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 work on software side-channel detections can 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 each leakage 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 information. 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 fined-grained vulnerability leakage information. We evaluated Abacus on OpenSSL and Mbed TLS. 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 confirmed 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 channels, such as cache attacks, memory page attacks, and controlled-channel attacks, are especially common and have been studied for years [6]–[11]. These vulnerabilities result from vulnerable software and shared hardware components. By observing the outputs or hardware behaviors, attackers can infer the program execution flow that manipulates secrets and guess the secrets such as encryption keys [12]–[15].

Various countermeasures have been proposed to defend against software-based side-channel attacks. Hardware-level solutions, such as reducing shared resources, adopting oblivious RAM, and using transnational memory [14], [16]–[18], need new hardware features or changes in modern complex computer systems, which is impractical and hard to adopt in reality. Therefore, a more promising and universal direction is software countermeasures, detecting and eliminating side-channel vulnerabilities from code base.

Regarding the root cause of software-based side channels, many of them originate from the following 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-access and control-flow, respectively. A central problem is how to identify these two code patterns automatically. Recent works [14], [16], [19]–[22] adopt static and dynamic analysis to detect side-channels. They are capable to find potential leak sites in real-world software, but fail to report how severe each potential leakage could be. Moreover, many of the reported vulnerabilities are typically hard to exploit and leak very little information. For example, DATA [16] reports 2,246 potential leakage site for the RSA implementation in OpenSSL. After some inspections, 1,510 are dismissed, but it still leaves 460 data-access leakages and 278 control-flow leakages. For software developers, it is hard to fix all these vulnerabilities, let alone the majority of them are negligible. That is, some vulnerabilities can be exploited to recover the full secret keys [23], but many other vulnerabilities prove to be less severe in reality.

To assess the sensitive level of side-channel vulnerabilities, we need a proper quantification metric. Static methods [22], [24], usually with abstract interpretation, can give a leakage upper bound, which is useful to justify the implementation is secure when they report zero or little leakage. However, they cannot indicate how serious the leakage is because of the over-approximation method they apply. For example, CacheAudit [22] 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 actual 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 leakage precisely. Unlike previous works that only consider the "average" information leakage, we study the problem based on real attack scenarios. The average information assumes that the target program has *variable* or *random* sensitive information as input when an attack is launched. However, for real-world attacks, an adversary may run the target problem again and over again with *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 attackers' observations.

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 determine the input information uniquely, 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 to model the relation between the original sensitive input and the attacker's observations. We run the instruction level symbolic execution on the whole execution trace to generate the constraints. 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 based works [14], [20], [21] either only analyze small programs or apply some domain knowledge to simplify the execution. We systematically analyze the bottleneck of the 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. We assume that adversaries can exploit secret-dependent control-flow transfers and data-access patterns when the program processes different sensitive data. First, we collect the dynamic execution trace for each input of the target libraries and then run symbolic execution on the traces. 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 symbol. Those formulas can precisely model side-channel vulnerabilities. Then we extend the problem to multiple leakages and related leakages and introduce the 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 and mbed TLS. The experimental result confirms that Abacus can precisely identify previously known vulnerabilities, report how much information is leaked and which byte in the original sensitive buffer is leaked. Although some of the analyzed crypto libraries have a number of side-channels, they actually leak very little information. Also, we perform the analysis of widely deployed software countermeasures against side channels. Abacus also discovers new vulnerabilities. With the help of Abacus, we confirm that those vulnerabilities are serious.

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. Some initial results indicate the sampling method suffers from the curse of dimensionality problem. Therefore, we design a guided sampling method and provide the corresponding error estimate.
- We implement the proposed method into a practical 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. Moreover, we analyze the root cause of many address-based side-channels. We find many of them 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. Fundamentally, these differences were caused by shared hardware components (e.g., CPU cache, TLB, and DRAM) in modern computer systems [25], [26]. Depending on the layer causing side-channels, we can classify them into the following types of side-channel attacks.

For example, cached-based side-channels [10]–[12], [23], [27]–[29] rely on the time differences between cache miss and cache hit. We introduce two common attack strategies, namely Prime+Probe [11] and Flush+Reload [29]. 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. If not, it will be fast. By knowing which cache set the target program accesses, the attacker can infer part of the sensitive information. While Flush+Reload targets a single cache line, it requires the attacker and the victim share the same memory address space. 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 know whether the victim has accessed the "monitored memory" and 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 the shielding systems by observing

```
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

the page fault sequences after restricting some code and data pages.

The key intuition is that each side-channel attacks above 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, many cache channels can observe cache accesses at the level of a cache line. For most CPU, one cache line holds 64 bytes of data. Hence according to the cache associativity, the low 6 bits of the address is irrelevant in causing those cached-based side-channels.

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. 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 uses Shannon entropy [19], min-entropy [30], and max-entropy [22], [31]. In these frameworks, the input sensitive information K is considered a random variable.

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

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

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 in one chance using the most common password, which is defined as

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

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

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

As it is easy to compute, most recent works [22], [31] use max-entropy as the definition of the amount of leaked information.

To illustrate how these definitions work, we consider the following code fragment.

