TANA: Precise Fine-grained Side-channel Inforamtion Leakage Quantification in Binaries

Anonymous

***Abstract*—Side-channel attacks allow attackers to infer some sensitive information based on non-functional characteristics. Existing works on address-base side-channel detection can provide a list of potential side-channel leakage sites. We observe the following limitations in the previous work: 1) Some vulnerabilities could be more severe than others, but the existing work cannot tell precisely the difference between those leakages. 2) An attacker usually exploits multiple leakages at one time. However, no existing tool can precisely report the sum of the information leakage from multiple sites given the information dependency among them.**

—FIXME: EMPHASIZE NOVELTY AND METHOD— To overcome the above limitations, we propose a novel method to more precisely quantify the information leakage. Previous work only considers the "average" information leakage which often neglect severe scenarios. In contrast, our method is precise and fine-grain. Our method can also precisely combine two leakages. Our results are superisingly different compared to previous results and much more useful in practice.

—FIXME: NEED A BETTER NAME— We have developed a tool called TANA, which can not only find the side-channel vulnerabilities but can estimate how many bits are actually leaked. TANA works in three steps. First, the application is executed to record the trace. Second, TANA runs the instruction level symbolic execution on the top of the execution trace. TANA will find side-channel information leakages and model each leakage as one unique math constraint. Finally, TANA will classify those constraints into independent multiple groups and run the multiple step Monte Carlo to estimate the information leakage. TANA can report a very fined-grained vulnerability result compared to the existing tools. We apply the tool on OpenSSL, MbedTLS and libjpeg and find several serious side channel vulnerabilities. We also evaluate the vulnerabilities from previous research. The result confirms our intuition; it indicates most of the reported vulnerabilities are actually hard to exploit in practice.

I. INTRODUCTION

Side channels are inevitable in modern computer systems. The attacker can infer the sensitive information by observing the execution behaviour. The software-based attacks(e.g., Cache-based side channels [1]–[4], controlled channel attacks [5]) are especially common because these attacks typically don't need any physical access. After examing the root cause of those vulnerabilities, we believe that the implementation of those software, which has different memory access patterns depending on the input secret during the execution, cause the following address-based side channels. Those side-channel vulnerabilities can be exploited by observing the different memory access patterns. An attacker can observe the pattern and infer the sensitive information.

Various countermeasures [6]–[8] have been proposed to defend against the address-based side-channel attacks. The basic idea is simple, identifying and eliminating secret-dependent memory access patterns. However, it can be very tedious and error-prone to find each leakage site manually. To address the problem, a lot of methods have been proposed to automatically detect information leakages. While those tools can typically report a list of potential leakages sites, they still face the following limitations.

First, existing tools [7], [8] can discover a list of potential leakage sites. But they fail to report how severe each potential leakage site could be. Most of leakages found by those tools are typically hard to be exploited and leak very little information (e.g., one bit of the length of the key [7]). It is worthwhile to have a tool to tell how much information is actually leaks. For example, some secret dependent memory access patterns in OpenSSL have been known for years. But the authors still don't fix them all because they think those vulnerabilities arent not severe enough. The tool based on static analysis [6] with abstract interpretation, can give inforamtion leakage upper bound, which is useful to justify the implementation is secure enough. But they cant indicate the leakage site is severe or not due to the over approximation when those tools calculate inforamtion leakage. The dynamic method will take a concrete input and run the actual program. Those tools are typically very precise in term of true leakage. But none of those tools can actually report how many bits are actually leaked. For example, DATA [8] reports more than 2000 potential leakage sites for the RSA implementation of OpenSSL. But most of them were dismissed by the author after some manual inspections.

Second, many open source libraries may have multiple leakage sites. A side-channel attacker [2], [5], [9] can exploit more than one leakage sites at one time. The attacker may retrieve part of the sensitive information from one site, part of the sensitive information from another site and combine them together. It is hard know how much information is actually leaked in total. Adding those leakages simply gets a very rough upper bound estimate of the total information leakage if those leakages arent independent. No existing tools can practically estimate the total information leakage from multiple leakage sites in real world open source libraries.

