrete public input that can lead to the worst case leakage. They assume the target program has multiple different input secrets and calculate the average leakage for one-fixed public input. CHALICE [30] quantifies the leaked information for a given cache behavior. It symbolically reasons about cache behavior and estimates the amount of leaked information based on cache miss/hit. Their approach only scale to small programs, which limits its usage in real-world applications. On the contrary, Abacus assesses 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 refers to the problem of computing the number of models for a propositional formula (#SAT). There are two approaches to solving the problem, exact model counting and approximate model counting. We focus on approximate model counting since it is our approach. Wei and Selman [62] introduce ApproxCount, a local search based method using Markov Chain Monte Carlo (MCMC). ApproxCount has the better scalability than exact model counters. Other approximate model counter includes SampleCount [63], Mbound [64], and MiniCount [65]. Unlike ApproxCount, these model counters can give lower or upper bounds with guarantees. Despite the rapid development of model counters for SAT and some research [72], [73] on Modulo Theories model counting (#SMT), they cannot be directly applied to side channel leakage quantification. ApproxFlow [56] uses ApproxMC [74] for information flow quantification, but it has only been tested with small programs.

IX. Conclusion

This paper presents a novel method to quantify memory-based side-channel leakage. We implement the method in a prototype called Abacus and show its effectiveness in finding and quantifying side-channel leakage. With the new definition of information leakage that models actual side-channel attackers, quantifying the number of leaked bits helps understand the severity level of side-channel vulnerabilities. The evaluation confirms that 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] Y. Yarom, D. Genkin, and N. Heninger, "Cachebleed: a timing attack on openssl constant-time RSA," *Journal of Cryptographic Engineering*, vol. 7, no. 2, 2017.

[28] D. Moghimi, J. Van Bulck, N. Heninger, F. Piessens, and B. Sunar, "CopyCat: Controlled Instruction-Level Attacks on Enclaves," in *USENIX Security 20*.

[29] 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*.

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

[31] 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.

[32] 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.

[33] A. Askarov and S. Chong, "Learning is change in knowledge: Knowledge-based security for dynamic policies," in *IEEE 25th Computer Security Foundations Symposium (CSF) 2012*. IEEE, pp. 308–322.

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

[35] 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.

[36] 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.

[37] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "Bitblaze: A new approach to computer security via binary analysis," in *International Conference on Information Systems Security*. Springer, 2008, pp. 1–25.

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

[39] 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*.

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

[41] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'08/ETAPS'08, pp. 337–340.

[42] P. Godefroid, M. Y. Levin, D. A. Molnar *et al.*, "Automated whitebox fuzz testing." in *NDSS*, vol. 8, 2008, pp. 151–166.

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

[44] 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.

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

[46] 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.

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

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

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

[50] 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*.

[51] 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*.

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

[53] 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*.

[54] 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.

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

[56] 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.

[57] 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.

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

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

[60] 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*.

[61] C. S. Pasareanu, Q.-S. Phan, and P. Malacaria, "Multi-run side-channel analysis using symbolic execution and Max-SMT," in *IEEE 29th Computer Security Foundations Symposium (CSF) 2016*. IEEE, pp. 387–400.

[62] 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.

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

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

[65] 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.

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

[67] 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.

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

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

[70] D. E. Robling Denning, *Cryptography and Data Security*. Addison-Wesley Longman Publishing Co., Inc., 1982.

[71] 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.

[72] 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.

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

[74] 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.

[75] C. E. Shannon, "A mathematical theory of communication," *Bell system technical journal*, vol. 27, no. 3, pp. 379–423, 1948.
[76] 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.
[77] 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 [75]. 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 work on information leakage quantification typically use Shannon entropy [17], [76], min-entropy [32], and max-entropy [20], [77]. 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], [77] 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