```
1  uint_8 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 < 8) {
6      A();                   // branch 1
7  }
8  if (t2 > 0) {              // branch 2
9      B();
10 }
```

Figure 3: Side-channel leakage

In this paper we assume an attacker can observe whether branch 1 and branch 2 are executed or not. Therefore, an attacker can have four different observations depending on the value of key: $\varnothing$ for neither branch 1 nor branch 2 is executed, $\{1\}$ for only branch 1 is executed, $\{2\}$ for only branch 2 is executed, and $\{1, 2\}$ for both branch 1 and branch 2 are executed. Now the question is how much information can be leaked from the above code if an attacker knows which branch is executed?

Table I: The distribution of observation

Observation (o)	$\varnothing$	$\{1\}$	$\{2\}$	$\{1,2\}$
Number of Solutions	32876	20	32634	16
Possibility (p)	0.5016	0.0003	0.4980	0.0002

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. The three types of leakage metrics are calculated as follows.

Min Entropy. As $p_{max} = 0.5016$, with the definition, min-entropy equals to

$$\textit{min-entropy} = -\log_2 0.5016 = 0.995 \text{ bits}$$

Max Entropy. Depending on the value of key, the code can run four different branches which corresponding to four different observations. Therefore, with the max entropy definition, the leakage equals to

$$\textit{max-entropy} = -\log_2 4 = 2.000 \text{ bits}$$

Shannon Entropy. Based on Shannon entropy, the leakage equals to

$$\begin{aligned}\textit{Shannon-entropy} = -(&0.5016 * \log_2 0.5016\\ &+ 0.0003 * \log_2 0.0003\\ &+ 0.4980 * \log_2 0.4980\\ &+ 0.0002 * \log_2 0.0002)\\ &= 1.006 \text{ bits}\end{aligned}$$

In the next section, we will show that these measures work well only theoretically in a static analysis setting where only

assume the average leakage. Generally, they do not apply to dynamic analysis or practical 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 that 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 memory or cache. However, for this project, we assume the attacker can have noise-free observations. 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 paper and propose our method.

A. Problem Setting

Existing static-based side-channel quantification works [19], [22] define information leakage with max entropy or Shannon entropy. These definitions provide a strong security guarantee when one tries to prove a program is secure. 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 an attack scenario, the leakage could be much severe.

```
1 char key[9] = input();
2 if(strcmp(key, "password")){
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 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 sensitive input, when an attacker observes the code executes branch 1.

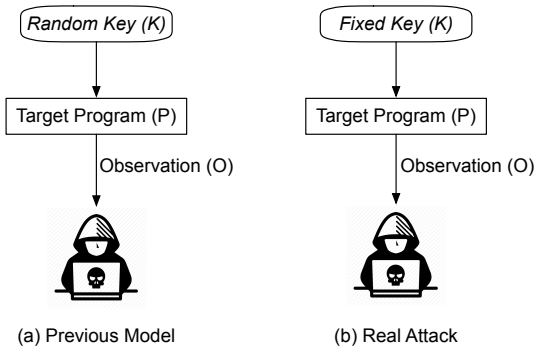


Figure 5: The gap between the real attack and previous model

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, the input values 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 and guess the value bit by bit, as in Figure 5(b). Like the previous example, the existing static method does not work well in those situations. We want to produce 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 side-channel. However, the above methods all fail in real attack models. This is the first challenge we face **(Challenge C1)**.

B. Notations

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

We assume a program (β) has K as the sensitive input. K should be a finite set of keys. The program also takes known messages M as the input. The model applies to most cryptosystems. For example, during the AES encryption, β is the encryption function. K is AES key, 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:

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

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 (MI) is a measure of the mutual dependence between the two variables. Here we use MI to describe the information dependence between K and 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 multiple times 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

$$\textit{Information leakage} = \textit{Initial uncertainty} - \textit{Remaining uncertainty}.$$

Now we compare 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. 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 the sensitive input 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*

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

The leakage $L_{\beta(k) \rightarrow o}$ based on the observation (o) is

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

With the new 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 or traces leak much more information than some others. In contrast, an *average* estimate based on random secret input information of 1 or 2 bits, 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)	$\varnothing$	$\{1\}$	$\{2\}$	$\{1, 2\}$
Number of Solutions	32876	20	32634	16
Leaked Information (bits)	1.0	11.7	1.0	12.0

D. Our Conceptual Framework

We now discuss how to model the observation (o), which is 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 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. For a specific branch, the condition is either true or false. Therefore, the branch condition is always a boolean variable.

We think 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 on secret-dependent control-transfers.

2) Secret-dependent Data Accesses: Similar to secret-dependent control 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 branch condition 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 control-transfers.

With the above definitions, we can model an attacker's observation with 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, 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], [32] all have large performance bottlenecks. As a result, those approaches either only apply to small-size programs [32] or apply some domain knowledge to simplify the analysis. Those tools interpret each instruction and update memory cells and registers with formulas that captured the semantics of the execution and search different input values that can lead to different execution behaviors using constraint solver. 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.

Existing binary analysis tools [33], [34] usually translate machine instructions into intermediate languages (IR). The reason is that the number of machine instructions is enormous, and the semantics of each instruction is complex. Intel Developer Manual [35] introduces more than 1000 different x86 instructions. It is tedious and hard to implement the manual rule for each instruction. On the contrary, 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, also introduce significant overhead [36].

First, transferring machine instructions into IR is time-consuming. For example, REIL IR [37], adopted in CacheS [21], has multiple transform processes, from binary to VEX IR, BAP IR, and finally REIL IR. As IR can also introduce additional conditional jump instructions, in order to precisely identify secret-dependent control-flows, we need to rule out conditional jump instructions introduced by IR, which is also time-consuming. Second, 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 [36] 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.

B. 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 [38]) to find satisfying k_1 and k_2.

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)

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 practical 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.

C. Counting the Number

The problem of quantifying the amount of leaked information can be reduced to the problem of computing the number of items in K^o, according to Definition 1 introduced in §III. 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.

