The Pennsylvania State University
The Graduate School

PRECISE AND SCALABLE SIDE-CHANNEL ANALYSIS

A Dissertation in
Information Sciences and Technology
by
Qinkun Bao

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

August 2021

The dissertation of Qinkun Bao was reviewed and approved by the following:

Dinghao Wu
Professor of Information Sciences and Technology
The Pennsylvania State University
Dissertation Advisor
Chair of Committee

Peng Liu
Professor of Information Sciences and Technology
The Pennsylvania State University

Benjamin Hanrahan
Assistant Professor of Information Sciences and Technology
The Pennsylvania State University

Sencun Zhu
Associate Professor of Computer Science and Engineering
The Pennsylvania State University

James Larus
Professor and Dean of the School of Computer and Communication Sciences
École Polytechnique Fédérale de Lausanne (EPFL)
Special Member

Mary Beth Rosson
Director of Graduate Programs and Professor of Information Sciences and Technology
The Pennsylvania State University

Abstract

Side channels are ubiquitous in modern computer systems as sensitive information can leak through many mechanisms such as power consumption, execution time, and even electromagnetic radiation. Among them, address-based side-channel attacks, such as cache-based attacks, memory page attacks, and controlled-channel attacks, are especially problematic as they do not require physical proximity. Hardware countermeasures, which usually require changes to the complex underlying hardware, are hard to adopt in practice. On the contrary, software approaches are generally easy to implement. While some existing tools can detect side-channel leakages, many of these approaches are computationally expensive or imprecise. Besides, many such vulnerabilities leak a negligible amount of sensitive information, and thus developers are often reluctant to address them. Existing tools do not provide sufficient information, such as the amount of information leaked through side channels, to evaluate the severity of a vulnerability.

In this dissertation, we present methods to detect and quantify address-based side-channel vulnerabilities in real-world applications. First, a new method to detect address-based side-channel vulnerabilities for the binary code is proposed. We examine the bottleneck in the symbolic approaches and improve the analysis precision and performance. Second, we propose a new program analysis method to precisely quantify the leaked information in a single-trace attack. We model an attacker's observation of each leakage site as a constraint and run Monte Carlo sampling to estimate the number of leaked bits for each leakage site. Finally, we extend our approach to quantify side-channel leakages from multiple trace attacks. Unlike previous side-channel detection tools, our approach can identify severe side-channel leakages without false positives.

We implement the approaches and apply them to popular cryptography libraries. The evaluation results confirm that our side-channel detection method is much faster than state-of-art tools while identifying all the known leakages reported by previous tools. The experiments also show that our side-channel analysis reports quite precise leakage information that can help developers better triage the reported vulnerabilities. This dissertation research develops fundamental and practical techniques for precise side-channel analysis in software systems. We have also released our research software prototypes. As a result, developers can use our tools to develop more secure systems and the academic and industry communities can further advance side-channel analysis on top of our research.

Contents

Chapter 1

Chapter 2

Chapter 3

List of Figures

List of Tables

Acknowledgments

I am tremendously fortunate to have been advised by Dinghao Wu. I thank Dinghao for inviting me to EPFL on his sabbatical. A large portion of the research presented in the dissertation was finished when I worked in Switzerland. I also thank him for treating me as an independent researcher during my Ph.D. study. I had a very hard time during the pandemic. Without his encouragement and support, I could not have finished the program.

The members of my committee, Peng Liu, Sencun Zhu, Benjamin Hanrahan, and James Larus, provided helpful comments on my dissertation. Prof. Liu and Prof. Zhu greatly improved this dissertation with their comments from a security research angle. Prof. Hanrahan and Prof. Larus helped me improve the grammar of my dissertation proposal and my dissertation. Prof. Larus is a senior researcher and my last English teacher. I learned a lot on the written English from him. I am extremely grateful for his wise counsel. Jim's comments on the early draft are also very close to what I got from reviewers later. However, I did not take his comments seriously at the very beginning.

Pursuing a Ph.D. can be extremely stressful, and summer internships are always the relaxing and productive moments during my study. I would like to thank Kang Li, Zhaofeng Chen, Mingshen Sun, Greg Zaverucha, Kevin Kane, and Brian LaMacchia for their help during my internships at Baidu X-Lab and Microsoft Research. I also thank my friends and collaborators at Penn State University and EPFL.

Finally, I would like to thank my parents for their continuous emotional and financial support. My parents always hope that I can study in an institution that is near to them. However, they still expressed their full support for me when I decided to come to USA as a graduate student.

This research was supported in part by the National Science Foundation (NSF) under grant CNS-1652790, and the Office of Naval Research (ONR) under grants N00014-16-1-2912, N00014-16-1-2265, and N00014-17-1-2894. Any opinions, or conclusions in the dissertation do not reflect the views of funding agencies.

Chapter 1 | Introduction

Software is becoming increasingly complex nowadays. Half a century ago, the Appollo 11's flight system software [2] was done with less than one hundred thousand lines of code. Today, a mobile instant messaging (IM) application such as Telegram [3] contains more than two million lines of code. It is estimated that engineers built Google's infrastructure with 2 billion lines of code [4]. The complexity in today's software systems makes it hard for developers to understand the codebase and find defects manually. So an automated bug detection tool is often needed. In this dissertation, we present techniques to analyze side-channel vulnerabilities in real-world applications. The side-channel vulnerability is a type of information leakage bugs in software systems. Unlike many ordinary bugs, attackers exploit side-channel vulnerabilities from the implementation of the software rather than weaknesses in the implementation itself. So it is sometimes harder for developers to identify side-channel vulnerabilities.

1.1 Side-channels

A side-channel attack is any attack that allows adversaries to infer sensitive information based on "side-channel" information. In 1996, Kocher published the first side-channel attack in the literature [5]. He demonstrated that an attacker could break cryptosystems such as RSA or Diffie-Hellman by measuring the execution time of a simple modular exponentiation operation. The simple modular exponentiation calculates the remainder of an exponent divided by an integer. The simple modular exponentiation operation executes faster when the exponent bit is 0 and slower when the exponent bit is 1. As a result, an attacker can infer the whole secret key by measuring the execution time of the operation. Since then, researchers have proposed numerous side-channel attacks which exploit different kinds of "side-channel" information, including execution timing [6–8], power consumption [9], electromagnetic radiation [10, 11], and even sound [12].

A large portion of side-channel attacks is the joint result of the underlying computer architecture and the software stack. One example is cache channel attacks. The CPU cache is a smaller but faster memory to bridge the gap of the access time between the main memory and CPU registers by storing copies of recently accessed data. Because the size of the cache is significantly smaller than the size of the main memory, the CPU may only find part of the data in the cache. Accessing data from the cache is faster than accessing data from the main memory. Such differences can be reflected in the variance in the program execution time. Hence we have cache channel attacks.

Various countermeasures have been proposed to defend against side-channel attacks. Existing countermeasures can be divided into two categories, hardware-based methods and software-based methods. Hardware-level solutions, such as reducing shared resources, oblivious RAM, and transnational memory [13–16], need to adopt new hardware features and change today's modern complex computer systems, which are hard to adopt in reality. Therefore, a promising and universal direction is software countermeasures, detecting and eliminating side-channel vulnerabilities in existing software codebase. For example, when we examine recent side-channel attacks, many of them exploit two kinds of code patterns: *secret-dependent control-flow transfers* and *secret-dependent data accesses*.

```
1 int secret[32];
2 ...
3 while(i>0){
4     r = (r * r) % n;
5     if(secret[--i] == 1)
6         r = (r * x) % n;
7 }
```

Figure 1.1: Secret-dependent control-flow transfer

Secret-dependent control-flow transfers happen when the value of secret can affect which branch the code executes. Figure 1.1 shows an example. Depending on the value of `secret`, the code may or may not execute line 6. An attacker can retrieve some information by observing the execution control-flow.

Secret-dependent data accesses happen when the value of secret data affects which memory

```
static char FSb[256] = {...}
...
uint32_t a = *RK++ ^ \
(FSb[(secret)) ^
(FSb[(secret >> 8)] << 8 ) ^
(FSb[(secret >>16)] << 16 ) ^
(FSb[(secret >>24)] << 24 );
...
```

Figure 1.2: Secret-dependent data access

location the code reads or writes. Figure 1.2 shows an example. FSb is an array. Depending on the value of secret, the code may read or write different items in the array.

The key intuition is that the above side-channel attacks happen when a program accesses different memory addresses with different sensitive inputs. As shown in Figure 1.1 and Figure 1.2, if a program shows different patterns in control transfers or data accesses when the program processes different sensitive inputs, the program could have side-channel vulnerabilities. Various techniques can be used to exploit side-channel leakage sites at different granularities. For example, cache channels can observe cache accesses at the level of cache sets [17], cache lines [6] or other granularities [18]. Other types of side-channel attacks such as controlled-channel attacks [19], can observe a memory access at the level of memory pages. So a natural defense approach is to remove the above code patterns in the software.

However, eliminating side-channel vulnerabilities from real-world software codebase is not easy. First, side-channels are a low-level problem in computer systems. As a non-functional property, it can be difficult for programmers to reason about or test. While recent work [13, 14, 20–23] can detect many side-channel vulnerabilities, it suffers from either expensive computation costs or high false positives and false negatives. Second, most vulnerabilities pose a negligible risk. Although some vulnerabilities result in the key being entirely compromised [6, 24], many others are less severe in reality. Therefore, developers are often reluctant to fixing all of them. We need a proper quantification metric to assess the sensitivity of side-channel vulnerabilities, so a developer can efficiently triage them. Third, many side-channel vulnerable implementations (e.g., using the Chinese Remainder Theorem optimization in RSA and lookup tables in AES) have better performance. This is a dilemma between the performance and the security.

In summary, it is hard to apply software approaches in practice due to the following reasons.

1. It is hard to identify side-channel vulnerabilities in real-world software. In general, many side-channel vulnerabilities are low-level problems. Control-flows or memory accesses may not be side-channel vulnerabilities if they are independent of the secret inputs. Recent work can identify side-channel leakages. But these tools can only work on small programs or need to apply domain knowledge of the target program to simplify the analysis.

2. Side-channel vulnerabilities are ubiquitous. While some vulnerabilities can result in the key being entirely compromised, many other vulnerabilities are hard to exploit. Moreover, some vulnerable implementations perform better. Recent work can identify thousands of leaked points, but most of them are not patched by developers because these leakages are either false positives or leaked very little information.

In this dissertation, we try to address the above problem from the following two aspects.

1. **Detection.** We propose a fast and precise side-channel detection method. Unlike previous trace-comparison methods, our method has fewer false positives and false negatives. Compared with methods based on symbolic execution and abstract interpretation, our methods can be applied to real-world cryptography libraries directly. Symbolic execution provides fine-grained information, but it is expensive to compute. Therefore, prior symbolic execution work [13, 21, 22] either analyzes only small programs or applies domain knowledge [13] to simplify the analysis. We examine the bottleneck of the trace-oriented symbolic execution and optimize it to work on real-world applications.

2. **Quantification.** We propose methods to quantify precisely the amount of information leakage that an attacker can retrieve during side-channel attacks. For the first project, we define the amount of leaked information as the cardinality of possible secrets based on an attacker's observation. For the second project, we we quantify the amount of leaked information based on channel capacity. Before an attack, an adversary has a large but finite input space. Whenever the adversary observes a leakage site, they can eliminate some impossible inputs and reduce the input space's size. In an extreme case, if the input space's size reduces to one, an adversary has determined the input, which means all secret information (e.g., the entire secret key) is leaked. By counting the number of inputs [25], we can quantify the information leakage precisely.

1.2 Contribution

This dissertation makes the following contributions. First, we propose a novel method that can detect side-channel vulnerabilities in real-world applications fast and precisely. Second, we propose methods that can quantify precisely each side-channel leakages. In addition, we study two kinds of threat models: single-trace attacks, and multiple-trace attacks. In summary, we make the following contributions:

On detecting side-channel leakages from real-world software, we make the following contributions:

- We design a symbolic analysis method that can detect secret-dependent control-flow transfers and secret-dependent data accesses. The key idea is to only symbolically execute machine instructions that are relevant to vulnerabilities while executing other instructions natively. We implement the above method in a tool called Lorem. The evaluation results shows that Lorem is 3 to 100 times faster than existing tools while identifying all vulnerabilities when we run it on the same library. We do not need to apply domain knowledge to analyze part of the original program, which simplify the analysis. In addition, Lorem finds new leakages which are not identified by previous tools. We confirmed these findings with developers.

On quantifying fine-grained leaked information in a single trace attack, we make the following contributions:

- We propose a method that can quantify fine-grained leaked information from side-channel vulnerabilities that result from actual attack scenarios in a single trace attack. Our approach differs from previous ones in that we precisely calculate the amount of leakage. We model the information quantification problem as a counting problem and use a Monte Carlo sampling method to estimate the information leakage.

- We implement the method into a tool called Abacus and apply it to several pieces of real software. Abacus[1] successfully identifies previous unknown and known side-channel vulnerabilities and calculates the corresponding information leakage. Our results are useful in practice. The leakage estimates and the corresponding inputs that triggers the vulnerability can help developers triage and fix the vulnerabilities.

[1] Abacus is open source under the MIT license. `https://github.com/s3team/Abacus`

On quantifying side-channel leakages through a multiple-trace attack, we make the following contributions:

- We propose a novel method that can detect and analyze the effect of each side-channel leakage site automatically. Our analysis combines the information from both the source code and the run-time execution, which makes the method more effective in finding the leakages. By applying a chi-squared test, the method can find independent leakages. Attackers can combine multiple leakage sites to retrieve more information as well.

- We analyze the amount of information leakage from side-channel vulnerabilities based on the channel capacity. The theory can serve as the foundation of the future work on quantifying the information leakage based on the fuzzing.

- We implement above methods in a tool called Quincunx and evaluate the tool with several benchmarks and real-world applications such as OpenSSL, mbedTLS, Libgcrpyt, TinyDNN, and GTK. The results show that our tool can detect and quantify the side-channel leakages effectively. Moreover, the leakage reports given by the Quincunx can help developers fix the reported vulnerabilities.

1.3 Dissertation Organization

The rest of the dissertation is organized as follows. We present the related work of the dissertation in Chapter 2. Chapter 3 presents a novel method to detect side-channel vulnerabilities in binary executables. Chapter 4 presents a method that can quantify each side-channel leakage in a single trace attack. Chapter 5 extends the approach and quantifies the side-channel leakage in a multiple-trace attack. Chapter 6 and Chapter 7 discuss limitations and future work of the dissertation research.

Chapter 2 | Related Work

This dissertation studies address-based side-channel attacks. We review the related work in this chapter. The related work covers the following aspects: side-channel attacks, detections, mitigation, information quantification, and current binary analysis tools.

2.1 Side-channel Attacks

While there are many kinds of side-channel attacks, we focus on address-based side-channel attacks in this dissertation. Address-based side-channel attacks happen when a program accesses different memory locations as it processes different secrets. In modern computer systems, an attacker may share hardware resources (e.g., Cache, TLB, and DRAM) with the victim [26, 27]. An attacker can probe the shared hardware resource to infer the memory accesses of the victim. While the root cause of the attack is the same, an attacker can exploit side-channel leakage sites in different methods to retrieve the information at different granularities.

Table 2.1 shows some representations of recent side-channel attacks. For a detailed overview of recent side-channel attacks, we refer to the survey paper by Ge et al. [26]. We classified these side-channel attacks into two categories: cache-level channels and page-level channels.

2.1.1 Cache-level Channel

A CPU cache is an essential part of modern computer systems. It is a smaller and faster memory that is designed to reduce the average data access costs to the main memory by storing copies of data from the main memory. When a CPU reads data from a memory location, it first checks the cache. If it finds that the cache has a copy of data from the main memory, a cache hit occurs.

Table 2.1: Here are some representations of the vast side-channel attacks: the shared resources, the granularity of the attacker can observe, is there any published attacks on non-cryptography libraries, and if the type of attack has been used to exploit multiple leakage sites.

Attacks	Shared Resources	Granularity	Non-Crpyto	Multiple
CacheBleed [18]	Cache	Sub Cache Line (<64 B)	✗	✗
CopyCat [28]	Page Tables	Instruction	✗	✗
Prime + Probe [17]	Cache	Cache Set (64-512 B)	✓	✗
Prime + Abort [7]	Cache	Cache Set (64-512 B)	✓	✗
Flush + Reload [29]	Cache	Cache Line (64 B)	✓	✓
Flush + Flush [30]	Cache	Cache Line (64 B)	✓	✗
Controlled-channel Attack [31]	Page Tables	Page (4 KB)	✓	✗
TLBleed [32]	MMUs	TLB Set (4 KB)	✓	✗
Page Cache Attacks [33]	Page Cache	Page (4 KB)	✗	✓

Otherwise, the CPU needs to read from the main memory, and a cache miss happens.

Cache channels [6, 6, 17, 18, 34, 35] rely on the time difference between a cache miss and cache hit. The core-private caches (e.g., L1 and L2) are traditionally set-associate cache. The cache is divided into different cache sets, and each cache set is made up of several cache lines. Typically, the smallest storage unit in a CPU cache is a cache line. Cache channel attacks can observe cache accesses at the level of cache sets, cache lines or other granularities. We introduce two common attack strategies, namely Prime+Probe [17] and Flush+Reload [29].

Prime+Probe. Prime+Probe targets a single cache set. An attacker primes the cache set with its own data and waits until the victim executes the program. Then the attacker probes the cache by reading previously loaded data. If the victim accesses the cache set and evicts the data loaded by the attacker, the attacker will experience a slow measurement. Under the circumstance, the attacker can know that the victim must have touched some data that maps to the same cache set.

Flush+Reload. Flush+Reload allows an attacker to observe memory accesses at the granularity of a cache line instead of a cache set. While Flush+Reload has a fine-grained granularity compared with Prime+Probe, it requires the attacker and victim to share some memory. It also has two stages. During the "flush" stage, the attacker flushes the "monitored memory" from the cache and waits for the victim to access the memory, which may be a load of the sensitive information to the cache line. In the next phase, the attacker reloads the "monitored memory". The attacker's process reads all data in the 'monitored memory". If the time is relatively fast, it indicates that the data has been accessed by the victim program and the cache has loaded the data. On X86-64 machines, the two steps can be combined by measuring the time differences of the clflush instruction.

Based on the above attacks, some work [36] suggests ensuring secret-dependent memory accesses within the size of one cache line should be secure enough. However, recent studies show that having different memory accesses within a cache line is also dangerous. One example is the CacheBleed attack [18]. It is a cache channel attack that can exploit cache-bank collisions.

2.1.2 Page-level Channel

Page channel attacks allow an attacker to observe the memory access at the granularity of memory pages. One example is the controlled-channel attack [31]. It is a primary threat to the trusted execution environments (TEEs). A privileged adversary (e.g., kernel) can infer sensitive information by manipulating the enclave page accesses. A malicious OS can proactively revoke a virtual enclave memory page from the virtual memory. If the victim enclave program accesses the memory in the page, then the OS can observe the page faults and know pages that are accessed. While the granularity of controlled-channel attacks is more coarse-grained compared with cache channel attacks, controlled-channel attacks allow the attacker to have a noise-free observation. Other examples such as TLB side-channel attacks [32] and DDRAM attacks [33] share the same page-level granularity.

Based on the above discussion, we consider three types of granularities in the dissertation: byte (1 byte), cache line size (64 bytes), page size (4 KBs). To the best of our knowledge, these three types of granularities can cover most of the side-channel attacks in the literature.

2.1.3 Attack Models

Previous work [23] classifies side-channel attacks into three types based on the type of information that an attacker can learn during an attack.

Timing-based Attacks

In timing-based attacks, an adversary measures and compares the execution time of victim programs. It could be the overall execution time or the execution time of some interesting components (e.g., a function). The execution time is correlated to some events (e.g., cache hits and misses, page faults, the number of instructions, optimizations). Timing-based attacks are perilous because these attacks are able to be carried out remotely over the networks [37].

Examples of these attacks include remote timing attacks [37], Bernstein's attack on AES [38], and Kocher's original attack [5] from CRYPTO 96.

Access-based Attacks

In access-based attacks, an adversary can learn the memory addresses that are accessed during the execution after the termination of a victim program. For example, a cache-based channel attack can know which memory access incurs a cache hit and which memory access incurs a cache miss to infer addresses that the victim program accesses. Since the attacker probes the victim program after the execution, the attacker does not know the order of memory accesses. Access-based attacks often happen in the cloud computing environment when the attacker and victim share the same hardware (e.g., LLC cache). Examples of these attacks include cache template attacks [34], Prime+Probe attacks [17], and Flush+Reload attacks [29].

Trace-based Attacks

In trace-based attacks, an adversary can interrupt and probe the victim program at any point. The attacker can learn the order of memory accesses. Suppose the victim program has two functions a and b. In an access-based attack, the attacker can only learn both a and b are executed. In a trace-based attack, the attacker can know the execution sequence of two functions (e.g., `aabbb`). Controlled-channel attacks [28, 31] belong to this category where a malicious operating system interrupts the victim application and probes the program at any time. In general, the trace-based attack has the strongest assumption about the threat model, but it can also retrieve more information compared to the other two types of attacks.

2.1.4 Target Applications

There is a large number of studies [6, 17, 18, 34, 35] on side-channel attacks. Most early works targeted retrieving encryption keys from cryptography libraries. In recent years, researchers start to focus on more general and diverse applications. In summary, the target applications of current side-channel attacks can be classified into three categories.

Table 2.2: Some common vulnerable components in cryptography implementations and corresponding software mitigation methods.

