Abacus: Precise, Scalable, and Fine-grained Side-channel Information Leakage Quantification for Production Software

Anonymous

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

To overcome this limitation, we propose a novel method to precisely quantify the leaked information from side-channel vulnerabilities. Our quantification method is dramatically different from previous methods, and the results are confirmed to be much more precise and usable in practice. We model an attacker's observation of each leakage site as a constraint. We scale symbolic execution to production software to generate the constraints and then run Monte Carlo sampling to estimate the number of leaked bits for each vulnerability. By using the Central Limit Theorem, we can also give the error bound for estimation.

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

I. Introduction

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

Regarding the root cause of software-based side channel attacks, many of them originate from two specific circumstances: data flow from secrets to load addresses and data flow from secrets to branch conditions. We refer to them as secret-dependent memory-accesses and control-flows, respectively. A central problem is eliminating these two code patterns. Recent works [14], [16]–[20], find plenty of side-channel vulnerabilities. For example, DATA [16] reports 2,246 potential leakage sites for the RSA implementation in OpenSSL. However, we find most of the reported vulnerabilities are not fixed because of the following reasons. First, many vulnerable implementations have better performance and are well-known for years. For example, symmetric encryptions like AES and DES use lookup tables (T-tables), which is fast but notoriously known to be vulnerable to side channels. As for asymmetric encryptions, many implementations of RSA, adopt the CRT optimization, which is faster but vulnerable to fault attacks [21]. Second, side-channels are numerous and it is hard to fix all these vulnerabilities, let alone the majority of them are negligible. That is, some vulnerabilities can result in the key being entirely compromised [21], [22], but many other vulnerabilities prove to be less severe in reality. Therefore, we need a proper quantification metric to assess the sensitive level of side-channel vulnerabilities.

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

To overcome these limitations, we propose a novel method to quantify information leakages precisely. Unlike previous works that only consider the "average" information leakage, we study the problem based on real attack scenarios. The average information leakage model assumes the target program has *variable* or *random* sensitive information as inputs when an attack is launched. However, for real-world attacks, an adversary may run the target problem with the *fixed* unknown sensitive information as the input. Therefore, the previous threat model cannot model real attack scenarios. In contrast, our method is more precise and fine-grained. We quantify the amount of leaked information as the cardinality of the set of possible inputs based on an attacker's observation to each

Network and Distributed Systems Security (NDSS) Symposium 2020
23-26 February 2020, San Diego, CA, USA
ISBN 1-891562-61-4
https://dx.doi.org/10.14722/ndss.2020.23xxx
www.ndss-symposium.org

leakage site. Before an attack, an adversary has a large but finite input space. Every time when the adversary observes a leakage site, he can eliminate some potential inputs and reduce the size of the input space. The smaller the input space is, the more information is obtained. In an extreme case, if the size of the input space reduces to one, the adversary can uniquely determine the input, which means all the secret information (e.g., the whole secret key) is leaked. By counting the number of distinct inputs, we can quantify the information leakage precisely.

We use constraints generated from symbolic execution to model the relation between the original sensitive input and the attacker's observations. Symbolic execution can provide fine-grained information but is usually believed to be an expensive operation in terms of performance. Therefore, existing dynamic symbolic execution works [14], [18], [19] either only analyze small programs or apply some domain knowledge [14] to simplify the analysis. We examine the bottleneck of the trace-oriented symbolic execution and optimize it to be scalable to real-world cryptosystems.

We apply the above technique and build a tool called Abacus, to discover potential information leakage sites as well as estimate how many bits they can leak for each leakage site. First, we collect dynamic execution traces for each target libraries and then run symbolic execution on each instruction. In this way, we model each side-channel leakage as a logic formula. The sensitive input is divided into several independent bytes, and each byte is regarded as a unique and free symbol. Those formulas can precisely model side-channel vulnerabilities. Then we use the conjunction of those formulas to model the same leaks in the source code but appears in the different location of the execution trace file (e.g., leakages inside a loop). Finally, we introduce Monte Carlo sampling method to estimate the single and combined information leakage.

We apply Abacus on both symmetric and asymmetric ciphers from real-world crypto libraries, including OpenSSL, mbedTLS and Monocypher. The experimental result confirms that Abacus can precisely identify previously known vulnerabilities, 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 has no false positives. Also, we show the widely deployed software countermeasures can mitigate side channels. Newer version of crypto libraries leak less amount of information compared to previous versions. Abacus also discovers new vulnerabilities. With the help of Abacus, we confirm that those vulnerabilities are severe.

In summary, we make the following contributions:

- We propose a novel method that can quantify fine-grained leaked information from side-channel vulnerabilities to match real attack scenarios. Our method is different from previous ones in that we model real attack scenarios more precisely, while the previous research only models the "average" or "random" case. We transfer the information quantification problem into a counting problem and use the Monte Carlo sampling method to estimate the information leakage.

```
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

- We implement the proposed method into a tool and apply it on several real-world software. Abacus successfully identifies memory-related side-channel vulnerabilities and calculates the corresponding information leakage. Our results are surprisingly different, much more useful in practice. The information leakage results provide detailed information that can help developers to fix the reported vulnerabilities.

II. Background and Threat Model

In this section, we first present an introduction to address-based side-channel attacks and show many of them are caused by two specific side-channel vulnerabilities: secret-dependent control-flow transfers and secret-dependent memory accesses. Therefore, we will focus on identifying and quantifying those leakages in the paper. After that, we discuss existing information leakage quantification metrics.

A. Address-based Side-channels

Side channels can leak sensitive information unconsciously through different execution behaviors caused by shared hardware components (e.g., CPU cache, TLB, and DRAM) in modern computer systems [24], [25].

For example, cached-based side-channels [10]–[12], [22], [22], [26], [27] rely on the time difference between cache misses and cache hits. We introduce two common attack strategies, namely Prime+Probe [11] and Flush+Reload [22]. Prime+Probe targets a single cache set. An attacker preloads the cache set with its own data and waits until the victim executes the program. If the victim accesses the cache set and evicts part of the data, the attacker will experience a slow measurement. While Flush+Reload targets a single cache line, it requires the attacker and victim share some memory. During the "flush" stage, the attacker flushes the "monitored memory" from the cache and also waits for the victim to access the memory, who will load the sensitive information to the cache line. In the next phase, the attacker reloads the "monitored memory". By measuring the time difference brought by cache hit and miss, the attacker can further infer the sensitive information. Some other types of side-channels target different hardware layers other than CPU cache. For example, the controlled-channel attack [6], where an attacker works in the kernel space, can infer sensitive data in shielding systems by observing the page fault sequences after restricting some code and data pages.

The key intuition is that above side-channel attacks happen when a program accesses different memory addresses if the program has different sensitive inputs. As shown in Figure 1 and Figure 2, if a program shows different patterns in control

transfers or data accesses when the program processes different sensitive inputs, the program could possibly have side channels vulnerabilities. Different kinds of side-channels can be exploited to retrieve information in various granularities. For example, cache channels can observe cache accesses at the level of cache sets [11], cache lines [22] or other granularities. Other kinds of side-channels like controlled-channel attack [6], can observe the memory access at the level of memory pages.

B. 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 [28]. Considering a char variable a with one byte storage size in a C program, its value ranges from 0 to 255. Assume a has a uniform distribution. If we observe that a equals 1, the probability of this observation is $\frac{1}{256}$. So we get $-\log(\frac{1}{256}) = 8$ bits information, which is exactly the size of a char variable in the C program.

Existing works on information leakage quantification typically use Shannon entropy [17], [29], min-entropy [30], and max-entropy [20], [31]. 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 the 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.

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

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

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

As it is easy to compute, most recent works [20], [31] 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 3. It has two secret-dependent control-flows, A and B.

In this paper we assume an attacker can observe the secret-dependent control-flows in Figure 3. 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 doo is executed, and $\neg B$ for function doo is not executed. Now the question is how much information can be leaked from the above code if an attacker knows which branch is executed?

