# CleverHans: Precise Fine-grained Side-channel Information Leakage Quantification in Binaries

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 many potential vulnerabilities. However, in practice many such vulnerabilities leak 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. We model each leakage as a constraint and use these constraints to divide the input space into subsets such that only one subset satisfies every constraint. The cardinality of the subset corresponds to the amount of leaked information from those leakages. We use symbolic execution 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 CleverHans, which can not only find the side-channel vulnerabilities but also estimate how many bits are leaked. CleverHans outperforms existing dynamic side-channel detection tools in terms of performance and accuracy. Also, CleverHans can report a very fined-grained vulnerability leakage information. We evaluated CleverHans on OpenSSL, MbedTLS, and found several severe side-channel vulnerabilities. FIXME: *Need to update the rest of the abstract after the evaluation section is done.* Compared with previous research, the results are surprisingly different. First, it shows that most of the reported vulnerabilities are hard to exploit in practice. Second, we also find several sensitive vulnerabilities that are missed by existing tools. We confirmed those vulnerabilities with manual checks and software developers.**

## I. Introduction

Side channels are inevitable in modern computer systems as the sensitive information may be leaked by many kinds of inadvertent behaviors, such as power, electromagnetic radiation and even sound [**?**]. 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 [**?**]. 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 manipulate secrets and guess the secrets such as encryption keys [**?**].

Various countermeasures have been proposed to defend against software-based side-channel attacks. Hardware level solutions, including reducing shared resources, adopting oblivious RAM, and using transnational memory [1]–[3] need new hardware features or changes to 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.

Regarding the root cause of software-based side channels, many of them are caused by the following two specific types: data flow from secrets to load addresses and data flow from secrets to branch conditions. We call them secret-dependent control-flow and memory-access correspondingly. Therefore, a central problem is identifying those two code patterns automatically. Recent works [**?**], [**?**], [2] adopt static and dynamic analysis to detect side-channels. They can find many potential leak sites in real-world software, but fail to report how severe a potential leakage could be. Many of the reported vulnerabilities are typically hard to exploit and leak very little information. For example, DATA [**?**] reports 2,246 potential leakage site for the RSA implementation in OpenSSL. After some inspectations, 1,510 are dismissed, but it still leaves 278 control-flow and 460 data-access patterns. For software developers, it is hard for them to fix all those vulnerabilities, let alone the majority of them are negligible. While some vulnerabilities can be used to recover the full secret keys [**?**], many other vulnerabilities prove to be less serious in reality.

To assess the sensitive level of side-channel vulnerabilities, we need a proper quantification metric. Static methods, 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 over-approximation [**?**]. For example, CacheAudit [**?**] 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 real environment. Although they are very precise in term of true leakages, no existing tool can precisely assess the severity of the vulnerabilities they discover.

To overcome these limitations, we propose a novel method to quantify information leakage more precisely. Different from previous works, which only consider the "average" information leakage, we study the problem based on real attack scenarios. The average information assumes that the target program will have *variable* sensitive information 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 such as the key. Therefore, the previous threat model cannot catch 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 big 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 actually gained. 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 more precisely.

We use constraints to model the relation between the original sensitive input and each leakage site. We run the instruction level symbolic execution on the whole execution trace to generate the constraints. Symbolic execution can provide the fine-grained information but is usually believed to be an expensive operation in terms of performance. Therefore, existing dynamic symbolic execution based works [**?**] either only analyze small programs or apply some domain knowledges to simplify the execution. We systematically analyze the bottleneck of the symbolic execution and optimzie it scalable to real-world cryptosystems.

We apply the above technique and build a tool called CleverHans,[1] which could discover potential information leakage sites as well as estimating how many bits they can leak for each leakage site. We assume that adversaries can exploit secret-dependnet 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 math 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 a monte carlo sampling method to estimate the single and combined information leakage. In fact, if an application has a different sensitive input but still satisfies the formula, the code can still leak the same information.

We apply CleverHans on both symmetric and asymmetric ciphers from real-world crypto libraries including OpenSSL and mbedTLS. The experimental result confirms that CleverHans can precisely identify the previous known vulnerabilities, reporting 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. Finally, we present new vulnerabilities. With the help of CleverHans, we confirm those vulnerabilities are easily to be exploited. Our results are superisingly different compared to previous results and much more useful in practice.

In summary, we make the following contributions:

- We propose a novel method that can quantify fine-grained leaked information from side-channel vulnerabilities. We model each side-channel vulnerabilities as math formulas and mutiple side-channel vulnerabilities can be seen as the conjunction of those formulas, which precisely models the program semantics.
- We transfer the information quantification problem into a probabilty distribution problem and use the Monte Carlo sampling method to estimate the information leakage. Some initial results indicate the the sampling method suffers from the curse of dimensionality problem. We therefore design a guided sampling method and provide the corresponding error esitimate.
- We implement the proposed method into a practical tool and apply it on several real-world software. CleverHans successfully identifies the address-based side-channel vulnerabilities and provides the corresponding information leakge. The information leakage result provides the detailed information that help developers to fix the reported vulnerabilities.

## II. Background and Threat Model

In this section, we first present an introduction to software-based side-channel attacks. Moreover, we analyze the root cause of many software-based side-channels. We find many of them are caused by two specific side-channel vulnerabilities, secret-dependent control-flow transfers and secret-dependent memory accesses. Therefore, we will focus on identifying and quantifying those leakages in the paper. After that, we discuss existing methods on side-channel detection and quantification.

### A. Software-based Side-channels

Side-channels are information channels that can leak sensitive information unconsciously through different execution behaviors. Fundamentally, those differences were caused by shared hardware components (e.g., the CPU cache, the TLB and DRAM) in modern computer systems. 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 rely on the time differences between cache miss and cache hit. We introduce two common attack strategies, namely Prime+Probe and Flush+Reload. Prime+Probe targets a single cache set. The attacker preloads cache set with its own data and wait until the victim execute 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 locations of the sensitive information. Flush+Reload targets a single cache line. It requires the attacker and the victim share the same memory address space. During the "flush" stage, an attacker flushes the "monitored memory" from the cache. Then the attacker waits for the victim to access the memory. In