Implementation	Common Vulnerable Point	Mitigation
AES	T-table and S-table lookups	AES instructions (AES-NI) Scalar bit-slicing Preloading tables
DES	Table lookups	Scalar bit-slicing (No implmentation)
RSA	CRT optimization Modular exponentiation	Verifying the consistency with the public key Montgomery modular multiplication Base blinding and exponent blinding
(EC)DSA	Scalar multiplication Modular inversion Modular reduction	Constant time scalar implementation Blinding

Cryptography Libraries

The first side-channel attack on cryptography libraries can be traced back to Kocher's timing attacks [5] in 1996. It exploits a simple modular exponentiation algorithm function in many cryptography (Diffie-Hellman, RSA) implementations. The function does not run in a fixed time. By measuring the timing differences between the modular exponentiation algorithm computes exponent bits (0 or 1), Kocher's attack can recover the whole secret key bit by bit. Since then, researchers start to break all kinds of cipher implementation through side-channel attacks. Table 2.2 summarizes some common vulnerable points in cryptography library implementations. Note that cryptography libraries usually have several implementations for some ciphers. For example, AES has more than five different implementations in OpenSSL. Depending on the build configuration and the return value of CPUID, the code may execute any implementation of them. The reference implementation usually adopts T-Tables and S-Tables to speed up the computation. However, such implementations are vulnerable to side-channels and can be exploited to retrieve the whole key [39]. In addition to the reference implementation, these libraries also have side-channel mitigated versions. However, recent studies show these versions (e.g., AES-NI) also have side-channel vulnerabilities caused by compiler optimizations and human mistakes [14]. The situation is even worse for asymmetric ciphers (e.g., Diffie-Hellman, RSA, ECDSA). These ciphers have a lot of big number calculations. It is hard to implement these calculations to be completely side-channel resistant. Recent work [18, 29, 40] demonstrated several end-to-end attacks to recover the private key of these ciphers.

Machine Learning Applications

Recent side-channel attacks on machine learning applications mainly focus on recovering two types of sensitive information: the input of the model and the architecture of deep learning models. Hua et al. [41] were the first in the literature to reverse engineer the architecture of convolutional neural networks (CNNs) through the memory access patterns. Wei et al. [42] perform side-channel attacks on an FPG-based convolution neural network by using the power consumption to recover inputs to the networks. For an image classification task trained by the MNIST dataset, their attacks can achieve up to 89% accuracy. CSI NN [43] reverse engineered the multi-layer perception and convolution neural network based on the electromagnetic signals (EMs). Cache Telepathy [44] introduces the cache-based side-channel attack to extract DNNs' architectures. We find that all existing approaches need a lot of domain knowledge and expertise to launch side-channel attacks.

Media and Graphic Libraries

Recent work also studies side-channel attacks in media and graphic libraries. Gruss et at. [34] show an attack that can infer keystrokes by exploiting secret-dependent memory accesses in the GDK library. In the controlled-channel attack [31], Xu et al. demonstrate an attack to recover the outline of an image by exploiting vulnerabilities in the inverse DCT function in libjpeg. Wang et al. [45] present a cache attack on graphic libraries. Their approaches use a machine learning approach to pick cache lines that can be exploited to leak the most information.

Others

Some other side-channel targets also include spelling checking tools and font libraries [31]. The attack on font libraries is essentially the same as the attack on the graphic library GDK. By exploiting the differences of memory accesses when the program renders different characters, an attacker can recover parts of the original sensitive information. Other side-channel attacks target operating system kernels. One example is using the side-channel attack to break the kernel address space randomization (KASLR). Many exploits rely on the knowledge of the memory location of a certain kernel function. Since Linux 4.12, the kernel loads core kernel image and device drivers at random positions during the boot time. Jang et al. [46] rely on side-channels to reveal the mapping status of each page. After that, they detect kernel modules with unique size signatures to break KASLR. The second example relies on the kernel space side-channel vulnerabilities [47] to infer TCP sequence numbers, which can be used by a third party to hijack connections between a client and a server [48].

2.2 Defenses and Mitigation

Both hardware [16, 49–53] and software [15, 54–57] side-channels mitigation techniques have been proposed recently. Hardware countermeasures, including partitioning hardware resources [49], randomizing cache accesses [50, 52], and designing new architecture [58], require changes to complex processors and are difficult to adopt. On the contrary, software approaches are usually easy to implement. Coppens et al. [54] develop a compiler extension to eliminate key-dependent control-flow transfers. Crane et al. [56] mitigate side-channels by randomizing software. As for crypto libraries, the basic idea is to eliminate key-dependent control-flow transfers and data accesses. Common approaches include bit-slicing [59, 60] and unifying control-flows [54].

2.2.1 Software Countermeasures

Software Vulnerability Identification

For many software countermeasures, the first step to identify side-channel leakages in software manually. It can be done manually or by some program analysis tools. There are several existing works on side-channel leakage detections. As shown in Table 2.3, existing methods can be classified into two categories, tools based on the trace comparison and tools based on the abstraction.

Trace comparison. If a program has two different execution traces under two different sensitive inputs, then attackers can distinguish the two inputs by observing the execution traces. Therefore, we can conclude the program is vulnerable to side-channel attacks. The method is similar to the UNIX tool diff. The approach does not need to model or abstract the program's semantics. In general, the approach is quite fast. On the negative side, comparing the trace can find the leakage, but it cannot find the complex relationship between the leakage site and the original secret input. For example, if two leakage sites leak the same information, these approaches cannot find out the two leakages are the same.

Abstraction. The second category of these tools belongs to the field of abstractions. Tools based on these approaches use abstraction to generate approximations to model the semantics of computer programs. After that, by analyzing the model, these tools can identify side-channel leakages. Compared with the trace comparison approach, the method is more precise.

These methods usually follow the steps below. Suppose a program P has k as the input. These approaches first retrieve the relationship between the k and temporary values during the execution. After that, they should have a cache or memory access model and model the side-channel leakage as a math constraint. By using SMT solvers to find different secret inputs that lead accesses to different locations in the memory or the cache, these approaches can determine if the program is vulnerable to side-channel attacks. Unlike to the trace comparison, the method is quite precise. Moreover, it can model multiple leakages precisely as the conjunction of these formulas. However, these approaches have the following limitations.

- Many tools can only analyze the source code or the intermediate representation level. However, side-channel are low-level problems in general. Most existing tools are research prototypes. For some projects, the tools can only handle the benchmarks in their papers properly.

- These tools heavily rely on SMT solvers. Despite SMT solvers having made incredible progress in performance, SMT solving is still the main bottleneck of these tools. SMT solvers are not good at non-linear arithmetic. However, non-linear arithmetics are quite common in cryptography algorithms. Recent studies show existing solvers still have bugs when dealing with floating number calculations. These calculations are also common in media libraries and machine learning applications.

Table 2.3 covers some representations of existing side-channel leakage detection tools. CacheAudit [23] uses abstract domains to compute an over-approximation of cache-based side-channel information leakage upper bound. However, it is difficult to judge the sensitive level of the side-channel leakage based on the leakage provided by CacheAudit. Besides, CacheAudit is not scalable enough to analyze ciphers like RSA implementations. CacheS [22] improves the approach from CacheAudit with a new abstract model that only tracks secret-related code. Like CacheAudit, CacheS cannot measure the amount of leaked information in each side-channel vulnerability. CaSym [21] introduces a static cache-aware symbolic reasoning technique to cover multiple paths in a target program. CaSym can work on two threat models: trace-based attacks and access-based attacks. Similarly, their approaches cannot evaluate the sensitive level for each side-channel vulnerability, and it only works on small code snippets.

The dynamic approach, usually consisting of taint analysis and symbolic execution, can perform a very precise analysis. CacheD [13] takes a concrete execution trace and runs symbolic execution on the trace to get a formula for each memory address. CacheD is quite precise in avoiding false positives. However, CacheD is not able to detect secret-dependent control-flows. We adopted a similar approach to model the secret-dependent data accesses in the dissertation. Abacus differs from CacheD in that we do not use traditional taint tracking or domain knowledge to cut the execution trace when identifying secret-dependent data access vulnerabilities. Abacus works on machine instructions directly instead of intermediate representations. Moreover, Abacus finds secret-dependent control-flows at the same time and gives a precise quantification of the leakage. DATA [14] detects address-based side-channel vulnerabilities by comparing and aligning different execution traces under various test inputs. After collecting execution traces, DATA aligns them and finds the differences. It uses statistical hypothesis testing to find true leakages. However, both imperfect trace alignment and the statistical testing that DATA uses can produce false positives. MicroWalk [20] adopts a similar approach as DATA by comparing all accessed addresses. It uses mutual information (MI) between sensitive input and execution state to quantify the information.

Software Patch

Once we have identified side-channel leakages, there are two common approaches to mitigate side-channel attacks. The first approach is to adopt unified behavior code when computing the secret data. For example, a control flow leakage happens when different secret inputs lead to different paths. Software developers can mitigate the leakage by executing both branches and select the result with bit-wise operations [54]. However, for exiting code base, it is hard to migrate all the code into a side-channel free implementation. The second approach is to introduce randomized behaviors during the execution to obfuscate the information that an attacker can retrieve. One example is cryptographic blinding [64]. Developers can use a random value to blend the secret key but still generate the same encrypted data with a carefully designed algorithm. Binding values alters the program execution to some unpredictable states and introduce the randomization to the execution. However, the binding value can also be retrieved by an attacker through other methods, including side-channel attacks. An attacker can launch an attack to retrieve the binding value to recover the secret input.

Table 2.3: Many leakage detection tools are only evaluated on cryptography libraries. All of them cannot reason about the total effect of multiple leakages.

Tool	S/D	Method	AVX/SIMD	Multiple	Evaluation	Available
CacheAudit [23]	Static	Abstraction	✗	✗	Crypto Algorithm	✓
CacheD [13]	Dynamic	Symbolic Execution + Taint Analysis	✗	✗	Libgcrypt, OpenSSL	✗
CachSym [21]	Static	Symbolic Execution	✗	✗	Crypto Snippet	✗
CacheS [22]	Static	Abstract Interpretation	✗	✗	Libgcrypt, OpenSSL, mbedTLS	✗
DATA [14]	Dynamic	Trace Alignment + Comparison	✓	✗	OpenSSL, PyCrypto	✓
ctgrind [61]	Dynamic	Taint Analysis	✓	✗	Crypto Library	✓
Satcco [62]	Dynamic	Trace Comparison	✓	✗	OpenSSL, GnuTLS, mbedTLS, WolfTLS, LibreSSL	✗
ANABLEPS [63]	Dynamic	Trace Comparison	✓	✗	gsl, Hunspell, PNG, Freetype, QRcodegen, Genometools	✗
MicroWalk [20]	Dynamic	Trace Comparison	✓	✗	Intel IPP, Microsoft CNG	✓

New Implementation

Recent work [65–68] proposes to design and implement clean-slate side-channel free libraries from scratch. Researchers [69, 70] design and implement new constant-time cryptographic algorithms (e.g., ChaCha20) to defend against timing side-channel attacks. Unfortunately, writing side-channel resistant cryptography code and validating the security property using pen-and-paper arguments can be tedious and error-prone. Computer-aided cryptography [71] is proposed to address the challenge, which encompasses formal verification methods to design, analyze and implement the cryptography libraries. Formal verification on side-channel leakages starts with defining a leakage model, which abstracts what an attacker can observe during an attack. After defining a leakage model, the implementation is secure (under the leakage model) to side-channel attacks if and only if it can be proved that observations are independent of the input secrets. The property is an example of *noninterference* or *secure information flow*. Eldib et al. [72] propose using a SMT solver based method to formally verifying masked implementations. Bayrak et al. [73] verify power-based side-channel countermeasures by transferring the verification problem into a set of Boolean satisfiability problems. However, achieving noninterferecne for real-world software is often impractical and researchers also seek to compromised solutions such as quantitative information flow (see Section 2.3).

2.2.2 Hardware Countermeasures

Resource Partitioning

Side-channel attacks usually require attackers and victims to share some hardware resources (e.g., CPU cache, MMU). A way to defend these attacks is to partition the resources and ensure the attacker and the victim never share the same hardware resource [49, 74, 75]. For example, in a cloud computing environment, several virtual machines (VMs) can be hosted on the same physical machine and these VMs share the same last-level cache (LLC). Researchers proposed different methods to partition the LLC such as cache coloring [74], using Intel Cache Allocation Technology [75]. Recent work also uses machine checkable methods to verify side-channel resistance in hardware [16, 51, 53].

Randomization

This type of defense randomizes the behaviors of hardware resources to make it harder for attackers to infer secrets based on side-channel information. For example, Random Permutation Cache [76] uses a dynamically randomized memory to cache mapping table for each different process. Vattikonda et al. [77] add noises to the result of the system timer to mitigate timing side-channel attacks.

2.2.3 Discussion on Existing Countermeasures

Despite a lot of hardware countermeasures that have been proposed in academia, few of them have been adopted in real-world production systems. In general, hardware countermeasures usually require new hardware designs and are harder to adopt in practice. In addition, some hardware methods (e.g., Oblivious RAM [78]) may introduce expensive computation overhead. Therefore, many practical solutions are still software countermeasures. However, software methods are not very general, and it is hard to identify many side-channel leakages in large software manually. In this dissertation, we propose side-channel analysis methods to address the problem.

2.3 Information Quantification

Proposed by Denning [79] and Gray [80], Quantitative Information Flow (QIF) aims to provide an estimation of the amount of leaked information from the sensitive information given the public output. McCamant and Ernst [81] quantify the information leakage as the network flow capacity. They model possible information leakage as a network of limited-capacity channel. After that, they use the maximum rate of the channel to represent the amount of secret information that the execution can reveal. Backes et al. [82] propose an automated method for QIF by computing an equivalence relation on the set of input keys. However, the evaluation shows the approach only works on small programs. Besides, their approach cannot handle programs with bitwise operations. However, such bitwise operations are quite common in cryptographic libraries. Phan et al. [83] propose symbolic QIF. The goal of their work is to ensure a program is *noninterference*. They adopt an over-approximation method to estimate the total information leakage, and their method does not work for secret-dependent memory access side-channels. Pasareanu et al. [84] combine symbolic analysis and Max-SMT solving to synthesize the concrete public input that can lead to the worst-case leakage. They assume the target program has multiple different input secrets and calculate the average leakage for one-fixed public input. CHALICE [85] quantifies the leaked information for a given cache behavior. It symbolically reasons about cache behaviors and estimates the amount of leaked information based on cache miss/hit. Their approach only scales to small programs, which limits its usage in real-world applications. On the contrary, Abacus [1] assesses the sensitive level of side channels with different granularities. It can also analyze side channels in real-world cryptographic libraries.

Model counting refers to the problem of computing the number of satisfying solutions for a propositional formula (#SAT). There are two approaches to solving the problem, exact model counting, and approximate model counting. Exact model counting for propositional formulas (#SAT) should determine the number of solutions. Intuitively, such a model counter should traverse the whole search space to find all the satisfying solutions. Consequently, the exact model counter is very slow. These model counters have difficulty in scaling up to real-world programs. On the other hand, the approximate model counter is much faster compared with the exact model counter. The related work focuses on approximate model counting since it is close to our approach. Wei and Selman [86] introduce ApproxCount, a local search-based method using Markov Chain Monte Carlo (MCMC). ApproxCount has better scalability than exact model counters. Other approximate model counter includes SampleCount [87], Mbound [88], and MiniCount [89]. Unlike ApproxCount, these model counters can give lower or upper bounds with guarantees. Despite the rapid development of model counters for SAT and some research [90,91], existing tools cannot be directly applied to side-channel leakage quantification. ApproxFlow [92] uses ApproxMC [93] for information flow quantification, but it has only been tested with small programs.

2.4 Trace-based Binary Analysis

Since our work relies on binary analysis, we briefly mention the related work here. Trace-based binary analysis can be seen as a type of dynamic analysis. It analyzes one concrete program execution path each time rather than trying to cover as many paths as possible. Compared to static analysis, trace-based analysis is usually more precise as it can incorporate runtime information. It usually has two steps. The first step is to collect trace information. Existing approaches often rely on binary instrumentation frameworks or emulators (e.g., Intel Pin [94], Valgrind [95], and QEMU [96]). The second step to analyze the trace using techniques such as symbolic execution and taint analysis. Depending on when the analysis is launched, there are two kinds of approaches, online analysis and offline analysis. As the name suggests, online analysis is performed during the program execution. The target program's data can be modified directly in memory, which is useful to tasks such as fuzzing because the analysis can force the program to execute a specific branch. Offline analysis or post-analysis happens after the program finishes executions. Trace-based binary analysis has been adopted in several tasks such as algorithm detection, vulnerability detection, and malware analysis. We only mention a few of the most related works here. Xu et al. [97] introduce bitwise symbolic execution, which detects crypto algorithms in obfuscated binaries. Ming et al. [98] use the method called sliced segment equivalence checking to do binary diffing. Jia et al. [99] perform an offline analysis on execution traces to identify heap overflows.

There are several well-known binary analysis frameworks that can help users to perform analysis on binary executables. BitBlaze [100] is a binary analysis platform that provides both static and dynamic analysis. It uses the tracecap plugin to collect execution traces and convert the trace into Vine IR. Binary Analysis Platform (BAP) [101] is another binary analysis platform. Similar to BitBlaze, it uses a pintrace tool to collect execution traces and convert the trace into BAP IRs. Angr [102], the most popular binary analysis platform recently, also supports trace-based analysis. It converts X86 instructions into VEX IRs and interprets each IR in Python.

All above binary analysis frameworks adopt IR layers, which makes it easy to implement the analysis and support multiple architectures. However, it can have very low efficiency and long traces. All IRs must be interpreted no matter symbolic variables are involved or not. As side-channels are low-level problems in general, the IR will lose some information. For example, IR-level branches are not necessarily a super-set or a sub-set of machine-code level branches. Some IR-level branches can be compiled into non-branch instructions like conditional moves. The above limitations motivate us to implement a new trace-based binary analysis framework.

Chapter 3 | Fast and Precise Side-channel Vulnerability Detection

3.1 Problem

Side channels are ubiquitous in modern computer systems as sensitive information can leak through many mechanisms such as power consumption, execution time, electromagnetic radiation, and even sound [9–12, 103]. Among them, software side-channel attacks, such as cache attacks, memory page attacks, and controlled-channel attacks, are especially problematic as they do not require physical proximity [17, 19, 34, 104–106]. These attacks arise from shared micro-architectural components in computer architectures. By observing inadvertent interactions between two programs, attackers can infer program execution flows that manipulate secrets and guess secrets such as encryption keys [13, 35, 107, 108].

The cause of most side-channel attacks is the combination of the hardware design and the vulnerable software. Existing countermeasures can be classified into two categories: fixing the software and adopting new hardware. Based on the discussion in Chapter 1 and Chapter 2, fixing side-channel leakage sites in software is usually easier than adopting new hardware to defend against side-channel attacks. In order to eliminate these leakage sites, developers need to find potential leakage sites in a large codebase. However, the manual process is tedious and error-prone. Recent work [13] also shows fixing old vulnerabilities can sometimes introduce new leakages.

Recent work has made good progress in detecting side-channel leakages automatically. Some tools (e.g., CacheD [13], CaSym [21]) have successfully identified unknown leakage sites in software products automatically. However, these tools have limitations.

First, although some tools [21] can find side-channel leakages in real-world software products, they can only analyze one code fragment at a time due to the expensive performance overhead. As a result, users of these tools sometimes need to cut off manually irrelevant code based on their experience. The trimming process is tedious and needs domain knowledge of the target software (e.g., which part of the code is more likely to have side-channel leakage sites). More importantly, if serious leakage sites are on the trimmed code, these tools will miss leakage sites as well.

Second, many tools perform the analysis at the level of intermediate representations (IR) instead of the machine code. It is a design decision to facilitate the implementations. While analyzing the IRs can simplify the implementation, it is not suitable for side-channel analysis. Side-channel leakages are a low-level issue and only the low-level analysis can produce the most accurate results. For example, the IR-level branches are not necessarily a superset or a subset of machine-code level branches. Besides, compiler optimizations can eliminate the IR-level branches with conditional moves, and the converse could also happen. Moreover, translating the machine code into IRs causes a significant overhead for the trace analysis, as we will discuss in the dissertation.

To overcome the above problems, we propose a fast and precise method to detect the side-channel leakages in real-world software products. We examine the bottleneck of current symbolic execution approaches and optimize it to work on real-world cryptography systems. Unlike previous methods, our tool analyzes the machine instructions, which can give more precision than traditional IR-based approaches. We evaluate our tool on popular cryptography libraries including OpenSSL, mbedTLS, Libgcrypt, and Monocypher. The evaluation results show that our tool can identify previous side-channel leakages as well as find new leakages. Compared with recent tools [13, 14, 20], our tool is 3–100x faster while finding all the leakages when we evaluate on the same benchmarks. In addition, our tool finds new side-channel vulnerabilities missed by the previous work.

3.2 Background and Threat Model

3.2.1 Micro-architectures

Many side-channel attacks are architecture-dependent. To facilitate the illustration, we present some necessary backgrounds in this section.

During the evolution of the computer architecture, researchers found that CPUs process data much faster than the main memory (DRAM) can supply. As a result, CPUs adopted a hierarchy memory architecture to bridge the performance gap between the CPU and the main memory. By caching extra copies of recently accessed data and code in a smaller but faster memory, the CPU can get its most recently accessed code and data in a shorter time.

Figure 3.1 shows the overview of one example of the hierarchy memory architecture. For a CPU with multiple cores, each core has two private level-1 (L1) cache: an instruction cache (iCache), and a data cache (dCache). Because instructions and data have different access patterns, separating iCache and dCache makes it possible for CPUs to fetch the code and data simultaneously and improve the performance. Followed by the level-1 cache, each core also has the unified, private level-2 cache for both the code and the data. The level-3 cache is also called the last level cache (LLC). The LLC is shared among all CPU cores within a CPU package. The main memory is at the bottom of the memory hierarchy.