```
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  }
8  if (t2 > 0){               // leakage site B
9      doo();
10 }
```

Figure 3: Side-channel leakage

Table I: 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 I describes the probability of each observation. We use the above three types of leakage metrics to calculate the amount of leaked information for leak A and leak B.

Min Entropy. As $p_{Amax} = 0.9998$ and $p_{Bmax} = 0.5$, with the definition, min-entropy equals to

$$min\text{-}entropy_A = -\log_2 0.9998 = 0.000 \text{ bits}$$
$$min\text{-}entropy_B = -\log_2 0.5 = 1.000 \text{ bits}$$

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

$$max\text{-}entropy = -\log_2 2 = 1.000 \text{ bits}$$

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

$$\begin{aligned} Shannon\text{-}entropy_A &= -(0.9998 * \log_2 0.9998 \\ &\quad + 0.0002 * \log_2 0.0002 \\ &= 0.000 \text{ bits} \\ Shannon\text{-}entropy_B &= -(0.5 * \log_2 0.5 \\ &\quad + 0.5 * \log_2 0.5 \\ &= 1.000 \text{ bits} \end{aligned}$$

In the next section, we will show that these measures work well only theoretically in a static analysis setting. Generally, they do not apply to dynamic analysis or real settings. We will present that the static or theoretical results could be dramatically different from the real world, and we do need a better method to quantify the information leakage from a practical point of view.

C. Threat Model

We consider an attacker shares the same hardware resource with the victim. The attacker attempts to retrieve sensitive information via memory-based side-channel attacks. The attacker has no direct access to the memory or cache but can probe the memory or cache at each program point. In reality,

the attacker will face many possible obstacles, including the noisy observations, limited observations on the memory or cache. However, for this project, we assume the attacker can have noise-free observations like previous works [14], [18], [20]. The threat model captures most of the cache-based and memory-based side-channel attacks. We only consider the deterministic program for the project and assume an attacker has access to the source code of the target program.

III. Abacus Leakage Definition

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

A. Problem Setting

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