[1] CleverHans is a horse that can "count". Our tool uses an advanced method to count the number of leaked bits from side channels.

```
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 next phase, the attacker reload the "monitored memory". By measuring the time difference, the attacker can infer the sensitive information.

There are some other types of side-channels which target different hardware layers other than CPU cache as well. For example, the controlled-channel attack [4], where an attacker works in the kernel space, can infer sensitive data in the shielding systems by observing the page fault sequences after restricting some code and data pages.

The key intuition is that each side-channels above happens when the program accesses different memory addresses if the program has different sensitive inputs. More specifically, if a program show 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. So the low 6 bits 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 $p(e)$, if the event $e$ happens, then we receive

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

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

Existing works on information leakage quantification typically uses Shannon entropy, min-entropy [5], and max-entropy. In these frameworks, the input sensitive information $K$ is viewed as a random variable.

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

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

Shannon entropy can be used to quantify the initial uncertainty about the sensitive information. It measures how much information is in a system.

Min-entropy describes in the information leaks from the most likely outcome. For example, min-entropy can be used to describe the best chance of success in guessing one's password in one chance, which is defined as

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

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

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

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

To illustrate how above 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 if 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 executed, $\{1\}$ for only branch 1 executed, $\{2\}$ for only branch 2 executed, and $\{1, 2\}$ for both branch 1 and branch 2 executed. Now the question is how much information can be leaked from the above code if an attacker can know which branch is executed?

If $key$ has the uniform distribution, we can calculate the corresponding possibility by counting the number of possible solutions.

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

Table I: The distribution of observation

Now we provide a concrete example to show how the two types of quantification definition works and show that how our method is different.

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

$$min\text{-}entropy = -\log_2 0.5016 = 1.0 \ bits$$

**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

$$max\text{-}entropy = -\log_2 4 = 2.0 \ bits$$

**Shannon Entropy.** If the key satisfies the uniform distribution, the probability of each observations is shown is the table II.

Therefore, the leakage equals to $0.5016 * \log_2 0.5016 + 0.0003 * \log_2 0.0003 + 0.4980 * \log_2 0.4980 + 0.0002 * \log_2 0.0002 = 1.0$ bits. Shannon entropy, according to the definition, equals to

$$Shannon\text{-}entropy = 1.0 \ bits$$

We will show that while those measures works well in static analysis, they don't apply to the dynamic analysis in the next section.

### C. Threat Model

We consider an attacker who 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. CleverHans Leakage Definition

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

### A. Problem Setting

Existing static-based side-channel quantification works [1], [6] define information leakage using max entropy or Shannon entropy. These definitions provide a strong security guarantee when trying to prove a program is secure enough if zero bit of information is leaked reported by their approaches. However, it is useless if the tool report the program leaks some information. Moreover, those metrics does not apply to static method.