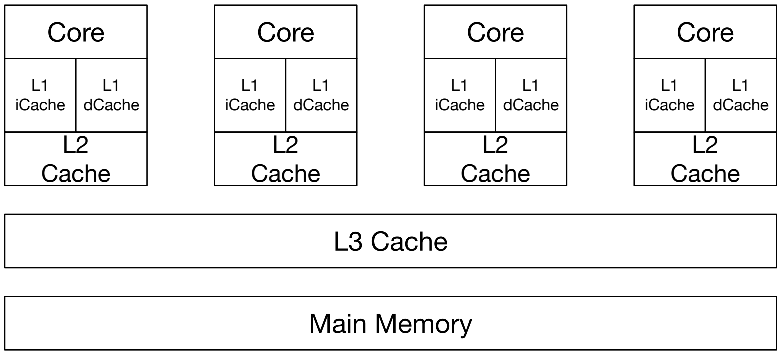


Figure 3.1: Computer memory hierarchy. The hierarchy is designed to minimise the access time with a relatively low cost. It is the root cause of many side-channel attacks.

The current computer memory hierarchy opens the way for side-channel attacks from two aspects. First, the architecture relies on the system software to manage the memory, which becomes a problem in a threat model when the operating system is not untrusted (e.g., Intel SGX). Second, the size of the cache is smaller than the main memory. It is possible that different units in the main memory share the same address in the cache.

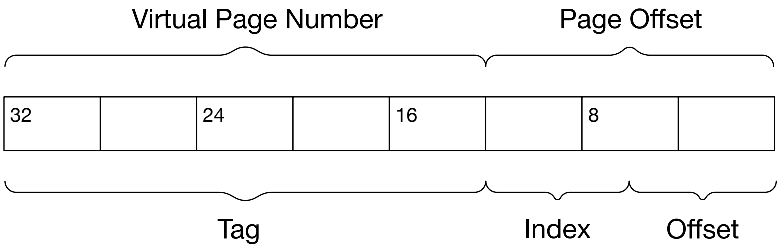


Figure 3.2: Memory addressing

In modern CPUs, L1 and L2 caches are traditional N-way set associate caches. That is, the cache is divided into several cache sets, and each set contains several cache lines. The cache line is the atomic unit. As shown in Figure 3.2, we can determine whether the data is in the cache with the following steps. Given an address, the CPU uses the Index field of the address to locate the cache set that the address should reside in. After that, it uses the Tag field to match every cache line inside the set line. If the CPU can locate a cache line with the same tag and the valid bit is set, then a cache hit occurs. The CPU uses a similar process to manage the main memory. The main memory is divided into many units called pages. As shown in Figure 3.2, the translation process keeps the bottom bits same while using the top bits to map the Physical Page Numbers (PPN) to Virtual Page Numbers (VPN).

3.2.2 Threat Model

We assume that an attacker shares the same hardware platform with the victim. The attacker attempts to retrieve sensitive information through address-based side-channel attacks. The attacker has no direct access to the victim's data, but he can probe the memory or cache at each program point. Here are a few examples.

1. A host machine has several virtual machines (VMs). The victim runs the application inside one VM. An attacker can control a different VM and probe the process running on other VMs.

2. In a shielding system [109, 110], a malicious operating system can extract sensitive information from protected user-level applications.

3. A ring-3 application can infer some sensitive information inside the kernel.

In reality, attackers may face many possible obstacles, such as the noisy observations of the memory or cache. However, in this research, we assume the attacker has noise-free observations as in previous work [13, 21, 23]. The threat model captures most address-based side-channel attacks and applies to the three attack: 1) time-based attacks, 2) access-based attacks, 3) trace-based attacks. We only consider deterministic programs and assume an attacker has access to the source code or binary executable of the target program.

3.3 Limitations of Previous Side-channel Leakage Detection Tools

In this section, we introduce two limitations of current side-channel leakage detection tools. The first is that some tools have some false positives or false negatives. The second is that current tools are not fast enough to analyze side-channel leakages in real-world software.

3.3.1 Imprecision

In this dissertation, we study two types of side-channel leakage code patterns. The intuition is that the target application shows different control flows or data flows when it processes different sensitive input data. We refer them as *secret-dependent control flow transfers* and *secret-dependent data accesses*. Unlike previous work, our tool works on the machine code, which is more precise than previous tools that analyze the source code or the intermediate representation.

3.3.1.1 Control Flow

If the input-sensitive data can affect the victim program's control flow, then an attacker can infer the sensitive data by observing the control flow during the execution. Figure 3.3(a) shows such an example. The function takes a public value m and a secret k as the inputs. The code at line 2 ensures that the lower 7 bits of the value k do not affect the value b. However, depending on the value of m, the code snippet in Figure 3.3(a) may have a leakage site. If the eighth bit of m is 1, then an attacker can infer the eighth bit of k by observing the branch at line 4 or line 6. The right figure shows the corresponding machine code. It has a conditional jump instruction jne. The sensitive input k can affect the program counter (the rip register), which is the root

```
int example1(uint8_t k, uint32_t m) {
  uint32_t b = (k & m) >> 7; // m = 0
  if (b) {
    ...    // Branch 1
  } else {
    ...    // Branch 2
  }
  ...
}
```

```
push  ebp
mov   ebp, esp
movzx eax, [addr_k]
and   eax, [addr_me]
shr   eax, 0x7
test  eax, eax
jne   branch 2
...
```

(a) A false negative

```
int example2(uint16_t k) {
  int res;
  if (k > 8) {
    res = 0;
  } else {
    res = 1;
  }
  return res;
}
```

```
xor   eax, eax
cmp   [addr_k], 8
setbe al
xor   edx, edx
ret
```

(b) A false positive

Figure 3.3: Secret-dependent control-flow transfers

cause of the side-channel leakage here.

However, a if-else branch does not always indicate there is a potential leakage site. Figure 3.3(b) shows a different situation. While the sensitive input k can affect the if-else branch, the compiler removes the branch with a conditional set instruction. The opposite situation is the same. For example, recent work tries to use bit masking to rewrite the program with control flows. However, for some situations [54], compilers (e.g., GCC) optimize the code too much (from a security view) and remove the unnecessary copy by reintroducing conditional jump instructions.

3.3.1.2 Data Flow

Figure 3.4 shows an example of secret-dependent data access. T is an array with 128 elements. The size of each element is one byte. So the total size of the array is 128 bytes. Suppose an attacker can observe the memory access at the granularity of one cache lines (64

```
uint8_t T[128];                              ...
...                                          lea    edx, [addr_T]
int example3(uint8_t k, uint32_t m){         mov    eax, [addr_k]
  uint32_t index = m;                        and    eax, 0x7f
  index = index + k % 128;                   add    eax, [addr_m]
  uint8_t t = T[index];                      movzx
  ...                                        eax, [addr_T + eax*1]
}                                            ...
```

(a) A true leakage

```
uint8_t T[32];                               ...
...                                          lea    edx, [addr_T]
int example4(uint8_t k, uint32_t m){         mov    eax, [addr_k]
  uint32_t index = m;                        and    eax, 0x3f
  index = index + k % 32;                    add    eax, [addr_m]
  uint8_t t = T[index];                      movzx
  ...                                        eax, [addr_T + eax*1]
}                                            ...
```

(b) A false positive

Figure 3.4: Secret-dependent data accesses

bytes), then it is not possible to hold all the data inside the one cache line. An attacker can get some information of the value secret in Figure 3.4(a) by observing the cache line access. Figure 3.4(b) shows a different example. In this example, only 32 different elements can be accessed in the array and the size of the array is 32 bytes, which is smaller than the size of a cache line. Depending on the base address of the array, the array can be stored in one cache line or two consecutive cache lines. Therefore, such a code may still be vulnerable to side-channel vulnerabilities. However, existing tools (e.g., CacheD [13]) use a concrete base address. Under the circumstance, these tools may miss such vulnerabilities.

3.3.2 Performance

The second limitation of current side-channel detection tools is the performance bottleneck, especially for tools based on symbolic analysis. For example, CaSym [21] can only analyze small programs. CacheD performs better than CaSym. However, it still cannot handle large programs such as RSA directly. It uses some domain knowledge to analyze only part of the program. The expensive cost comes from the symbolic execution. Symbolic execution can be used to explore all the possible paths, which is useful for many tasks. Nevertheless, it is significantly less likely for cache side-channel bugs in crypto primitives. Cryptography primitives are more likely to have an even coverage of different paths (based on pseudo-random cipher states), and a branch-based side-channel is vulnerable from the very first branch on secret data, making complex path conditions less often relevant. Therefore, we choose to analyze a trace instead of the whole path. Another bottleneck is that the IR. While the IR can significantly decrease the difficulty of the implementation, it also introduces an average of 10x of the overall overhead. Third, using SMT solvers can be convenient because of its generality, and it is commonly used for its precision when checking branch feasibility in symbolic execution. But the decision problems that an SMT solver handles are typically at least NP-hard. On the other hand, we can use some methods to avoid the expensive SMT solving and find side-channel leakages more efficiently.

3.4 Method

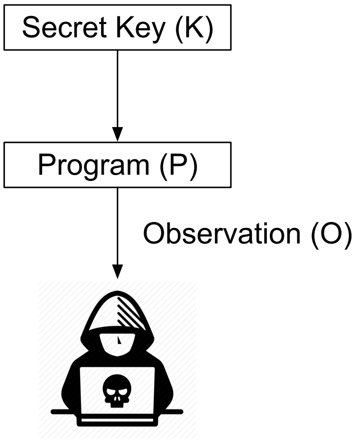


Figure 3.5: A side-channel attack

In this section, we give necessary definitions and notations for dealing with programs and side-channels.

Shown in Figure 3.5, a side-channel attack can be formulated into the following steps. A program (β) has K as its sensitive input (e.g., the encryption key) and M as its the public input (e.g., the plaintext). In a real execution, an adversary may have some observations (O) of the program. Examples of these observations include the timing, CPU usages, and electromagnetic signals (EM). In this dissertation, we use secret-dependent control flows and secret-dependent data accesses as observations.

With the above definition, we have the following mapping between β, K, M, and O:

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

We model a side-channel in the following way. An adversary does not have access to K, but he knows β, M, and O. For one execution of a deterministic program, once $k \in K$ and $m \in M$ are fixed, the observation ($o \in O$) should also be determined. An attacker knows β, o, and m. The attacker wants to infer the value of k. Moreover, we assume an attacker can change the public input ($m \in M$) while keeping the secret input k. The threat model is similar to a chosen-plaintext attack. We now discuss how to model observations (O), which are the direct information that an adversary gets during the attack.

For one execution, a program (β) has many temporary values ($t_i \in T$). Once β (program), k (secret), and m (message, public) are determined, t_i is also fixed. Therefore, $t_i = f_i(\beta, k, m)$, where f_i is a function that maps between t_i and (β, k, m). In this dissertation, we consider two code patterns that can be exploited by an attacker, *secret-dependent control transfers* and *secret-dependent data accesses*.

3.4.1 Secret-dependent Control Transfers

A control-flow path is secret-dependent if there are two different input-sensitive keys (K) that can lead to different branch conditions. We define a branch to be secret-dependent if

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

where PC represents the path condition. An adversary can observe which branch the code executes if the branch condition equals t_b. We use the constraint $c_i : f_i(\beta, k, m) = t_b$ to model the observation (o) on secret-dependent control-transfers. Note that in the above definition, m is also a variable. So it is possible for some $m \in M$, we cannot find a pair of two different k_{i1} and k_{i2} satisfying the above inequation. For example, in Figure 3.3, if $m = \texttt{0}$, the function example1 will always execute the branch 2. However, if $m = \texttt{0xffffffff}$, the function example1 executes the branch 1 when $k = \texttt{0}$, and the function example1 executes the branch 2 when $k = \texttt{0xf000000}$.

3.4.2 Secret-dependent Data Accesses

Similar to secret-dependent control-flow transfers, a data access operation is secret-dependent if different input sensitive keys (K) cause accesses to different memory addresses. We use the model from CacheD [13]. The low L bits of the address are generally unimportant in side-channels.

A data access is secret-dependent if

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

If the memory access equals to t_b, we can use the constraint $c_i : f_i(\beta, k, m) >> L = t_b >> L$ to model the observation on secret-dependent data accesses. Let us take a look at the examples in Figure 3.4(a). Suppose the base address of the array T is 10. We symbolize both the k and m. The formula that represents the memory access at line 6 is $10 + m + (k \mod 128)$. Therefore, we have the formula $(10 + m + (k_1 \mod 128)) >> 6 \neq (10 + m + (k_2 \mod 128)) >> 6$. Here is one possible solution: $m = 0, k_1 = 0, k_2 = 127$. In fact, for any m, we can always find possible k_1 and k_2 that satisfy the above inequation. It is a true leakage. The leakage can also be identified by previous tools such as CacheD. Similarly, we can model the memory access at line 6 with the formula: $10 + m + (k_1 \mod 32) >> 6 \neq 10 + m + (k_2 \mod 32) >> 6$. If $m = 0$, we cannot find satisfiable k_1 and k_2 for the above inequation, which indicates that it will always access the same cache line at line 6. However, it is possible that for some $m \in M$, we can find such a pair of k_1 and k_2 ($m = 53, k_1 = 1, k_2 = 0$). If the actual public input is 0, existing tools (e.g., CacheD) can miss such a vulnerability. On the other hand, our tool can identify the leakages precisely.

3.5 Scalability

3.5.1 Trace-oriented Symbolic Execution

Symbolic execution is notorious for its expensive performance cost. Previous trace-oriented symbolic execution work [13, 85] has serious performance bottlenecks. As a result, these approaches either apply only to small programs [85] or require domain knowledge [50] to simplify the analysis. These tools interpret each instruction and update memory cells and registers with formulas that captured the semantics of the execution and search different input values that can lead to different execution behaviors using a constraint solver. We implement the approach presented in Section 3.4 and model the side-channels as formulas. While the tools can analyze some simple cases such as AES, it cannot handle complicated examples such as RSA. We observe that finding side-channels using symbolic execution differs from traditional symbolic execution, and the symbolic analysis approach can be optimized to be as efficient as other methods.

3.5.2 Interpret Instructions Symbolically

Current binary analysis frameworks [101, 102, 111] translate machine instructions into intermediate languages (IR) to simplify analysis since the variety of machine instructions is enormous, and their semantics is complex. The Intel Developer Manual [112] documents more than 1000 different x86 instructions. Unfortunately, the IR layer, which reduces the workload of these tools, is not suitable for side-channels analysis because IR-based or source code side-channels analyses do not represent the executed instructions accurately enough to analyze fully their control and memory accesses. For example, a compiler may use conditional moves or bitwise operations to eliminate branches. Also, as IRs are not a superset or a subset of ISA, it is hard to rule out conditional jumps introduced by IR and add real branches eliminated by IR transformations.

Moreover, the IR causes significant overhead [113]. Translating machine instructions into IR is time-consuming. For example, REIL IR [114], adopted in CacheS [22], has multiple transform processes, from binary to VEX IR, BAP IR, and finally REIL IR. Also, IR increases the total number of instructions. For example, x86 instruction *test eax, eax* transfers into 18 REIL IR instructions.

Our Solution We abandoned IR layers and expended the effort to implement symbolic execution directly on x86 instructions. Table 3.1 shows that eliminating the IR reduces the number of instructions examined during analysis. Previous works [113] also adopted a similar approach to speed up fuzzing. Our implementation differs from that work in two aspects: 1) We use complete constraints. 2) We run the symbolic execution on one execution path each time. Our approach is approximately 30 times faster than using an IR (transferring ISA into IR and symbolically executing it).

Table 3.1: The number of x86, REIL IR, and VEX IR instructions on the traces of crypto programs.

	Number of x86 Instructions	Number of VEX IR	Number of REIL IR
AES OpenSSL 0.9.7	$1,704$	$23,938$ (15x)	$62,045$ (36x)
DES OpenSSL 0.9.7	$2,976$	$41,897$ (15x)	$100,365$ (33x)
RSA OpenSSL 0.9.7	$1.6 * 10^7$	$2.4 * 10^8$ (15x)	$5.9 * 10^8$ (37x)
RSA mbedTLS 2.5	$2.2 * 10^7$	$3.1 * 10^8$ (15x)	$8.6 * 10^8$ (39x)

3.5.3 Constraint Solving

As discussed in the previous section, the problem of identifying side-channels can be reduced to the question below.

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

Existing approaches rely on satisfiability modulo theories (SMT) solvers (e.g., Z3 [115]) to find satisfying assignments to k_1 and k_2. While this is a universal approach to solving constraints, for constraints of this form, using custom heuristics and testing is much more efficient in practice. Constraint solving is a decision problem expressed in logic formulas. SMT solvers transfer the SMT formula into the boolean conjunctive normal form (CNF) and feed it into the internal boolean satisfiability problem (SAT) solver. The translation process, called “bit blasting”, is time-consuming. Also, as the SAT problem is a well-known NP-complete problem, it is hard to deal for huge formulas. Despite the rapid improvement in SMT solvers in recent years, constraint solving remains one of the obstacles to scaling the analysis of real-world crypto-systems.

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

3.6 Design and Implementation

3.6.1 Trace Logging

The trace information can be logged via some emulators (e.g., QEMU [96]) or dynamic binary instrumentation tools (DBI). In this work, we run a program with the concrete input under the DBI to record 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 registers.
- The memory address and the length of the sensitive information. Most crypto libraries store sensitive information in arrays, variables or contiguous buffer.

3.6.2 Instruction Level Symbolic Execution

The primary 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 original input that was used to generate the execution trace but still satisfies these constraints, an attacker will have the same observations on control-flow transfers and data-access patterns.

The tool runs symbolic execution on top of execution traces. At the beginning of the symbolic execution, the tool creates new symbols for each byte in the raw buffer. For other data in the register or memory at the beginning, we use concrete values from the runtime execution information. 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 of concrete values and symbols accumulated through the symbolic execution. We implement several rules to simplify the formulas. For each formula, the tool will check whether 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.

3.6.2.1 Verification and Optimization

We run the symbolic execution (SE) on the x86 instructions. In other words, we do not rely on any intermediate languages to simplify the implementation 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, mistakes are inevitable. Therefore, we verify the correctness of the SE engine during the execution. The tool will collect the runtime execution 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 do not match, we check the code and fix the error. Also, if the formula does not contain any symbols, the tool will use the concrete value instead of symbolic execution.

3.6.2.2 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, such as CALL, JUMP, and RET, transfer control flows from one code segment location to another. Since the transfer is independent of the input sensitive information, an attacker is not able to infer any sensitive information from the control-flow. So unconditional control-transfer does not leak any information based on our threat model. During the symbolic execution, we update the register information and memory cells with new formulas accordingly.

The conditional control-flow transfer instructions, such as conditional jumps, depending on CPU states, may or may not transfer control flows. For conditional jumps, the CPU will test if certain condition flags (e.g., CF = 0, ZF =1) are met and jump to certain branches, respectively. The symbolic engine computes the flag and represents the flag in a symbol formula. Because we are running on a symbolic execution on an execution trace, we know which branch is executed. If a conditional jump uses the CPU status flag, we will generate the constraint accordingly.

For examples, considering the following x86 code snippet.

```
...
0x0000e781      add dword [local_14h], 1
0x0000e785      cmp dword [local_14h], 4
```

```
0x0000e789      jne 0xe7df
0x0000e78b      mov dword [local_14h], 0
...
```

At the address 0x0000e781, the value at the address of local_14h can be written as $F(\vec{K})$. At the address 0x0000e785, 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 (0x0000e789), is stored as a *formula tuple (address, formula)*. Each formula tuple represents one potential leakage site.

3.6.2.3 Secret-dependent Data Accesses

Like input-dependent control-flow transfers, an adversary can also infer sensitive information from the data access patterns 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 symbol 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 does not leak any sensitive information according to our threat model. Otherwise, we 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 CacheD [13], the formula can be written as

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

where L represents the minimum memory address granularity that an attacker can observe. For example, Flush and Reload attacks target on one cache line, which means the value of L is 6 for 64 byte cache line.

3.6.2.4 Information Flow Check

Lorem is designed to help software developers find and understand the side-channel vulnerabilities. To ease the procedure of fixing bugs, we also track the information flow for each byte of the input buffer. The step can be seen as the multiple-tag taint analysis. With the help of the information from symbolic execution, we can implement a simple but relatively precise information flow track. At the beginning of the analysis, Lorem keep track of each byte in the original buffer. When Lorem symbolically executes each instruction in the trace, it will check every value read from registers or memories. 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 original information passes through the instruction. Since each byte in the sensitive buffer is represented as a symbol with a unique ID, Lorem can know which byte in the origin buffer goes through the instruction.

3.7 Evaluation

We evaluate our method on real-world crypto libraries, which include OpenSSL, mbedTLS, Libgcrypt and Monocypher. OpenSSL is the most commonly used crypto libraries in today's software. mbedTLS (previous known as PolarSSL) is designed to be easy to understand and fit on embedded devices. We also evaluate Monocypher, a new cryptographic library that resists to most side-channel attacks. According to the manual of Monocypher, Monocypher is designed to have no secret dependent indices and secret dependent branches. Therefore, Monocypher should be secure under our threat model. Libgcrypt is a cryptographic library from the GunPG. As some previous work (e.g., CacheD) choose to evaluate their tools on Libgcrypt, we can compare the evaluation results by applying our tool on Libgcrypt.

For some crypto libraries, we evaluate multiple versions. After that, we compare the evaluation results with the version history to track these leakages' patches. In summary, we evaluate our tool on the following libraries.

- OpenSSL: 0.9.7, 1.0.2f, 1.0.2k, 1.1.0f, 1.1.1, 1.1.1g
- MbedTLS: 2.5, 2.15
- Libgcrypt: 1.6.1, 1.7.3, 1.8.5
- Monocyper: 3.0

We refer to the test cases of these libraries and write simple programs that use the crypto libraries. For symmetric encryption, the length of keys is 256 bits. For RSA, the length of the keys is 1024 bits. For ECDSA, the length of the key is 384 bit. The input public message is "Hello World!". We mark variables and buffers that store the sensitive data. For DES and AES, we mark symmetric keys as secrets. For RSA, we mark private keys as secrets. For ECDSA, we mark nonces and private keys as secrets.