```
1 char key[9] = input();
2 if(strcmp(key, "password")){  // leakage site C
3     pass();                   // branch 1
4 }else{
5     fail();                   // branch 2
6 }
```

Figure 4: A dummy password checker

We consider a dummy password checker shown in Figure 4. The password checker will take an 8-byte char array (exclude NULL character) as the input and check if the input is the correct password. If an attacker knows the code executes branch $\{1\}$ by side-channel attacks, he can infer the password equals to "password", in which case the attacker can entirely retrieve the password. Therefore, the total leaked information should be 64 bits, which equals to the size of the original sensitive input if the code executes branch 1.

However, previous static-based approaches cannot precisely reflect the amount of the leakage. According to the definition of Shannon entropy, the leakage will be $\frac{1}{2^{64}} * \log_2 \frac{1}{2^{64}} + \frac{2^{64}-1}{2^{64}} * \log_2 \frac{2^{64}-1}{2^{64}} \approx 0$ bits. Because the program has two branches, tools based on max-entropy will report the code has $\log_2 2 = 1$ bit leakage.

Both approaches fail to tell how much information is leaked during the execution precisely. The problem with existing methods is that they are static-based and input values and real runtime information are neglected by their approaches. They assume an attacker runs the program multiple times with many different or random sensitive inputs. As shown in Figure 5(a), previous models, both Shannon entropy and max entropy, give an "average" estimate of the information leakage. However, it is not the typical scenario for an adversary to launch a side-channel attack. When a side-channel attack happens, the adversary wants to retrieve the sensitive information, in which case the sensitive information is fixed (e.g., AES keys). The adversary will run the attack over and over again with fixed input and guess the value bit by bit (e.g., Kocher's timing attacks [32]), as in Figure 5(b). We want to have a theory for dynamic analysis that if the theory says an attack leaks x bits of secret information from a side-channel vulnerability, then x should be useful in estimating the sensitive level of the vulnerability. However, the above methods all fail in real attack models. This is the first challenge we face **(Challenge C1)**.

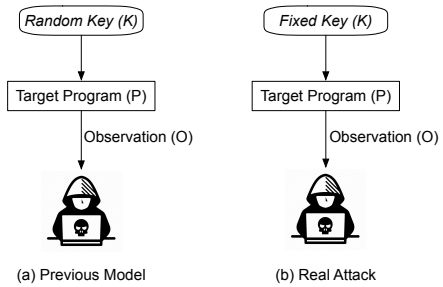


Figure 5: The gap between the real attack and previous models

B. Notations

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

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

With the above definitions, we define the following mapping between β, K, M, and O:

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

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

$$K^o = \{k \in K \mid \beta(k, m) \rightarrow o\}$$

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

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

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

Now we present our metric to quantify the amount of leaked information from dynamic analysis.

In information theory, the mutual information (I) is a measure of the mutual dependence between two variables. Here 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 $P(k_i, o_i)$ is the joint discrete distribution 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 with the condition O. It quantifies the uncertainty of K given the value of O. In other word, the conditional entropy $H(K|O)$ marks the uncertainty about K after an adversary has gained some observations (O).

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

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

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

Next we compare the Eq. (2) with the above slogan, 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

$$Leakage = H(K;o) = H(K) - H(K|o)$$

For a program (β) without knowing any domain information, all possible sensitive inputs should appear equally. Therefore, for any $k \in K$, $p(k) = \frac{1}{|K|}$. So 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 wants to guess a 128-bit encryption key from a program. Without any domain knowledge, he can find the key by performing exhaustive search over 2^{128} possible keys. However, the program has a side-channel leakage site. After the program finishes execution, the attacker get some leaked information and only need to find the key by performing exhaustive search over 2^{120} possible keys. Then we can say that 8 bits of the information is leaked. In this example, 2^{128} is the size of K and 2^{120} is the size of K^o.

With the definition, if an attacker observes that the code in Figure 4 runs the branch 1, then the $K^{o^1} = \{\text{“password”}\}$. Therefore, the information leakage $L_{P(k)=o^1} = \log_2 2^{64} - \log_2 1 = 64$ bits, which means the key is totally leaked. If the attacker observes the code hits branch 2, the leaked information is $L_{P(k)=o^2} = \log_2 2^{64} - \log_2(2^{64}-1) \approx 0$ bit.

We can also calculate the leaked information from the sample code in Figure 3. As the size of input sensitive information is usually public. The problem of quantifying the leaked information has been transferred into the problem of estimating the size of input key $|K^o|$ under the condition $o \in O$. The result is shown in Table II. We can see that some branches (e.g., A) or traces leak much more information than some others. Those kinds of branches can be vulnerable when an attacker's data is incorporate. For example, the code will skip some of the calculation if the value is 0 in big number multiplication.

In contrast, an *average* estimate based on random secret input information is around 1 bit, as shown in §II-B and Table I, is not very useful in practice as an attacker is able to get much more leaked information in some attack scenarios.

Table II: New leakage modeling results

Observation (o)	A	$\neg A$	B	$\neg B$
Number of Solutions	65526	10	32768	32768
Leaked Information (bits)	0.0	14.7	1.0	1.0

D. Our Conceptual Framework

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

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

In the paper, we consider two code patterns that can be exploited by an attacker, *secret-dependent control transfers* and *secret-dependent data accesses*. In other words, an adversary has observations based on control-flows and data accesses.

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

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

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

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

We consider a data access is secret-dependent if:

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

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

With the above definitions, we can model an attacker's observation by a series of math formulas. For example, in Figure 3, if an attacker observes the code executes the branch 1, we have $c_5 : k_1 + k_2 < 8$ to describe an attacker's knowledge and $K^{o5} = \{k_1, k_2 \,|\, (k_1 + k_2) < 8\}$. If an attacker observes the code executes the branch 2, we have $c_8 : k_1 - k_2 > 0$ and $K^{o8} = \{k_1, k_2 \,|\, (k_1 - k_2) > 0\}$.

IV. Scalable to Real-world Crypto Systems

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

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

- Symbolic execution (**Challenge C2**)
- Constraint solving (**Challenge C3**)
- Counting the number of items in K^o (**Challenge C4**)

A. Trace-oriented Symbolic Execution

While symbolic execution can capture fine-grained semantics of programs, it is also notorious for its unbearable performance cost. Previous trace-oriented symbolic execution based works [14], [33] all have large performance bottlenecks. As a result, those approaches either only apply to small-size programs [33] or apply some domain knowledge to simplify the analysis. We implement the approach presented in §III and model the side-channels as formulas. While the tool can finish analyzing some simple cases like AES, it can not handle complicated cases like RSA. We observe that finding side-channels using symbolic execution is different from traditional general symbolic execution and can be optimized to be as efficient as other methods with approaches below.

1) Interprete Instructions Symbolically: Existing binary analysis tools [34], [35] usually translate machine instructions into intermediate languages (IR) to simplify analysis. The reason is that the number of machine instructions is enormous, and the semantics of each instruction is complex. Intel Developer Manual [36] introduces more than 1000 different x86 instructions. IR typically has fewer instructions compared to the original machine ISA. However, the IR layer, which predigest the implementation and reduce the workload of those tools, is not suitable for side-channels analysis. Memory-based side-channels are very low issues. So, in general, IR-based or source code side-channels analyses are not accurate enough. In many cases, compilers can 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 design also introduces significant overhead [37]. Transferring machine instructions into IR is time-consuming. For example, REIL IR [38], 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. If we assume the time of symbolically executing one instruction is constant, the design of adopting IR layers can introduce large overhead.