D. 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 $\frac{|K|}{|K^o|}$. K^o represents the set that contains every input keys that satisfy the constraint. As the cardinality of K is known, the primary problem is to estimate 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 $c_t(\xi_1, \xi_2, \ldots, \xi_n) = c_{\xi_1} \wedge c_{\xi_2} \wedge \ldots \wedge c_{\xi_n}$. The problem of estimating the total leaked information can be reduced to the problem of counting the number of different solutions that satisfies the constraint $c_t(\xi_1, \xi_2, \ldots, \xi_n)$. 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-C, 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 the 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) $\forall \quad i, j \in \{1, 2, 3, \ldots, m\}$ *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 the 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 A.

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 satisfies 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 B.

4) Error Estimation: In this part, we analyze the accuracy of the result from the Monte Carlo approximation. We use the central limit theorem (CLT) and uncertainty propagation theorem to estimate errors of the number of leaked bits for each site.

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 describe standard deviation. We can claim that we have 95% 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 differentiation. For Abacus, we want the error of estimated leaked information (ΔL) to be less than 1 bit. So we can get $\frac{\Delta p}{p \ln 2} \leq 1$. 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, the Monte Carlo simulation will terminate.

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

The shortcomings of the existing work inspire us to design a new tool to detect and quantify side-channel vulnerabilities in binaries. The tool 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. Previous works [14], [16] have demonstrated the effectiveness of the dynamic analysis. We follow their approaches and run the target binary under dynamic binary instrumentations (DBI) to record execution traces and the runtime information.
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. The above design significantly reduces the overhead of symbolic executions, which makes the tool scale to real-world programs.
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 theorems (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 IV. 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 part of the 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 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 with the implementation.

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, OpenSSL and mbed TLS. OpenSSL is the most commonly used crypto libraries in today's software. mbed TLS (previous known as PolarSSL) is designed to be easy to understand and fit on small embedded devices.

We build the source code into 32-bit x86 Linux executables with the GCC 8.0 under Ubuntu 14.04. Although we use use symbol information to track back leakage sites in the source code, our tool can also work on stripped binaries. 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. 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 the side-channel vulnerabilities? (§VI-C, §VI-D, and §VI-E)

A. Evaluation Result Overview

Table V shows the overview of evaluation results. Abacus found 883 leakages in total from real-world crypto system 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 finds that secret-dependent memory accesses cause most leakages. Abacus also identifies that most side-channel vulnerabilities leak very little information in practice, which confirms our initial assumptions. Without our tool, developers will not be able to distinguish those "vulnerabilities" from severe ones and ignore them for sure. However, we do find some vulnerabilities that Abacus reports with more severe leakages. Some of them have been confirmed by existing research that those vulnerabilities can be exploited to realize real attacks.

All the symmetric encryption implementations in OpenSSL and mbed TLS have significant leakages due to the implementation of the lookup table to speed up the computation. Every

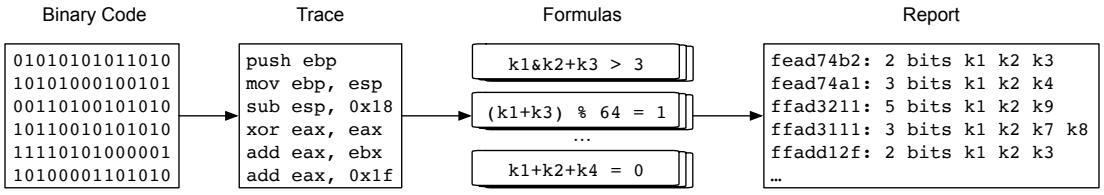


Figure 6: The workflow of Abacus.

Table V: Evaluation results overview. We evaluate two versions of mbed TLS and five versions of OpenSSL. CF represents secret-dependent control-flow transfers and DF represents secret-dependent data-flow transfers. Side-channel leakages can be found by symbolic execution and we run Monte Carlo to estimate the amount of leakage information. A summary of all vulnerabilities with the amount of leak information can be found in the appendix.

Algorithm	Implementation	Leakage Sites	CF	DF	# Instructions	Max Leakage	Sym. Exe.	Monte Carlo
						bits	ms	ms
AES	Mbed TLS 2.5	68	0	68	39,855	8.6	512	1,052
AES	Mbed TLS 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	Mbed TLS 2.5	15	0	15	4,596	1.1	58	162
DES	Mbed TLS 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
							minutes	minutes
RSA	Mbed TLS 2.5	6	6	0	22,109,246	9.6	39	41
RSA	Mbed TLS 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	128,061,910		209m	2,861m

leakage found during the evaluation belongs to the type of secret-dependent memory accesses. We believe that the secret-dependent control-flow transfers have been widely studied in the past few years, and developers have patched most of those leakages. One method to address the leakage is to use bit-slicing. We will analyze the corresponding countermeasure in the following sections.

Abacus finds several leakage sites for both implementations of DES and AES in OpenSSL and mbed TLS. Abacus confirms that all those leakages come from table lookups. mbed TLS 2.15 and 2.5 have the same implementations of 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 OpenSSL and mbed TLS. However, we find the new implementation of OpenSSL instead uses typical four 1K tables. It only uses 1K of the tables. This implementation is rather easy but is still vulnerable to a side channel attack. However, the countermeasures do somehow decrease the total amount of leaked information.

We also evaluates our tool on the RSA implementations. With the optimizations introduced in §IV, we do not apply any domain knowledge to simplify the analysis. Therefore, our tools can identify all the leakage sites reported by CacheD [14] in a shorter time. Our tool finds that most leakages in RSA occur in the big number implementation. 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 the next section.