We build the source program into 32-bit x86 Linux executables with GCC 7.5 running on Ubuntu 16.04. While our tool can work on the stripped binaries, in the evaluation, we keep the debug information to get the fine-grained information (e.g., line numbers). We run our experiments on a 2.90GHz Intel Xeon(R) E5-2690 CPU with 128GB RAM. We run the experiments simultaneously, but the execution time is calculated on a single-core.

During our evaluation process, we are interested in the following aspects:

1. Is Lorem effective to detect side-channels in real-world crypto systems? (Effectiveness)

2. How much performance overhead does the method introduce? (Performance)

3.7.1 Evaluation Result Overview

Table 3.2 summarizes the results. Lorem finds 913 leaks in the crypto libraries, among which, 241 are due to secret-dependent control-flow transfers and 672 are due to secret-dependent data accesses.

The evaluated algorithms can be classified into two categories: symmetric encryption and asymmetric encryption. Most of the side-channel leakages in symmetric implementations are from the lookup tables. The new implementation of OpenSSL has adopted some methods (e.g., one single S-box instead of four lookup tables, smaller lookup tables) to mitigate the problem.

We also evaluate our tool on the RSA implementation. With the optimization introduced in the previous sections, we need not apply domain knowledge to simplify the analysis. Our tool identifies all leakage sites reported by CacheD [13] and find a new leak in shorter time. We also find newer versions of RSA in OpenSSL have fewer leaks. Our tool can finish all the analysis in less than 6 hours.

3.7.2 Comparison with the Existing Tools

In this section, we compare Lorem with the existing trace-based side-channel detection tools on vulnerability detections. For other tools, we use the reported results in their papers [13]. As

Table 3.2: Evaluation results overview: Algorithm, Implementation, Side-channel Leaks (Leaks), Secret-dependent Control-flows (CF), Secret-dependent Data-flows (DF), The number of instructions (# Instructions), The Execution Time, and The Memory Usage.

Algorithm	Implementation	# Leaks	# CF	# DF	# Instructions	Time	RAM (MB)
						ms	
AES	mbedTLS 2.5	68	0	68	39,855	512	47
AES	mbedTLS 2.15	68	0	68	39,855	520	47
AES	OpenSSL 0.9.7	75	0	75	1,704	231	15
AES	OpenSSL 1.0.2f	88	0	88	1,350	36	15
AES	OpenSSL 1.0.2k	88	0	88	1,350	35	16
AES	OpenSSL 1.1.0f	88	0	88	1,420	36	16
AES	OpenSSL 1.1.1	88	0	88	1,586	43	16
DES	mbedTLS 2.5	15	0	15	4,596	58	8
DES	mbedTLS 2.15	15	0	15	4,596	57	8
DES	OpenSSL 0.9.7	6	0	6	2,976	163	11
DES	OpenSSL 1.0.2f	8	0	8	2,593	166	11
DES	OpenSSL 1.0.2k	8	0	8	2,593	165	11
DES	OpenSSL 1.1.0f	8	0	8	4,260	182	21
Triple DES	OpenSSL 1.1.0f	9	0	9	13,105	369	56
DES	OpenSSL 1.1.1	6	0	6	8,272	229	43
						seconds	
Chacha20	Monocypher 3.0	0	0	0	149,353	2	15
Poly1305	Monocypher 3.0	0	0	0	1,213,937	15	43
Argon2i	Monocypher 3.0	0	0	0	4,595,142	37	102
Ed25519	Monocypher 3.0	0	0	0	5,713,619	271	98
ECDSA	mbedTLS 2.5	6	6	0	4,214,946	48	689
ECDSA	mbedTLS 2.15	4	4	0	4,192,558	102	680
ECDSA	OpenSSL 1.0.2f	5	4	1	8,248,322	101	980
ECDSA	OpenSSL 1.0.2k	5	4	1	8,263,599	100	906
ECDSA	OpenSSL 1.1.0f	5	4	1	6,100,465	76	705
ECDSA	OpenSSL 1.1.1	0	0	0	10,244,076	121	1,048
ECDSA	OpenSSL 1.1.1g	0	0	0	9,266,191	102	1,001
						minutes	
RSA	mbedTLS 2.5	6	6	0	22,109,246	39	3,708
RSA	mbedTLS 2.15	12	12	0	24,484,441	44	4,012
RSA	OpenSSL 0.9.7	107	105	2	17,002,523	23	3,601
RSA	OpenSSL 1.0.2f	38	27	11	14,468,307	29	3,301
RSA	OpenSSL 1.0.2k	36	27	9	15,285,210	40	3,402
RSA	OpenSSL 1.1.0f	31	22	9	16,390,750	34	3,701
RSA	OpenSSL 1.1.1	4	4	0	18,207,016	8	4,181
RSA	OpenSSL 1.1.1g	8	8	0	18,536,796	5	3,901
RSA	Libgcrypt 1.6.1	11	9	2	9,527,231	2	2516
RSA	Libgcrypt 1.7.3	14	14	0	10,513,606	14	2,701
RSA	Libgcrypt 1.8.5	8	8	0	27,407,986	113	4,701
Total		913	241	672	167,155,052	341m	

other tools do not quantify leakage sites, we only include the time of detecting vulnerabilities to perform a fair comparison.

The comparison result with CacheD [13, 14] is shown in Table 3.3. Note that one statement in the source code can be compiled into several machine instructions. So it is possible that one statement can have multiple leakage points. In the circumstance, we consider it takes only one leakage. We have confirmed that Lorem can identify all the secret-dependent data access vulnerabilities reported by CacheD. In addition, Lorem finds many new ones. CacheD fails to detect some vulnerabilities for two reasons. First, CacheD can only detect secret-dependent data access vulnerabilities. Lorem can detect secret-dependent control-flows as well. Second, according to the CacheD paper, CacheD times out after 20 hours when processing asymmetric ciphers. CacheD applies some domain knowledge to simplify and speed up the analysis. While these optimizations do not introduce false positives, they may miss some vulnerabilities. Third, CacheD only symbolizes the encryption keys. Some side-channel leakages occurs when the input messages equal to some certain values. It is possible that CacheD misses some leakages as we have discussion the previous sections. We notice that the number of instructions in these traces are different due to the different analysis starting functions and build options during the evaluation. Table 3.3 shows that Lorem is faster than CacheD. Lorem is much faster than CacheD when analyzing the same number of instructions. For example, when we test Lorem on AES from OpenSSL 0.9.7, Lorem is over 100x faster than CacheD.

Table 3.3: Comparison with CacheD: Time, Secret-dependent Control-flows (CF), Secret-dependent Data-flows (DF), The Number of Instructions (# Instructions).

	CacheD			Abacus		
Benchmark	Time (s)	# Instructions	DF	Time (s)	# Instructions	CF+DF
AES OpenSSL 0.9.7	43.4	791	48	0.23	1,704	0+75
AES OpenSSL 1.0.2f	48.5	2,410	32	0.04	1,350	0+88
RSA OpenSSL 0.9.7	199.3	674,797	1	1,351.6	17,002,523	105+2
RSA OpenSSL 1.0.2f	165.6	473,392	1	1,753.3	14,468,307	27+11
RSA Libgcrypt 1.6.1	11542.3	26,848,103	2	128.1	9,527,231	9+2
RSA Libgcrypt 1.7.3	10788.9	27,775,053	0	891.7	10,513,606	14+0
Total	22,788.0	55,738,546	84	4,125.0	51,514,721	155+178
# of Instructions per second		CacheD: 2,445			Abacus: 12,489	

DATA [14] identifies side-channel leakages by finding differences in execution traces of the test program under *various secret inputs*. According to the original DATA paper, it uses 443 different traces to analyze the side-channel vulnerabilities in symmetric cyphers and 450 different traces to analyze the side-channel vulnerabilities in asymmetric cyphers. On the other hand, Lorem detects side-channel vulnerabilities from one execution trace. Lorem uses symbolic analysis to extract formulas that model each side-channel leakage. After that, we sample the formula with *various secret inputs* to detect and quantify each leakage site. In theory, DATA might have better code coverage than Lorem because it uses more execution traces, but Lorem has the following advantages. a) Lorem is faster than DATA. For example, it takes 116 minutes for DATA to detect vulnerabilities in the RSA implementation in OpenSSL 1.1.0f. Lorem only spends 34 minutes, as shown in Table 3.2. It takes 13 minutes and 20 minutes for DATA to analyze the side-channel leakages in AES and DES, respectively. On the other hand, Lorem finishes its analysis in less than ten seconds while finding all the leakages reported by DATA. b) Because Lorem does not exexute the test program again when we have a new *secret input*, Lorem can test more input secrets on these formulas within the same time to achieve better precision. c) DATA tries to use leakage models (domain knowledge) to classify each leakage. The strength of Quincunx is that it does not need such domain knowledge. DATA reports 278 control-flow and 460 memory-access leaks for the RSA implementation in OpenSSL 1.1.0f. Among these leakages, they find one new vulnerability in RSA after some manual analysis. Lorem also finds a new vulnerability. For each leakage site, Lorem can provide concrete examples to trigger the issue.

3.7.3 Case Studies

3.7.3.1 DES and AES

Our tool confirms that both implementations in OpenSSL and mbedTLS have side-channel leakages sites. Moreover, we find that all the leakages belong to the secret-dependent data accesses. The reference implementation of AES uses four lookup tables to speed up the computation. However, such implementation are vulnerable to side-channel attacks. Previous work has shown an end-to-end attack to fully recover the key. Our findings is similar to the previous work. We also notice that recent versions of OpenSSL (after 1.0.1) use a modified implementation with smaller S-tables. However, Lorem also finds leakage sites in the version.

3.7.3.2 RSA

We also evaluate our tool on RSA implementations. With the optimizations introduced in this chapter, we do not apply any domain knowledge to simplify the analysis. Therefore, our tools can identify all the leakage sites reported by CacheD [13] in a shorter time. Our tool finds that most leakages in RSA occur in the big number implementation. We also find the newer versions of RSA in OpenSSL tend to have fewer leakages.

Even for the up-to-date version of OpenSSL, Lorem still find several side-channel leakages. Figure 3.6 shows an unknown leakage site in OpenSSL 1.1.1. It is a secret-dependent control flow transfers. However, as the branch is inside a loop and a bit shift function causes the branch leak different bits from the sensitive buffer. The function can leak the last digit from big number **X**. The leakage is severe due to the function BN_rshift1. Each time, function BN_rshift1 shifts X right by one and places the result in X. Therefore, an attacker can infer multiple bits of X by observing the branch at line 3.

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

Figure 3.6: Previously unknown sensitive secret-dependent branch leaks from function int_bn_mod_inverse in OpenSSL 1.1.1.

3.7.3.3 Monocypher

Monocypher is a small, easy to use cryptographic library with performance comparable to LibSodium [66] and NaCl [65]. We choose four ciphers that are designed to be side-channel resistant from the library. These ciphers have no data flow from secrets to branch conditions and load addresses. Monocypher should be side-channel resistant under our threat models. We analyze these ciphers with Lorem, and it reports no leaks. This indicates that Lorem is effective for validating countermeasures.

Chapter 4 | Precise Analysis on Single-trace Attacks

4.1 Problem

In Chapter 3, we present a method to identify address-based side-channel leakage sites in cryptography libraries [13, 35, 107, 108]. However, many reported leakage sites are not patched by the developers. Recent work on side-channel vulnerability identification [13, 14, 20–23] also faced the same situation. For example, DATA [14] reports 2,248 potential leakage sites for the RSA implementation in OpenSSL 1.1.0f. Further analysis shows 1,510 leaks can be dismissed, but that still leaves 460 data-access leaks and 278 control-flow leaks. Many of these vulnerabilities have not been fixed by the developers for a variety of reasons.

First, some vulnerable implementations perform better. For example, RSA implementations usually adopt the Chinese Remainder Theorem (CRT) optimization, which is faster but vulnerable to fault attacks [24]. Second, fixing old vulnerabilities can introduce new weaknesses. Third, most vulnerabilities pose a negligible risk. Although some vulnerabilities result in the key being entirely compromised [6, 24], many others are less severe in reality. Therefore, we need a proper quantification metric to assess the sensitivity of side-channel vulnerabilities, so a developer can efficiently triage them.

Previous work on side-channel quantification [23, 82] can identify numerous leakages or even provide an upper bound on the amount of leakage, which is useful to verify that an implementation is secure if it incurs zero leakage. However, these techniques cannot quantify the severity of a leak because they over approximate the leakage. For example, CacheAudit [23] reports that the upper-bound leakage of AES-128 exceeds the original key size. Besides, existing side-channel quantification work [23, 82] assumes an attacker runs the target program multiple times with different input secrets and calculates an "average" estimation, which is different from real attack scenarios when the secret that an attacker wants to retrieve is fixed. As a consequence, these results are less useful in assessing the severity level of each leakage site.

To overcome these limitations, we propose a novel method to quantify information leakage precisely. We quantify the number of bits that can be leaked during a real execution and define the amount of leaked information as the cardinality of possible secrets based on an attacker's observation. Before an attack, an adversary has a large but finite input space. Whenever the adversary observes a leakage site, he can eliminate some impossible inputs and reduce the input space's size. In an extreme case, if the input space's size reduces to one, an adversary has determined the input, which means all secret information (e.g., the entire secret key) is leaked. By counting the number of inputs [25], we can quantify the information leakage precisely. We use symbolic execution to generate constraints to model the relation between the original sensitive input and an attacker's observations. Symbolic execution provides fine-grained information, but it is expensive to compute. Therefore, prior symbolic execution work [13, 21, 22] either analyzes only small programs or applies domain knowledge [13] to simplify the analysis. We examine the bottleneck of the trace-oriented symbolic execution and optimize it to work for real-world crypto-systems.

We have implemented the proposed technique in a tool called Abacus and demonstrated it on real-world crypto libraries, including OpenSSL, mbedTLS, Libgcrypt, and Monocypher. We collect execution traces of these libraries and apply symbolic execution to each instruction. We model each side-channel leak as a logic formula. These formulas precisely model side-channel vulnerabilities. Then we use the conjunction of these formulas to model the leaks at a statement that appears in different location in the execution trace file (e.g., leaks inside a loop). Finally, we introduce a Monte Carlo sampling method to estimate the information leakage. The experimental results confirm that Abacus precisely identifies previously known vulnerabilities and reports how much information is leaked and which byte in the original sensitive buffer is leaked. We also test Abacus on side-channel-free algorithms. Abacus produces no false positives. The result also shows the newer version of crypto libraries leak less information than earlier versions. Abacus also discovers new vulnerabilities. With the help of Abacus, we confirm that some of these vulnerabilities are severe.

In summary, we make the following contributions:

- We propose a novel method that can quantify fine-grained leaked information from side-channel vulnerabilities that result from actual attack scenarios. Our approach differs from previous ones in that we model real attack scenarios for one execution. We model the information quantification problem as a counting problem and use a Monte Carlo sampling method to estimate the information leakage.

- We implement the method into a tool and apply it to several pieces of real software. Abacus successfully identifies previous unknown and known side-channel vulnerabilities and calculates the corresponding information leakage. Our results are useful in practice. The leakage estimates and the corresponding trigger inputs can help developers to triage and fix the vulnerabilities.

4.2 Threat Model

The threat model in this chapter is similar to the threat model in our previous work from Chapter 3. Prior work [13, 21, 23] also shares the same threat model. That is, we assume an attacker can share some hardware resource with the victim. The attacker has the ability to probe the victim at any time of the execution. We also assume an attacker has noise-free observations. The attacker has the access to the victim's binary executable. We believe that the majority of address-based side-channel attacks in the literature applies to the thread model.

4.3 Background

Given an event e that occurs with the probability $p(e)$, we receive

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

bits of information by knowing the event e happens according to information theory [116]. Considering a char variable a with one byte storage size in a C program, its value ranges from 0 to 255. Assume a has a uniform distribution. If we observe that a equals 1, the probability of this observation is $\frac{1}{256}$. So we get $-\log(\frac{1}{256}) = 8$ bits information, which is exactly the size of a char variable in the C program.

Existing work on information leakage quantification typically uses Shannon entropy [20, 117], min-entropy [118], and max-entropy [23, 119]. In these frameworks, the input sensitive information K is considered as a random variable.

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

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

Shannon entropy can be used to quantify the initial uncertainty about the sensitive information. It measures the amount of information in a system.

Min-entropy describes the information leaks for a program with the most likely input. For example, min-entropy can be used to describe the best chance of success in guessing one's password using the most common password.

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

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

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

As it is easy to compute, most recent works [23, 119] use max-entropy as the definition of the amount of leaked information.

To illustrate how these definitions work, we consider the code fragment in Figure 4.1. It has two secret-dependent control-flows, A and B.

```
uint8_t key[2], t1, t2;
get_key(key);              // 0 <= key[0], key[1] < 256
t1 = key[0] + key[1];
t2 = key[0] - key[1];
if (t1 < 4){               // leakage site A
    foo();
}
if (t2 > 0){               // leakage site B
    doo();
}
```

Figure 4.1: Side-channel leakage

In this chapter, we assume an attacker can observe the secret-dependent control-flows in Figure 4.1. Therefore, an attacker can have two different observations for each leak site depending on the value of the key: A for function foo is executed, $\neg A$ for function foo is not executed, B for function doo is executed, and $\neg B$ for function doo is not executed. Now the question is how much information can be leaked from the above code if an attacker knows which branch is executed?

Table 4.1: The distribution of observations

Observation (o)	A	$\neg A$	B	$\neg B$
Number of Solutions	65526	10	32768	32768
Possibility (p)	0.9998	0.0002	0.5	0.5

Assuming key is uniformly distributed, we can calculate the corresponding possibility by counting the number of possible inputs. Table 4.1 describes the probability of each observation. We use the above three types of leakage metrics to calculate the amount of leaked information for leak A and leak B.

Min Entropy

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

$$\begin{aligned} \textit{min-entropy}_A &= -\log_2 0.9998 = 0.000 \text{ bits} \\ \textit{min-entropy}_B &= -\log_2 0.5 = 1.000 \text{ bits} \end{aligned}$$

Max Entropy

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

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

Shannon Entropy

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

$$\begin{aligned} \textit{Shannon-entropy}_A = -(&0.9998 * \log_2 0.9998 \\ &+ 0.0002 * \log_2 0.0002) \end{aligned}$$

$$
\begin{aligned}
&= 0.000 \text{ bits} \\
\mathit{Shannon\text{-}entropy}_B &= -(0.5 * \log_2 0.5 + 0.5 * \log_2 0.5) \\
&= 1.000 \text{ bits}
\end{aligned}
$$

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

4.4 Leakage Definition

In this section, we discuss how Abacus quantifies the amount of leaked information. We first present the limitation of existing quantification metrics. Then, we introduce our model, the mathematical notation used in the chapter, and our method.

4.4.1 Problem Setting

Existing static side-channel quantification works [20, 23, 120, 121] define information leakage using max entropy or Shannon entropy. If zero bits of information leakage is reported, a program is secure. However, when a tool using these metrics reports leakage, it is the "average" leakage. In a real attack, the leakage could be dramatically different.

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

Figure 4.2: A dummy password checker

Consider a password checker sketched in Figure 4.2. The password checker takes an 8-byte char array (exclude NULL character) and checks if the input is the correct password. If an attacker uses a side-channel attack to determine that the code executes branch $\{1\}$, they can infer that the password equals to "password", in which case the attacker retrieves the full password. Therefore, the total leaked information should be 64 bits, which equals to the size of the original sensitive input when the code executes branch 1.

However, prior static approaches cannot precisely capture the amount of leakage. According to the definition of Shannon entropy, the leakage will be

$$\frac{1}{2^{64}} * \log_2 \frac{1}{2^{64}} + \frac{2^{64}-1}{2^{64}} * \log_2 \frac{2^{64}-1}{2^{64}} \approx 0 \,\text{bits}.$$

Max-entropy is defined from the number of possible observations. Because the program has two branches, tools based on max-entropy will report the code has a $\log_2 2 = 1.0$ 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 so input values and real runtime information are neglected by their approaches. They assume an attacker runs the program multiple times with many different or random sensitive inputs. As shown in Figure 4.3(a), previous methods, both Shannon entropy and max entropy, give an "average" estimate of the information leakage. However, it is not the typical scenario for an adversary to launch a side-channel attack. When a side-channel attack happens, the adversary wants to retrieve the sensitive information, in which case the sensitive information is fixed (e.g., AES keys). The adversary will run the attack over and over again with fixed input and guess the value bit by bit (e.g., Kocher's timing attacks [5]), as in Figure 4.3(b). We want to have a theory for dynamic analysis that if the theory says an attack leaks x bits of secret information from a side-channel vulnerability, then x should be useful in estimating the sensitive level of the vulnerability. However, the above methods all fail in real attack models.

4.4.2 Precise Analysis

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

We assume a program (β) has K as its sensitive input. The input K should be a finite set of keys. The program also takes known messages M as its input. During an AES encryption, for example, β is the encryption function. Here K is the set of all possible AES keys, and M represents the set consisting of all plaintext messages to be encrypted. In a real execution, an adversary may have some observations (O) of the program. This dissertation only uses secret-dependent control flows and secret-dependent data accesses as observations.