Our Solution to Challenge C2: We adopt the approach from QSYM [37] and implement the symbolic execution directly on the top of x86 instructions. Table III shows that eliminating the IR layer can reduce the number of instructions executed during the analysis.

Table III: 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)

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

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

Existing approach relies on satisfiability modulo theories (SMT) solvers (e.g, Z3 [39]) to find satisfying k_1 and k_2. We argue that while it is a universal approach to solving constraints with SMT solvers, for constraints with the above formats, using custom heuristics and testing is much more efficient in practice. Constraint solving is a decision problem expressed in logic formulas. SMT solvers transfer the inputted SMT formula into the boolean conjunctive normal form (CNF) and feed it into the internal boolean satisfiability problem (SAT) solver. The translation process, called "bit blasting", is time-consuming. Also, as the SAT problem is a well-known NP-complete problem, it is also hard to deal when it comes to realistic uses with huge formulas. Despite the rapid development of SMT solvers in recent years, constraint solving remains one of the obstacles to achieve the scalability for real-world cryptosystems.

Our Solution to Challenge C3: Instead of feeding the formula $f_a(k_1) \neq f_a(k_2)$ into a SMT solver, we just randomly pick up $k_1, k_2 \in K$ and test them if they can satisfy the formula. Our solution is based on the following intuition. For most combination of (k_1, k_2), the formula $f_a(k_1) \neq f_a(k_2)$ holds. As long as f_a is not a constant function, such k_1, k_2 must exist. For example, suppose each time we only have 5% chance to find such k_1, k_2, then after we test with different input combination with 100 times, we have $1-(1-0.05)^{100} = 99.6\%$ chance find such k_1, k_2. Such random algorithms work well for our problem.

B. Counting the Number

According to Definition 1 introduced in §III, the problem of quantifying the amount of leaked information can be reduced to the problem of computing the number of items in K^o. However, we find that while there are various propositional model counters (e.g., #SAT), they are not sufficient scalable for production cryptosystem analysis.

One straightforward method approximating the number of solutions is based on Monte Carlo sampling. However, the number of satisfying values could be exponentially small. Consider the formula $f_i \equiv k_1 = 1 \wedge k_2 = 2 \wedge k_3 = 3 \wedge k_4 = 4$, where k_1, k_2, k_3, and k_4 each represents one byte in the original sensitive input buffer, there is only one satisfying solution of total 2^{32} possible values, which requires exponentially many samples to get a tight bound. Monte Carlo method also suffers from the curse of dimensionality. For example, the length of an RSA private key can be as long as 4096 bits. If we take each byte (8 bits) in the original buffer as one symbol, the formula can have as many as 512 symbols.

Our Solution to Challenge C4: We adopt multiple-step Monte Carlo sampling methods to count the number of possible inputs that satisfy the logic formula groups. The key idea is to split those constraints into several small formulas and sample them independently.

C. Information Leakage Estimation

In this section, we present the algorithm to calculate the information leakage based on Definition 1 (§III), answering to **Challenge C4**.

1) Problem Statement: For each leakage site, we model it with a math formula constraint with the method presented in §III-D. Suppose the address of the leakage site is ξ_i, we use c_{ξ_i} to denote the constraint. For multiple leakage sites, we take the conjunction of those constraints to represent those leakage sites.

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 has 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 native method for approximating the result is to pick elements k from K and check if the element also contained in K^o. Assume q elements satisfy this condition. In expectation, we can use $\frac{k}{q}$ to approximate the value of $\frac{|K|}{|K^o|}$.

However, as discussed in §IV-B, the above sampling method will typically fail in practice due to the following two problems:

1) The curse of dimensionality. $c_t(\xi_1, \ldots, \xi_n)$ is the conjunction of many constraints. Therefore, the input variables of each constraints will also be the input variables of the $c_t(\xi_1, \ldots, \xi_n)$. The sampling method will fail as n increases. For example, if the program has 2 byte input equals to 2, the whole search space is a 256^2 cube. If we want the sampling distance between each point equals to d, we need $256^2 d$ points. If the program has 10 byte input, we need $256^{10} d$ points if we still we want the sampling distance equals to d.
2) The number of satisfying assignments could be exponentially small. According to Chernoff bound, we need exponentially many samples to get a tight bound. On an extreme situation, if the constraint only has one unique satisfying solution, the simple Monte Carlo method cannot find the satisfying assignment even after sampling many points.

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

1) $c_t(\xi_1, \xi_2, \ldots, \xi_n)$ is the conjunction of several short constraints c_{ξ_i}. The set containing the input variables of c_{ξ_i} is the subset of the input variables of

$c_t(\xi_1, \xi_2, \ldots, \xi_n)$. Some constraints have completely different input variables from other constraints.

2) Each time when we sample $c_t(\xi_1, \xi_2, \ldots, \xi_n)$ with a point, the sampling result is *Satisfied* or not *Not Satisfied*. The result is randomly generated in a way that does not depend on the result in previous experiments. Also, as the amount of leaked information is calculated by $\log$ function, we do not need to precisely 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) $\forall i, j \in \{1, 2, \ldots, m\}$ *and* $i \neq j$, $\pi(g_i) \cap \pi(g_j) = \varnothing$
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 that we want to reduce the dimensions. Consider the example in the previous section,

$$c : (k_1 = 1) \wedge (k_2 = 2) \wedge (k_3 > 4) \wedge (k_3 - k_4 > 10)$$

A good partition of F would be

$$g_1 : (k_1 = 1) \quad g_2 : (k_2 = 2) \quad g_3 : (k_3 > 4) \wedge (k_3 - k_4 > 10)$$

So instead of sampling in the four dimension space, we can sample each constraint in the less dimension space and combine them together with Theorem 1 .

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

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

With Theorem 1, we can transfer the problem of counting the number of solutions to a complicated constraint in high-dimension space into counting solutions to several small constraints. The algorithm to compute the Maximum Independent Partition of the $c_t(\xi_1, \xi_2, \ldots, \xi_n)$ is shown in Appendix B.

3) Multiple Step Monte Carlo Sampling: After we split those constraints into several small constraints, we count the number of solutions for each constraint. Even though the dimension has been significantly reduced after the previous step, this is still a #P problem. For our project, we apply the approximate counting instead of exact counting for two reasons. First, we do not need to have a very precise result of the exact number of total solutions since the information is defined with a logarithmic function. We do not need to distinguish between constraints having 10^{10} or $10^{10} + 10$ solutions; they are very close to after taking logarithmic. Second, the precise model counting approaches, like Davis-Putnam-Logemann-Loveland (DPLL) search, have difficulty scaling up to large problem sizes.

We apply the "counting by sampling" method. The basic idea is as follows. 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 less numbers of input compared to g_i. For example, if c_{i_j} has only one 8-bit input variable, we can find the exact solution set $K_{c_{i_j}}$ of c_{i_j} by trying every possible 256 solutions. After that, we can only generate random input numbers for the rest input variables in constraint g_i. With this simple yet effective trick, we can reduce the number of input while still ensure the accuracy. The detailed algorithm is shown in Appendix C.

4) Error Estimation: In this part, we will show that even we use approximating methods, our result can have 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 can get $\hat{p} = \frac{n_s}{n}$. If we repeat the experiment multiple times, each time we can get a $\hat{p}$. As each $\hat{p}$ is independent and identically distributed, according to the central limit theorem, the mean value should follow normal distribution $\frac{\bar{p} - E(p)}{\sigma\sqrt{n}} \to N(0, 1)$. 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 the calculate the standard deviation, we can claim that we have 95%[1] confidence that the error $\Delta p = \bar{p} - E(p)$ falls in the interval:

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

Since we use $L = \log_2 p$ to estimate the amount of leaked information, we can have the following error propagation formula $\Delta L = \frac{\Delta p}{p \ln 2}$ by taking the derivative from Definition 1. For Abacus, we want the error of estimated leaked information (ΔL) to be less than 1 bit. So we can 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 the above inequation, Monte Carlo simulation will terminate.

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