In addition to identifying side-channel leakages, 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 the number of satisfying keys is very small and the leakage is quite severe. *That is, timeout indicates severe leakage.* During the evaluation, we find Abacus can quantify every side-channel leakage for every symmetric encryption. For asymmetric encryptions, Monte Carlo sometimes times out. We manually check those leakage sites and find most of them are quite severe. We will present the details in the subsequent sections.

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		

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

As shown in Table VI, Abacus can not only identify all the leakage sites reported by CacheD [14], but also many new ones. CacheD fails to detect many other 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 suffers from some performance issues and uses some domain knowledge to simplify symbolic execution and has to trim the traces before processing. The design does not introduce false positives, but can neglect some vulnerabilities. On the contrary, Abacus does not apply any domain knowledge and can find more 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 more than 100x faster than CacheD.

Since DATA [16] compares 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 data leaks. Among those leakages, they found two vulnerabilities. On the contrary, Abacus can report how many bits is actually leaked, which eases the pain to discover sensitive leakages.

C. Vulnerability Case Study

1) AES in mbed TLS: During our evaluation, we find mbed TLS 2.5 and 2.15.1 have the same implementation of AES. Our tool provides the same leakage report for both versions. Abacus identifies that most leakages are in function *mbedtls_internal_aes_decrypt*. (Other leakage sites are in function *mbedtls_aes_setkey_enc*.) All leakages are caused by secret-dependent memory accesses. Shown in Figure 7, there are seven leakage sites in total. Leakage 1, 2, 3 are the same and leakage 4, 5, 6, 7 are the same. They both use a pre-computed lookup table to speed up computation. However, Abacus reports leakage 1, 2, 3 typically 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 accesses a smaller lookup table (2K). Therefore, leakage 4, 5, 6, 7 leak less information.

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

Figure 7: Function *mbedtls_internal_aes_encrypt*

```
1  ...
2  if( mbedtls_mpi_cmp_int( N, 0 ) < 0 || ( N->p[0] & 1 ) == 0 )
3      return( MBEDTLS_ERR_MPI_BAD_INPUT_DATA );
4  ...
```

Figure 8: Function *mbedtls_mpi_exp_mod*

2) RSA in mbedTLS: Abacus identifies several side-channel leakages for the RSA implementation in Mbed TLS. Here we introduce and analyze two cases.

Abacus reports one bit information is leaked from the branch at line 2 in Figure 8 and it leaks less information than other leakages. Function *mbedtls_mpi_exp_mod* performs sliding-window exponentiation for big numbers. The leakage is caused by checking the signed bit of the big number N. Therefore, the leakage can only tell whether N is greater than zero, which is one bit leak, not severe.

Function *mpi_mul_hlp*, shown in Figure 9, is notoriously for a series of timing attacks. Recent patches have fixed many leakages in function *mpi_mul_hlp*. Abacus reports 8 bits of information is leaked from line 5. *mpi_mul_hlp* is a helper function to perform mbedtls_mpi multiplication. As the code will be executed many times, each time it will leak independent information. The leakage is severe compared to the previous one.

D. Case Study of RSA in OpenSSL

For the crypto libraries, it is likely that an updated version has less vulnerabilities compared to the previous versions because software developers have patched some of those vulnerabilities.

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 10, confirms

```
1  ...
2  do {
3      *d += c; c = ( *d < c ); d++;
4  }
5  while( c != 0 );
6  ...
```

Figure 9: Function *mpi_mul_hlp*

our assumptions. The newer version of OpenSSL leaked less amount of information compared to the previous versions. After version 0.9.7g, OpenSSL adopted a fixed-window mod_exp 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 quite effectively mitigated most of leakages because the other four versions have fewer leakages than 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 vulnerabilities for RSA (CVE-2016-0702). RSA changelog also claims OpenSSL 1.1.1 adopted "numerous side-channel attack mitigation." The result confirms our assumptions.

E. Analysis of Software Countermeasures

1) Bit-slicing: Bit-slicing is an efficient method to construct constant-time implementation for side channel to mitigation. The basic concept is to implement a function in terms of single-bit logical gate operations, such as AND, XOR, OR, and NOT. Since the table lookups and conditional jumps are replaced with single-bit logical gates, with no secret-dependent memory addresses or control flow, both the data access and control flow types of side-channel leakages are mitigated.

We would like to test Abacus on bit-slicing. We adopted the SBOX implementations, commonly used in block ciphers such as DES and AES, with and without bit-slicing, and apply Abacus to confirm the mitigation. The SBOX implementation with and without bit-slicing are shown in Figure 11 and Figure 12, respectively, in Appendix C. Consider an SBOX take some bits derived from a password as input and output 2-bit transform result. The plain implementation has a range check on the password input and a secret-dependent table lookup while bit-slicing does not. Abacus reports that there are both control-flow and data access types of leakage in the non-bit-slicing implementation (line 6 and line 7 in Figure 12). The number of leaked bits is 5.0 and 4.4, respectively. At line 6, according to Definition 1, the input set K is $[0, 2^8-1]$, the observed input set K^o is $[0, 2^3-1]$. Thus, the leakage $L_{\beta(k)\to o}$ based on the observation (o) is $L_{\beta(k)\to o} = \log_2|K| - \log_2|K^o| = 8-3 = 5$ bit, which confirms the result from Abacus. Similarly, we can verify the result of the other leakage site. Abacus reports no leakage on the bit-slicing implementation.