To overcome the above limitations, we propose a novel method to more precisely quantify the information leakage. Previous work only considers the "average" information leakage which often neglects severe scenarios. The average information assumes that the target program will have **different** sensitive information as the input when the attacker launch the attack, which often does not match the attack scenario. During the real side-channel attacks, an adversary may run the target problem again and over again with the fixed unknown sensitive information as the input. Therefore, the previous threat model can't catch the real attack scenarios.

In contrast, our method is more precise and fine-grained. For our analysis, the input key is fixed. The average leakage does not fit in this scenario. The key insight is ... **FIXME:** *Qinkun: complete this and i will revise.* **Qinkun's Response:** *During the real side-channel attacks, an adversary may run the target problem again and over again with the fixed unknown sensitive information as the input and do the profiling. Therefore, the previous threat model can't catch the real attack scenarios.* By fixing with more constraints, we are able to obtain more precise information leakage. Our results are superisingly different compared to previous results and much more useful in practice.

We define the amount of leaked information in a simple but effective way. It is interesting to mention that the definition here is different from the definition in several static analysis tools. Our method is based on ... **FIXME:** *Qinkun: explain in high level, but give a novelty sense on how our method work, the method or theory, not workflow, not implementation.* **Qinkun's Response:** *Here is the intuition. Before the attack, the adversary has a big but finite input space. Every time the adversary observes one leakage site, he can eliminate some potential input and reduce the size of the input space. The smaller the input space is, the more information is actually leaked. On an extreme situation, if the size of the input space reduce to one, the adversary can determine the input information uniquely, which means the total information is leaked. We define the leaked information as the proportion of the number of possible input that satisfies the attacker's observation.*

Our method can identify and quantify address-based sensitive information leakage sites in real-world applications automatically. Adversaries can exploit different control-flow transfers and data-access patterns when the program processes different sensitive data. We refer them as the potential information leakage sites. Our tool can discover and estimate those potential information leakage sites as well as how many bits they can leak. We are also able to report precisely how many bits can be leaked in total if an attacker observes more than one site. We run symbolic execution on execution traces. 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 every the side-channel vulnerability. In other words, if the application has a different sensitive input but still satisfies the formula, the code can still leak the same information. Those information leakage sites may spread in the whole program and their leakages may not be dependent. Simply adding them up can only get a coarse upper bound estimate. In order to accurately calculate the total information leakage, we must know the dependent relationships among those multiple leakages sites. Therefore, we introduce a monte carlo sampling method to estimate the total information leakage.

We implement the proposed method as a tool named TANA which can precisely discover and quantify the information leakage vulnerabilities. We apply TANA on several crypto and non-crypto libraries including OpenSSL, MbedTLS and libjpeg. The experiment result confirms that TANA can precisely identify the pre-known vulnerabilities, report how much information is leaked and which byte in the original sensitive buffer is leaked. The result shows that while those crypto libraries have a number of side-channels, most of them actually leak very little information. Also, we also use the tool to analysis the two reported side-channel attack in the libjpeg library. Finally, we present new vulnerabilities. With the help of TANA, we confirm those vulnerabilities are easily to be exploited.

In summary, we make the following contributions:

- We propose a novel method that can quantify fine-grained side-channel information leakages. We model each information leakage vulnerability as a formula and mutiple side-channel vulnerabilities can be seen as the disjunctions of those formula, 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 to several real-world software. TANA successfully identify the address-based side-channel vulnerabilities and provide the corresponding information leakge. The information lekage result provide the detailed information that help developers to fix the vulnerability.

II. Background

In this section, we first present a basic introduction about the memory-based side-channel attacks. Those attacks are what we attempt to study in the paper. After that, we will present existing methods on side-channel detections and quantifications.

A. Address-based Side-Channels

Address-based side-channels are information channels that can leak sensitive information unintendedly through the different behaviors when the program accesses different memory addresses. Fundamentally, those differences were caused by the memory hierarchy design in modern computer systems. When the CPU fetches the data, it first searches the cache, which stores copies of the data that was frequently used in main memory. If the data doesn't exist in the cache, the CPU will read the data from the main memory (RAM). According to the layer that causes the side-channels, we classify the address-based side channels into two catogories: cache-based side-channel attacks and memory-based side-channel attacks.

1) Cache-based Attacks: In general, the cached-based side-channels seek information relying on the time differences between the cache miss and cache hit. Here we introduce two common cache attacks: PRIME+PROBE, FLUSH+RELOAD.