V. Design and Implementation

In this section, we describe the design of Abacus by focusing on how our design solves the challenges discussed in the previous section.

A. Design

Abacus has three steps, as shown in Figure 6. First, we run the target program with a concrete input (sensitive information) under the dynamic binary instrumentation (DBI) frameworks to collect execution traces. After that, we run the symbolic execution to capture the fine-grained semantic information of each secret-dependent control-flow transfers and data-accesses. Finally, we run Monte Carlo (MC) simulations to estimate the amount of leaked information.

1) *Execution trace generation.* The design goal of Abacus is to estimate the information leakage as precisely as possible. Therefore, we sacrifice the soundness for precision in terms of program analysis. We run the target binary under dynamic binary instrumentations (DBI) to to record execution traces and the runtime information. Once sensitive information is loaded into memory, we start to collect the trace.
2) *Instruction level symbolic execution.* We model attackers' observations from side-channel vulnerabilities with logic formulas. Each formula captures the fined-grained information between input secrets and leakage sites. In consideration of precision and performance, we remove the intermediate language(IR) layer of the symbolic execution. Also, the engine only symbolically executes the instruction that might be affected by the input key. We use random testing instead of SMT solvers to find satisfying variables.
3) *Leakage estimation.* We transfer the information leakage quantification problem into the problem of counting the number of assignments that satisfy the formulas which model the observations from attackers. 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 gives us the *precision guarantee*.

B. Implementation

We implement Abacus with 16,729 lines of code in C++ and Python. It has three components, Intel Pin tool that can collect the execution trace, the instruction-level symbolic execution engine, and the backend that can estimate the information leakage. The breakdown is shown in Table V-B. The tool can also report the memory address of the leakage site. To assist developers to fix the bugs, we also have several Python scripts that can report the leakage location in the source code with the debug information and the symbol information. A sample report can be found in the appendix.

Our current implementation supports most Intel 32-bit instructions, including bitwise operations, control transfer, data movement, and logic instructions, which are essential in finding memory-based side-channel vulnerabilities. For other instructions that the current implementation does not support, the tool will use the real values to update the registers and memory cells. Therefore, the tool may miss some leakages but will not give us any new false positives. As our proposed method is architecture-independent, there is no fundamental difficulties to extend the tool to support 64-bit architecture.

Table IV: 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

VI. Evaluation

We evaluate Abacus on real-world crypto libraries including OpenSSL, mbedTLS and Monocypher. OpenSSL is the most commonly used crypto libraries in today's software. mbedTLS (previous known as PolarSSL) is designed to be easy to understand and fit on embedded devices. We also evaluate Monocypher, a new cryptographic library that resists to most side-channel attacks. Monocypher is designed to have no secrets dependent indices and no secret dependent branches. Therefore, Monocypher should be secure under our threat model.

We build the source code into 32-bit x86 Linux executables with GCC 8.0 under Ubuntu 14.04. Although we use use symbol information to track back leakage sites into the source code, our tool can work on stripped binaries as well. We develop a Pin tool based on Intel Pin (version 3.7) to record the execution trace. We run our experiments on a 2.90GHz Intel Xeon(R) E5-2690 CPU with 128GB RAM memory. The execution time is calculated on a single-core. During our evaluation process, we are interested in the following aspects:

1) **Identifying side-channels leakages.** The first step of Abacus is to identify side-channel 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? Are the numbers 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 V shows the overview of evaluation results. Abacus finds 883 leakages in total from real-world crypto libraries. Among those 883 leak points, 205 of them are leaked due to secret-dependent control-flow transfers and 678 of them are leaked due to secret-dependent memory accesses.

Abacus also identifies that most side-channel vulnerabilities leak very little information in practice, which confirms our initial assumptions. However, we do find some vulnerabilities that Abacus reports with severe leakages. Some of them have been confirmed by existing research that those vulnerabilities can be exploited to realize real attacks. Without our tool, developers will not be able to distinguish those "vulnerabilities" from severe ones and ignore others for sure easily.

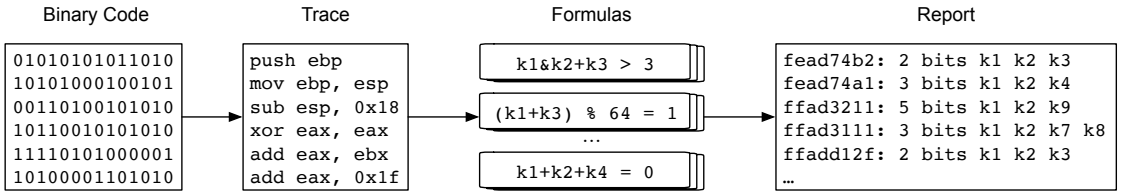


Figure 6: The workflow of Abacus.

Table V: Evaluation results overview. We evaluate two different versions of mbedTLS, five different versions of OpenSSL and Monocypher 3.0. CF represents secret-dependent control-flow transfers and DF represents secret-dependent data-flow transfers. A summary of vulnerabilities with the amount of leaked information can be found in Appendix §F.

Algorithm	Version	Leakage Sites	CF	DF	# Instructions	Max Leakage	Sym. Exe.	Monte Carlo
						bits	ms	ms
AES	mbedTLS 2.5	68	0	68	39,855	8.6	512	1,052
AES	mbedTLS 2.15	68	0	68	39,855	9.1	520	1,057
AES	OpenSSL 0.9.7	75	0	75	1,704	10.6	231	9,199
AES	OpenSSL 1.0.2f	88	0	88	1,350	12.0	36	1,924
AES	OpenSSL 1.0.2k	88	0	88	1,350	12.5	35	1,961
AES	OpenSSL 1.1.0f	88	0	88	1,420	12.6	36	2,161
AES	OpenSSL 1.1.1	88	0	88	1,586	4.4	43	1,631
DES	mbedTLS 2.5	15	0	15	4,596	1.1	58	162
DES	mbedTLS 2.15	15	0	15	4,596	1.0	57	162
DES	OpenSSL 0.9.7	6	0	6	2,976	7.6	163	4,677
DES	OpenSSL 1.0.2f	8	0	8	2,593	9.8	166	6,509
DES	OpenSSL 1.0.2k	8	0	8	2,593	10.1	165	5,975
DES	OpenSSL 1.1.0f	8	0	8	4,260	8.8	182	5,292
DES	OpenSSL 1.1.1	6	0	6	8,272	7.5	229	5,152
							seconds	seconds
Chacha20	Monocypher 3.0	0	0	0	149,353	0	2	0
Poly1305	Monocypher 3.0	0	0	0	1,213,937	0	15	0
Argon2i	Monocypher 3.0	0	0	0	4,595,142	0	37	0
Ed25519	Monocypher 3.0	0	0	0	5,713,619	0	271	0
							minutes	minutes
RSA	mbedTLS 2.5	6	6	0	22,109,246	9.6	39	41
RSA	mbedTLS 2.15	12	12	0	24,484,441	8.7	44	251
RSA	OpenSSL 0.9.7	107	105	2	17,002,523	17.2	23	428
RSA	OpenSSL 1.0.2f	38	27	11	14,468,307	16.2	29	436
RSA	OpenSSL 1.0.2k	36	27	9	15,285,210	14.2	40	714
RSA	OpenSSL 1.1.0f	31	22	9	16,390,750	17.2	34	490
RSA	OpenSSL 1.1.1	27	20	7	18,207,020	14.9	7	501
Total		886	219	667	139,733,961		214m	2,861m

All symmetric encryption implementations in OpenSSL and mbedTLS have significant leakages due to the lookup table implementation. Every leakages found during the evaluation belong to the type of secret-dependent memory accesses. Abacus finds several leakage sites for both implementations of DES and AES in OpenSSL and mbedTLS. Abacus confirms that all those leakages come from table lookups. mbedTLS 2.15 and 2.5 have the same implementation of both DES and AES, so they have the same leakage report. One proper fix would be a scalar bit-sliced implementation. However, we do not see the bit-sliced implementation of AES and DES in various versions of mbedTLS. However, we find the new implementation of OpenSSL instead uses typical four 1K tables. It only uses one 1K of the tables. This implementation is rather easy but does somehow decrease the total amount of leaked information as the quantification result shown in the next section.

We also evaluates our tool on the RSA implementation. With the optimization introduced in §IV, we do not apply any domain knowledge to simplify the analysis. Therefore, our tools can not only identify all the leakage sites reported by CacheD [14], but find new leakages in a shorter time. We also find the newer versions of RSA in OpenSSL tend to have fewer leakages detected by Abacus. We will discuss the version changes and corresponding leakages in §VI-C2.

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

B. Comparison with the Existing Tools

Abacus is designed to quantify side-channel leakages. But it can detect side-channels leakages as well. In this section, we

Table VI: 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		

compare Abacus with the existing trace-based side-channel detection tools.

As shown in Table VI, Abacus not only discovers all the leakage sites reported by CacheD [14], but also find many new ones. CacheD fails to detect many vulnerabilities for two reasons. First, CacheD can only detect secret-dependent memory access vulnerabilities. But Abacus can detect secret-dependent control-flows as well. Second, CacheD uses some domain knowledge to simplify symbolic execution and has to trim the traces before processing, which does not introduce false positives, but can neglect some vulnerabilities. The table VI shows that Abacus is three times faster than CacheD. As the time of symbolic execution grow 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] need 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 analysis the RSA of implementation in OpenSSL 1.1.0f. Abacus only spends 34 minutes according to Table V. Also, DATA reports report 278 control-flow and 460 memory-access leaks. Among those leakages, they found two vulnerabilities. On the contrary, Abacus can report how many bits is actually leaked for each vulnerability, which eases the pain to identify real sensitive leaks.