With the above definition, we have the following mapping between β, K, M, and O:

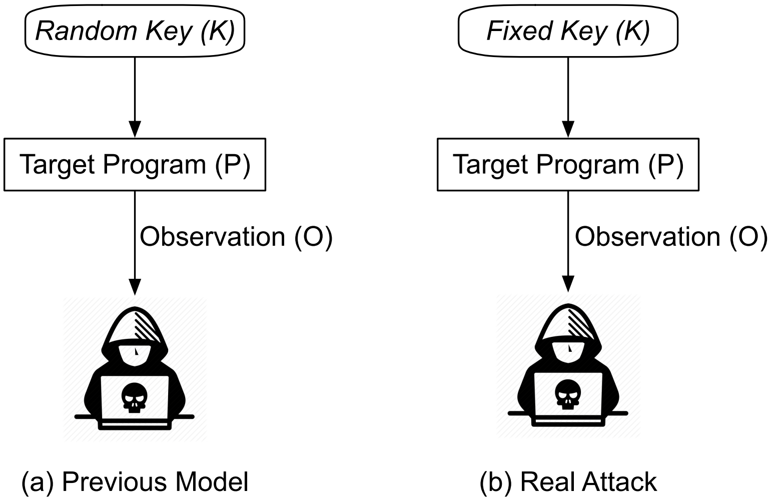


Figure 4.3: The gap between real attacks and previous models

$$\beta(K, M) \to O.$$

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

Then the problem of quantifying the amount of leaked information can be restated as the following question:

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

In information theory, the mutual information (I) is a measure of the mutual dependence between two variables. Here we use I to describe the dependence between original sensitive keys (K) and attackers' observations (O), which is defined as

$$I(K; O) = \sum_{k \in K} \sum_{o \in O} p(k, o) \log_2 \frac{p(k, o)}{p(k)p(o)}, \tag{4.1}$$

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

$$I(K; O) = H(K) - H(K|O), \tag{4.2}$$

where $H(K|O)$ is the entropy of K with the condition O. It quantifies the uncertainty of K given the value of O. In other word, the conditional entropy $H(K|O)$ marks the uncertainty about K after an adversary has gained some observations (O).

$$H(K|O) = -\sum_{o\in O} p(o) \sum_{k\in K} p(k|o)\log_2 p(k|o) \tag{4.3}$$

In this research, we hope to give a very precise definition of information leakages. Suppose an attacker runs the target program with one fixed input, we want to know how much information he can infer by observing the memory access patterns (o). We come to the simple slogan [118] that

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

Next we compare the Eq. (4.2) with the above slogan, we find that $H(K)$ is the *Initial uncertainty* and $H(K|O)$ is the *Remaining uncertainty*. During a real attack, the observation (o) is known. Thus we have $H(K|O) = H(K|o)$.

Therefore, we define the amount of leaked information as

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

For a program (β) without knowing any domain information, all possible sensitive inputs should appear equally. Therefore, for any $k \in K$, $p(k) = \frac{1}{|K|}$. So we have

$$H(K) = \sum_{k\in K} \frac{1}{|K|}\log_2|K| = \log_2|K|.$$

For any $k' \in K \setminus K^o$, $p(k'|o) = 0$. We get

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

$$= \log_2 |K^o|.$$

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

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

The amount of leaked information $L_{\beta(k)\rightarrow o}$ based on the observation (o) is

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

The above definition [122] can be understood in an intuitive way. Suppose an attacker wants to guess a 128-bit encryption key from a program. Without any domain knowledge, he can find the key by performing exhaustive search over 2^{128} possible keys. However, the program has a side-channel leakage site. After the program finishes execution, the attacker gets some leaked information and only needs to find the key by performing exhaustive search over 2^{120} possible keys. Then we can say that 8 bits of the information is leaked. In this example, 2^{128} is the size of K and 2^{120} is the size of K^o.

With the definition, if an attacker observes that the code in Figure 4.2 runs the branch 1, then the $K^{o^1} = \{\text{“password”}\}$. Therefore, the information leakage $L_{P(k)=o^1} = \log_2 2^{64} - \log_2 1 = 64$ bits, which means the key is totally leaked. If the attacker observes the code hits branch 2, the leaked information is $L_{P(k)=o^2} = \log_2 2^{64} - \log_2 (2^{64} - 1) \approx 0$ bit.

We can also calculate the leaked information from the sample code in Figure 4.1. As the size of input sensitive information is usually public. The problem of quantifying the leaked information has been transferred into the problem of estimating the size of input key $|K^o|$ under the condition $o \in O$. The result is shown in Table 4.2. We can see that some branches (e.g., A) or traces leak much more information than some others. These kinds of branches can be vulnerable when an attacker's data is incorporated. For example, the code will skip some of the calculation if the value is 0 in big number multiplication.

In contrast, an *average* estimate based on random secret input information is around 1 bit, as shown in the previous section and Table 4.1, is not very useful in practice as an attacker is able to get much more leaked information in some attack scenarios. As the size of input-sensitive information is usually public, the problem of quantifying the leaked information is equivalent to the problem of estimating the size of input key $|K^o|$ under the condition $o \in O$.

Table 4.2: New leakage modeling results

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

4.5 Approximate Model Counting

In the previous section, we propose an information leakage definition for realistic attack scenarios to model two types of address-based side-channel leakages and show how to quantify them by calculating the number of input keys (K^o) that satisfy the formulas. Intuitively, we can use symbolic execution to capture math formulas and model counting to obtain the number of satisfying input keys (K^o). However, some preliminary experiments showed that this approach was far too expensive to use with real-world applications. In this section, we discuss the bottlenecks in this approach and propose a practical solution.

According to Definition 1 introduced in the previous section, the problem of quantifying the amount of leaked information can be reduced to the problem of computing the number of items in K^o. However, we find that while there are various propositional model counters (e.g., #SAT), they are not sufficiently scalable for production cryptosystem analysis. Besides, there is no open source modulo theories counter (#SMT) available.

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

We adopt multiple-step Monte Carlo sampling methods to count the number of possible inputs that satisfy the logic formula groups. The key idea is to split these constraints into several small formulas and sample them independently. We will introduce the method in the following subsection. In this section, we present the algorithm to calculate the information leakage based on Definition 1.

4.5.1 Problem Statement

For each leakage site, we model it with a constraint using the method presented in the previous section. Suppose the address of the leakage site is ξ_i, we use c_{ξ_i} to denote the constraint that models its side-channel leakage. For multiple leakage sites, we take the conjunction of these constraints to represent these leakage sites.

According to Definition 1, to calculate the amount of leaked information, the key is to calculate the cardinality of K^o. Suppose an attacker can observe n leakage sites, and each leakage site has the following constraints: $c_{\xi_1}, c_{\xi_2}, \ldots, c_{\xi_n}$ respectively. The total leakage can be calculated from the constraint $c_t(\xi_1, \xi_2, \ldots, \xi_n) = c_{\xi_1} \wedge c_{\xi_2} \wedge \ldots \wedge c_{\xi_n}$. The problem of estimating the total leaked information can be reduced to the problem of counting the number of different solutions that satisfies the constraint $c_t(\xi_1, \xi_2, \ldots, \xi_n)$. A simple method is to pick elements k from K and check if an element is also contained in K^o. Assume q elements satisfy this condition. In expectation, we can use $\frac{k}{q}$ to approximate the value of $\frac{|K|}{|K^o|}$.

However, the above sampling method fails in practice due to the following two problems.

1. The curse of dimensionality problem. $c_t(\xi_1, \ldots, \xi_n)$ is the conjunction of many constraints. Therefore, the input variables of each constraints will also be the input variables of the $c_t(\xi_1, \ldots, \xi_n)$. The sampling method fails as n grows. For example, if the program has 2 input whose size is one byte, the whole search space is a 256^2 cube. If we want the sampling distance between each point equals to d, we need $256^2 d$ points. If the program has 10 byte input, we need $256^{10} d$ points if we still we want the sampling distance equals to d.

2. The number of satisfying assignments could be exponentially small. According to the Chernoff bound [123], we need exponentially many samples to get a tight bound. On an extreme situation, if the constraint only has one unique satisfying solution, the simple Monte Carlo method cannot find the satisfying assignment even after sampling many points.

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

1. $c_t(\xi_1, \xi_2, \dots, \xi_n)$ is the conjunction of several short constraints c_{ξ_i}. The set containing the input variables of c_{ξ_i} is the subset of the input variables of $c_t(\xi_1, \xi_2, \dots, \xi_n)$. Some constraints have completely different input variables from other constraints.

2. Each time when we sample $c_t(\xi_1, \xi_2, \dots, \xi_n)$ with a point, the sampling result is *Satisfied* or not *Not Satisfied*. The outcome does not depend on the result of previous experiments. Also, as the amount of leaked information is calculated by a $\log$ function, we need not exactly count the number of solutions for a given constraint.

In regard to the above problems, we present our methods. First, we split $c_t(\xi_1, \xi_2, \dots, \xi_n)$ into several independent constraint groups. After that, we run a multi-step sampling method for each constraint.

4.5.2 Maximum Independent Partition

For a constraint c_{ξ_i}, we define function π, which projects the constraint into a set of different input symbols. For example, $\pi(k1 + k2 > 128) = \{k1, k2\}$.

Definition 2. *Given two constraints c_m and c_n, we call them independent iff*

$$\pi(c_m) \cap \pi(c_n) = \varnothing.$$

Based on Definition 2, we can split the constraint $c_t(\xi_1, \xi_2, \xi_3, \dots, \xi_n)$ into several independent constraints. There are many partitions. For our project, we are interested in the following one.

Definition 3. *For the constraint $c_t(\xi_1, \xi_2, \dots, \xi_n)$, we call the constraint group $g_1, g_2, \dots, g_m$ the maximum independent partition of $c_t(\xi_1, \xi_2, \dots, \xi_n)$ iff*

1. $g_1 \wedge g_2 \wedge \dots \wedge g_m = c_t(\xi_1, \xi_2, \dots, \xi_n)$,

2. $\forall i, j \in \{1, \dots, m\} \quad$ *and* $\quad i \neq j, \quad \pi(g_i) \cap \pi(g_j) = \varnothing$,

3. For any other partitions $h_1, h_2, \dots, h_{m'}$ satisfy 1) and 2), $m \geq m'$.

The reason we want a good partition of constraints is we want to reduce the dimensions. For example, a good partition of $F : (k_1 = 1) \wedge (k_2 = 2) \wedge (k_3 > 4) \wedge (k_3 - k_4 > 10)$ would be $g_1 : (k_1 = 1) \quad g_2 : (k_2 = 2) \quad g_3 : (k_3 > 4) \wedge (k_3 - k_4 > 10)$ We can sample each constraint independently and combine them with Theorem 1.

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

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

With Theorem 1, we transfer the problem of counting the number of solutions to a complicated constraint in a high-dimension space into counting solutions to several small constraints. We compute the maximum independent partition by iterating each ξ_i and applying the function π over the constraint ξ_i.

We apply Algorithm 1 to get the Maximum Independent Partition of the $c_t(\xi_1, \xi_2, \ldots, \xi_n)$.

Algorithm 1: The Maximum Independent Partition

```
input  :$c_t(\xi_1, \xi_2, \ldots, \xi_n) = c_{\xi_1} \wedge c_{\xi_2} \wedge \ldots \wedge c_{\xi_m}$
output :The Maximum Independent Partition of $G = \{g_1, g_2, \ldots, g_m\}$
Insert $c_{\xi_1}$ to $G$ as a new entry
for $i \leftarrow 2$ to $n$ do
    $S_{c_{\xi_i}} \leftarrow \pi(c_{\xi_i})$
    for $g_i \in G$ do
        $S_{g_j} \leftarrow \pi(g_j)$
        $S \leftarrow S_{c_{\xi_i}} \cap S_{g_j}$
        if $S \neq \varnothing$ then
            $g_j \leftarrow g_i \wedge g_{\xi_i}$
            continue
        end
        Insert $c_{\xi_i}$ to $G$ as a new entry
    end
end
```

4.5.3 Multiple-step Monte Carlo Sampling

After we split these constraints into several small constraints, we count the number of solutions for each constraint. Even though the dimension has been significantly reduced by the previous step, this is still a #P problem. For our project, we apply the approximate counting instead of exact counting for two reasons. First, we do not need to have a very precise result of the exact number of total solutions since the information is defined with a logarithmic function. We do not need to distinguish between constraints having 10^{10} or $10^{10} + 10$ solutions; they are very close to after taking logarithmic. Second, the precise model counting approaches, such as

Davis-Putnam-Logemann-Loveland (DPLL) search, have difficulty scaling up to large problem sizes.

We apply the “counting by sampling” method. For the constraint $g_i = c_{i_1} \wedge c_{i_2} \wedge, \ldots, \wedge c_{i_j}$ $\wedge, \ldots, \wedge c_{i_m}$, if the solution satisfies g_i, it should also satisfy any constraint from c_{i_1} to c_{i_m}. In other words, $K_{c_g i}$ should be the subset of K_{c_1}, K_{c_2}, ..., K_{c_m}. We notice that c_i usually has fewer inputs than g_i. For example, if c_{i_j} has only one 8-bit input variable, we can find the exact solution set $K_{c_{i_j}}$ of c_{i_j} by trying every possible 256 solution. After that, we only generate random input numbers for the other input variables in constraint g_i. With this simple yet effective trick, we reduce the number of inputs while still ensuring accuracy. The details of the algorithm is shown in Algorithm 2.

4.5.4 Error Estimation

Our result has a probabilistic guarantee that the error of the estimated amount of leaked information is less than 1 bit under the Central Limit Theorem (CLT) and uncertainty propagation theorem [124].

Let n be the number of samples and n_s be the number of samples that satisfy the constraint C. Then we get $\hat{p} = \frac{n_s}{n}$. If we repeat the experiment multiple times, each time we get a $\hat{p}$. As each $\hat{p}$ is independent and identically distributed, according to the central limit theorem, the mean value should follow the normal distribution

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

Here $E(p)$ is the mean value of p, and σ is the standard variance of p. If we use the observed value $\hat{p}$ to calculate the standard deviation, we can claim that we have 95%[1] confidence that the error $\Delta p = \bar{p} - E(p)$ falls in the interval:

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

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

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

Algorithm 2: Multiple Step Monte Carlo Sampling

```
**Input:** The constraint $g_i = c_{i_1} \wedge c_{i_2} \wedge \ldots \wedge c_{i_m}$
**Output:** The number of assignments that satisfy $g_i$ $|K_{g_i}|$
$n$: the number of sampling times
$S_{c_i}$: the set contains input variables for $c_i$
$n_s$: the number of satisfying assignments
$N_{c_t}$: the set contains all solution for $c_t$
$r$: times of reducing $g$
$k$: the input variable
$R$: a function that produces a random point from $S_{c_i}$
$r \leftarrow 1, n \leftarrow 0$
**for** $t \leftarrow 1$ **to** $m$ **do**
    $S_{c_t} \leftarrow \pi(c_t)$
    **if** $|S_{c_t}| = 1$ **then**
        $N_{c_t} \leftarrow$ Compute all solutions of $c_i$
        $N_{c_t} = \{n_1, \ldots, n_m\},\ S_{c_t} = \{k\}$
        $g_i = g_i(k = n_1) \wedge \ldots \wedge g_i(k = n_m)$
        $r \leftarrow r + 1$
    **end**
**end**
**while** $n \leq \frac{8p}{1-p}$ **do**
    $S_{g_i} \leftarrow \pi(g_i)$
    $v \leftarrow R(S_{g_i})$
    **if** $v$ *satisfies* $g_i$ **then**
        $n_s \leftarrow n_s + 1$
    **end**
    $n \leftarrow n + 1,\ p = \frac{n_s}{n}$
**end**
$|K_{g_i}| \leftarrow n_s|K|/(n * r * \text{range(k)})$
```

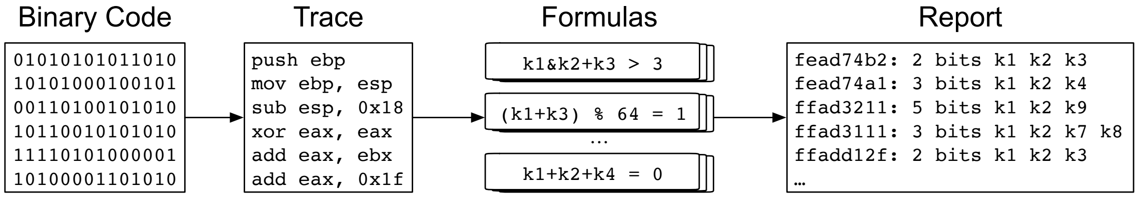
get $\frac{\Delta p}{p \ln 2} \leq 1$. Therefore, as long as

$$n \geq \frac{1.96^2(1-p)}{p(\ln 2)^2},$$

we have 95% confidence that the error of estimated leaked information is less than 1 bit. During the simulation, if n and p satisfy this inequality, the Monte Carlo simulation will terminate.

4.6 Design and Implementation

Figure 4.4 shows the three steps of Abacus. First, we run the target program with a concrete input (sensitive information) under the dynamic binary instrumentation (DBI) framework to collect an execution trace. After that, we run the symbolic execution to capture fine-grained semantic information for each secret-dependent control-flow transfer and data access. Finally, we run Monte Carlo (MC) simulation to estimate the amount of leaked information.



```
01010101011010
10101000100101
00110100101010
10110010101010
11110101000001
10100001101010
```

```
push ebp
mov ebp, esp
sub esp, 0x18
xor eax, eax
add eax, ebx
add eax, 0x1f
```

```
k1&k2+k3 > 3
(k1+k3) % 64 = 1
...
k1+k2+k4 = 0
```

```
fead74b2: 2 bits k1 k2 k3
fead74a1: 3 bits k1 k2 k4
ffad3211: 5 bits k1 k2 k9
ffad3111: 3 bits k1 k2 k7 k8
ffadd12f: 2 bits k1 k2 k3
…
```

Figure 4.4: The workflow of Abacus.

4.6.1 Execution Trace Generation

The design goal of Abacus is to estimate the information leakage as precisely as possible. We run the target binary under a dynamic binary instrumentation (DBI) tool to record execution traces and runtime information. Once the sensitive information is loaded into memory, we start to collect the trace. In this step, we mark variables and buffers that hold the sensitive data by either annotating the source code (make_abacus_symbolic) or telling the DBI tool of the memory address and the length of secrets.

4.6.2 Instruction Level Symbolic Execution

We model an attacker's observations from side-channel vulnerabilities with logic formulas. Each formula captures the fine-grained information between the input secrets and leakage sites. The engine only symbolically executes instructions that might be affected by the input sensitive data. Abacus works on one path at a time. The memory model is conceptually similar to other offline executors (e.g., SAGE [125] and the trace-based executor of BitBlaze [111]). That is, we use symbolic execution to track secrets. When secrets are loaded into the memory, Abacus starts to interpret instructions symbolically. We treat secrets as symbols (S). For other variables, we use concrete values (C) from the execution. We do not know which instruction may manipulate a secret until we execute it. For each instruction, if all its operands and implicit memory accesses are concrete values, we perform concrete calculation and update the destination with the concrete value according to the instruction semantics. Otherwise, we symbolically interpret the instruction and update the destination with a formula.

4.6.3 Leakage Estimation

We change the information leakage quantification problem into a counting problem. We propose a Monte Carlo method to estimate the number of satisfying solutions. With the help of the Central Limit Theorem (CLT), we also give an error estimate with the probability, which provides us with the *precision guarantee*.

4.6.4 Implementation

Abacus consists of 16,729 lines of code in C++17 and Python. It has three components: an Intel Pin tool that collects the execution trace, the instruction-level symbolic execution engine, and the back-end that estimates the information leakage. Table 4.3 shows the breakdown of the implementation of Abacus.

Table 4.3: Abacus' main components and sizes

Component	Lines of Code (LOC)
Trace Logging	501 lines of C++
Symbolic Execution	14,963 lines of C++
Data Flow	451 lines of C++
Monte Carlo Sampling	603 lines of C++
Others	211 lines of Python
Total	16,729 lines

Our current implementation supports most Intel 32-bit instructions that are essential to find address-based side-channel vulnerabilities, including bitwise operations, control transfer, data movement, and logic instructions. The tool uses concrete values to update the registers and memory for instructions that the implementation does not support. Therefore, the tool may miss some leakages but will not raise false positives.

4.7 Evaluation

4.7.1 Overview

In this section, we evaluate Abacus on a set of popular crypto libraries. In particular, we are interested in following aspects:

- Can Abacus precisely report the number of leaked bits in crypto libraries?
- Are the numbers of leaked bits reported by Abacus useful to justify the severity levels of the side-channel vulnerabilities?

Our testbed is comprised of several popular cryptography libraries. We choose them based on the following aspects. First, they should be widely used in many popular software. In general, it is hard to implement a secure cryptography library from scratch. So most software only use a few number of well-known cryptography libraries. Second, we choose some libraries which are well studied in the previous work. We can refer to these conclusions to verify our results. Based on the two requirements, we evaluate Abacus on the following libraries: OpenSSL, Libgcrypt, mbedTLS, and Monocypher. OpenSSL, Libgcrypt, and mebdTLS are the most widely used cryptography libraries. OpenSSL and mbedTLS have been widely studied in the previous research. Monocyper is designed to be side-channel resistant. We use the library to test if our tool has any false positives. We also test Abacus on some simple countermeasure implementations.

We write simple encryption and decryption function with the above libraries. After that, we build the source code into 32-bit binary executions. The configuration of our experiment is shown as follows.

- CPU: 2.90GHz Intel Xeon(R) E5-2690 CPU
- Memory: 128 GB
- OS: Ubuntu 18.04 LTS
- Compiler: GCC 7.5

Table 4.4 shows an overview of the evaluation results. Abacus also finds that most side-channel vulnerabilities leak very little information, which confirms our initial assumption.

Table 4.4: Evaluation results overview: Side-channel Leaks (Leaks), Secret-dependent Control-flows (CF), Secret-dependent Data-flows (DF), the number of instructions (# Instructions), Symbolic Execution (SE) and Monte Carlo (MC) time.