```
1 uint8_t password = input();
2 if(password == 0x1b){
3     pass();     //branch 1
4 }else{
5     fail();     //branch 2
6 }
```

Figure 4: A dummy password checker

We consider the above dummy password checker in 4. The program will take a 8 bit number 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 0x1b, in which case the attacker can fully retrieve the password. Therefore, the total information leakage should be 8 bits, which equals to the size of the sensitive input if an attacker observes the code executes branch 1 .

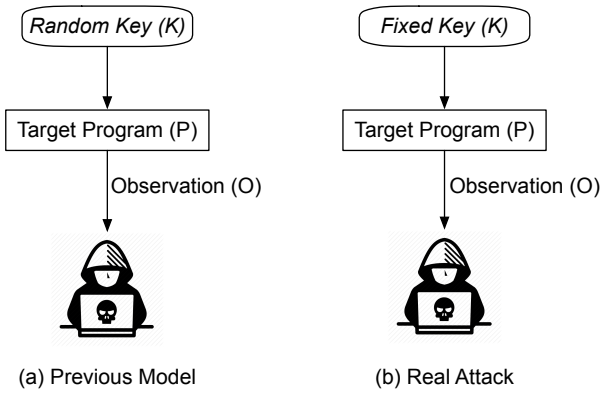



Figure 5: The gap between the real attack and previous model

However, previous static-based approach can not precisely reflect the sensitivity of the leakage. According to the definition of Shannon entropy, the leakage will be $\frac{1}{256} * \log_2 \frac{1}{256} + \frac{255}{256} * \log_2 \frac{255}{256} = 0.24$ 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 the previous definition. They assume the attacker runs the program multiple times with many different sensitive information as the input. 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. Like the previous example, the existing static method does not work well in those situations. We want to have a theory for dynamic analysis that if the theory says an attack leaks $x$ bits of secret information from side-channel vulnerabilities, then $x$ should be useful in estimating the sensitive level of side-channels. However, the above method all fails in the real attack model.

### B. Notations

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

We assume the program ($\beta$) 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 of the

cryptosystems. For example, during the AES encryption, $\beta$ is the encryption function. $K$ is AES key and $M$ is the message 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). For the paper, we consider the secret-dependent control-flows and secret-dependent memory accesses as the observations.

With the above definitions, we have the following mapping between $\beta$, $K$, $M$, and $O$:

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

An adversary does not have access to $K$, but he should know $\beta$, $M$, and $O$. For one execution of a deterministic program, once $k \in K$ and $m \in M$ are fixed, the observation ($o \in O$) is also determined. As an attacker, he knows $\beta$, $o$, and $m$. The attacker wants to infer value of $k$. We use $K^o$ to denote the set of $k$ that still produces the same observations:

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

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 $\beta$, $m$, and $o$?

### C. Theoretical Analysis

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

In information theory, the mutual information (MI) of is a measure of the mutual dependence between the two variables. Here we use MI to describe the leakage 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 $K$ conditioned on $O$. 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 the 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 the project, we hope to give a very precise definition of information leakages. Suppose an attacker run 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 [5] that

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

Now we come compare the equation 2 with the above slogan, we will find $H(K)$ is the *Initial uncertainty* and $H(K|O)$ is *Remaining uncertainty*. During a side-channel 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 ($\beta$) without knowing any domain information, any 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 $k' \in K - K^o$, $p(k'|o) = 0$. We can get the following equation:

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

**Definition 1.** *Given a program $\beta$ 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 the attacker observes that the code 4 runs the branch 1, then the $K^{o^1} = \{0\text{x}1\text{b}\}$. Therefore, the information leakage $L_{P(k)=o^1} = \log_2 256 - \log_2 1 = 8$ bits, which means the key is totally leaked. If the attacker observes the code runs branch2, the leaked information is $L_{P(k)=o^2} = \log_2 256 - \log_2 255 = 0$ bit.

We can also calculate the leaked information from the sample code 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 $o \in O$.

| 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 |

Table II: The distribution of observations

### D. Our Conceptual Framework

We now discuss how we model the observation (o), which is the direct information that an adversary can get during the attack.

During the execution, a program ($\beta$) have many temporary values ($t_i \in T$). Once $\beta, k, m$ is determined, $t_i$ is also fixed. Therefore, $t_i = f_i(\beta, k, m)$. $f_i$ is a function that can uniquely map the one to one relation between $t_i$ and ($\beta$, $k$, $m$).

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

*1) Secret-dependent Control Transfers:* We think a control-flow is secret-dependence if different input sensitive keys ($K$) can lead to different branch conditions. For a specific branch, the branch 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 is secret-dependence if different input sensitive keys ($K$) can lead to different memory addresses. We use the model from [2]. The low $L$ bits of the address is irrelevant in side-channels.

We think 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 following definition, we can model an attacker's observation with math formulas. For example 3, if an attacker observes the code executes 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 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 the section, we articulate several challenges and existing problems in applying the method above in real-world crypto systems.

We use symbolic execution to get the function $f_i$ and model counting to get the number of items in $K^o$. However, both symbolic execution and model counting are well-known for theirs expensive performance cost, which limit their usage in real-world crypto systems.

### A. Scalability and Performance

After we transfer each potential leaks sites into formula. We can group several formulas together to estimate the total information leakage. One naive way is to use the Monte Carlo sampling estimate the number of input keys. With the definition 1, we can estimate the total information leakage.

However, some pre-experiments show that above approach suffers from the unbearable cost, which impede its usage to detect and quantify side-channel leakages in real-world applications. We systematically analyze the performance bottlenecks of the whole process. In general, the performance suffers from the two following reasons.

- Symbolic Execution (Challenge III(a))
- Monte Carlo Sampling (Challenge III(b))

*1) Symbolic Execution:* Symbolic execution interprets each instruction and update the memory cells and registers with a formula that captured the semantics of the execution. Unfortunately, the number of machine instructions are huge and the semantics of each instruction is complex. For example, the Intel Developer Manual [7] introduces more than 1000 different X86 instructions. It is tedious to manually implement the rules for every instructions.

Therefore, existing binary analysis tools [8], [9] will translate machine instructions into intermediate languages (IR). The IR typically has fewer instructions compared to the original machine instructions. The IR layer designs, which significantly simplify the implementations, also introduce significant overhead as well [10].

**Our Solution to Challenge III(a):** We adopt the similar approach from [10] and implement the symbolic execution directly on the top X86 instructions.

*2) Monte Carlo Sampling:* For an application with $m$ bytes secret, there are total $2^{8m}$ possible inputs. Of the $2^{8m}$ possible inputs, we want to estimate the number of inputs that satisfy those formulas. Then we can use the definition /refdef to calculate the information leakage.

A Monte Carlo method for approximating the number of $|K_o|$ is to pick up $M$ random values and check how many of them satisfy those constrains. If $l$ values satisfy those constrains, then the approximate result is $\frac{l*2^{8m}}{M}$.

However, the number of satisfying values could be exponentially small. Consider the formula $F = k_1 = 1 \wedge k_2 = 2 \wedge k_3 = 3 \wedge k_4 = 4$, $k_1$, $k_2$, $k_3$ and $k_4$ each represents one byte in the orginal sensitive input, there is only one possible solution of $2^{32}$ possible values, which requres exponentially many samples to get a tight bound. The naive Monte Carlo Method also suffers from the curse of dimensionality. For example, the libjpeg libraries can transfer the image from one format into another format. One image could be 1kMB. If we take each byte in the original buffer as symbols, the formula can have at most 1024 symbols.

**Our Solution to Challenge III(b):** We adopt the Markov Chain Monte Carlo to estimate the number of possible input that satisfies the logic formula groups. The key idea is that we have one group of input that satisfies the logic formula constrains. We will introduce the method in the following subsection.

### B. Information Leakage Estimation

In this section, we present the alogorithm to calculate the information leakage based on the definition 1, answering to **Challenge III(b)**.

*1) Problem Statement:* With instruction level symbolic execution (see §V-C), we can generate the constraint for each unique leakage site on the execution trace. The only variables in those constraints are the sensitive data represented with $k_1, k_2, \ldots, k_n$. Suppose the address of the leakage site is $\xi_i$,

we use $C_{\xi_i}$ to denote the constraint. For multiple leakage sites, as each leakage site has one constraint, we use 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 calculating $\frac{|K|}{|K^o|}$. $K^o$ represents the set that contains every input keys that satisfy the constraint. As the cardinality of $K$ is known, the key problem is to know 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 $F(C_{\xi_1}, C_{\xi_2}, \ldots, C_{\xi_n}) = C_{\xi_1} \wedge C_{\xi_2} \wedge \ldots \wedge C_{\xi_n}$. 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 $F(C_{\xi_1}, C_{\xi_2}, \ldots, C_{\xi_n})$.

A native method for approximating the result is to pick $k$ elements from $K$ and check how many of them are also contained in $K^o$. If $q$ elements are also in $K^o$. In expectation, we can use $\frac{k}{q}$ to approximate the value of $\frac{|K|}{|K^o|}$.

However, as discussed in §IV-A2, the above sampling method will typically fail in practice due to the following two problems:

1) The curse of dimensionality. $F(C_{\xi_1}, \ldots, C_{\xi_n})$ is the conjunction of many constraints. Therefore, the input variables of each constraints will also be the input variables of the $F(C_{\xi_1}, \ldots, C_{\xi_n})$. The sampling method will fail as $n$ increases. For example, when $n$ equals to 2, the whole serach space is a $256^2$ cube. If we want the sampling distance between each point still to 1, we need $256^2$ points. When $n$ equals to 10, we need $256^{10}$ points if we still we want the distance between each points equals to 1.
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 can't find the satisfying assignment after sampling many points.

However, despite the two problems. We also observe the constrain $F(C_{\xi_1}, C_{\xi_2}, \ldots, C_{\xi_n})$ has the following characteristics:

1) $F(C_{\xi_1}, C_{\xi_2}, \ldots, C_{\xi_n})$ is the conjunction of several short constraints $F(C_{addr_i})$. The set contaning the input variables of $F(C_{addr_i})$ is the subset of the input variables of $F(C_{\xi_1}, C_{\xi_2}, \ldots, C_{\xi_n})$. Some constraints have completely different input variables from other constraints.