2) Smaller Lookup Tables: Considering the cache-collision timing attack [39], the probability of leakage decreases when the lookup table entry or element size gets smaller. We take AES as an example, for the last round of encryption, the algorithm uses a specialized table with 4-byte size entries, of which only one byte from each element actually contributes to the result. Hence, a significant method to reduce the information leakage in AES is to reduce the size of the lookup table. We designed a demo to illustrate this countermeasure, a lookup table with one-byte size elements (see Figure 13) will leak less information than a table with 4-byte element size (see Figure 14). The encrypt function for each table to process the same amount of information, respectively, (see Figure 15 and Figure 16). Abacus reports three leakage sites for each lookup table, with 4.0 bits for the table with smaller entries and 2.0 bits for the table with bigger entries, confirming the theory.

VII. Discussions and Limitations

In the section, we discuss the limitations, usages, and some future work. 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 [40], [41], also has the common limitations of dynamic approaches. Abacus may only cover part of the code. We may neglect some side-channel vulnerabilities not covered by the traces analyzed. However, this is not a crucial problem for analyzing crypto libraries because crypto libraries are designed to have the same code coverage for various inputs. Our evaluation also confirms the above point. For symmetric encryptions, during our evaluation there is no secret-dependent control-flow transfers. RSA implementations have several secret-dependent control-flow transfers. Most of them are bound checks, which do not leak much information and have negligible effects on the whole code coverage as well.

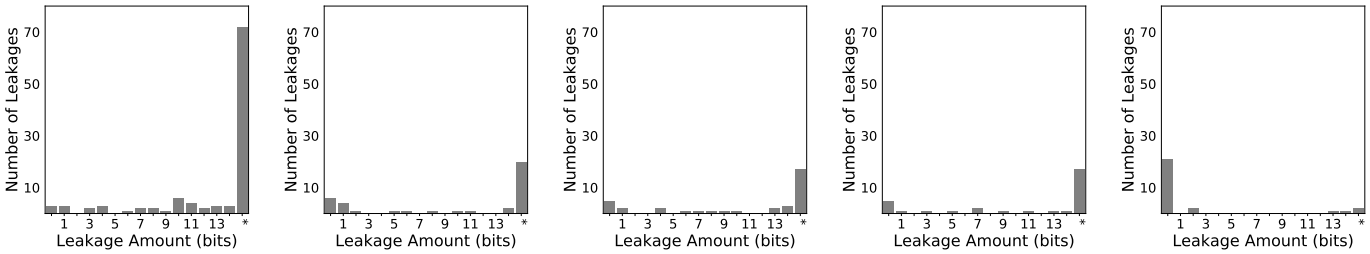
One of the motivations of Abacus is that while recent works have reported lots of potential side-channel vulnerabilities, most of them are not patched by developers. Our evaluation result confirms that most of them are not severe and can be safely ignored. For RSA, the latest OpenSSL only has one leakage site that can leak more than 3 bits while there are 22 leakage sites according to Abacus. DES implementation of OpenSSL has several sensitive leakages, but given the end life status of DES, it is still unpatched for the worth of engineering effort. In the future, we would like to apply Abacus on other libraries including non-crypto ones.

VIII. Related Work

There is a vast amount of work on side channel detection [14], [16], [19]–[22], [42], mitigation [17], [18], [43]–[49], information quantification [24], [32], [50]–[53], and model counting [32], [54]–[57]. Here we only present the closely related work to ours. Due to space limit, we do not include side channel attack work.

A. Detection

There are a large number of works on side-channel vulnerability detections in recent years. CacheAudit [22] uses abstract domains to compute the over approximation of cache-based side-channel information leakage upper bound. However, due to over approximation they make, CacheAudit can indicate the program is side-channel free if the program has zero leakage. However, it is less useful to judge the sensitive level of the side-channel leakage based on the leakage provided by CacheAudit. CacheS [21] 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 [20] 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.



(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 10: RSA implementations in different versions of OpenSSL. We round the number of leaked information into the nearest integer. The mark $*$ means timeout, which indicates more severe leakages (see §VI-A).

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. During the symbolic execution, every value except the sensitive key uses the concrete value. 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 [19] 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.

B. Mitigation

Both hardware [18], [43]–[46] and software [17], [47]–[49] side-channels mitigation methods have been proposed recently. Hardware countermeasures, including parting the hardware computing resource [43], randomizing cache accesses [44], [46], and designing new architecture [58], 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. [47] introduced a compiler-based approach to eliminate key-dependent control-flow transfers. Crane et al. [49] 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 [59], [60] and unifying control-flows [47].

C. Quantification

Proposed by Denning [61] and Gray [62], 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 [52] quantify the information leakage as the network flow capacity. Backes et al. [24] 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. For side-channels leakage quantification, we can think the attacker's observation is the public output. Phan et al. [53] 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 [32] 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.

D. 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 [54] 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* [55], *Mbound* [56], and *MiniCount* [57]. 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 [63], [64] on Modulo Theories model counting (#SMT). They cannot be directly applied to side channel leakage quantification. ApproxFlow [50] uses ApproxMC [65] 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 definition and 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)*. USENIX Association, 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)*. USENIX Association, 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)*. USENIX Association, 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)*. USENIX Association, 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. IEEE Computer Society, 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)*. USENIX Association, 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)*. USENIX Association, 2018.

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

[18] 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.

[19] 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. ACM, 2018.

[20] 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.

[21] 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)*. USENIX Association, 2019.

[22] 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.

[23] 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)*. USENIX Association, 2014.

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

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

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

[27] Y. Yarom, D. Genkin, and N. Heninger, "Cachebleed: a timing attack on openssl constant-time rsa," *Journal of Cryptographic Engineering*, vol. 7, no. 2, 2017.