Name	# Leaks	# CF	# DF	# Instructions	Detection	Quantification
					ms	ms
AES[1]	68	0	68	39,855	512	1,052
AES[2]	68	0	68	39,855	520	1,057
AES[4]	75	0	75	1,704	231	9,199
AES[5]	88	0	88	1,350	36	1,924
AES[6]	88	0	88	1,350	35	1,961
AES[7]	88	0	88	1,420	36	2,161
AES[8]	88	0	88	1,586	43	1,631
DES[1]	15	0	15	4,596	58	162
DES[2]	15	0	15	4,596	57	162
DES[4]	6	0	6	2,976	163	4,677
DES[5]	8	0	8	2,593	166	6,509
DES[6]	8	0	8	2,593	165	5,975
DES[7]	8	0	8	4,260	182	5,292
DES[8]	6	0	6	8,272	229	5,152
					seconds	seconds
Chacha20[3]	0	0	0	149,353	2	0
Poly1305[3]	0	0	0	1,213,937	15	0
Argon2i[3]	0	0	0	4,595,142	37	0
Ed25519[3]	0	0	0	5,713,619	271	0
ECDSA[1]	6	6	0	4,214,946	48	31
ECDSA[2]	4	4	0	4,192,558	102	1639
ECDSA[5]	5	4	1	8,248,322	101	62
ECDSA[6]	5	4	1	8,263,599	100	58
ECDSA[7]	5	4	1	6,100,465	76	42
ECDSA[8]	0	0	0	10,244,076	121	0
ECDSA[9]	0	0	0	9,266,191	102	59
					minutes	minutes
RSA[1]	6	6	0	22,109,246	39	41
RSA[2]	12	12	0	24,484,441	44	251
RSA[4]	107	105	2	17,002,523	23	428
RSA[5]	38	27	11	14,468,307	29	436
RSA[6]	36	27	9	15,285,210	40	714
RSA[7]	31	22	9	16,390,750	34	490
RSA[8]	4	4	0	18,207,016	8	53
RSA[9]	8	8	0	18,536,796	5	780
RSA[10]	11	9	2	9,527,231	2	38
RSA[11]	14	14	0	10,513,606	14	503
RSA[12]	8	8	0	27,407,986	113	6560
Total	904	241	663	167,141,947	341m	10,232m

[1] mbedTLS 2.5 [2] mbedTLS 2.15 [3] Monocypher 3.0
[4] OpenSSL 0.9.7 [5] OpenSSL 1.0.2f [6] OpenSSL 1.0.2k [7] OpenSSL 1.1.0f [8] OpenSSL 1.1.1 [9] OpenSSL 1.1.1g
[10] Libgcrypt 1.6.1 [11] Libgcrypt 1.7.3 [12] Libgcrypt 1.8.5

However, Abacus finds some vulnerabilities with severe leakages. Prior research [107, 126] has confirmed that some of these vulnerabilities can be exploited in real attacks. With our tool, developers can distinguish non-critical "vulnerabilities" from the severe ones.

Symmetric encryption implementations in OpenSSL and mbedTLS have significant leakage due to their lookup table implementations. Abacus confirms that these leakage come from table lookups. The new implementation of OpenSSL has adopted several methods (e.g., one single S-box instead of four lookup tables, smaller lookup tables) to mitigate the problem. These changes are rather easy but significantly decrease the total amount of leaked information as the quantification result indicates.

Abacus can estimate how much information is leaked from each vulnerability. During the evaluation, for each leakage site, Abacus will stop once 1) it has 95% confidence that the error of the estimated leaked information is less than 1.0 bit, which gives the leakage quantification a *precision guarantee*, or 2) it cannot reach the termination condition after 10 minutes. In the latter case, it means Abacus cannot estimate the amount of leakage with a probabilistic guarantee. Abacus times out on around 10% reported side-channel leakage sites. We manually check these leakage sites and find most of them are quite severe. We will present the details in the subsequent sections.

4.7.2 Case Studies

4.7.2.1 Symmetric Ciphers: DES and AES

We test both DES and AES ciphers from mbedTLS and OpenSSL. Both cipher implementations apply lookup tables, which improve performance but can also introduce side-channels as well. During our evaluation, we find mbedTLS 2.5 and 2.15.1 have the same implementation of AES and DES. Therefore, our tool reports the same leakages for both versions.

We find that the DES implementations in both mbedTLS and OpenSSL have several severe information leakages in the key-schedule function. We do not see any mitigation in the new version. We think it is not seen as worth the engineering efforts given the life cycles of DES.

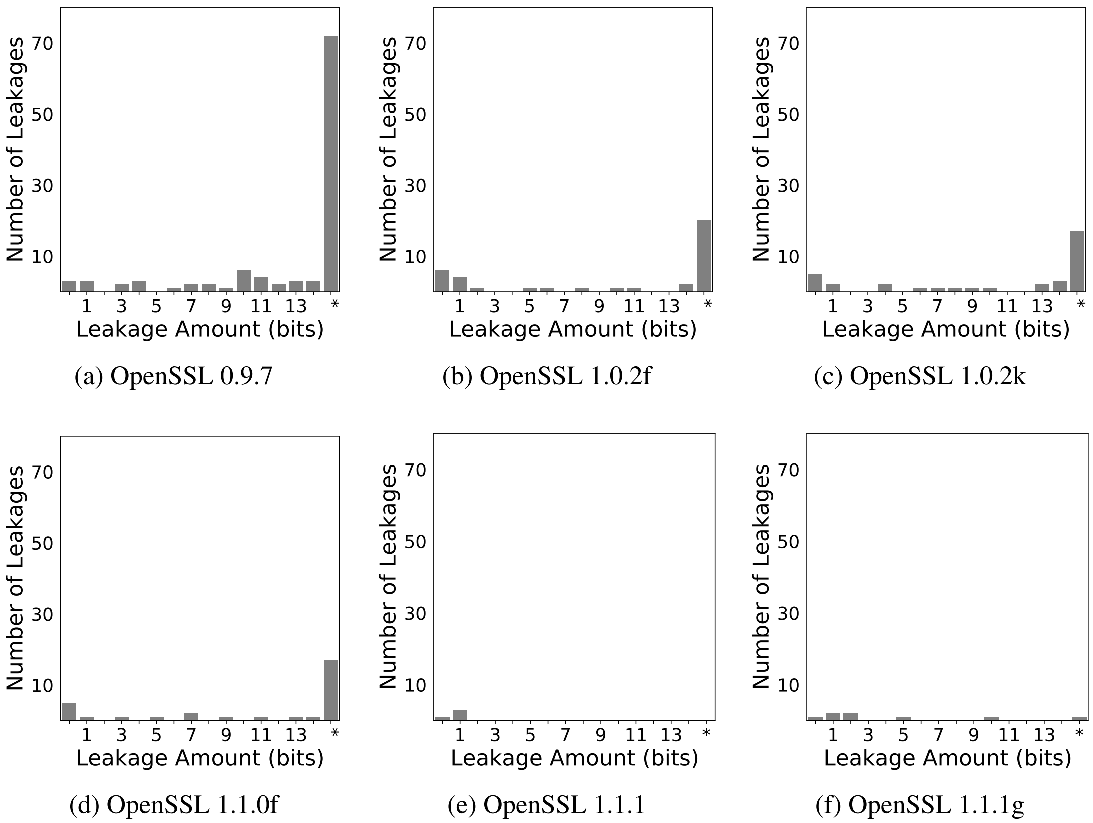
Abacus shows that the AES in OpenSSL 1.1.1 has less leakage than other versions. OpenSSL 1.1.1 uses 1KB lookup tables with 8-bit entries, unlike older versions that use a table with 32-bit entries. Our tool suggests a smaller lookup table might mitigate side-channel vulnerabilities.

During our evaluation, we find mbed TLS 2.5 and 2.15.1 have the same implementation of

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

Figure 4.5: Function *mbedtls_internal_aes_encrypt*

AES. Our tool provides the same leakage report for both versions. Abacus identifies that most leakages are in function *mbedtls_internal_aes_decrypt*. (Other leakage sites are in function *mbedtls_aes_setkey_enc*.) All leakages are caused by the secret-dependent memory accesses. Shown in Figure 4.5, there are seven leakage sites in total. Leakage 1, 2, 3 are the same and leakage 4, 5, 6, 7 are the same. They both use a pre-computed lookup table to speed up the computation. However, Abacus reports leakage 1, 2, 3 typically leak more information (7.6 - 8.1 bits) compared to leakage 4, 5, 6, 7 (4.0 bits). We check the source code and find leakage 1, 2, 3 use secret to access the lookup table *RT0, RT1, RT2, RT3*, which is 8K each. On the contrary, leakage 4, 5, 6, 7 each accesses a smaller lookup table (2K). Therefore, leakage 4, 5, 6, 7 leak less information.



(a) OpenSSL 0.9.7 (b) OpenSSL 1.0.2f (c) OpenSSL 1.0.2k

(d) OpenSSL 1.1.0f (e) OpenSSL 1.1.1 (f) OpenSSL 1.1.1g

Figure 4.6: Side-channel leakages in different implementations of RSA in OpenSSL. We round the number of leaked information into the nearest integer. The mark $*$ means timeout.

4.7.2.2 Asymmetric Ciphers: RSA

We also evaluate Abacus on RSA. As shown in Figure 4.6, the result indicates that the newer versions of OpenSSL leak less information than the earlier versions. After version 0.9.7g, OpenSSL adopts a fixed-window mod_exp_mont implementation for RSA. With this design, the sequence of squares and multiples and the memory access patterns are independent of the secret key. Abacus's result confirms the new exponentiation implementation has mitigated most leakages effectively because the four newer versions have fewer leakages than version 0.9.7. OpenSSL version 1.0.2f, 1.0.2k, and 1.1.0f almost have the same amount of leakage. We check the ChangeLog and find only one change to patch vulnerability CVE-2016-0702. Abacus finds OpenSSL 1.1.1 and 1.1.1g have significantly less leaked information than other versions. We check the ChangeLog of these two versions and find a claim that the new RSA implementation adopts "numerous side-channel attack mitigation", which proves the

```
1 # define mul_add_c2(a,b,c0,c1,c2)                        \
2     t=(BN_ULLONG)a*b;                                    \
3     tt=(t+t)&BN_MASK;                                    \
4     if (tt < t) c2++;                                    \
5     t1=(BN_ULONG)Lw(tt);                                 \
6     t2=(BN_ULONG)Hw(tt);                                 \
7     c0=(c0+t1)&BN_MASK2;                                 \
8     if ((c0 < t1) && (((++t2)&BN_MASK2) == 0)) c2++; \
9     c1=(c1+t2)&BN_MASK2; if ((c1) < t2) c2++;
```

Figure 4.7: Macro sqr_add_c2 in OpenSSL 0.9.7

effectiveness of our quantifying method.

Our quantification result shows vulnerabilities that leak significant amounts of information are more likely to be fixed in the updated version. As presented in Figure 4.6, OpenSSL 0.9.7 has several severe leaks from function bn_sqr_comba8, which is a main component of the OpenSSL big number implementation. Shown in Figure 4.7, it has a secret-dependent control flow at line 8. The value of the function parameter a is derived from the secret key. As function bn_sqr_comba8 calls the macro (sqr_add_c2) multiple times, and the code can leak some information each time. Abacus indicates the vulnerability is quite severe. It was patched in OpenSSL 1.1.1. In Figure 4.8, control-flows transfers are replaced so there are no leaks in the function sqr_add_c2 in OpenSSL 1.1.1. We note that line 4 and 9 in Figure 4.7 both contain `if` branches. However, they are not leaks because most compilers use *add with carry* instruction to eliminate the branch. In addition, branches can be compiled into non-branch machine instructions with conditional moves. We notice a bitwise operation in Libgcrypt 1.8.5 is compiled to a conditional jump, which leads to a side-channel leakage. Therefore, source-level code reviews are not accurate enough to detect side-channels.

For vulnerabilities that leak less amount of information, developers are more reluctant to fix them or fixing them unintentionally. For example, OpenSSL 0.9.7 adds a fixed windows version of function BN_mod_exp_mont_consttime to replace the original function BN_mod_exp_mont. Abacus detects a minor vulnerability in the original function that can leak the last bit of the big number `m`. In the updated version, developers make the fixed windows the default option and rewrite most of the function. However, the leakage site still exists in OpenSSL 1.1.1.

To evaluate the effectiveness of the leaked bits reported by Abacus, we conduct a case

```
# define mul_add_c2(a,b,c0,c1,c2)      do { \
    BN_ULONG ta = (a), tb = (b);            \
    BN_ULONG lo, hi, tt;                    \
    BN_UMULT_LOHI(lo,hi,ta,tb);             \
    c0 += lo; tt = hi+((c0<lo)?1:0);        \
    c1 += tt; c2 += (c1<tt)?1:0;            \
    c0 += lo; hi += (c0<lo)?1:0;            \
    c1 += hi; c2 += (c1<hi)?1:0;            \
    } while(0)
```

Figure 4.8: Macro sqr_add_c2 in OpenSSL 1.1.1

study of all the leaked sites identified by Abacus in OpenSSL 1.1.0f. Table 4.5 shows the result. First, all leakage sites that leak more than 5 bits information are fixed by developers. Second, for all leakages that are not patched by developers, the quantification result shows that they leak less than 1.0 bits information. The evaluation results show that Abacus can help developers find severe leakages automatically.

4.7.3 Benchmarks

We also tested Abacus on a set of small benchmarks.

Bit-slicing

Bit-slicing is an efficient method to construct constant-time implementations to mitigate side-channels. The basic concept is to implement a function in terms of single-bit logical gate operations, such as AND, XOR, OR, and NOT.

Since the table look-ups and conditional jumps are replaced with single-bit logical gates, with no secret-dependent memory addresses or control flow, both the data access and control flow types of side-channel leakages are mitigated.

We test Abacus on bit-slicing. We adopt the SBOX implementations, commonly used in block ciphers such as DES and AES, with and without bit-slicing, and apply Abacus to confirm the mitigation. The SBOX implementation with and without bit-slicing are shown in Figure 4.10 and Figure 4.9, respectively. Considering an SBOX take some bits derived from a password as input and output 2-bit transform result, the plain implementation has a range check on the password input and a secret-dependent table lookup while bit-slicing does not.

Table 4.5: Leaked Functions in RSA implemented by OpenSSL 1.1.0f. According to Abacus [1], the mark “$*$” means timeout, which indicates more severe leakages. 0.0 means very small amount of leakage, but not exactly zero.

File	Line Number	Vulnerable Function	Result (bits)	Fixed
bn_lib.c	143	BN_num_bits_word	*	✓
bn_lib.c	144	BN_num_bits_word	*	✓
bn_lib.c	145	BN_num_bits_word	17.2	✓
bn_lib.c	1029	bn_correct_top	*	✓
bn_lib.c	639	BN_ucmp	*	✓
ct_b64.c	164	__udivdi3	5.9	✓
bn_div.c	330	BN_div	*	✓
bn_gcd.c	192	int_bn_mod_inverse	1.0	✗
bn_gcd.c	215	int_bn_mod_inverse	7.9	✓
bn_gcd.c	237	int_bn_mod_inverse	8.2	✓
bn_gcd.c	218	int_bn_mod_inverse	14.9	✓
bn_gcd.c	240	int_bn_mod_inverse	9.2	✓
bn_lib.c	147	BN_num_bits_word	*	✓
bn_lib.c	152	BN_num_bits_word	12.6	✓
bn_lib.c	153	BN_num_bits_word	*	✓
bn_lib.c	156	BN_num_bits_word	*	✓
bn_div.c	384	BN_div	17.2	✓
bn_div.c	330	BN_div	11.9	✓
bn_div.c	334	BN_div	3.8	✓
bn_exp.c	622	BN_mod_exp_mont_consttime	1.0	✗
bn_exp.c	741	BN_mod_exp_mont_consttime	1.0	✗
bn_mont.c	138	BN_from_montgomery_word	*	✓
bn_mont.c	139	BN_from_montgomery_word	*	✓
bn_mont.c	140	BN_from_montgomery_word	*	✓
bn_mont.c	142	BN_from_montgomery_word	*	✓
bn_mont.c	152	BN_from_montgomery_word	*	✓
bn_asm.c	733	bn_sqr_comba8	*	✓
bn_asm.c	592	bn_mul_comba8	*	✓
bn_mont.c	98	BN_from_montgomery_word	0.0	✗
bn_div.c	330	BN_div	0.3	✗
bn_div.c	330	BN_div	0.3	✓

Abacus reports that there are control-flow and data access types of leakages in the non-bit-slicing implementation (line 6 and line 7 in Figure 4.9). The number of leaked bits is 5.0 and 4.4, respectively. At line 6, according to Definition 1, the input set K is $[0, 2^8 - 1]$, the observed input set K^o is $[0, 2^3 - 1]$. Thus, the leakage $L_{\beta(k)\to o}$ based on the observation (o)

```
uint8_t password = input();

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

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

Figure 4.9: SBOX without bitslicing

```
uint8_t password = input();
a = *password & 0b001;
b = (*password & 0b010) >> 1;
c = (*password & 0b100) >> 2;
na = ~a & 1;
nb = ~b & 1;
nc = ~c & 1;
t0 = (b & nc);
t1 = (b | nc);
l = (a & nb) | t0;
r = (na & t1) | t0;
ret = l << 1 + r;
```

Figure 4.10: SBOX with bitslicing

is $L_{\beta(k) \to o} = \log_2 |K| - \log_2 |K^o| = 8 - 3 = 5$ bit, which confirms the result from Abacus. Similarly, we can verify the result of other leakage sites. Abacus reports no leakage on the bit-slicing implementation.

Lookup Table

```
static const uint8_t T[1024] = {
      0x63U, 0x7cU, 0x77U, 0x7bU, 0xf2U, 0x6bU, 0x6fU, 0xc5U,
      0x30U, 0x01U, 0x67U, 0x2bU, 0xfeU, 0xd7U, 0xabU, 0x76U,
...
output = (T[(key[0]>>24)] << 24) ^
         (T[(key[1]>>16) & 0xff] << 16) ^
         (T[(key[2]>>8) & 0xff] << 8) ^
         (T[(key[3]) & 0xff]);
```

Figure 4.11: Lookup tables with small entries.

```
static const uint32_t T[256] = {
    0xc66363a5U, 0xf87c7c84U, 0xee777799U, 0xf67b7b8dU,
    0xfff2f20dU, 0xd66b6bbdU, 0xde6f6fb1U, 0x91c5c554U,
    0x60303050U, 0x02010103U, 0xce6767a9U, 0x562b2b7dU,
    0xe7fefe19U, 0xb5d7d762U, 0x4dababe6U, 0xec76769aU};
...
output = (T[(key[0]>>24)] << 24) ^
         (T[(key[1]>>16) & 0xff] << 16) ^
         (T[(key[2]>>8) & 0xff] << 8) ^
         (T[(key[3]) & 0xff]);
```

Figure 4.12: Lookup tables with big entries.

Considering the cache-collision timing attack [126], the probability of leakage decreases when the lookup table entry or element size gets smaller. We analyze the table lookup with Abacus. It is a common operation used by symmetric ciphers such as AES. Figure 4.12 is from the original AES reference implementation. It uses a lookup table and each entry is 4 bytes, which is vulnerable to side-channel attacks. Most cryptography libraries adopted the mitigated version, shown in Figure 4.11. Abacus reports three leakage sites for each lookup table, with 4.0 bits for the table with smaller entries and 2.0 bits for the table with bigger entries, confirming the theory. The evaluation result shows the version with smaller lookup tables leaks less information, which confirm the effectiveness of the mitigation technique.

Password Checker

```
bool pwcheck(uint8_t *key, uint8_t *pub) {
  for (int i = 0; i < 2; ++i) {
    if (key[i] != pub[i]) {
      return false;
    }
  }
  return true;
}
```

Figure 4.13: A vulnerable password checker.

We illustrate the quantification method on two password checking examples. As shown in Figure 4.13, the version is vulnerable to side-channel attacks because of the early return at