PRIME+PROBE targets a single cache set. It has two phases. During the “prime” phase, the attacker fills the cache set with his own data. In the second “probe” phase, the attacker accesses the cache set again. If the victim accesses the cache set and evicts part of the data, the attacker will experience a slow mesurement. If not, the mesurement will be fast.

FLUSH+RELOAD targets a single cache line. It requires the attacker and victim share the same memory. It also have two phases. During the ”flush” phase, the attacker will flush the “monitered memory” from the cache. Then the attacker wait for the victim to access the memory. In the next phase, the attacker reload the “monitered memory”. If the time is short, which indicates there is a cache hit and the victim reolads the memory before. On the other hand, the time will be longer since the CPU need to reolad the memory into the cache line.

2) Memory-based Attack: Memory-based side-channel attack [5] exploits the different behaviors when the program accesses different page tables. For example, the controlled-channel attack [5], which works in the kernel space, can infer the sensitive data in the shielding systems by observing the page fault sequences after restricting some code and data pages.

After examing the memory-based side-channels attack. We find the root cause of those attacks are secret-dependent memory access and control flow transfers.

```
1  void sample_encrypt_setup
2                  (encrypt_ctx *ctx,
3                  const unsigned char *key)
4  {
5      int i, j, k;
6      unsigned char *m, T[256];
7      m = ctx->m;
8      for(i = 0 ; i < 256; i++, k++)
9      {
10         // Secret Dependent Memory Access
11         m[i] = Table[key[k] % 256];
12         // Secret Dependent Control Flow Transfer
13         if(key[k] == 0)
14         {
15             m[i] = k * key[i];
16             m[i] = m[i] % 256;
17         }
18     }
19
20 }
```

Listing 1: Sample code shows secret-dependent memory access and secret-dependent control-flow transfer.

For example, the above code 1 show an simple encryption function that has the two kinds of side-channels. At the line 11, dependending on the value of key, the code will access the different entry in the predefined table. At the line 13, the code will do a series of computation and determine if the code in the if branch is executed or not. Such vulnerabilities are called the memory-based side-channles. We identify and quantify the leakage of the two kinds of vulnerabilities in the paper.

B. Information Leakage Quantification

Given an event e which 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.

The above definition is obvious. Suppose a char variable a in C program has the size of one byte (8 bits), so the value in the variable can range from 0 - 255. We assume the a has the uniform distribution. If at one time we observe the a equals to 1, the probability will be $\frac{1}{256}$. 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 are based on mutual information or min-entropy [10]. In their frameworks, the input sensitive information K is viewed as a random variable. Let k_i be one of the possible value of K. The Shannon entropy $H(K)$ is defined by

$$H(K) = -\sum_{k_i \in K} P(k_i) \log_2 P(k_i)$$

The Shannon entropy can be used to quantify the initial uncertainty about the sensitive information. Suppose a program (P) with the K as the sensitive input, an adversary has some observations (O) through the side-channels. In this work, the observations are referred to the secret-dependent control-flows and secret-dependent data-accesses patterns. The conditional entropy $H(K|O)$ is

$$H(K|O) = -\sum_{o_j \in O} P(o_j) \sum_{k_i \in K} P(k_i|o_j) \log_2 P(k_i|o_j)$$

Intuitively, the conditional information marks the uncertainty about K after the adversary has gained some observations (O).

Many previous works use the mutual information $I(K; O)$ to quantify the leakage which is defined as follows:

$$\textit{Leakage} = I(K; O) = \sum_{k_i \in K} \sum_{o_j \in O} P(k_i, o_j) \log_2 \frac{P(k_i, o_j)}{P(k_i)P(o_j)}$$

where $P(k_i, o_i)$ is the joint discrete distribution of K and O. Alternatively, the mutual information can also be computed with the following equation:

$$\textit{Leakage} = I(K; O) = H(K) - H(K|O) = H(O) - H(O|K)$$

For a deterministic program, once the input K is fixed, the program will have the same control-flow transfers and data-access patterns. As a result, $P(k_i, o_j)$ will always equals to 1 or 0. So the conditional entropy $H(O|K)$ will equal to zero. So the leakage defined by the mutual information can be simplified into:

$$\textit{Leakage} = I(K; O) = H(O)$$

In other words, once we know the distribution of those memory-access patterns. We can calculate how much information is actually leaked.