C. Case Studies

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

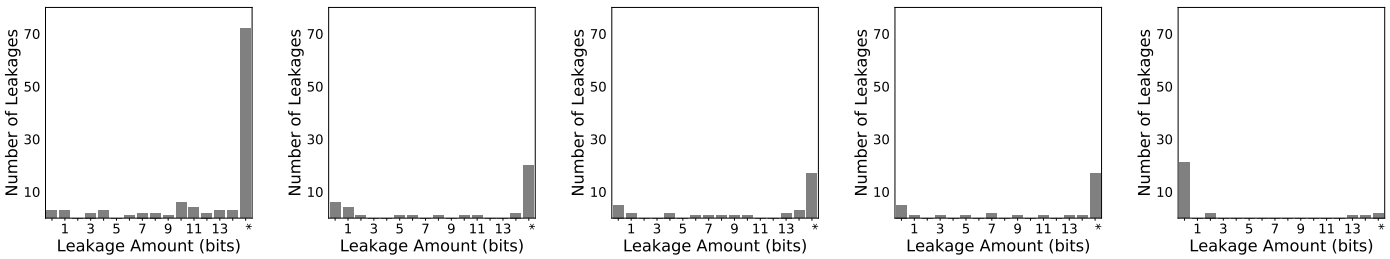
According to Abacus, we find DES implementations in both mbedTLS and OpenSSL have several sensitive information leakages in the key schedule function. However, leakages in OpenSSL are more severe. We do not see any mitigation in the new version. We think it is not seen as worth the engineering efforts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

Abacus shows that the AES in OpenSSL 1.1.1 has less amount of leakages compared to other versions by our tools. (e.g., the max leakage of AES in OpenSSL 1.1.1 is 4.4 bits, but other version have leakages that can leak can around 10 bits.) We find that OpenSSL 1.1.1 version instead uses the 1KB lookup tables with 32 bit entries like older versions, it uses a tables with 8 bit entries. In other words, our tools shows that lookup tables with smaller entries will leak less amount of information. Our tool suggests a smaller lookup table can mitigate side-channels vulnerabilities. For example, Shown in Appendix 10, Abacus identify seven leakages from function mbedtls_internal_aes_decrypt. However, Abacus reports leakage 1, 2, 3 leak more information compared to leakage 4, 5, 6, 7. We check the source code and find leakage 1, 2, 3 use secret to access the lookup table RT0, RT1, RT2, RT3, which is 8K each. On the contrary, leakage 4, 5, 6, 7 each access a smaller lookup table (2K).

2) Asymmetric Ciphers: RSA: We also evaluate Abacus on RSA. We find developers are more interested in fixing side-channel vulnerabilities for RSA implementations. We test five versions of OpenSSL (0.9.7, 1.0.2f, 1.0.2k, 1.1.0f, 1.1.1). The result, as shown in Figure 7, indicates that the newer version of OpenSSL leaks less amount of information compared to the previous versions. After version 0.9.7g, OpenSSL adopts a fixed-window mod_exp_mont implementation for RSA. With the new design, the sequence of squares and multiples and the memory access patterns are independent of the secret key. Abacus's result confirms the new exponentiation implementation has effectively mitigated most of leakages because the other four versions have fewer leakages compared with version 0.9.7. OpenSSL version 1.0.2f, 1.0.2k and 1.1.0f almost have the same amount of leakage. We check the changelog and find only one change for patching the vulnerability CVE-2016-0702. Abacus find OpenSSL 1.1.1 version has significantly less amount of leaked information compared to other versions. We check the changelog of OpenSSL 1.1.1 and find it claims the new RSA implementation adopts "numerous side-channel attack mitigation", which proves the effectiveness of our quantifying method.

Our quantification result shows vulnerabilities that leak more information identified by Abacus are more likely to be fixed in the updated version. As presented in Figure 7, 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 8, 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 times. Abacus thinks the vulnerability is quite serious. The vulnerability has been patched in OpenSSL 1.1.1. Seen in Figure 9, control-flows transfers are replaced. So there is no leaks in the function sqr_add_c2 in OpenSSL 1.1.1. We mention that even line 4 and 9 in Figure 8 both have if branches. However, it is not a secret-dependent control-flow transfers because most compilers will use *add with carry* instruction to remove the branch. Besides, branches can also be compiled to non-branch machine instructions like conditional moves. Therefore, simple code reviews are not accurate enough to detect side-channels.

On the other hand, for those 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_constime to replace original function BN_mod_exp_mont. Abacus detects a minor vulnerability in the original function that can leaks the last bit of the big number **m**. In the updated version, developers make



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

Figure 7: RSA implementations in different versions of 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 8: 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 9: Macro sqr_add_c2 in OpenSSL 1.1.1

the fixed windows become the default option and rewrite most parts of the function. However, the vulnerability still exits in OpenSSL 1.1.1, despite the vulnerability is blunt. Details of the vulnerability can been seen in Appendix§ D.

Even for the up-to-date version of OpenSSL, Abacus still find several side-channel leakages. One of the vulnerabilities has the similar leaked branch as the one in § D. However, as the branch is inside a loop and a bit shift function causes the branch leak different bits from the sensitive buffer. The vulnerability is far severe by our tool. Details of the leak can be found in Appendix §E and Table XIV.

3) Monocypher: Monocypher is a small, easy to use cryptographic library with a comparable performance as LibSodium [40] and NaCl [41]. 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. Therefore, Monocypher should be safe under our threat models. We analyze those ciphers with Abacus, and it reports no leaks from the implementation. This indicates that Abacus can be used for countermeasure confirmation.

VII. Discussions and Limitations

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

Abacus can be used by software developers to find sensitive vulnerabilities and reason about countermeasures. Abacus estimates the amount of leaked information for each side-channel leakage in one particular execution. Abacus is useful for software engineers to test programs and fix vulnerabilities. Abacus works on native x86 execution traces. The design, which is very precise in terms of true leakages compared to other static source code method [42], [43], can omit some leakages on other traces. The amount of leaked information also depends on the secret key. However, as the tool is intended for debugging and testing. we think it is software engineers' responsibility to select the input key and trigger the path in which they are interested. For the crypto software, this is typically not a problem since virtually all keys follow the same or similar computational paths.

We use the amount of leaked information to represent the sensitivity level of each side-channel vulnerability. Although imperfect, Abacus produces a reasonable measurement for each leak. For example, the simple modular exponentiation is notoriously famous for various side-channel attacks [32]. During the execution, the single leak points will be executed multiple times and each time it leaks one different bit. Therefore, Abacus reports that the vulnerability can leak the whole key. However, not every leak point inside a loop is severe. Because very often the leakages may leak the same bit in the original key and those leaks are not independent. Abacus can capture the most fine-grained information by modeling each leak during the execution as a math formula and by using the conjunction of those formulas to describe the total effect.

Abacus reaches full precision if the number of estimated leaked bits equals to Definition 1. According to the threat model, the length of the secret is known, so the only estimated value is the number of possible secrets under the attacker's observation ($|K^o|$). During the symbolic execution, Abacus may lose precision from the memory model it uses in theory. However, we do not find false positives caused by the imprecise memory model during the evaluation. During the symbolic

execution, for each constraint it generates, the tool will check the correctness of the constraint by giving the constraint real values (the value of sensitive keys). Sampling can also introduce imprecision but with a probabilistic guarantee, as long as the sampling can terminate under the Central Limit Theorem (CLT) condition in probability theory. However, during the evaluation, we find that Abacus cannot estimate the amount of leakage for some leakage sitesin a reasonable time, which means the number of K^o is very small. According to Definition 1, it means the leakage is very severe.

VIII. Related Work

There is a vast amount of work on side channel detection [14], [16]–[20], [44], mitigation [45]–[53], information quantification [23], [33], [54]–[57], and model counting [33], [58]–[61]. Here we only present the closely related work to ours. Due to space limit, we do not include side-channel attacks work.

A. Detection and Mitigation

There are a large number of works on side-channel vulnerability detections in recent years. CacheAudit [20] uses abstract domains to compute the over approximation of cache-based side-channel information leakage upper bound. However, it is less useful to judge the sensitive level of the side-channel leakage based on the leakage provided by CacheAudit. CacheS [19] improves the work of CacheAudit by proposing the novel abstract domains, which only track secret-related code. Like CacheAudit, CacheS cannot provide the information to indicate the sensitive level of side-channel vulnerabilities. CacheSym [18] introduces a static cache-aware symbolic reasoning technique to cover multiple paths for the target program. Still, their approaches cannot assess the sensitive level for each side-channel vulnerability does not scale to real-world crypto libraries.