2) For each constraint $F(C_{addr_i})$, the satisfying assignments are close to each other, which means if we find one satisfying assignment, we are more likely to find other satisfying assignments nearby than randomly pick one point in the whole searching space.

In regard to the above problems, we present our methods. First, we split $F(C_{\xi_1}, C_{\xi_2}, \ldots, C_{\xi_n})$ into several independent constraint groups. After that, we run random-walk based sampling method on each constraint.

*2) Maximum Independent Partition:* For a constraint $C_{\xi_i}$, we define function $\pi$, which maps the constraint into the set consisting of input symbols. For example, $\pi(k1+k2 > 128) = \{k1, k2\}$.

**Definition IV.1.** 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 , we can split the constraint $F(C_{\xi_1}, C_{\xi_2}, \ldots, C_{\xi_n})$ into several independent constraints. There are many partitions. For our project, we are interested in the following one.

**Definition IV.2.** For the constraint $F(C_{\xi_1}, C_{\xi_2}, \ldots, C_{\xi_n})$, we call the constraint group $G_1, G_2, \ldots, G_m$ the maximum independent partition of $F(C_{\xi_1}, C_{\xi_2}, \ldots, C_{\xi_n})$ iff

1) $G_1 \wedge G_2 \wedge \ldots \wedge G_m = F(C_{\xi_1}, C_{\xi_2}, \ldots, C_{\xi_n})$
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,

$$F : k_1 = 1 \wedge k_2 = 2 \wedge k_3 = 3 \wedge k_4 = 4$$

A good partition of $F$ would be

$$G_1 : k_1 = 1 \quad G_2 : k_2 = 2 \quad G_3 : k_3 = 3 \quad G_4 : k_4 = 4$$

So instead of sampling in the four dimension space, we can sample each constraint in the one dimension space and combine them together with IV-B2 .

**Theorem IV.1.** Let $G_1, G_2, \ldots, G_m$ be a maximum independent partition of $F(C_1, C_2, \ldots, C_n)$. Let $K_c$ is the input set that satisfies constrain $c$. We can have the following equation in regard to the size of $K_c$

$$|K_{F(C_{\xi_1}, C_{\xi_2}, \ldots, C_{\xi_n})}| = |K_{G_1}| * |K_{G_2}| * \ldots * |K_{G_n}|$$

With Theorem IV.1, we can transfer the problem of counting the number of solutions in a large constraint with high dimensions into counting solutions of several small constraints. We apply the following algorithm to get the Maximum Independent Partition of the $F(C_{\xi_1}, C_{\xi_2}, \ldots, C_{\xi_n})$.

*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 reduced greatly 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 don't need to have a very precise result of the exact number of total solutions. The information is defined with a logarithmic function. We don't need to distinguish between a constraint having $10^{10}$ and $10^{10} + 10$ solutions. Second, as the constraint could be very complicated. The exact model

**Algorithm 1:** The Maximum Independent Partition

**input :** $F(C_{\xi_1}, C_{\xi_2}, \ldots, C_{\xi_n}) = C_{\xi_1} \wedge C_{\xi_2} \wedge \ldots \wedge C_{\xi_m}$
**output:** The Maximum Independent Partition of $G = \{G_1, G_2, \ldots, G_m\}$

**for** $i \leftarrow 1$ **to** $n$ **do**
  $S_{C_i} \leftarrow \pi(C_{\xi_i})$
  **for** $G_i \in G$ **do**
    $S_{G_i} \leftarrow \pi(G_i)$
    $S \leftarrow S_{C_i} \cap S_{G_i}$
    **if** $S \neq \varnothing$ **then**
      $G_i \leftarrow G_i \wedge C_{\xi_i}$
      **break**
    **end**
    Insert $C_{\xi_i}$ to $G$
  **end**
**end**

**Algorithm 2:** Metropolis 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
$P$: a probability generator
$k$: the input assignment
$n_s$: the number of satisfying assignments
$\#k$: the satisfying number of k **FIXME:** *this number is not used syntactically*
Initialization:
$\#k_0 \leftarrow \sum_{j=1}^{m} C_{i_j}(k_0)$
**for** $t \leftarrow 1$ **to** $n$ **do**
  $p \leftarrow P$
  **if** $p \geq 0.5$ **then**
    $v \leftarrow$ **RandomWalk** $v$
  **end**
  **else**
    $v \leftarrow$ **MetropolisMove** $v$
  **end**
  **if** $v$ *satisfies* $G_i$ **then**
    $n_s \leftarrow n_s + 1$
  **end**
**end**
$|K_{G_i}| \leftarrow n_s|K|/n$

counting approaches, like 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 also satisfies any constraint from $C_{i_1}$ to $C_{i_m}$. In other words, $K_{C_g i}$ should be the subset of $K_{C_1}, K_{C_2}, \ldots, K_{C_m}$. We notice that $C_i$ usually has less numbers of input compared to $G_i$. For example, if $C_{i_j}$ has only one 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 the simple trick, we can reduce the number of input while still ensure the accuracy.

*4) Error Estimation:* In this part, we analyze the accuracy of the Monte Carlo approximate result. We use central limit theorem (CLT) and propagation of uncertainty 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 run the same experiment multiple times, each time we can get a $p$. As each experiment is independent from others, according to central limit theorem, the sample mean $\bar{p}$ should satisfy normal distribution.

$$\frac{\bar{p} - E(p)}{\sigma\sqrt{n}} \rightarrow N(0,1)$$

Here $E(p)$ is the mean value of $p$ and $\sigma$ is the standard variance of $p$. If we use the observed value $\hat{p}$ to the represent 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 CleverHans, we want the error of estimated leaked information ($\Delta L$) is 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 the error of estimated leaked information is less than 1 bit. During the simulation, if $n$ and $p$ satisfy the above equation, the Monte Carlo Simulation will terminate.

## V. Design and Implementation

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

### A. Workflow

The shortcomings of the existing work inspire us to design a new tool to detect and quantify information leakage vulnerabilities in binaries. The tool has three steps. First, we run the target program with the concrete input (sensitive information) under the dynamic binary instrumentation (DBI) frameworks to collect the execution traces. After that, we run the symbolic execution to capture the fined-grained semantic information of each secret-dependent control-flow transfers and data-accesses. Finally we run the Markov Chain Monte Carlo (MCMC) to estimate the information leakage.

1) *Execution trace generation.* The design goal of CleverHans is to estimate the information leakage as precisely as possible. Therefore, we sacrifice the soundness for the precision in terms of program analysis. Previous works [2], [3] have demonstrated the effectiveness of the dynamic program analysis. We run the target binary under

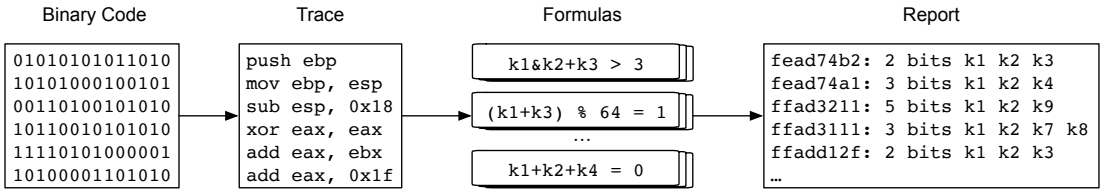



Figure 6: The workflow of CleverHans.

the dynamic binary instrumentation (DBI) to record the execution trace and the runtime information.

2) *Instruction level symbolic execution.* We model the attacker's observation about the side-channel vulnerabilities with math formula. Each formula capture the fined-grained information between the input secrets and the leakage site. For the consideration of precision and performance, we remove the intermediate language(IR) layer of the symbolic execution. Also, the engine only symbolically execute the instruction that might be affected the input key. The above design significantly reduces the overhead of the symbolic execution, which make the tool scales 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 models the observations from attacker. We propose a markov monte carlo method to estimate the number of satisfied solutions. With the help of Chebyshev's Inequality, we also give the an error estimate with the probability.

### B. Trace Logging

The trace information can be logged via some emulators (e.g., QEMU) or dynamic binary instrumentation tools (DBI). We run a program with the concrete input under the DBI to record the execution trace. The trace data has the following information:

- Each instruction mnemonics and its memory address.
- The operands of each instruction and their concrete values during the runtime.
- The value of eflags register.
- The memory address and the length of the sensitive information. Most software developers stores sensitive information in an array, a variable or a buffer, which means that those data is stored in a contiguous area in the memory. We use the symbol information in the binary to track the address in the memory.

### C. Instruction Level Symbolic Execution

The main purpose of the step is to generate constraints of the input sensitive information from the execution trace. If we give the target program a new input which is different from the origin input that was used to generate the execution trace but still satisfies those constraints, the new execution trace still have the same control flow and data access patterns.

The tool runs the symbolic execution on top of the execution traces. At the beginning of the symbolic execution, the tool creates fresh symbols for each byte in the sensitive buffer. For other data in the register or memory at the beginning, we use concrete values from the runtime information collected during the runtime. During the symbolic execution for each instruction, the tool updates every variable in the memory and registers with a math formula. The formula is made up with concrete values and the input key as the symbols accumulated through the symbolic execution. For each formula, the tool will check weather it can be reduced into a concrete values (e.g., $k_1 + 12 - k_1 = 12$ ). If so, the tool will only use the concrete values in the following symbolic execution.

*1) Verification and Optimization:* We run the symbolic execution (SE) on top of x86 instructions. In other words, we don't rely on any intermediate languages to simplify the implemetation of symbolic execution. While the implementation itself has a lot of benefits (Better performance, accurate memory model), we need to implement the symbolic execution rules for each X86 instruction. However, due to the complexity of X86, it is inevitable to make mistakes. Therefore, we verify the correctness of the SE engine during the execution. The tool will collect the runtime information (Register values, memory values) and compare them with the formula generated from the symbolic execution. Whenever the tool finishes the symbolic execution of each instruction, the tool will compare the formula for each symbol and its actual value. If the two values don't match, we check the code and fix the error. Also, if the formula doesn't contain any symbols, the tool will use the concrete value instead of symbolic execution.

*2) Secret-dependent control-flows:* An adversary can infer sensitive information from secret dependent control-flows. There are two kinds of control-transfer instructions: the unconditional control-transfer instructions and the conditional transfer instructions. The unconditional instructions, like CALL, JUMP, RET transfer control from one code segment location to another. Since the transfer is independent from the input sensitive information, an attacker was not able to infer any sensitive information from the control-flow. So the unconditional control-transfer doesn't leak any information based on our threat model. During the symbolic execution, we just update the register information and memory cells with new formulas accordingly.

The conditional control-flow transfer instructions, like conditional jumps, depending on CPU states, may or may not transfer control flows. For conditional jumps, the CPU will test if certain condition flag (e.g., CF = 0, ZF =1) is met and jump to the certain branches respectively. The symbolic

engine will compute the flag and represent the flag in a symbol formula. Because we are running on a symbolic execution on a execution trace, we know which branch is executed. If a conditional jump uses the CPU status flag, we will generate the constraint accordingly.