Another common method is based on the maximal leakage [6], [10], [11]. The formal information theory can prove that the maximal leakage is the upper bound of the mutual information (Channel Capacity).

$$Leakage = \log(C(O))$$

$C(O)$ represents the number of different observations that an attacker can have.

Now we provide a concrete example to show how the two types of quantification definition works.

```
unsigned char key = input();
// key = [0 ... 255]
if(key = 128)
    A(); // branch 1
else if (key < 64)
    B(); // branch 2
else if (key < 128)
    C(); // branch 3
else
    D(); // branch 4
```

Listing 2: A simple program

Maximal leakage Depending on the value of key, the code can run four different branches which corrosponding to four different observations. Therefore, by the maximal leakage definition, the leakage equals to $\log 4 = 2$ bits.

Mutual Information If the key satisfies the uniform distribution, the probability of the code runs each branch can be computed with the following result: Therefore, the leakage

Branch	A	B	C	D
P	1/256	64/256	64/256	127/256

TABLE I: The distribution of observations

equals to $\frac{1}{256}\log\frac{1}{256} + \frac{1}{4}\log\frac{1}{4} * 2 + \frac{127}{256}\log\frac{127}{256} = 1.7$ bits.

III. 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. But 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.

IV. Overview

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.

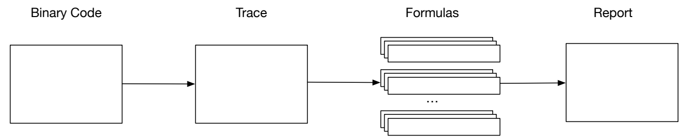


Fig. 1: The workflow of TANA

Finally we run the Markov Chain Monte Carlo (MCMC) to estimate the information leakage.

1) *Execution trace generation.* The design goal of TANA 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 [7], [8] have demonstrated the effectiveness of the dynamic program analysis. We run the target binary under 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.

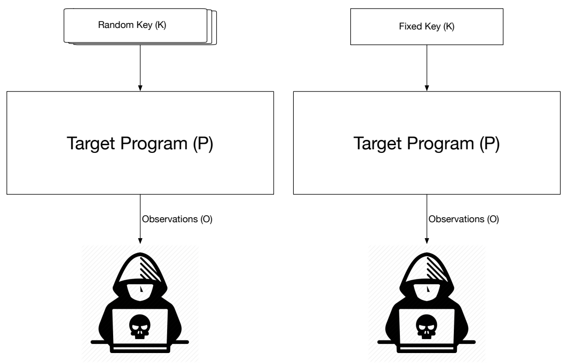
V. Chanllenges

In this section, we articulate several chanllenges and existing problems in quantifying the side-channel vulnerability leakages. We describe the chanllenges and then briefly present the corresponding solution.

A. Information Leakage Definition

Existing static-based side-channel quantification works [6] define information leakage as the mutual information or the maximal leakage. Those definitions provide a strong security guarantee when trying to prove a program is secure enough if the their methods calculate the program leaks zero bits of information.

However, the above definition is less useful to justify the sensitive level of leakage sites. Considering the example code 2 in the previous section, if an attacker knows the code executes branch A by some observations, the attacker can know the key actually equals to 128. Suppose it is a dummy



(a) Mutual Information (b) Real Attack

password checker, 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 unsigned char. According to the mutual definition, however, the leakage will be 1.7 bits. The maximal information leakage is 2 bits. Both approaches fail to tell how much information is actually leaked during the execution precisely.

The problem with the 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 the mutual information and the max-leakage give an "average" estimate of the information leakage. But it isn't the typical scenario for an adversary to launch the 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 doesnt work well in those situations.

Solution: In the project, we hope to give a very precise definition of information leakages. Suppose an attacker run the target program mutiple times with one fixed input, we want to know how much information he can infer by oberserving the memory access patterns. We come to the simple slogan [10] where the information leakage equals:

initial uncertainty - remaining uncertainty

If an adversary has zero knowledge about the input before the attack. The initial uncertainty equals to the size the input. As for the remaining uncertainty, we come to the original definition of the information content.