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

Both hardware [45]–[49] and software [50]–[53] side-channels mitigation methods have been proposed recently. Hardware countermeasures, including parting the hardware computing resource [45], randomizing cache accesses [46], [49], and designing new architecture [62], which need to change the hardware and is usually hard to adopt in reality. On the contrary, software approaches are usually easy to implement. Coppens et al. [51] introduced a compiler-based approach to eliminate key-dependent control-flow transfers. Crane et al. [53] mitigated side-channels by randomizing software. As for crypto libraries, the basic idea is to eliminate key-dependent control-flow transfers and data accesses. Common approaches include bit-slicing [63], [64] and unifying control-flows [51].

B. Quantification

Proposed by Denning [65] and Gray [66], Quantitative Information Flow (QIF) aims at providing an estimation of the amount of leaked information from the sensitive information given the public output. If zero bit of the information is leaked, the program is called non-interference. McCamant and Ernst [56] 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. [57] propose symbolic QIF. The goal of their work is to ensure the program is non-interference. They adopt an over approximation way of estimating the total information leakage and their method does not work for secret-dependent memory access side-channels. CHALICE [33] quantifies the leaked information for a given cache behavior. CHALICE symbolically reason about cache behavior and estimate the amount of leaked information based on cache miss/hit. Their approach can only scale to small programs, which limits its usage in real-world applications. On the contrary, Abacus can assess the sensitive level of side-channels with different granularities. It can also analyze side-channels in real-world crypto libraries.

C. Model Counting

Model counting usually refers to the problem of computing the number of models for a propositional formula (#SAT). There are two directions solving the problem, exact model counting and approximate model counting. We focus on approximate model counting since it shares similar idea as our approach. Wei and Selman [58] introduce ApproxCount, a local search based method using Markov Chain Monte Carlo (MCMC). ApproxCount has the better scalability compared to exact model counters. Other approximate model counter includes SampleCount [59], Mbound [60], and MiniCount [61]. Compared to ApproxCount, those model counters can give lower or upper bounds with guarantees. Despite the rapid development of model counters for SAT and some research [67], [68] on Modulo Theories model counting (#SMT). They cannot be directly applied to side channel leakage quantification. ApproxFlow [54] uses ApproxMC [69] for information flow quantification, but it's only tested with small programs while Abacus can scale to production crypto libraries.

IX. Conclusion

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

REFERENCES

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

[2] M. Kar, A. Singh, S. Mathew, A. Rajan, V. De, and S. Mukhopadhyay, "8.1 improved power-side-channel-attack resistance of an aes-128 core via a security-aware integrated buck voltage regulator," in *2017 IEEE International Solid-State Circuits Conference (ISSCC)*. IEEE, 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 *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[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 *2015 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 *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[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 *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[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 *26th {USENIX} Security Symposium ({USENIX} Security 17)*, 2017.

[10] D. Gruss, R. Spreitzer, and S. Mangard, "Cache template attacks: Automating attacks on inclusive last-level caches," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015.

[11] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 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 *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, ser. SP '11, 2011.

[14] S. Wang, P. Wang, X. Liu, D. Zhang, and D. Wu, "CacheD: Identifying cache-based timing channels in production software," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017.

[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 *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[17] J. Wichelmann, A. Moghimi, T. Eisenbarth, and B. Sunar, "Microwalk: A framework for finding side channels in binaries," in *Proceedings of the 34th Annual Computer Security Applications Conference*, ser. 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 (S&P)*, 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 *28th USENIX Security Symposium (USENIX Security 19)*, 2019.

[20] G. Doychev, D. Feld, B. Kopf, L. Mauborgne, and J. Reineke, "CacheAudit: A tool for the static analysis of cache side channels," in *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*. USENIX, 2013.

[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 *23rd USENIX Security Symposium (USENIX Security 14)*, 2014.

[23] M. Backes, B. Köpf, and A. Rybalchenko, "Automatic discovery and quantification of information leaks," in *2009 30th IEEE Symposium on Security and Privacy*, 2009.

[24] 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.

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

[26] 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.

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

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

[29] 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.

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

[31] G. Doychev and B. Köpf, "Rigorous analysis of software countermeasures against cache attacks," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2017, 2017.

[32] 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.

[33] S. Chattopadhyay, M. Beck, A. Rezine, and A. Zeller, "Quantifying the information leak in cache attacks via symbolic execution," in *Proceedings of the 15th ACM-IEEE International Conference on Formal Methods and Models for System Design*, ser. MEMOCODE '17. ACM, 2017.

[34] 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.

[35] 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.

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

[37] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM: A practical concolic execution engine tailored for hybrid fuzzing," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

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

[39] 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. Springer-Verlag, 2008.

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

[41] 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.

[42] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, "Verifying constant-time implementations," in *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, 2016.

[43] 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.

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

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

[46] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ser. ISCA '07. ACM, 2007.

[47] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers, "A hardware design language for timing-sensitive information-flow security," *SIGPLAN Not.*, vol. 50, no. 4, 2015.

[48] 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 *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14. ACM, 2014.

[49] M. Werner, T. Unterluggauer, L. Giner, M. Schwarz, D. Gruss, and S. Mangard, "Scattercache: Thwarting cache attacks via cache set randomization," in *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, 2019.

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

[51] B. Coppens, I. Verbauwhede, K. D. Bosschere, and B. D. Sutter, "Practical mitigations for timing-based side-channel attacks on modern x86 processors," in *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, ser. SP '09. IEEE Computer Society, 2009.

[52] 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.

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

[54] 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.

[55] 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.

[56] S. McCamant and M. D. Ernst, "Quantitative information flow as network flow capacity," in *PLDI 2008: Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation*, 2008.

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

[58] 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.

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

[60] C. P. Gomes, A. Sabharwal, and B. Selman, "Model counting: A new strategy for obtaining good bounds," in *AAAI*, 2006, pp. 54–61.

[61] 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.

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

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

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

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

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

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

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

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

APPENDIX A
AES LOOKUP TABLES LEAKAGE