```
1  804841c;  movzx edx, byte ptr [ebp+var_D];     var_D = k1;   edx = zeroext(k1)
2  8048420;  movzx eax, byte ptr [ebp+var_C];     var_C = k2;   eax = zeroext(k2)
3  8048424;  add eax, edx;                        eax = zeroext(k1) + zeroext(k2);  edx = zeroext(k1)
4  8048426;  mov byte ptr [ebp+var_B], al;        var_B = k1 + k2
5  8048429;  movzx edx, byte ptr [ebp+var_C];     var_C = k2;   edx = zeroext(k2)
6  804842d;  movzx eax, byte ptr [ebp+var_A];     var_A = k3;   eax = zeroext(k3)
7  8048431;  sub edx, eax;                        eax = zeorext(k3);   edx = zeroext(k2) - zeroext(k3)
8  8048433;  mov eax, edx;                        eax = zeroext(k2) - zeroext(k3)
9  8048435;  mov byte ptr [ebp+var_9], al;        var_9 = k2 - k3
10 8048438;  movzx eax, byte ptr [ebp+var_B];     eax = k1 + k2
11 804843c;  cmp al, byte ptr [ebp+var_9];        CF =  (k1 + k2) > (k2 - k3)
12 804843f;  jb 0x8048451;                        Formula: k1 ≤  k3
13 8048441;  sub esp, 0xc;                        esp = C1 - 0xc
```

Figure 7: The workflow of CleverHans. **FIXME:** *fix the caption, fix the drawing.* **FIXME:** *duplicate label.*

For examples,