We quantify the information leakage with the following definition.

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

$$P(k) = o$$

The leakage L_{Pko} based on the oberservation is

$$L_{Pko} = log_2|K| - log_2|K^o|$$

where

$$K^o = \{k^{'} | k^{'} \in K \ and \ P(k^{'}) = o\}$$

With the new definition, if the attacker observes that the code 2 runs the branch 1, then the $K^{o^1} = \{128\}$. Therefore, the information leakage $L_{Pko^1} = log_2 256 - log_2 1 = 8$ bits, which means the key is totally leaked. If the attacker observes the code runs other branches, the leaked information is shown in the following table.

Branch	1	2	3	4
K^o	1	64	64	127
L_{Pko}(bits)	8	2	2	1

TABLE II: The leaked information by the definition 1

With the same definition 1, if the attacker observes that the code run branch 2, the information leakage will be 2 bits. The conclusion is consistent with the intuition. Because if the branch 2 was executed, we can know the key is less than 64. So we know the most and the second significant digits of the value key equals to 128.

B. 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 [2], [5]. For example, the controlled-side channel attack [5], the author demonstrates an attack against a popular spell checking tool, Hunspell. By observing four sets of secret-dependent memory accesses sites in two functions $HashMgr :: addword$ and $HashMgr :: lookup$, the author can recover the word in the original text.

For the Hunspell, the attacker manually studies the source code of Hunspell, figure out the relation of those vulnerabilities and launch the attack. In order to precisely quantify the total information leakage, we need to know the relation of those leakage sites.

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

Listing 3: Multiple leakages

Considering the running example in 3, 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.

It is very hard to estimate total leakages. For example, 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.

For the real program, it is tough to estimate the total information leakage for the following reasons. First, the real-world applications have more than thousands of lines of code. Side-channel vulnerabilities could exist in many different functions of the source 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 simply add them up. Third, leakage sites are in the different block of the control-flow graph, which means that only one of the two leakages site may execute during the exectution.

Suppose one program has two side-channel vulnerabilities A and B, which leaks L_A and L_B bits during the execution. The total leaked information is noted as L_{Total}. The relation between A and B have the following three cases.

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:

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

3) Mutual Exclusive Leakages: If A and B are mutual exclusive leakges, then only A or B can be observed for one fixed input. The total information leakage will be L_A or L_B.

According to above definition, leakage 1, 2, 3 are independent leakages. Leakage 4, 5 are mutual exclusive leakages.

Solution: We run the symbolic execution on the top of the execution traces. At the beginning of the exectution, each byte in the sensitive buffer is modeled with with a symbol. After that, the symbolic exectution engine interprets each instruction of the exectution traces. So every values in the registers or memory cells is modeled with a math formula.

Given a program P, k is the sensitive input. The k should be a value in a memory cell or a sequential buffer (e.g., an array). We use k_i to denote the sensitive information, where i is the index of the byte in the original buffer. We can have the following equations. The t1, t2, t3, is the temporary values during the execution.

$$t_1 = f_1(k_1, k2, ...k_n)$$

$$t_2 = f_2(k1, k2, ...k_n)$$

$$t_3 = f_3(k1, k2, ...k_n)$$

...

$$t_m = f_m(k1, k2, ...k_n)$$

After that, we model each potential 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.

4) Secret-dependent Control Flow: Here is an example of the secret-dependent control-flows. Consider the code snippet in List 1. Here the key is the confidential data. The code will have different behaviours (time, cache access) dependenting on which branch is actually executing. By observing the behaviour, the attacker can infer which branch actually executed and know some of the sensitive information. One of the famous leakage example is the square and multiply in many RSA implementations.

For example, the attacker knows the key equals to zero if he observes the code run the branch1. Because key has 256 different possibilities. The original key has lg256 = 8 bits information. If the attacker can observe the code run branch 1. Then he will knows the key equals to zero. If the code run branch 2, the attacker can infer the key doesnt equal to zero.

```
Branch 1
temp = 0xb;
0 =< key <= 256;
temp = key/2;
```

Information Leakage = -log(1/p) = -log(1/256) = 8 bits

```
Branch 2
temp != 0;
0 =< key <= 256;
temp = key/2;
```

Information Leakage = -log(255/256) bits

```
T[64]; // Lookup tables with 64 entries
index = key % 63;
temp = T[index];
// Secret-dependent memory access
```

The simple program above is an example of secret dependent memory access. Here T is a precomputed tables with sixty-four entries. Depending on the values of key, the program may access any values in the array. Those kind of code patterns may wildly exist in many crypto and media libraries.