```
bool pwcheck(uint8_t *key, uint8_t *pub) {
  bool matched = true;
  for (int i = 0; i < 2; ++i) {
    if (pub[i] != key[i])
      matched = false;
  }
  return matched;
}
```

Figure 4.14: A safe password checker.

line 4. If the first bytes of key and pub are the same, the function will continue running and access the second byte. Figure 4.14 fixes the vulnerability because the function will continue comparing the second byte even if the first byte is different. Abacus identifies the leakage site in Figure 4.13 successfully, and estimates the amount of leakage is 16.0 bits.

Chapter 5
Precise Analysis on Multiple-trace Attacks

5.1 Introduction

Side-channel attacks allow attackers to infer sensitive information based on the non-functional behaviors of the computer system. Examples of these non-functional behaviors include acoustics, CPU usages, timing, and EM signals [10, 43, 127, 128]. In general, if a program has different execution behaviors when it processes various keys, an attacker can infer nontrivial information by observing secret-dependent behaviors. While separate side-channel attacks may have different forms and patterns, a large portion of them share the same fundamental reasons. That is, the memory access pattern is dependent on the original sensitive input.

While removing secret-dependent memory-access patterns in existing software completely is hard and tedious, fixing some of the most dangerous leakages appears to be feasible. Previous studies [1, 13, 14, 20, 22, 23, 23, 61–63] on side-channel detection have shown their effectiveness in finding such code patterns. Developers can run a tool to analyze the side-channels leakages and fix these patterns. However, fixing all these leakages seems not feasible in practice. Tremendous efforts have been made to remove the side-channel vulnerabilities in cryptography libraries in the past decade. However, there are still several side-channel leakages in the recent versions of OpenSSL [129]. Besides, many side-channel vulnerable versions usually have better performance, such as the T-table lookups in AES, CRT optimizations in RSA, and Fast Fourier Transform (FFT) in many media processing libraries. For many non-cryptography developers, side-channels are not in their threat model. However, recent studies [31, 34, 130] have shown a series of attacks on the non-cryptography libraries.

Even a minor leakage can be severe for cryptography libraries because the leakage can reduce the encryption algorithm's strength. However, some attacks also exploit non-cryptography libraries and applications such as machine learning applications [44,131], graphic libraries [45], and spell checking tools [31]. Unlike side-channel attacks on cryptography libraries that exploit one vulnerability at a time, attacks on non-cryptography rely on multiple side-channel leakage sites to retrieve more information. A program usually has more than two leakage sites. However, no existing tools that can estimate the total effect of multiple leakage sites. Are they independent or have the same root cause? For real-world applications with thousands of lines, attacker can exploit more than one leakage sites to retrieve more information. Attackers can guess the values of many temporary values. These temporary values may contain some knowledge of the original secret buffer.

Many side-channel leakage detection tools rely on techniques such as symbolic executions, taint analysis, and abstract interpretations [13,21–23]. Due to the limitations of existing binary analysis techniques, it is hard to apply existing methods to analyze side-channels in non-cryptography libraries. For many other applications like machine learning, media, and graphic libraries built on the top architecture-dependent libraries (e.g., Intel MKL [132]), these tools are not able to handle them. Moreover, some binary analysis frameworks (e.g., Angr [102]) are limited in analyzing floating-point instructions. However, the functionality of these applications heavily depends on floating-point calculations.

While many existing works can detect side-channel leakages, few of them can assess the severity levels of these vulnerabilities [1, 13, 14, 20, 22, 23, 23, 53, 61–63, 133, 134]. Some existing tools [23,82,84,92,122,135,136] quantify the information leakage by giving an upper bound estimation of each leakage site. This is helpful to justify that the software is secure if the reported leakage is zero. However, these tools cannot report to software developers how severe each leakage site would be because of the over-approximation they apply when they estimate the amount of leakage. For example, CacheAudit [23] estimates that a 128-bit AES encryption can leak more than 128 bits. As a result, even these tools report that some leakages can leak lots of information, it does not mean the program has truly severe side-channel vulnerabilities.

To solve the above problem, we propose a method to estimate the effect of side-channel leakages automatically. We treat the side-channel attack as a communication system and use the channel capacity to quantify the amount of the information flow from the original secret to an attacker's observation. The method also allows us to combine multiple leakages sites to estimate the overall effect. The attack can be seen as a process that reduces the search space of the original sensitive inputs. We use the side-channel vulnerability to divide the input space. If the patterns can uniquely distinguish the input space, then we think the information is leaked totally. However, in real cases, many of these side-channel leakage patterns are not sufficient to distinguish each leakage site but can still discern some secret inputs. In these cases, we call them partial leakages.

The method consists of three phases. In the first phase, we fuzz the target program with various inputs and collect the memory access information. In the second phase, we map the memory access information with the source code. With the debug information, we can precisely know the address access information of the source code. In the third phase, we examine the source code. If one function has two memory access patterns, then the function is vulnerable to side-channel attacks. Based on the distribution of memory access patterns under different inputs, we quantify each function's side-channel leakage.

In this work, we make the following contributions:

- We propose a method that automatically detects and analyzes the effect of each side-channel leakage site. Our analysis combines information from both the source code and the runtime execution, which makes the method more effective in finding leakages. The technique can combine multiple leakage sites to retrieve more information as well.

- We analyze the amount of information leakage from side-channel vulnerabilities based on the channel capacity. The information leakage our tool reports for each leakage site is quite precise and thus can reflect the severity level of the leakage.

- We implement the above method in a tool called Quincunx and evaluate the tool with several benchmarks and several real-world applications such as OpenSSL, mbedTLS, TinyDNN, and GTK. The results show that our tool can detect and quantify the side-channel leakages effectively. Moreover, the leakage reports given by the Quincunx can help developers fix the reported vulnerabilities.

5.2 Background and Threat Model

In this chapter, we treat the side-channel attack as a communication system between two parties. Here we first give an overview of the background knowledge of information theory.

5.2.1 Channel Capacity

Given a discrete random variable K, k is one of the possible values occurring with the probability $p(k)$. The Shannon entropy $H(K)$ is defined as

$$H(K) = -\sum_{k \in K} p(k) \log_2 p(k). \tag{5.1}$$

Shannon entropy can be used to quantify the initial uncertainty about sensitive information. It measures the amount of information in a system. For example, an AES encryption program takes a 128-bit key as the input. Suppose each key has the equal opportunity and $p_k = 1/2^{128}$, then the encryption system has 128 $bits$ information according to Shannon entropy.

For a deterministic program, the program should always have the same memory access patterns under the same input. Therefore, an address-based side-channel attack can be seen as a communication channel between the memory access patterns and secret inputs. In this chapter, we use the channel capacity to describe the amount of information leakages through a channel. In information theory, the channel capacity is used to quantify the maximum rate of information can be reliably transmitted over a channel. If we use K to represent the input secrets and O to represent the output observation. The channel capacity (C) is defined as

$$C(K;O) = \max_{p(x)} I(K;O). \tag{5.2}$$

Here $I(K;O)$ is the mutual information between K and O.

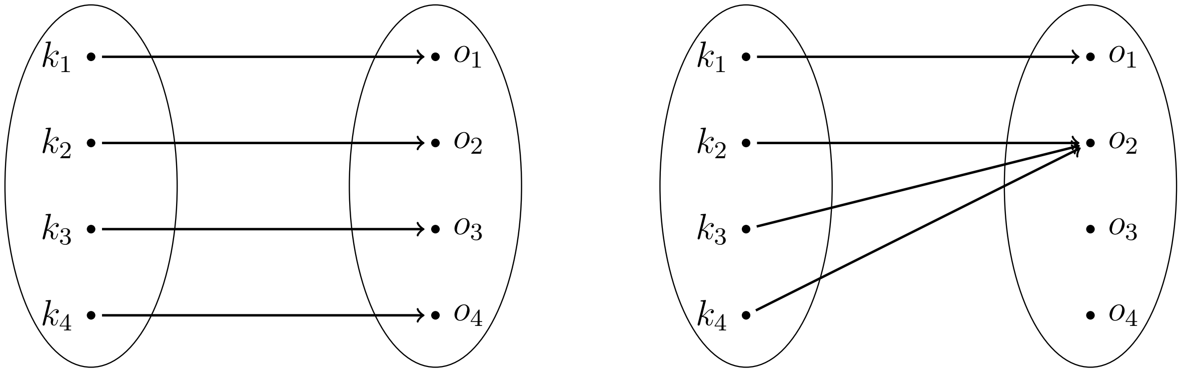
$$I(K;O) = \sum_{k \in K} \sum_{o \in O} p(k,o) \log_2 \frac{p(k,o)}{p(k)p(o)} \tag{5.3}$$

While the channel capacity in general situations is hard to compute, we consider two special channels.

5.2.1.1 Noiseless Lossless Channel

An information channel is noiseless and lossless if every k corresponds to exactly one unique o. The channel capacity of a noiseless lossless channel equals

$$C(K;O) = \max_{p(k)} I(K;O) = \max_{p(k)} H|O| = \max_{p(k)} H|K|, \tag{5.4}$$



(a) Noiseless Lossless

(b) Noiseless Loss

Figure 5.1: Two kinds of information channels

where $|K|$ represents the number of possible variables in K.

Figure 5.1(a) shows an example of the type of the channel. Under the circumstance, we always have $P(o_i/k_i) = 1$ and $P(k_i/o_i) = 1$. As we can see, the channel has the unique output ($o \in O$) for every input ($k \in K$).

5.2.1.2 Noiseless Loss Channel

In reality, noiseless loss channels are more common. The channel capacity of a noiseless loss channel equals

$$C(K;O) = \max_{p(x)} I(K;O) = \max_{p(x)} H|O|. \tag{5.5}$$

Figure 5.1(b) shows an example of this type of channels. Once the input value is determined, the output value is also determined. Shown in Figure 5.1(b), the channel has the unique output ($o \in O$) for every input ($k \in K$).

5.2.2 Threat Model and Notation

We consider an access-based attack [26], where an adversary can probe one time for each run only after the termination of a deterministic program. The attacker may not know the memory accesses of the victim program directly. Still, he can infer whether some memory addresses are accessed or not at various granularities after *each run* of the victim program. We call the memory access information as the observation (O) in the rest of the chapter. It is a typical scenario for most side-channel attacks. We also assume that the attacker can have access to the source code of the victim program (β). It is a valid assumption since most side-channel attacks target open-source software. Similar to recent work, we assume the attacker has a noise-free observation. The program (β) has K as the sensitive input, which is a finite set consisted of every possible key ($k \in K$). The program also takes known messages M as the input.

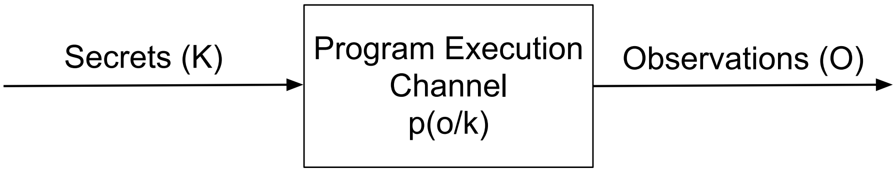


Figure 5.2: The relationship between the side-channel attack and the channel capacity. For a deterministic program, the side-channel attack can be seen as a communication channel between the observation ($o \in O$) and the secret ($k \in K$).

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

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

Side-channel attacks can be seen as a process that infers the secret (K) based on the observations (O). As shown in Figure 5.2, we assume that an attacker can observe the memory addresses of the victim program. He wants to infer the original secret (k) based on the observation. We also assume the attacker knows the known message $m \in M$. In the rest of the chapter, we use $\beta(K) \rightarrow O$ to denote a victim program that may have side-channel leakage sites.

5.3 Method

In this section, we first provide two examples that show how we rank the side-channel vulnerability. After that, we prove that our method is a conservative estimate of the amount of each leakage.

5.3.1 An Example

Figure 5.3 shows a function that is vulnerable to side-channel attacks. The function takes an input secret from the caller. Depending on the value of the secret, it may access different values at line 7. Suppose an attacker can observe which item in the table T is accessed. He can infer the secret based on the observation.

```
1 void bar(uint8_t secret)
2 {
3   uint8_t T[128];
4   uint8_t index = 0, t;
5   int i;
6   index = (index+secret)%128;
7   t = T[index];
8   ...
9 }
```

S \ T	0	1	2	3	4	5	6	7	...
0	A								
1		A							
2			A						
3				A					
4					A				
5						A			
6							A		
7								A	

Figure 5.3: A severe side-channel leakage.

To assess the severity level of the vulnerability, we use the sampling method to estimate the leakage. Suppose we give the function input from $0, 1, \ldots, 7$ and observe the array T, we can find each different input has one unique observation. Therefore, the secret can be uniquely determined.

```
1 void bar(uint8_t secret)
2 {
3   uint8_t T[128];
4   uint8_t index = 0, t;
5   int i;
6   index = (index+secret)%128;
7   t = T[index % 4];
8   ...
9 }
```

S \ T	0	1	2	3	4	5	6	7	...
0	A								
1		A							
2			A						
3				A					
4	A								
5		A							
6			A						
7				A					

Figure 5.4: A minor side-channel leakage.

On the other hand, the second example also has side-channel leakages at line 7. However, the leakage is slightly different. Still, we test the program with input from $0, 1, \ldots, 7$. However, we find the attacker cannot uniquely determine the secret based on the observation of array T, for example. Both 0 and 4 read the first item of the table. Therefore, the vulnerability is less severe than the previous one.

Let us formally define the amount of leaked information based on the above analysis.

Definition 4. *Suppose we have a program β with the input space K and observation set O. We randomly select $k_1, k_2, \ldots, k_n \in K$ as the input and get observations $o_1, o_2, \ldots, o_n \in O$.*

Assume $K' = \{k_1, k_2, \dots, k_n\}$ ($K' \subseteq K$) and $O' = \{o_1, o_2, \dots, o_n\}$ ($O' \subseteq O$). We denote it as

$$\beta(K') \to O'.$$

The amount of leaked information $L_{\beta(K')\to O'}$ based on the observations (O') is

$$L_{\beta(K')\to O'} = H(O') = -\sum_{o\in O'} p(o)\log_2(p(o)).$$

With Definition 4, we can calculate the information leakage in Figure 5.3 and Figure 5.4, respectively. In Figure 5.3, we have 8 kinds of observations and each observation is uniformly distributed. Therefore, the information leakage is $\log_2 8 = 3$ bits. In Figure 5.4, we have 4 kinds of observations and each observation is also uniformly distributed. Therefore, the information leakage is $\log_2 4 = 2$ bits. We test the program with 8 inputs. Therefore, the Shannon entropy of the original K $H(K)$ is also $3\,\mathrm{bits}$, according to Equation 5.1. Therefore, the leakage in Figure 5.3 shows the input secret is leaked totally.

5.3.2 A Conservative Estimation

In this section, we prove Definition 4 is a conservative estimation of the amount of leaked information based on the channel capacity.

Theorem 2. *If an attacker launches a side-channel attack on a deterministic program, then the channel between the secret and the observation is a noiseless loss channel.*

Proof 1. *According to information theory,*

$$\begin{aligned} I(K;O) &= H(O) - H(O|K) \\ &= H(O) - \sum_{k\in K} p(o|k)\log_2 p(o|k) \end{aligned}$$

For a deterministic program, $p(o|k) = 1$. As a result, $H(O|K) = 0$.

$$C(K;O) = \max_{p(k)} I(K;O) = \max_{p(k)} H|O|$$

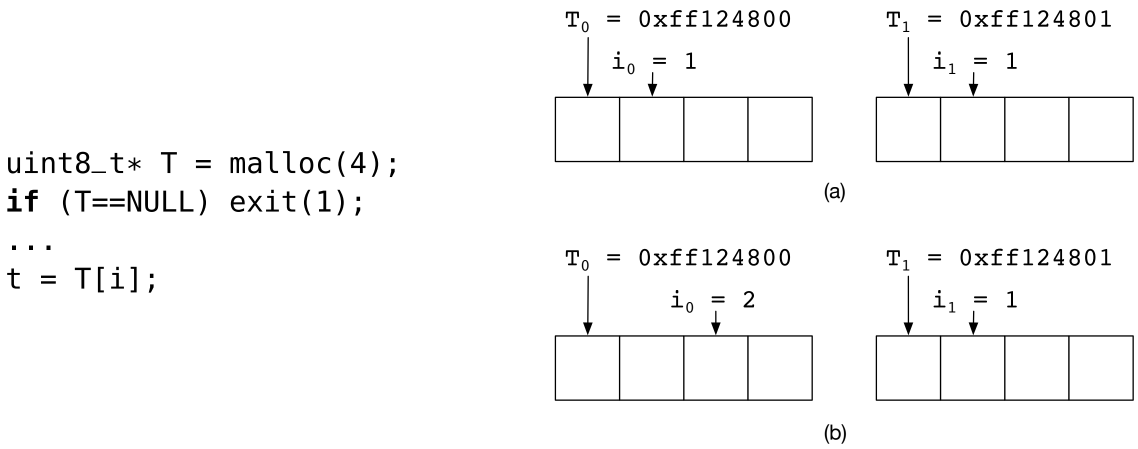
With Theorem 2, we can have

$$\max_{p(k)} H|O| >= \max_{p(k')} H|O'| >= H|O'|$$

Accordingly, if Definition 4 says the vulnerability leakage has m bits of leakages, then the vulnerability leaks at least m bits of information. We can use the definition to detect these severe leakages. In the examples of Figure 5.3 and Figure 5.4, the secret input ranges from 0 to 255. We can calculate the precise value of the amount of leaked by iterating each input keys. In Figure 5.3, each item in the array could be read. Therefore, the channel capacity in Figure 5.3 is $\log_2 128 = 7\,\mathrm{bits}$. In Figure 5.4, there could be 4 kinds of information, so the information leakages defined by channel capacity is $\log_2 4 = 2\,\mathrm{bits}$. Compared with the result from Definition 4, we can see that the amount of leakages by Definition 4 is a conservative estimation.

5.4 Align Address Access Across Multiple Executions

We compare the address accesses of different executions generated by different inputs to quantify the amount of leakage information. A meaningful comparison requires that we align executions across multiple runs before we quantify the leakage. In this work, we combine the information from both the source code and the binary executable to align the memory access during the execution.



```
uint8_t* T = malloc(4);
if (T==NULL) exit(1);
...
t = T[i];
```

Figure 5.5: Without proper address alignment, there could be many false positives and false negatives.

Considering the example in Figure 5.5, it read data from the array `T` based on the index (i). If the value of index (i) is associated with the secret information, the example is vulnerable to side-channel attacks. Unlike the example in Section 5.3, the array `T` is on the heap. Data on the heap are often allocated at different addresses each time. Suppose during the first run, `T` is allocated at the address `0xff124800`, and `T` is allocated at the address `0xff124801` during the second run. For the example in Figure 5.5(a), the index that is used to access the array is always 1, so it should not be a side-channel vulnerability here. However, the base addresses are different (`0xff124800` and `0xff124801`) so we get two different memory accesses. Under the circumstance, it is a false positive. Without proper alignment, it can cause false negatives, as the example shown in Figure 5.5(b). In the example, different keys lead to different indexes. However, as $T_0 + i_0$ equals to $T_1 + i_0$, Quincunx cannot observe the difference here, which is a false positive.

Our Strategy

We assign a value for each memory location [137]. The value should satisfy the following characteristics.

- **Uniqueness.** At any point during the execution, different memory cells must have a different value.
- **Alignment.** The same memory cell across multiple executions should share the same value.

There are two types of possible solutions to assign the value.

Runtime Information. One example here is the concrete memory address during execution. Such a value satisfies the uniqueness characteristic. However, as the example shown in Figure 5.5, it does not have the alignment property.

Source Code Information. Alternatively, we can use the information from the source code. Recent work [137] proposes using the execution point that allocates the memory cell as the index and tracking the pointer arithmetic to track the pointer that points to the memory. However, such a method may miss aliased symbols that point to the same memory cell.

We propose to combine both the information from the source code and the runtime to align the memory access during the execution. At the point the memory is allocated, we use the tuple `(start address, length)` to represent the buffer. In the following execution, we check if any memory access falls into the range of these tuples. If so, we add the memory access into the tuple. After the execution, the tuple can be mapped into the location of the source code, and we only compare memory accesses that belong to the same lines in the source code.

5.5 Multiple Leakages Sites

Real-world software can have many side-channel vulnerabilities. These vulnerabilities may spread throughout the whole program. An adversary can exploit more than one side-channel vulnerability to gain more information [19, 34]. In order to quantify the total information leakage precisely, we need to know the relation of these leakage sites.

```
 1 void function1(uint8_t* secret)
 2 {
 3   uint8_t k1, k2;
 4   k1 = secret[1];
 5   k2 = secret[2];
 6   if (k1 > k2)
 7     a();
 8   if (k1 + k2 < 256)
 9     b();
10   ...
11 }
```

(a) Independent Leakages

