TANA: Fined-grained Side-channel Inforamtion Leakage Quantifications 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-channels leakage sites. We observed that those works still have the following limitations: 1) Many software may have multiple information leakage sites. Some vulnerabilities could be more severe than others. But existing work couldnt tell the difference between those leakages. 2) An attacker could exploit multiple leakages at one time. However, no existing tool can report how much information is leaked in total.**

To overcome the above limitations: we proposed a tool called TANA, which can not only find the side channels but can estimate how many bits are actually leaked through the leakage. TANA works in three phases. 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 constraints. Finally, TANA will classify those constraints into independent multiple groups and run the multiple step monte carlo to estimate the information leakage. TANA will a very fined-grained result compared to 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 indicates most of the reported vulnerabilities are actually hard to exploit in practice.

I. Introduction

Side channels are inevitable in computer systems. The attacker can infer the sensitive information by observing execution behaviours. The software-based attackers(e.g., Cache-based side channels, controlled side attacks) are especially common because these attacks typically dont need any physical access. We observe that the root cause is that the implementation of the program may have different memory access patterns during the execution. Attackers can observe the pattern and infer the sensitive information based on source code.

Various countermeasures have been proposed to defend the address-based side-channel attacks. The basic idea is to identify and eliminate secret-dependent memory access patterns. It is very tedious and hard to manually find every the leakage site. To address the problem, a lot of methods have been proposed to automatically detect information leakage. While those tools can typically report a list of potential leakages sites, they still face a couple of limitations.

First, the existing tools can only report 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 exploit or leak too little information. It is worthwhile to have a tool to tell how much information is actually leaks. For example, many secret dependent memory access patterns in OpenSSL have been known and studied for years. But the authors still doesnt fix them all because they think they arent not severe. The tool based on static analysis with abstract interpretation, can give leakage leakage upper bound, which is useful to justify the implementation is secure enough. But they cant indicate the leakage site is severe enough due to over approximation. 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 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 information leakage sites. A side-channel attacker usually exploits multiple leakage sites at one time. The attacker may retrieve some information from one site and some other information from another site. It is hard to model how much information is actually leaked in totoel. Adding those leakage simply can get only get an upper bound estimate of total information leakage if those leakages arent independent. No existing tools can practically estimate the total information leakage from multiple leakage sites in open source libraries.

In the paper, we presented a tool that can automatically identify sensitive information leakage sites in real-world applications. Compared to existing tools, our tool can provide a list of information leakage sites as well as how many bits they can leak if the attacker can observe one site, two sites or multiple sites. We defined the amount of leaked information as how much uncertainty about the sensitive information the attacker could reduce after observing the side-channel. It is interesting to mention that the definition here is different than the definition in several static analysis tools. We will explain the reason in the following sections. After that, we run the symbolic execution on the execution trace. We model each side-channel leakage as several constraints. The only symbols in those constraints are sensitive input. Those constraints can unique model the side-channels. In other words, if the application has an different sensitive input, the code will still leak the same information about the input. Those information leakage sites may spread the whole program and their leakages arent dependent. Simply adding them up can only give the 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 from the constraints generated by previous symbolic execution.

We implemented the proposed method as a tool and applied it on a series of open source libraries. First, we found that most leakages reported in the previous research actually leaked very little information. Second, we also discovered several sensitive vulnerabilities in the media libraries.

II. Motivation

A. Information Leakage Quantification

Existing side-channel quantification works either define the amount of leaked information corresponds to the number of possible side-channel observations or the min-entropy. However, the definition has some limitations. Consider the following example,

unsigned char key = input(); //sensitive information, 4 bits information Unsigned char temp = key/2; if (temp == 0xb) //branch 0 else //branch 1

Suppose an attacker can observe which branch the code actually run (e.g., flush reload, controlled attack), the attacker can infer the value of key. For example, if the attacker observe the code executes the branch 0, then he will know the key equals 11*2 = 22. Because the attacker knows the key equals 22. In this case, there is no uncertainty about the key. As the key has 4 bits information in total, we think the amount of information leakage here is 4 bits. On the other hand, if the code executes the branch 2, the only thing that an attacker can know is that the key doesnt equal 22, which leaks very little information in this case. However, if we use the number of possible observations to quantify the amount of leakage, the code will have two different observations (branch 1 and branch 2). Therefore, the calculated information leakage here will be 1 bit.

We use the mutual information (MI) to quantify the information leakages. For a program P with sensitive information K as the input, the attacker may have some observations O during the execution. The information leakage is defined as the mutual information I(O; K) between O and K.

I(O; K) = H(O) - H(O—K)

I(O; K) represent how much uncertainty about K can be reduced if the attacker has the observation O. For a deterministic program, the program will have the same memory access behavior as long as the input is fixed. As the observation of the attacker correlate to the memory access behavior, we can have the following formula.

O = f(K)

The function f is determined by the program P. For our project, we can calculate the function f via symbolic execution.

I(O; K) = H(O) - H(f(K)—K) = H(O) = H(f(K))

So the mutual information between O and S equals to the self information of O.

H(O) = p(Oi)log(p(Oi))

For a determinist program, we can calculate the distribution of O as long as we know the distribution of input K. So we can calculate how much information is leaked.

For examples, given a program P, we have the sensitive input K. The K should be a value in a memory cell or a sequential buffer (e.g., an array). We use ki 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.

t1 = f1(k1, k2, k3 ... kn) t2 = f2(k1, k2, k3 ... kn) t3 = f3(k1, k2, k3 ... kn) ... tm = fm(k1, k2, k3 ... kn)

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.

B. 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

C. Seret-dependent Memory Access

T[64]; // Lookup tables with 64 entries index = key 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

III. Threat Model

We consider an attacker who shares the same hardware resource with the victim. The attacker attempt to retrieve sensitive information via side-channel attack. The attacker has no direct access to the memory or cache, but can probe the memory or cache at each program point. Similar to DATA[], the attacker will face many noisy observations or can only observe a limited of memory or caches in practice. For the project, we assume the attacker can have noise-free observations. This threat model captures most of the cache-based and memory-based side channel attacks.

IV. Design

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

Intermediate Language vs X86

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.

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 values The memory address of sensitive information and its length The value of eflags register 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 Optimizayion

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 local$_1 4hcanbewrittenasF(K).Attheaddresse785, thevaluewillbeupdatedwithF(K)+$
$1.Thenthecodecomparethevaluewith4andusetheresultasaconditionaljump.Basedontheresult, wecanhavethefollowingconstrain$
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