[28] 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.

[29] 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.

[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. ACM, 2017.

[32] 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.

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

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

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

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

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

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

[39] J. Bonneau and I. Mironov, "Cache-collision timing attacks against aes," in *Cryptographic Hardware and Embedded Systems - CHES 2006*, L. Goubin and M. Matsui, Eds. Springer Berlin Heidelberg, 2006.

[40] 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.

[41] 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.

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

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

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

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

[50] 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.

[51] 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.

[52] 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.

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

[54] 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.

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

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

[57] 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.

[58] 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.

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

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

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

[62] 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.

[63] 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.

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

[65] 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
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 B
ALGORITHM TO COMPUTE THE NUMBER OF SATISFYING ASSIGNMENTS

Algorithm 2: Multiple Step Monte Carlo Sampling

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

n: the number of sampling times
S_{c_i}: the set contains input variables for c_i
n_s: the number of satisfying assignments
N_{c_t}: the set contains all solution for c_t
r: times of reducing g
k: the input variable
R: a function that produces a random point from S_{c_i}
$r \leftarrow 1$, $n \leftarrow 0$
for $t \leftarrow 1$ **to** m **do**
  $S_{c_t} \leftarrow \pi(c_t)$
  if $|S_{c_t}| = 1$ **then**
    $N_{c_t} \leftarrow$ Compute all solutions of c_i
    $N_{c_t} = \{n_1, \ldots, n_m\},\ S_{c_t} = \{k\}$
    $g_i = g_i(k = n_1) \wedge \ldots \wedge g_i(k = n_m)$
    $r \leftarrow r + 1$
  end
end
while $n \leq \frac{6p}{1-p}$ **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
$|K_{g_i}| \leftarrow n_s|K|/(n * r * range(k))$

APPENDIX C
MITIGATION METHOD EVALUATION

```
uint8_t password = input();

a = *password & 0b001;
b = (*password & 0b010) >> 1;
c = (*password & 0b100) >> 2;

na = ~a & 1;
nb = ~b & 1;
nc = ~c & 1;

t0 = (b & nc);
t1 = (b | nc);

l = (a & nb) | t0;
r = (na & t1) | t0;

ret = l << 1 + r;
```

Figure 11: SBOX with BitSlicing

```
uint8_t password = input();

uint8_t SBOX[] = {1, 0, 3, 1, 2, 2, 3, 0};

if (password <= 0b111)      \\Leaks 5 bits of password
    ret = SBOX[password];   \\Leaks 4 bits of password
```

Figure 12: SBOX without BitSlicing

```
static const uint8_t T[16] = {
      0x63U, 0x7cU, 0x77U, 0x7bU, 0xf2U, 0x6bU, 0x6fU, 0xc5U,
      0x30U, 0x01U, 0x67U, 0x2bU, 0xfeU, 0xd7U, 0xabU, 0x76U};
```

Figure 13: One-byte Entry Lookup Table

```
static const uint32_t T[16] = {
    0xc66363a5U, 0xf87c7c84U, 0xee777799U, 0xf67b7b8dU,
    0xfff2f20dU, 0xd66b6bbdU, 0xde6f6fb1U, 0x91c5c554U,
    0x60303050U, 0x02010103U, 0xce6767a9U, 0x562b2b7dU,
    0xe7fefe19U, 0xb5d7d762U, 0x4dababe6U, 0xec76769aU};
```

Figure 14: Four-byte Entry Lookup Table

```
void encrypt_one(uint32_t *o, uint32_t *key, uint32_t l) {
    for (int i = 0; i < l; i+=4)
        output[i] = (T[(key[i]>>24)] << 24) ^
                    (T[(key[i+1]>>16) & 0xff] << 16) ^
                    (T[(key[i+2]>>8) & 0xff] << 8) ^
                    (T[(key[i+3]) & 0xff]);
}
```

Figure 15: Encrypt with One-byte Entry Lookup Table

```
void encrypt_four(uint32_t *o, uint32_t *key, uint32_t l) {
    for (int i = 0; i < l; i+=4)
        output[i] = (T[(key[i]>>24)] & 0xff000000) ^
                    (T[(key[i+1]>>16) & 0xff] & 0x00ff0000) ^
                    (T[(key[i+2]>>8) & 0xff] & 0x0000ff00) ^
                    (T[(key[i+3]) & 0xff] & 0x000000ff);
}
```

Figure 16: Encrypt with One-byte Entry Lookup Table

APPENDIX D
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 mbed TLS 2.5, OpenSSL 1.1.0f, and OpenSSL 1.1.1. The results are representative to other versions. All the results will be made available in electronic format online when the paper is published.

In all the tables presented in this appendix, the mark "$*$" means timeout, which indicates more severe leakages. See §VI-A for the details. Also note that we round the calculated numbers of leaked bits to include one digit after the decimal point, so 0.0 really means very small amount of leakage, but not exactly zero. See §IV-D4 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 1.1.0f

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 XII: 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 XIII: 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