Suppose the attackers observe the code accesses the first entry of the lookup tables. We can have the following formulas.

```
key mod 63 = 1
0 =< key <= 256
```

So the key can be one of the following values: 1 64 127 190 253

Information leakages = -log(5/256) = 5.6 bits

C. Scalability and Performance

After we transfer each potential leaks sites into logic 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 unberable 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.
- The Naive Monte Carlo Sampling.

1) Symbolic Execution: Symbolic execution interprets each instruction and update the memory cells and registers with a formula that captured the sementics of the exectution. Unfortunately, the number of machine instructions are huge and the sementics of each instruction is complex. For example, the Intel Developer Manual [12] introduces more than 1000 different X86 instructions. It is tedious to manually implement the rules for every instructions.

Therefore, existing binary analysis tools [13], [14] 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 [15].

Solution: We adopt the similar approach from [15] and implement the symbolic exectution 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 secrets, we want to know many of them of satisfy the given logic formula groups. Then we can use the definition /refdef to calculate the information leakage.

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

However, the number of satisfying values could be exponentially small. Considering 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 represent one byte in the sensitive buffer, there is only one possible solution of 2^{32} possible values. 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.

Solution: 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 can also estimate the error with chebyshev inequality. We will introduce the method in the following sections.

VI. DESIGN

In this section, we will explain the design decisions to realize TANA.

A. Trace Logging

The trace information can be logged via some emulators (e.g., QEMU) or Binary Instrumentation Tools. For our project, we write an Intel Pin Tool to record the execution traces. 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.

B. Instruction Level Symbolic Execution

The main purpose of the this step is to generate constraints of the input sensitive information for 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 will still have the same control flow and data access patterns.

The tool runs the symbolic execution on the 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 of the symbolic execution, we use the concrete value from the runtime information collected in the previous step. During the symbolic execution, the tool will maintain a symbol and a concrete value for every variables in the memory and registers. The formula is made up with concrete values and the input key as the symbols calculated through the symbolic execution. For each formula, the tool will check weather it can be reduced into a concrete values. If so, the tool will only use the concrete values in the following symbolic execution.

C. Verification and Optimization

We run the symbolic execution on the top of x86 instructions to achieve the better performance and accuracy of the memory model. In other words, we dont rely on any intermediate languages to simplify the symbolic execution. While the implementation itself has a lot of benefits, we need to implement the symbolic execution rules for more than one thousand x86 instructions. However, due to the complexity of X86, it is inevitable to make mistakes. Therefore, we verify the correctness of the symbolic execution. The tool will collect the runtime information (Register values, memory values) and compare them with the values 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 dont match, we check the code and fix the error. Also, if the formula doesnt contain the any symbols, the tool will use the concrete value instead of symbolic execution.

D. 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 doesnt leak any information based on our threat model. During the symbolic execution, we just update the register information and memory cells with the new formulas.

The conditional transfer instructions, like conditional jump, may or may not transfer control, depending the CPU state. For any conditional jump, the CPU will test if certain condition flag (e.g., CF = 0, ZF =1) is true or false and jump to the certain branches respectively. So 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 executes. If a conditional jump uses the CPU status flag, we will generate the constraint accordingly.

For examples,

```
...
0x0000e781        add dword [local_14h], 1
0x0000e785        cmp dword [local_14h], 4
0x0000e789        jne 0xe7df
0x0000e78b        mov dword [local_14h], 0
```

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

$F(K) + 1 = 4$

E. Secret-dependent data access

Like control-flows, 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 formula. As discussed before, every symbols in the formula is the input key. If the formula doesnt contain any symbols, the memory access is independent from the input sensitive information and wont leak any sensitive information according to our threat model. Otherwise, we will generate the constraint for the memory addressing.

Monte Carlo Volume Sampling From the above steps, we already have the constraints from the execution trace. The only variables in those constraints is the sensitive data. An adversary who wants to infer the sensitive data based on side-channel attacks cant observe the sensitive information directly. The adversary, however, can observe the memory access pattern of the software.