```
1
2  0x0000e781      add dword [local_14h], 1
3  0x0000e785      cmp dword [local_14h], 4
4  0x0000e789      jne 0xe7df
5  0x0000e78b      mov dword [local_14h], 0
6
```

At the beginning of the instruction segment, the value at the address of local14h can be written as $F(\vec{K})$. At the address e785, the value will be updated with $F(\vec{K})+1$. Then the code compares the value with 4 and use the result as a conditional jump. Based on the result, we can have the following formula:

$$F(\vec{K}) + 1 = 4$$

The formula, together with the memory address (0xe789) is store as a *formula tuple (address, formula)*. Each formula tuple represents one leakage site.

*3) Secret-dependent data access:* Like input-dependent control-flow transfers, an adversary can also infer sensitive information from the data access pattern as well. We try to find this kind of leakages by checking every memory operand of the instruction. We generate the memory addressing formulas. As discussed before, every symbols in the formula is the input key. If the formula does not contain any symbols, the memory access is independent from the input sensitive information and will not leak any sensitive information according to our threat model. Otherwise, we will generate the constraint for the memory addressing. We model the memory address with a symbolic formula $F(\vec{K})$. Because we also have the concrete value of the memory address $Addr1$. Inspired by the work from [2], the formula can be written as:

$$F(\vec{K}) >> L = Addr1 >> L$$

The $L$ represents the minimum memory address granularity that an attacker can observe. For example, Flush and Reload can distinguish between different cache lines, which means the value of L is 6.

*4) Information Flow Check:* CleverHans is designed to help software developers find and understand the side-channel vulnerabilities. To ease the procedure of fixing the bug, we also track the information flow for each byte of the input buffer. The step can be seen as the mutiple-tag taint analysis. With the help of the information from symbolic execution, we can implement a relative simple but relatively precise information flow track. At the begining of the analysis, CleverHans keep a track for each byte in the original buffer. When CleverHans symbolically executes each instruction in the trace, it will check every value read from registers or memory. If the value is concrete, it means the instruction has nothing to do with the original buffer. If the value is a formula, it means the orginal information pass through the instruction. Since each byte in the sensitive buffer is represented as a symbol with a unique ID, CleverHans can know which byte in the origin buffer actually goes through the instruction.

### *D. Challenge II: How to Combine the Leakage Information from Multiple Leak Sites*

Real-world software can have various side-channel vulnerabilities. Those vulnerabilities may spread in the whole program. An adversary may exploit more than one side-channel vulnerabilities to gain more information [4], [11]. In order to precisely quantify the total information leakage, we need to know the relation of those leakage sites.