Table XIV: Leakages in AES implemented by OpenSSL 1.1.0f

File	Line No.	Function	# Leaked Bits	Type
aes_core.c	676	AES_set_encrypt_key	4.1	DA
aes_core.c	676	AES_set_encrypt_key	5.6	DA
aes_core.c	676	AES_set_encrypt_key	4.2	DA
aes_core.c	677	AES_set_encrypt_key	3.9	DA
aes_core.c	698	AES_set_encrypt_key	4.0	DA
aes_core.c	698	AES_set_encrypt_key	4.3	DA
aes_core.c	699	AES_set_encrypt_key	3.7	DA
aes_core.c	700	AES_set_encrypt_key	4.2	DA
aes_core.c	648	AES_set_encrypt_key	4.1	DA
aes_core.c	664	AES_set_encrypt_key	3.9	DA
aes_core.c	665	AES_set_encrypt_key	4.0	DA
aes_core.c	662	AES_set_encrypt_key	4.2	DA
aes_core.c	665	AES_set_encrypt_key	3.9	DA
aes_core.c	665	AES_set_encrypt_key	4.1	DA
aes_core.c	668	AES_set_encrypt_key	3.8	DA
aes_core.c	665	AES_set_encrypt_key	4.0	DA
aes_core.c	665	AES_set_encrypt_key	4.1	DA
aes_core.c	665	AES_set_encrypt_key	3.8	DA
aes_core.c	665	AES_set_encrypt_key	3.9	DA
aes_core.c	665	AES_set_encrypt_key	3.9	DA
aes_core.c	665	AES_set_encrypt_key	4.0	DA
aes_core.c	665	AES_set_encrypt_key	3.8	DA
aes_core.c	665	AES_set_encrypt_key	4.2	DA
aes_core.c	668	AES_set_encrypt_key	4.2	DA
aes_core.c	665	AES_set_encrypt_key	3.9	DA
aes_core.c	662	AES_set_encrypt_key	4.0	DA
aes_core.c	665	AES_set_encrypt_key	3.8	DA
aes_core.c	661	AES_set_encrypt_key	4.4	DA
aes_core.c	665	AES_set_encrypt_key	3.8	DA
aes_core.c	662	AES_set_encrypt_key	4.0	DA
aes_core.c	665	AES_set_encrypt_key	4.0	DA
aes_core.c	665	AES_set_encrypt_key	3.9	DA
aes_core.c	665	AES_set_encrypt_key	4.1	DA
aes_core.c	665	AES_set_encrypt_key	3.8	DA
aes_core.c	665	AES_set_encrypt_key	3.8	DA
aes_core.c	665	AES_set_encrypt_key	4.2	DA
aes_core.c	665	AES_set_encrypt_key	4.3	DA
aes_core.c	662	AES_set_encrypt_key	4.0	DA
aes_core.c	665	AES_set_encrypt_key	3.8	DA
aes_core.c	661	AES_set_encrypt_key	4.0	DA
aes_core.c	805	AES_encrypt	6.2	DA
aes_core.c	805	AES_encrypt	6.5	DA
aes_core.c	806	AES_encrypt	11.6	DA
aes_core.c	806	AES_encrypt	5.6	DA
aes_core.c	882	AES_encrypt	11.6	DA
aes_core.c	882	AES_encrypt	11.4	DA
aes_core.c	797	AES_encrypt	7.3	DA
aes_core.c	797	AES_encrypt	11.3	DA
aes_core.c	916	AES_encrypt	5.4	DA
aes_core.c	917	AES_encrypt	12.6	DA
aes_core.c	924	AES_encrypt	10.9	DA
aes_core.c	914	AES_encrypt	11.1	DA
aes_core.c	920	AES_encrypt	7.5	DA
aes_core.c	920	AES_encrypt	12.6	DA
aes_core.c	920	AES_encrypt	6.7	DA
aes_core.c	920	AES_encrypt	6.9	DA
aes_core.c	929	AES_encrypt	6.6	DA
aes_core.c	932	AES_encrypt	6.5	DA
aes_core.c	934	AES_encrypt	10.7	DA
aes_core.c	884	AES_encrypt	12.5	DA
aes_core.c	885	AES_encrypt	7.0	DA
aes_core.c	886	AES_encrypt	9.0	DA
aes_core.c	891	AES_encrypt	5.9	DA
aes_core.c	890	AES_encrypt	9.2	DA
aes_core.c	897	AES_encrypt	7.5	DA
aes_core.c	897	AES_encrypt	11.1	DA
aes_core.c	896	AES_encrypt	9.8	DA
aes_core.c	898	AES_encrypt	5.2	DA
aes_core.c	896	AES_encrypt	10.3	DA
aes_core.c	899	AES_encrypt	5.8	DA
aes_core.c	902	AES_encrypt	7.3	DA
aes_core.c	902	AES_encrypt	10.5	DA
aes_core.c	910	AES_encrypt	3.9	DA
aes_core.c	910	AES_encrypt	4.2	DA
aes_core.c	944	AES_encrypt	3.8	DA
aes_core.c	944	AES_encrypt	4.4	DA
aes_core.c	944	AES_encrypt	4.4	DA
aes_core.c	944	AES_encrypt	3.8	DA
aes_core.c	950	AES_encrypt	4.1	DA
aes_core.c	952	AES_encrypt	4.7	DA
aes_core.c	951	AES_encrypt	4.0	DA
aes_core.c	951	AES_encrypt	4.1	DA
aes_core.c	961	AES_encrypt	3.9	DA
aes_core.c	957	AES_encrypt	3.9	DA
aes_core.c	957	AES_encrypt	3.9	DA
aes_core.c	958	AES_encrypt	5.4	DA
aes_core.c	958	AES_encrypt	3.9	DA
aes_core.c	960	AES_encrypt	3.9	DA