We use the Monte Carlo Sampling to calculate the ratio of input that satisfies those constraints. We will estimate how much information is actually leaked from each input key.

Normalizations The goal of the normalizations is to simplify the constraints. Each constraint will be evaluated multiple times during the following sampling, we would like to make those formula simpler to reduce the whole execution time. Each formula is implemented as a abstract syntax tree. We apply a series normalization rules (e.g. key1 xor key1 = 0) to simplify the formula.

Key1 + 1 + 2 + key3 ¡ key2 =¿ key1 + 3 ¡ key2

Split the independent constraints into multiple groups Each constraints may have multiple symbols as the input. If each two formulas have complete different inputs, then those leakages modeled by the constraints are independent.

Multiple Stage Monte Carlo Sampling After the above steps, each side channel leakage vulnerability is transfered into a formula. The only symbol in the orginal formula is the key. An adversary who wants to infer the sensitive data based on side-channel attacks cant observe the sensitive information directly.

The adversary, however, can observe the memory access pattern of the software. If the attacker can observe one leakage site, the attack is modeled as one formula. If he observe mutiple information leakage, the attack is modeled as DNF formula. Calculating the total information leakage can be reduced to the problem of approximating the number of solutions. One intution way is to use the Monte Carlo method. However, the number of satisfying keys could be exponentially small. Image an attacker who can recover the one unique key after the attacks, in such case, the DNF formula F = (f(k)) will only have one solution.

In order to estimate how much information is actually leaked from each input key. we have to calculate the ratio of input that satisfies those constraints. One intuition is to use the Monte Carlo method to approximate the number of solutions.

VII. Discussion on implementation

A. Binary code 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 questions. 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. Implementation

We implement the TANA with 12K lines of code in C++11. It has three components, Intel Pin tool that can collect the execution trace, the intruction-level symbolic execution engine and the backend that can estimate the information leakage.

IX. Evaluation

We evaluate TANA with the real-world crypto libraries and non-crypto. For crypto libraries, we choose OpenSSL and mbedTLS, two most commonly used crypto libraries in today's software. For OpenSSL, we compile it with the option *no-asm* to disable the assembly language routines. For non-crypto libraries, we study the libjepg, a commonly used lossy library implemented by the Independent JPEG Group. We build the source code into a 32 bit ELF binary with the GCC 8.0 on Ubuntu 14.04. Although our tool can work on stripped binaries, we enable the symbol information to trace back leakage sites into the source code.

We use Intel Pin version 3.7 to record the execution trace.

A. Evaluation Result Overview

X. Related Work

A. Side-channel vulnerabilities detection

There are a large number of works on side-channel vulnerability detections in recent years. CacheAudit [6] uses abstract domains to compute the overapproximation 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 [16] imporves the work of CacheAudit by proposing the novel abstract domains, which only precisely track secret related code. Like CacheAudit, CacheS can't 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 [7] 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 the similar idea to model the secret-dependent memory accesses. DATA [8] detects address-based side-channel vulnerabilities by comparing execution traces under various tests.

B. Quantification

Quantitative Information Flow (QIF) aims at providing an information leakage estimate for the sensitive information through program analysis.

References

[1] 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)*. San Diego, CA: USENIX Association, 2014, pp. 719–732. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom

[2] 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. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/gruss

[3] 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*, May 2015, pp. 605–622.

[4] 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. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 1–20. [Online]. Available: http://dx.doi.org/10.1007/11605805_1

[5] 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.

[6] 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. [Online]. Available: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/doychev

[7] 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. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/wang-shuai

[8] 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. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/weiser

[9] T. Hornby, "Side-channel attacks on everyday applications: distinguishing inputs with flush+ reload. black hat usa (2016)," 2011.

[10] 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.

[11] B. Kopf, L. Mauborgne, and M. Ochoa, "Automatic quantification of cache side-channels," in *Computer Aided Verification*, P. Madhusudan and S. A. Seshia, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 564–580.

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

[13] 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.

[14] 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.

[15] 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. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/yun

[16] "Identifying cache-based side channels through secret-augmented abstract interpretation," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, 2019. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/wang-shuai