```
1  unsigned char k1, k2, k3;
2  ...
3  t1 = T[k1 % 8];         // Leakage 1
4  t2 = T[k2 % 8];         // Leakage 2
5  t3 = T[k3 % 8];         // Leakage 3
6  bool flag = foo(t1, t2, t3);
7  if(flag) {
8      if(k1 > 128)        // Leakage 4
9          A();
10 }
11 else {
12     if(k2 > 128)        // Leakage 5
13         B();
14 }
15 if(k1 + k2 + k3 > 128) // Leakage 6
16     C();
```

Listing 1: Multiple leakages

Consider the running example in 1, in which $k1$, $k2$ and $k3$ are the sensitive key. The code has six different leakage. Leakage 1, 2, 3 are the secret-dependent data accesses and leakage 4, 5, 6 are the secret-dependent control-flow transfers. The attacker can infer the last three digits of $k1$, $k2$, $k3$ from leakage 1, 2, 3. So those leakages are independent. For leakage 1, 4, 6, however, we have no idea about the total information leakage.

Suppose one program has two side-channel vulnerabilities A and B, which can leak $L_A$ and $L_B$ bits respectively according to the definition 1. Depending on the relation between A and B, the total leaked information $L_{total}$ will be:

*1) Independent Leakages:* If A and B are independent leakages, the total information leakage will be:

$$L_{total} = L_A + L_B$$

*2) Dependent Leakages:* If A and B are dependent leakages, the total information leakage will be:

$$\max\{L_A, L_B\} \leq L_{total} < L_A + L_B$$

*3) Mutual Exclusive Leakages:* If A and B are mutual exclusive leakages, then only A or B can be observed for one fixed input.

$$L_{total} = \begin{cases} L_A, & \text{only } A \\ L_B, & \text{only } B \end{cases}$$

According to above definition, leakage 1, 2, 3 are independent leakages. Leakage 4, 5 are mutual exclusive leakages. For real-world applications, it is hard to estimate the total leaked information for the following reasons. First, the real-world applications have more than thousands of lines of code. One leakage site leaks the temporary value. But the value contains some information about the original buffer. It is hard to know how the the sensitive value affects the temporary value. Second, some leakages sites may be dependent. The occurrence of the first affects the occurrence of the second sites. We can't simply add them up. Third, leakage sites are in the different blocks of the control-flow graph, which means that only one of the two leakages site may be executed during the exectution.

**Our Solution to Challenge II:** Given a program $P$ with $k$ as the sensitive input, we use $k_i$ to denote the sensitive information, where $i$ is the index of the byte in the original buffer. We can represent each temporal values with a formula. There are two types of values in the formula: the concrete value and the symbolic value. We use the runtime information to simplify the formula. In other words, we only use symbolic values to represent the sensitive input. For other values that are independent from the sensitive input, we use the concrete value from the runtime information.

After that, we model each leakage sites as a math formulas. The attacker can retrieve the sensitive information by observing the different patterns in control-flows and data access when the program process different sensitive information. We refer them as the secret-dependent control flow and secret-dependent data access accordingly. For secret-dependent control transfers, we model the leakage using the path conditions that cause the control transfer. For secret-dependent memory accesses, we use a symbolic formula $F(\vec{K})$ to represent the memory address and check if different sensitive inputs can lead to different memory accesses. As long as we model each leakage with a formula. We can regard multiple leakges as the conjunction of those formulas.

### *E. Implementation*

We implement the CleverHans with 14320 lines of code in C++11. 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 tool can also report the memory address of the leakage site. To assist the developers 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 at the appendix.

Table III: CleverHans' 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 |

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 cell. Therefore, the tool may miss some leakages but will not give us any new false positives with the implementation.

## VI. Evaluation

We evaluate CleverHans with the real-world crypto libraries and non-crypto libraries. For crypto libraries, we choose OpenSSL, mbedTLS and NaCl. OpenSSL and mbedTLS are the two most commonly used crypto libraries in today's software. NaCl (pronounced "salt") is a new software library for encryption, decryption and signatures, etc. NaCl is designed to have no data flow from secrets to load address and no data flow from secrets to branch conditions. Therefore, NaCl should have no leakage under our attack model.

We build the source code into 32-bit x86 Linux executables with the GCC 8.0 on Ubuntu 14.04. Although we use use symbol information to track back leakage sites in the source code, our tool can work on stripped binaries as well. We use 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 two aspects:

1) Is CleverHans effective to detect side-channels in real-world crypto systems?
2) Can CleverHans precisely report the number of leaked bits in open source libraries?
3) Recent work has reported a number of side-channel vulnerabilities in open source libraries. Is the number of leaked bits reported by CleverHans useful to justify the sensitive level of side-channel vulnerabilities?

### *A. Evaluation Result Overview*

In this section, we present an overview of the evaluation result. CleverHans find 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.

For crypto libraries, CleverHans finds that secret-dependent memory accesses cause most leakages. CleverHans also identifies that most side-channel vulnerabilities leak very little

Table IV: Evaluation results overview. We evaluate two versions of mbedTLS and five versions of OpenSSL. CF represents secret-dependent control-flow transfers and DF represents secret-dependent data-flow transfers. Side-channel leakagas 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 | Lekage Sites | CF | DF | # Instructions | Max Leakeage | Sym. Exe. | Monte Carlo |
|---|---|---|---|---|---|---|---|---|
| | | | | | | bits | ms | ms |
| AES | mbed TLS 2.5 | 68 | 0 | 68 | 39,855 | 8 | 570 | 850 |
| AES | mbed TLS 2.15 | 68 | 0 | 68 | 39,855 | 8 | 550 | 829 |
| AES | openssl 0.9.7 | 75 | 0 | 75 | 1,704 | 10 | 319 | 7,720 |
| AES | openssl 1.0.2f | 88 | 0 | 88 | 1,350 | 12 | 72 | 1,500 |
| AES | openssl 1.0.2k | 88 | 0 | 88 | 1,350 | 11 | 83 | 1,441 |
| AES | openssl 1.1.0f | 88 | 0 | 88 | 1,420 | 12 | 87 | 1,454 |
| AES | openssl 1.1.1 | 88 | 0 | 88 | 1,586 | 8 | 91 | 1,250 |
| DES | mbed TLS 2.5 | 15 | 0 | 15 | 4,596 | 1 | 114 | 144 |
| DES | mbed TLS 2.15 | 15 | 0 | 15 | 4,596 | 1 | 106 | 137 |
| DES | openssl 0.9.7 | 6 | 0 | 6 | 2,976 | 7 | 149 | 4,193 |
| DES | openssl 1.0.2f | 8 | 0 | 8 | 2,593 | 9 | 239 | 5,311 |
| DES | openssl 1.0.2k | 8 | 0 | 8 | 2,593 | 9 | 235 | 5,080 |
| DES | openssl 1.1.0f | 8 | 0 | 8 | 4,260 | 9 | 256 | 5,027 |
| DES | openssl 1.1.1 | 6 | 0 | 6 | 8,272 | 7 | 235 | 4,584 |
| | | | | | | | minutes | minutes |
| RSA | mbed TLS 2.5 | 6 | 6 | 0 | 22,109,246 | 9 | 38 | 20 |
| RSA | mbed TLS 2.15 | 12 | 0 | 12 | 24,484,441 | 9 | 39 | 241 |
| RSA | openssl 0.9.7 | 105 | 103 | 2 | 16,980,109 | 13 | 28 | 266 |
| RSA | openssl 1.0.2f | 38 | 27 | 11 | 14,468,307 | 10 | 28 | 160 |
| RSA | openssl 1.0.2k | 36 | 27 | 9 | 15,285,210 | 12 | 39 | 282 |
| RSA | openssl 1.1.0f | 31 | 22 | 9 | 16,390,750 | 13 | 32 | 262 |
| RSA | openssl 1.1.1 | 26 | 20 | 6 | 18,207,020 | 12 | 7 | 455 |
| Total | | 883 | 205 | 678 | 128,042,089 | | 213m | 1,688m |

information in practice, which confirms our initial assumptions. However, we do find some sensitive leakages. Some of them have been confirmed by existing research that those vulnerabilities can be exploited to realize real attacks.

All the symmetric key implementations in OpenSSL and mbedTLS all yield significant leakages due to the implementation of the lookup table to speed up the computation. Every leakages 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 analyse the corresponding countermeasure in the following sections.

CleverHans find several leakage sites for both the implementation of DES and AES in OpenSSL and mbedTLS. CleverHans confirm that all those leakages come from table lookups implementations. mbedTLS 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 implementations. However, we do not see the bit-sliced implementation of AES and DES in various versions of OpenSSL and mbedTLS. However, we find the new implementation of OpenSSL instead use typical four 1K tables. It only uses 1K of 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 implementation. With the optimizations introduced in the paper, we do not apply any domain knowledge to simplify the analysis. Therefore, our tools can identify all the leakage sites reported by CacheD [2] in a shorter time. Our tool also finds that most leakages in RSA are caused in the big number implementation. We also find newer version of RSA in OpenSSL tend to have fewer leakages detected by CleverHans. We will discuss the version changes and corresponding leakages in the next section.

In addition to identifying side-channel leakages, CleverHans can also estimate how much information is leaked from each vulnerabilities. CleverHans achieve the goal by estimating number of keys that satisfy the constraints. During the Evaluation, for each leakage site, CleverHans will stop once 1) have 95% confidence possibility that the error of estimated leaked information is less than 1 bit. 2) can not reach the termination condition after 10 minutes. In the case, it means the satisfying keys is very small and the leakage is very serious. The tools mark the Monte Carlo is failed. During the evaluation, we find CleverHans can quantify every side-channels leakages for every symmetric encryptions. For asymmetric encryptions, Monte Carlo quantification sometimes fail. We manually check those leakages sites and think most of them are serious.

### *B. Case Study of vulnerabilities*

*1) AES in mbedTLS:* During our evaluation, we find mbedTLS 2.5 and 2.15.1 have the same implementation of AES. Our tool provide the same leakage report for both versions. CleverHans identify 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 the figure 8, 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 tables to speed up computation. However, CleverHans report 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. On the contrary, leakage 4, 5, 6, 7 access a smaller lookup tables (2K). Therefore, leakage 4, 5, 6, 7 leaks 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
13 AES_FROUND( Y0, Y1, Y2, Y3, X0, X1, X2, X3 );     // Leakage 3
14
15 X0 = *RK++ ^ \                                    // Leakage 4
16     ( (uint32_t) FSb[ ( Y0       ) & 0xFF ]       ) ^
17     ( (uint32_t) FSb[ ( Y1 >>  8 ) & 0xFF ] <<  8 ) ^
18     ( (uint32_t) FSb[ ( Y2 >> 16 ) & 0xFF ] << 16 ) ^
19     ( (uint32_t) FSb[ ( Y3 >> 24 ) & 0xFF ] << 24 );
20
21 // X1, X2, X3 do the same computation as X0
22 ...                                          // Leakage 5,6,7
23
24 PUT_UINT32_LE( X0, output,  0 );
25 ...
26 return( 0 );
27 }
```