```
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

APPENDIX B
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 $G = \{g_1, g_2, \ldots, g_m\}$

for $i \leftarrow 1$ **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**
      $g_j \leftarrow g_i \wedge g_{\xi_i}$
      break
    end
    Insert c_{ξ_i} to G
  end
end

APPENDIX C
ALGORITHM TO COMPUTE THE NUMBER OF SATISFYING ASSIGNMENTS

APPENDIX D
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.

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 $n \leq \frac{6p}{1-p}$ **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
$|K_{g_i}| \leftarrow n_s|K|/(n * r * range(k))$

```
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

APPENDIX E
UNKNOWN LEAKS IN OPENSSL 1.1.1

Shown in Table XIV, 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 unknown vulnerability. Despite the following vulnerability leaks through the same code patterns at line 3 as the vulnerability shown in §D, Abacus shows the following code can leak around 15 bits from the original key.

APPENDIX F
DETAILED EXPERIMENTAL RESULTS

Here we present the detailed experimental results. Due to space limitation, we select the representative implementations of AES, DES, and RSA 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.

```
1  int BN_mod_exp_mont(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 }
14
```

Figure 12: BN_mod_exp_mont in OpenSSL 1.1.1

```
1  while (!BN_is_bit_set(B, shift)) { /* note that 0 < B */
2      shift++;
3      if (BN_is_odd(X)) {
4          if (!BN_uadd(X, X, n))
5              goto err;
6      }
7      ...
8      if (!BN_rshift1(X, X))  // It causes the leak severe!
9          goto err;
10     ...
11 }
```

Figure 13: Unknown sensitive secret-dependent branch leaks from function int_bn_mod_inverse in OpenSSL 1.1.1. Same as the example in §D, the code can leak the last digit from big number **X**. However, the leak is more severe because of the function BN_rshift1. Each time function BN_rshift1 will shift **X** right by one and places the result in **X**. Therefore, an attacker can infer multiple bits of **X** by observing the branch at line 3.

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-C4 for the details of error estimate.

Table VII: 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 VIII: 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 IX: 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 X: 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 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	399	rsa_ossl_private_decrypt	0.0	CF
bn_lib.c	555	BN_ucmp	*	CF
bn_gcd.c	199	int_bn_mod_inverse	1.0	CF
bn_gcd.c	247	int_bn_mod_inverse	14.9	CF
bn_gcd.c	225	int_bn_mod_inverse	12.3	CF
ct_b64.c	168	__udivdi3	0.1	CF
bn_div.c	374	bn_div_fixed_top	*	CF
bn_lib.c	955	bn_correct_top	2.6	CF
ct_b64.c	168	__memset_sse2_rep	0.0	CF
ct_b64.c	168	__memset_sse2_rep	0.0	CF
ct_b64.c	168	__memset_sse2_rep	0.0	DA
ct_b64.c	168	__memset_sse2_rep	0.0	DA
bn_exp.c	317	BN_mod_exp_mont	1.0	CF
bn_asm.c	592	bn_mul_comba8	2.1	CF
bn_exp.c	383	BN_mod_exp_mont	0.9	CF
bn_lib.c	453	BN_bn2binpad	0.0	DA
bn_lib.c	450	BN_bn2binpad	0.0	CF
rsa_oaep.c	180	RSA_padding_check_PKCS1_OAEP_mgf1	0.0	DA
rsa_oaep.c	180	RSA_padding_check_PKCS1_OAEP_mgf1	0.0	DA
rsa_oaep.c	176	RSA_padding_check_PKCS1_OAEP_mgf1	0.0	CF
string3.h	90	SHA1_Final	0.0	CF
rsa_oaep.c	200	RSA_padding_check_PKCS1_OAEP_mgf1	0.0	CF
rsa_oaep.c	209	RSA_padding_check_PKCS1_OAEP_mgf1	0.0	CF
rsa_oaep.c	250	RSA_padding_check_PKCS1_OAEP_mgf1	0.0	CF
rsa_oaep.c	253	RSA_padding_check_PKCS1_OAEP_mgf1	0.0	CF
ct_b64.c	168	__memset_sse2_rep	0.0	DA
ct_b64.c	168	__memset_sse2_rep	0.0	DA

Table XV: Leakages in AES implemented by mbed TLS 2.5

File	Line No.	Function	# Leaked Bits	Type
aes.c	536	mbedtls_aes_setkey_enc	7.9	DA
aes.c	536	mbedtls_aes_setkey_enc	7.6	DA
aes.c	536	mbedtls_aes_setkey_enc	7.3	DA
aes.c	536	mbedtls_aes_setkey_enc	7.5	DA
aes.c	729	mbedtls_internal_aes_encrypt	3.9	DA
aes.c	729	mbedtls_internal_aes_encrypt	8.2	DA
aes.c	729	mbedtls_internal_aes_encrypt	4.2	DA
aes.c	729	mbedtls_internal_aes_encrypt	8.0	DA
aes.c	729	mbedtls_internal_aes_encrypt	4.3	DA
aes.c	729	mbedtls_internal_aes_encrypt	4.1	DA
aes.c	729	mbedtls_internal_aes_encrypt	8.6	DA
aes.c	729	mbedtls_internal_aes_encrypt	8.1	DA
aes.c	729	mbedtls_internal_aes_encrypt	7.6	DA
aes.c	729	mbedtls_internal_aes_encrypt	3.7	DA
aes.c	729	mbedtls_internal_aes_encrypt	8.4	DA
aes.c	729	mbedtls_internal_aes_encrypt	7.4	DA
aes.c	729	mbedtls_internal_aes_encrypt	8.0	DA
aes.c	729	mbedtls_internal_aes_encrypt	4.2	DA
aes.c	729	mbedtls_internal_aes_encrypt	3.9	DA
aes.c	729	mbedtls_internal_aes_encrypt	4.1	DA
aes.c	730	mbedtls_internal_aes_encrypt	3.9	DA
aes.c	730	mbedtls_internal_aes_encrypt	7.6	DA
aes.c	730	mbedtls_internal_aes_encrypt	4.0	DA
aes.c	730	mbedtls_internal_aes_encrypt	7.6	DA
aes.c	730	mbedtls_internal_aes_encrypt	4.1	DA
aes.c	730	mbedtls_internal_aes_encrypt	3.7	DA
aes.c	730	mbedtls_internal_aes_encrypt	7.6	DA
aes.c	730	mbedtls_internal_aes_encrypt	7.9	DA
aes.c	730	mbedtls_internal_aes_encrypt	8.1	DA
aes.c	730	mbedtls_internal_aes_encrypt	8.2	DA
aes.c	730	mbedtls_internal_aes_encrypt	7.2	DA
aes.c	730	mbedtls_internal_aes_encrypt	3.9	DA
aes.c	730	mbedtls_internal_aes_encrypt	7.6	DA
aes.c	730	mbedtls_internal_aes_encrypt	3.8	DA
aes.c	730	mbedtls_internal_aes_encrypt	4.1	DA
aes.c	730	mbedtls_internal_aes_encrypt	4.1	DA
aes.c	733	mbedtls_internal_aes_encrypt	4.0	DA
aes.c	733	mbedtls_internal_aes_encrypt	4.0	DA
aes.c	733	mbedtls_internal_aes_encrypt	4.3	DA
aes.c	733	mbedtls_internal_aes_encrypt	4.0	DA
aes.c	733	mbedtls_internal_aes_encrypt	3.9	DA
aes.c	733	mbedtls_internal_aes_encrypt	4.0	DA
aes.c	733	mbedtls_internal_aes_encrypt	4.2	DA
aes.c	733	mbedtls_internal_aes_encrypt	4.0	DA
aes.c	733	mbedtls_internal_aes_encrypt	3.9	DA
aes.c	733	mbedtls_internal_aes_encrypt	4.1	DA
aes.c	733	mbedtls_internal_aes_encrypt	4.2	DA
aes.c	733	mbedtls_internal_aes_encrypt	4.1	DA
aes.c	733	mbedtls_internal_aes_encrypt	4.0	DA
aes.c	733	mbedtls_internal_aes_encrypt	3.6	DA
aes.c	733	mbedtls_internal_aes_encrypt	4.0	DA
aes.c	733	mbedtls_internal_aes_encrypt	4.0	DA
aes.c	735	mbedtls_internal_aes_encrypt	1.9	DA
aes.c	735	mbedtls_internal_aes_encrypt	2.0	DA
aes.c	735	mbedtls_internal_aes_encrypt	2.0	DA
aes.c	735	mbedtls_internal_aes_encrypt	2.1	DA
aes.c	741	mbedtls_internal_aes_encrypt	2.1	DA
aes.c	741	mbedtls_internal_aes_encrypt	2.0	DA
aes.c	747	mbedtls_internal_aes_encrypt	1.9	DA
aes.c	741	mbedtls_internal_aes_encrypt	2.0	DA
aes.c	753	mbedtls_internal_aes_encrypt	2.1	DA
aes.c	741	mbedtls_internal_aes_encrypt	1.8	DA
aes.c	747	mbedtls_internal_aes_encrypt	2.2	DA
aes.c	747	mbedtls_internal_aes_encrypt	2.0	DA
aes.c	753	mbedtls_internal_aes_encrypt	2.0	DA
aes.c	747	mbedtls_internal_aes_encrypt	1.9	DA
aes.c	753	mbedtls_internal_aes_encrypt	1.9	DA
aes.c	753	mbedtls_internal_aes_encrypt	1.9	DA