Table XV: Leakages in AES implemented by OpenSSL 1.1.1

File	Line No.	Function	# Leaked Bits	Type
aes_core.c	665	AES_set_encrypt_key	4.2	DA
aes_core.c	665	AES_set_encrypt_key	3.9	DA
aes_core.c	665	AES_set_encrypt_key	3.9	DA
aes_core.c	665	AES_set_encrypt_key	4.0	DA
aes_core.c	665	AES_set_encrypt_key	4.2	DA
aes_core.c	665	AES_set_encrypt_key	3.9	DA
aes_core.c	665	AES_set_encrypt_key	3.7	DA
aes_core.c	665	AES_set_encrypt_key	4.1	DA
aes_core.c	665	AES_set_encrypt_key	4.1	DA
aes_core.c	665	AES_set_encrypt_key	3.7	DA
aes_core.c	665	AES_set_encrypt_key	4.1	DA
aes_core.c	665	AES_set_encrypt_key	4.3	DA
aes_core.c	665	AES_set_encrypt_key	3.8	DA
aes_core.c	665	AES_set_encrypt_key	4.1	DA
aes_core.c	665	AES_set_encrypt_key	3.8	DA
aes_core.c	665	AES_set_encrypt_key	4.2	DA
aes_core.c	665	AES_set_encrypt_key	3.8	DA
aes_core.c	665	AES_set_encrypt_key	4.1	DA
aes_core.c	665	AES_set_encrypt_key	4.1	DA
aes_core.c	665	AES_set_encrypt_key	4.1	DA
aes_core.c	665	AES_set_encrypt_key	4.2	DA
aes_core.c	665	AES_set_encrypt_key	4.2	DA
aes_core.c	665	AES_set_encrypt_key	4.0	DA
aes_core.c	665	AES_set_encrypt_key	4.4	DA
aes_core.c	665	AES_set_encrypt_key	4.1	DA
aes_core.c	665	AES_set_encrypt_key	4.2	DA
aes_core.c	665	AES_set_encrypt_key	4.1	DA
aes_core.c	665	AES_set_encrypt_key	3.9	DA
aes_core.c	665	AES_set_encrypt_key	4.2	DA
aes_core.c	665	AES_set_encrypt_key	3.9	DA
aes_core.c	665	AES_set_encrypt_key	3.7	DA
aes_core.c	665	AES_set_encrypt_key	4.0	DA
aes_core.c	665	AES_set_encrypt_key	3.9	DA
aes_core.c	665	AES_set_encrypt_key	4.0	DA
aes_core.c	665	AES_set_encrypt_key	4.1	DA
aes_core.c	665	AES_set_encrypt_key	3.9	DA
aes_core.c	665	AES_set_encrypt_key	4.3	DA
aes_core.c	665	AES_set_encrypt_key	3.9	DA
aes_core.c	665	AES_set_encrypt_key	4.0	DA
aes_core.c	665	AES_set_encrypt_key	4.1	DA
aes_core.c	884	AES_encrypt	4.2	DA
aes_core.c	884	AES_encrypt	8.1	DA
aes_core.c	884	AES_encrypt	4.2	DA
aes_core.c	884	AES_encrypt	8.1	DA
aes_core.c	890	AES_encrypt	3.8	DA
aes_core.c	890	AES_encrypt	4.0	DA
aes_core.c	890	AES_encrypt	8.1	DA
aes_core.c	890	AES_encrypt	7.7	DA
aes_core.c	896	AES_encrypt	7.9	DA
aes_core.c	896	AES_encrypt	4.0	DA
aes_core.c	896	AES_encrypt	7.3	DA
aes_core.c	896	AES_encrypt	4.3	DA
aes_core.c	902	AES_encrypt	7.3	DA
aes_core.c	902	AES_encrypt	7.9	DA
aes_core.c	902	AES_encrypt	4.1	DA
aes_core.c	902	AES_encrypt	3.9	DA
aes_core.c	914	AES_encrypt	4.3	DA
aes_core.c	914	AES_encrypt	8.1	DA
aes_core.c	914	AES_encrypt	4.2	DA
aes_core.c	914	AES_encrypt	7.3	DA
aes_core.c	920	AES_encrypt	4.1	DA
aes_core.c	920	AES_encrypt	4.0	DA
aes_core.c	920	AES_encrypt	8.2	DA
aes_core.c	920	AES_encrypt	7.6	DA
aes_core.c	926	AES_encrypt	8.2	DA
aes_core.c	926	AES_encrypt	3.9	DA
aes_core.c	926	AES_encrypt	8.2	DA
aes_core.c	926	AES_encrypt	3.9	DA
aes_core.c	932	AES_encrypt	8.1	DA
aes_core.c	932	AES_encrypt	7.6	DA
aes_core.c	932	AES_encrypt	4.0	DA
aes_core.c	932	AES_encrypt	4.0	DA
aes_core.c	944	AES_encrypt	3.9	DA
aes_core.c	944	AES_encrypt	4.3	DA
aes_core.c	944	AES_encrypt	3.9	DA
aes_core.c	944	AES_encrypt	4.0	DA
aes_core.c	951	AES_encrypt	3.8	DA
aes_core.c	951	AES_encrypt	3.9	DA
aes_core.c	951	AES_encrypt	3.9	DA
aes_core.c	951	AES_encrypt	3.9	DA
aes_core.c	958	AES_encrypt	4.2	DA
aes_core.c	958	AES_encrypt	3.7	DA
aes_core.c	958	AES_encrypt	3.8	DA
aes_core.c	958	AES_encrypt	4.0	DA
aes_core.c	965	AES_encrypt	4.0	DA
aes_core.c	965	AES_encrypt	4.0	DA
aes_core.c	965	AES_encrypt	3.7	DA
aes_core.c	965	AES_encrypt	3.7	DA