Figure 8: mbedtls_internal_aes_encrypt

*2) RSA in mbedTLS:* CleverHans identify several side-channel leakages for the RSA implementation in MbedTLS. Here we introduce and analyze two cases.

```
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 9: mbedtls_mpi_exp_mod

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

Figure 10: mpi_mul_hlp

### C. Case Study of RSA in OpenSSL

### D. Analysis of Software Countermeasures

*1) Bit-slicing:*

*2) Scatter and Gather:*

*3) Smaller Lookup Tables:*

## VII. Discussion

### A. Binary vs. Source Code

Many existing works find side-channels vulnerabilities from source code level or intermediate languages (e.g., LLVM IR). Those approaches will have the following limitations. First, many compilers can translate the operator into branches. For example, the GCC compiler will translate the ! operator into conditional branches. If the branch is secret-dependent, the attacker could learn some sensitive information. But those source-based methods will fail to detect those vulnerabilities. Second, some if else in the source code can be converted into single conditional instructions (e.g., cmov). Source-based methods will still regard them as potential leakages, which will introduce false positives.

### B. x86 vs. Intermediate Languages

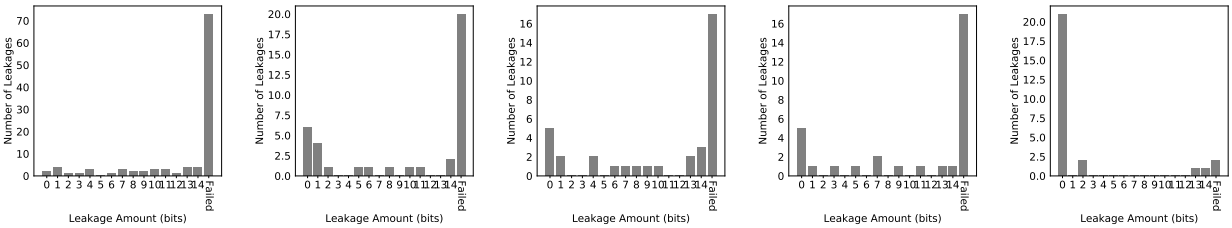
The tool takes in an unmodified binary and the marked sensitive information as the input. The tool will first start with logging the execution trace. Then the tool will symbolizes the sensitive information into multiple symbols and start with the symbolic execution. During the symbolic execution, the tool will find any potentials leakage sites as well as the path constraints. For each leakage site, whether it is the tool will also generate the constraint. Then, the tool will split the constraints into multiple group. After that, the tool will run monte carlo sampling to estimate the total information leakages.

## VIII. Related Work

### A. Side-channel Vulnerability Detection

There are a large number of works on side-channel vulnerability detections in recent years. CacheAudit [1] uses abstract domains to compute the over approximation of cache-based side-channel information leakage upper bound. However, due to over approximation, the leakage given by 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 [12] improves the work of CacheAudit by proposing the novel abstract domains, which only precisely track secret related code. Like CacheAudit, CacheS can not provide the information to indicate the sensitive level of side-channel vulnerabilities.

The dynamic approach, usually comes with taint analysis and symbolic execution, can perform a very precise analysis. CacheD [2] takes a concrete program 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 [3] detects address-based side-channel vulnerabilities by comparing execution traces under various tests. DATA can detect various side-channel vulnerabilities,




(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 11: RSA implementation in different versions of OpenSSL
For crypto libraries, it is likely that a newer version has less vulnerabilities compared to the previous version.

### B. Quantification

Quantitative Information Flow (QIF) aims at providing an information leakage estimate for the sensitive information given the public output. If zero bits of the information is leaked, the program is called non-interference. McCamant [13] quantify the information leakage as the network flow capacity. Phan [14] 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 doesn't work for secret-dependent memory access side-channels. CHALICE [15] quantifies the information for a given cache behaviours. But CHALICE can only work on PICA code (a MIPS like architecture), which limits its usage on x86 bianry. CHALICE can only scale to small programs, which limits its usage in real-world applications.

## 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 prototype called present CleverHans and show that CleverHans is effective at 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 CleverHans is useful in estimating the amount of leaked information in real-world applications.

## References

[1] 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)*. Washington, D.C.: USENIX, 2013, pp. 431–446.

[2] 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)*. Vancouver, BC: USENIX Association, 2017, pp. 235–252.

[3] 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)*. Baltimore, MD: USENIX Association, 2018, pp. 603–620.

[4] 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*, May 2015, pp. 640–656.

[5] G. Smith, "On the foundations of quantitative information flow," in *Foundations of Software Science and Computational Structures*, L. de Alfaro, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 288–302.

[6] 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. New York, NY, USA: ACM, 2018, pp. 161–173. [Online]. Available: http://doi.acm.org/10.1145/3274694.3274741

[7] Intel Corporation, *Intel® 64 and IA-32 Architectures Software Developer's Manual*, December 2009, no. 253669-033US.

[8] 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.

[9] 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. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 463–469.

[10] 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)*. Baltimore, MD: USENIX Association, 2018, pp. 745–761.

[11] 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)*. Washington, D.C.: USENIX Association, 2015, pp. 897–912.

[12] "Identifying cache-based side channels through secret-augmented abstract interpretation," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, 2019.

[13] 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*, Tucson, AZ, USA, Jun. 2008, pp. 193–205.

[14] Q.-S. Phan, P. Malacaria, O. Tkachuk, and C. S. Păsăreanu, "Symbolic quantitative information flow," *SIGSOFT Softw. Eng. Notes*, vol. 37, no. 6, pp. 1–5, 2012.

[15] 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, New York, NY, USA, 2017, pp. 25–35.