```
 1 void function2(uint8_t* secret)
 2 {
 3   uint8_t k1, k2;
 4   k1 = secret[1];
 5   k2 = secret[2];
 6   if (k1 + k2 > 128)
 7     c();
 8   if (k1 > 32)
 9     d();
10   ...
11 }
```

(b) Dependent Leakages

Figure 5.6: Multiple leakage sites. The two leakage sites in Figure (a) leak different information. But the leaked information in Figure (b) has some overlaps.

Considering the examples in Figure 5.6, Figure 5.6(a) show a code snippet with two leakage sites at line 6 and line 8, respectively. For example, if an adversary observes that function a executes, then the attacker can infer that the first byte in buffer secret is larger than the second byte in buffer secret. Similarly, if an adversary observes that function b executes, then the attacker can infer that the sum of the first two bytes in buffer secret is smaller than 256. The interesting part here is that if the attacker observes that function a executes, then he still has zero knowledge about the function b. The two leakage sites in Figure 5.6 (b) are different. If an attacker observes function c executes, then it is likely the attacker can observe that function d also executes.

Suppose a program has two side-channel vulnerabilities A and B, which can leak L_A and L_B bits, respectively, according to Definition 4. For the example in Figure 5.6, we use a to denote that the function a executes, and $\neg a$ to denote that the function a does not execute during the execution. Depending on the relation between A and B, the total leaked information L_{AB} can be calculated as follows.

5.5.1 Independent Leakages

Given a leakage site A, the site has different memory access patterns under different secrets. We call each memory access pattern as an observation. If any observation of A does not affect the probability of occurrence of any observation of B, we call the two leakage sites A and B are independent, $p_{ab} = p_a p_b$. If A and B are two independent leakages, the total amount of leaked information is:

$$
\begin{aligned}
L_{AB} &= -\sum p_{ab} \log_2 p_{ab} = -\sum p_a p_b \log_2 p_a p_b \\
&= -\sum p_a \log_2 p_a - \sum p_b \log_2 p_b \\
&= L_A + L_B
\end{aligned}
$$

5.5.2 Dependent Leakages

If A and B are dependent leakages, the total information leakage will be:

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

We use the chi-squared test to check if two leakage sites are independent. A chi-squared test can determine if there is a relationship between two categorical variables. Suppose we can summarize the data in a $r \cdot c$ two-way contingency table. The chi-squared test statistic and the degrees of freedom v are calculated with the formula below.

$$\tilde{\chi}^2 = \sum_{k=1}^{rc} \frac{(O_k - E_k)^2}{E_k}, \qquad v = (r-1)(c-1)$$

Here O is the observed value and E is the expected value. A low value means that there is a high correlation between the two variables.

The null hypothesis is that the two leakages are independent. We generate random inputs as the secret input and record the memory access for each execution. Each time, the execution is independent of the previous execution, and the target program is the same as the program executed in the previous experiment. For one type of event (e.g., function a executes), we use p to denote the probability of the event. Suppose the target program is executed n times, the probability that we observe the event happens $k\,(k < n)$ times is $P = \binom{k}{n} p^n (1-p)^{n-k}$. Therefore, the times of the same observation for each event should satisfy the binomial distribution. After that, we can use the chi-squared test to examine whether the two leakage sites are independent.

	a	$\neg a$	total
b	255 (255.5)	253 (252.5)	508
$\neg b$	260 (259.5)	256 (256.5)	516
total	515	509	1,024

(a)

	c	$\neg c$	total
d	809 (764.8)	82 (126.2)	891
$\neg d$	70 (114.2)	63 (18.8)	133
total	879	145	1,024

(b)

Figure 5.7: The contingency table for the experiments. The numbers in the parenthesis are expected values.

Assume we generate 1024 random inputs and record the memory access each time. After calculating the number of the same observation, we get the contingency table shown in Figure 5.7. Suppose the observations of the execution of function a and the execution of function b are independent ($p_{ab} = p_a \cdot p_b$), we calculate the expected value for the observation ab based on the frequency.

$$E_{ab} = 1024 \cdot p_{ab} = 1024 \cdot p_a \cdot p_b = 1024 \cdot \frac{515}{1024} \cdot \frac{508}{1024} = 255.5$$

Similarly, we calculate the rest expected values with the hypothesis that the leakages are independent and fill in them in Figure 5.7. The degree of freedom v is $(2-1) \cdot (2-1) = 1$.

Based on the equation, we calculate the chi-squared test statistic for two leakage sites in Figure 5.6 (a) and (b).

$$\tilde{\chi}_a^2 = \frac{(255.5 - 255)^2}{255.5} + \frac{(252.5 - 253)^2}{252.5} + \frac{(259.5 - 260)^2}{259.5} + \frac{(256.5 - 256)^2}{256.5} = 0.004$$

$$\tilde{\chi}_b^2 = \frac{(764.8-809)^2}{764.8} + \frac{(126.2-82)^2}{126.2} + \frac{(114.2-70)^2}{114.2} + \frac{(18.8-63)^2}{18.8} = 139.1$$

According to chi-squared distribution table, we have the probability $p \approx 0.95$ to accept the null hypothesis for the two leakages in Figure 5.6 (a). We have the probability $p \approx 0.00$ to accept the null hypothesis for the two leakages in Figure 5.6 (b). Therefore, we infer that the two leakages in Figure 5.6 (a) are likely to independent. And the two leakages Figure 5.6 (b) are dependent. In this chapter, for any two leakages, if $p < 0.03$, we think the two leakage sites are independent. We use the above method to test if any two leakage sites are independent for each leakage point. If any two leakages are independent, we combine them into one leakage and calculate the total amount of the leaked information.

5.6 Design

In this section, we explain the design of Quincunx. We first present the overview of Quincunx. After that, we discuss some design details to implement Quincunx.

5.6.1 Overview

Figure 5.8 shows an overview of our side-channel leakages quantification tool. Given the source code of a victim program, Quincunx can find the address-based side-channel leakages by fuzzing the victim program. To assist the illustration, we divide the workflow of Quincunx into three steps. For the first step, we build the target program from the source code. After that, we give them random inputs to the victim program and run the target program and record each of the addresses in the binary file is hit or not, and store the information in a bitmap. Next, we analyze the side-channel leakage from the address access bitmap. We split the address space into several segments to speed up the comparison according to the meta information. Finally, Quincunx generates the final leakage report. For each leakage site, Quincunx gives a conservative estimation of the information leakage. Every leakage site in the report is a true leakage, and there are no false positives.

5.6.2 Step 1: Fuzzing

After we build the software into a binary executable, we give the binary executable random inputs. For cryptography libraries, the input is the encryption key. For non-cryptography

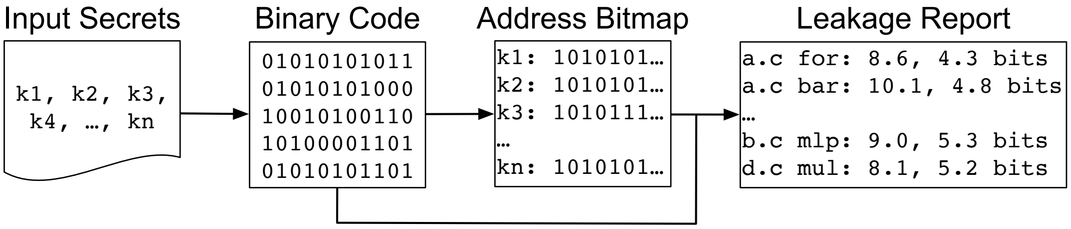


Figure 5.8: The workflow of Quincunx. We divide Quincunx into three steps to help the illustrate the process.

libraries, the input could be images, text, and information in another format. Encryption algorithms like AES and DES take an array representing the secrets. We generate the random input by flipping each bit of the keys. Some other programs also take a buffer as the input. The buffer may contain redundancy information and follow a specific format. So not every input that satisfies the input length is a valid input. Under the circumstance, we have a dictionary that contains the input. For example, crypto libraries usually take PKCS #1 as the input of the RSA private key. We generate a list of valid keys and use the keys to sample the program. Due to the enormous search space, we cannot iterate every input. However, our method can give a conservative estimate of each leakage site.

5.6.3 Step 2: Address Recording

Next, we record the memory access information based on the input secret. We use the dynamic binary instrumentation (DBI) tool to record whether a particular memory access information is accessed or not. Inspired by AFL and Valgrind, we use a bitmap to represent whether a particular memory address is accessed or not. For example, the bitmap 0101 of address l_1, l_2, l_3, l_4 means that l_2, l_4 are hit during the execution. During the execution, the direct information we get is the virtual memory address (VMA). However, we transfer the address into the offset LMA (Load Memory Address) in the original binary file. It has two benefits. First, modern computer systems employ the Address Space Layout Randomization (ASLR). With ASLR, the operating system randomly puts the code and data at various memory offset. As a result, the code will get different execution traces even for the same input because the image is loaded at a different address each time. Second, it helps us locate the leakage site in the source code.

We rely on the program header and symbol information to recover the offset in the binary. During the loading process, the operating system maps each segment into a contiguous memory region. So the offset of the virtual memory address between the code within the same segment is the same as the offset within the binary file. We choose the start address of some common functions (e.g., main) as the navigation function. Let b be the navigation function here. We get the virtual memory address of the start of the function (VMA_b) from the DBI tool and the load memory address of the function (LMA_b) from the symbol information. For any virtual address, we use Equation 5.6 to calculate the offset in the original address.

$$LMA_a = LMA_b - VMA_b - VMA_a \tag{5.6}$$

There are two kinds of memory access: access to the code and access to the data.

- Code Access. The memory address of the code is the value of the program counter rip register. However, the length of x86 instructions can be from 1 byte to 15 bytes. For instructions whose length is longer than one byte, we also update the address after the value rip until the address reaches the instruction's boundary.
- Data Access. For instructions with memory access, we identify all the operands of each instruction. Same as the code address, the instruction can read or write multiple bytes one time. Instructions like push and pop have implicit memory accesses. We all update the bitmap correspondingly.

5.6.4 Step 3: Leakage Detection and Quantification

In this step, we detect and quantify the side-channel leakage. For each $k \in K$, we have a boolean number (0 or 1) to describe whether the address l is accessed or not during the execution when the input is k. For the brevity of description, we use $B^l_{k_i}$ to denote the boolean number.

5.6.4.1 Multi-granularity Side-channel Detection

Quincunx can detect side-channel vulnerabilities in different granularities. For each granularity, we calculate a new bitmap that represents access information of each unit. We have the bitmap of the memory access at the byte level. An attacker who has the coarse-grained observation at address space may not be able to distinguish the address difference. For example, modern X64 computer systems use 48-bit virtual memory addresses. The top 36 bit of the virtual address is called Virtual Page Number (VPN), and the bottom 12 bits of the virtual address is called offset. The memory management unit (MMU) transfers the virtual address into the physical address by mapping the Virtual Page Number (VPN) into the Physical Page Number (PPN) while keeping the offset of the address. So two addresses with the same VPN but different offsets are not distinguishable by an attacker who can perform the page-level observation.

Suppose a program reads one byte at memory address `0x7ffff7ffd001` when the secret is k_1, and reads a different byte at memory `0x7ffff7ffda01` when the secret is k_2, an attacker who has the page-level observation cannot launch the attack but an attacker who launches the cache attack can know the input is k_1 or k_2. We update the new bitmap of the large unit by merging the bitmaps of small units that fall into the large unit. That is, $\forall l_1, l_2, \ldots, l_n \in l_N, B_{k_i}^{l_N} = B_{k_i}^{l_1} \vee, \ldots, \vee B_{k_n}^{l_n}$.

5.6.4.2 Leakage Detection

The access of one address is secret-dependent if $\exists\, k_{i_1}, k_{i_2} \in K,\ B_{k_{i_1}}^{l} \oplus B_{k_{i_2}}^{l} = 1$. In this step, we perform the logical exclusive OR operation on every boolearn value in the same address. If the result is 1, then the address is vulnerable to the side-channel attacks.

Example 1. *Suppose we sample a program from k_1 to k_8 and record memory accesses at the granularity of one byte from* `7ffff7ffd000` *to* `7ffff7ffd03f` *(one cache line). We perform a bit-wise operation of each address within the range. Because the cache line is always accessed. It is not a leak here.*

	k1	*k2*	*k3*	*k4*	*k5*	*k6*	*k7*	*k8*	*Result*
7ffff7ffd000	*1*	*0*	*1*	*0*	*1*	*1*	*0*	*1*	
7ffff7ffd000	*0*	*1*	*1*	*0*	*1*	*1*	*1*	*1*	
7ffff7ffd000	*1*	*1*	*1*	*1*	*1*	*1*	*0*	*1*	
7ffff7ffd000	*1*	*0*	*1*	*0*	*1*	*1*	*1*	*1*	
...									
7ffff7ffd03f	*1*	*0*	*1*	*0*	*1*	*1*	*1*	*0*	
cache line	*1*	*1*	*1*	*1*	*1*	*1*	*1*	*1*	*0 (No Leaks)*

5.6.4.3 Estimate the Leaked Information

According to Definition 4, we quantify the information leakage based on the distribution of the observation. If we calculate the information leakage from a $4\mathrm{MB}$ memory area, then the length of the byte-level bitmap is $4 \times 1024 \times 1024 = 4194304$. While it is possible to compute the distribution of the bitmap theoretically, it is hard to compare the bitmap of such length in practice.

To solve the problem, we use the debug information to split the bitmap into several segments and map the leakage site into the source code location. Each segment maps the address access situation of the function in the source code. In our setting, the maximal length of the bitmap of each segment is 1024, the length of `uint1024_t`. So we can put the bitmap into a variable. The split operation eases the computation. However, it also adds the limitation of the max address range when we quantify the side-channel leakage, as shown in Table 5.1. We believe the range of the memory is big enough for us to quantify most of the cache side-channel attacks.

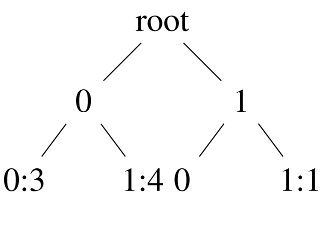
Granularity	Byte	Cache Line (64 Bytes)	Memory Page (4 KB)
Range	1024 Bytes	64 KB	4 KB

Table 5.1: The maximum range address range when we quantify the amount of the leakage with different granularity.

After that, we traverse the bitmap for each k and count it to get the distribution of the bitmap. The counting process is like traversing a binary tree.

Example 2. *Shown in Figure 5.9, we have the following observation from k_1 to k_8. We count the distribution of the observation. The process is like traversing a binary tree and increase the counter of the child node. According to Definition 4, the amount of the leakage is $1.4\ bits$.*

$$H(O') = -\frac{3}{8} \cdot \log_2 \frac{3}{8} - \frac{1}{2} \cdot \log_2 \frac{1}{2} - \frac{1}{8} \cdot \log_2 \frac{1}{8} = 1.4\,\text{bits}$$



```
k1  00
k2  01
k3  01
k4  01
k5  00
k6  01
k7  11
k8  00
```

Observation	P
00	3/8
01	1/2
10	0
11	1/8

Figure 5.9: We estimate the amount of leaked information by traversing a binary tree.

5.7 Implementation

We have implemented a prototype of Quincunx. It takes an ELF binary as well as the source code of the corresponding program. The front-end of Quincunx is implemented in Python. It can generate random inputs and test the program with these inputs many times. The address recording part is implemented as an Intel Pin tool plugin in C++ to record the bitmap for each input. The main component uses libelfin, an open-source library to parse ELF binaries and read DWARF debug information. The current implementation can support 64-bits ELF binary in Linux.

5.8 Evaluation

In this section, we evaluate Quincunx on real-world software trying to answer the following questions:

- **Effectiveness:** Can Quincunx quantify the information leakage with an conservative estimation? If so, what is the loss compared with the real leakage?
- **Usage:** Can Quincunx help developers find and fix side-channel leakages?

Evaluation setup. We run all the following experiments in a machine with Ubuntu 18.04 LTS. The machine equips with a 2.30GHz Intel Core i9-9880H CPU with 16GB RAM memory. We use the default release settings to build the software's source code into ELF 64-bit executables with GCC 7.5.

Benchmarks. We collect benchmarks from real-world applications and libraries. Our test programs cover applications from diverse functionalities, including small programs, cryptography, machine learning applications, and graphic libraries.

Cryptography Applications: Our benchmarks include most of the popular cryptography libraries, such as mebdTLS, OpenSSL, Libgcrypt, and wolfSSL. Encryption ciphers can be divided into two categories: symmetric ciphers and asymmetric ciphers. For symmetric ciphers, we choose AES. We also evaluate the tool on RSA, a widely used algorithm in the public-key cryptosystems.

Machine Learning Applications: We evaluate the tool on TinyDNN, a dependency-free deep learning framework in C++.

Graphic Libraries: We evaluate the tool on GDK, a popular graphic library. Previous research has demonstrated attacks to infer the keystroke [34, 45].

Table 5.2 presents the overview result. The minimum unit of the program during the experiment is one function. If a function has more than one access bitmap, then the function is vulnerable to side-channel attacks. We measure the side-channel leakages at two different granularities: the byte level and the cache line level. The result shows that programs have more vulnerable functions and tend to leak more information at the byte level than the cache line level. We run the target program under random inputs and record memory accesses. For cryptography programs, we run the target program with 4096 different inputs. During the fuzzing process, if the program does not have a new memory access bitmap after several rounds, we call the fuzzing step reaches a fixed point. From the side-channel detection aspect, it is not necessary to continue running the experiments since more rounds do not reveal new leakages. During the experiment, we find AES ciphers reach the fixed point after about 300 rounds, and RSA can reach the fixed point after around 500 rounds. Previous work [14] also reported a similar result. We continue running the experiment during our experiment until the observed value for the same observation is at least 5 to satisfy the chi-square test requirement. Therefore, we choose to run all the cipher with different inputs for 4096 times. If one program has more than two leaked functions, we run the chi-squared test and calculate the total amount of leakages.

5.8.1 Cryptography Libraries

AES encryption can be divided into three steps. In the first step, the round keys are derived from the encryption keys (AES key schedule), and a state array is initialized with the plain-text. In the second step, the AES performs eleven rounds of state manipulation. In the final stage, the state array is copied out as the encrypted data. The reference implementation uses lookup tables in the first two steps. Intuitively, the implementation is vulnerable to side-channel attacks. There are some mitigated versions of AES. We evaluate Quincunx on three variants of AES implementations: the reference implementation, the bit-sliced version, and the AES-NI implementation. For mebedTLS, we evaluate AES-NI and the reference implementation. We do not find any leakage sites in the AES-NI version. On the other hand, the reference implementation has several leakage sites. The bit-sliced version from OpenSSL and wolfSSL should be resistant to side-channel attacks. However, we find they only adopt the protection in the encryption function but leave the key expansion function unprotected. The T-Table

Table 5.2: Evaluation results overview: Name of the benchmark, size of the Input Data, number of leaked functions, maximum Leakage, size of the benchmark, and performance.

Benchmark	Input Data		# Leaked Functions		Max. Leak (bits)		Program Size		Performance		
	#	bits	1 Byte	64 Bytes	1 Byte	64 Bytes	LoC	Size	Fuzzing	Comparison	Chi-squared
One-byte Lookup Table	4096	12.0	1	1	7.5	3.6	18	104 KB	0.8 min	0.1 min	0 min
Four-byte Lookup Table	4096	12.0	1	1	9.6	8.5	18	104 KB	0.7 min	0.1 min	0 min
AES T-Table mbedTLS 2.15	4096	12.0	2	2	11.6	6.7	-	1.1 MB	4.1 min	0.8 min	0.1 min
AES NI mbedTLS 2.15	4096	12.0	0	0	0	0	-	1.1 MB	2.8 min	0 min	0 min
AES OpenSSL 1.1.0f	4096	12.0	1	0	1.4	0	-	1.0 MB	3.1 min	0.2 min	0 min
AES OpenSSL 1.1.1	4096	12.0	1	0	1.3	0	-	909 KB	2.9 min	0.2 min	0 min
AES wolfSSL 4.5.0	4096	12.0	1	0	0.11	0	-	1.1 MB	3.3 min	0.1 min	0 min
AES Libgcrypt 1.8.7	4096	12.0	0	0	0	0	-	4.4 MB	10.2 min	0 min	0 min
RSA mbedTLS 2.15	4096	12.0	7	6	10.3	6.0	-	1.8 MB	1321.1 min	63.5 min	5.1 min
RSA OpenSSL 1.1.0f	4096	12.0	12	6	9.7	8.1	-	11 MB	981.1 min	75.4 min	4.5 min
RSA OpenSSL 1.1.1	4096	12.0	6	4	6.3	2.1	-	13 MB	1021.1 min	45.1 min	5.5 min
RSA wolfSSL 4.5.0	4096	12.0	5	2	3.4	2.5	-	4.2 MB	1613.6 min	51.6 min	7.9 min
RSA Libgcrypt 1.8.7	4096	12.0	8	6	8.6	6.6	-	4.4 MB	1412.6 min	23.9 min	4.3 min
Tiny DNN	10000	-	17	14	17.1	16.1	-	4.4 MB	183.5 min	24.6 min	5.2 min
GDK 3.24.23	4096	16.0	33	31	12.1	6.0	-	48.5 MB	1060.8 min	24.1 min	1.5 min

implementation has side-channel leakage sites in both the key expanding function and the actual encryption function.

The RSA implementations blind the secret data on a random value before operating on it. We use a fixed RSA blinding value to avoid non-deterministic behaviors during the execution. After that, we compare the result with the previous work [1, 13]. The result shows that Quincunx can identify all leakage sites reported in the previous work. In addition, Quincunx identifies new leakages in the private key decode function (conv_ascii2bin) in OpenSSL 1.1.1. To evaluate the effectiveness of the quantification result, we compare the two RSA implementations between OpenSSL 1.1.0 and OpenSSL 1.1.1. We are interested if these leaked functions from OpenSSL 1.1.0 are fixed in OpenSSL 1.1.1. Note that Quincunx can analyze the side-channel leakages at both the byte granularity and the cache line granularity. We show the result analyzed with the cache line granularity.

Table 5.3: Leaked Functions in RSA implemented by OpenSSL 1.1.0. We manually check if these leaked functions are fixed in OpenSSL 1.1.1.

File	Vulnerable Function	A[a](bits)	Q[b](bits)	Fixed
bn_lib.c	BN_num_bits_word	(*, *, *, *, *, 17.2, 12.6)	5.7	✓
bn_lib.c	bn_correct_top	1.7	7.6	✓
bn_lib.c	BN_ucmp	*	6.7	✓
ct_b64.c	__udivdi3	5.9	4.3	✓
bn_gcd.c	int_bn_mod_inverse	(14.9, 9.2, 8.2, 7.9, 1.0)	7.7	✓[1]
bn_exp.c	BN_mod_exp_mont_consttime	(1.0, 1.0)	1.0	✗
bn_mont.c	BN_from_montgomery_word[2]	(*, *, *, *, *, *, 0.0)	9.7	✓
bn_mont.c	BN_from_montgomery_word	0.0	0.1	✗
bn_asm.c	bn_sqr_comba8	9.3	5.0	✓
bn_div.c	BN_div	(17.2, 11.9, 3,8, 0.3, 0.3)	7.3	✓
rsa_ossl.c	rsa_ossl_mod_exp	1.0	4.8	✓
encode.c	conv_bin2ascii	Not Found	1.2	✗

[a] Abacus [b] Quincunx

[1] The leakage site at line 144 (OpenSSL 1.1.1) are not fixed.

[2] We break the function into two parts during the analysis.

Table 5.3 shows the leaked function and the amount of leakage reported by Quincunx and Abacus [1]. According to Abacus, the mark “$*$” means timeout, which indicates more severe leakages. 0.0 means very small amount of leakage, but not exactly zero. As the result indicates, the leakage sites that leak a lot of information (> 5.0 bits) are all fixed in OpenSSL 1.1.1. The result indicates that Quincunx can help developers identify side-channel leakages and pick severe ones from them automatically. As expected, we find that most of the leakages are from the big number calculations. OpenSSL has a side-channel mitigated big number implementation. As long as big numbers are derived from the secret data, the big number should be enabled with the flag BN_FLG_CONSTTIME. However, the flag is off by default. We find that the developers forget to propagate the flag in the caller, which results in the vulnerabilities in BN_div and int_bn_mod_inverse.

For leakage sites that are not fixed by the developers, Quincunx reports that they leak less amount of information. For example, function BN_mod_exp_mont_consttime check if the modulus is odd. However, it is not a side-channel leakage vulnerability since the modulus is a prime number and all prime numbers except 2 are odd. Quincunx reports that the leakage site can leak one bit of information.

Previous work [1] reports some leakage sites leak little information, but Quincunx reports the vulnerabilities are severe. Abacus quantifies the amount of leaked information for one execution. So it is possible that some leakage sites leak little information for one input but leak more information for a different input. We run the chi-squared test to determine if any two leaked functions have independent leakages. However, we cannot conclude that any two leakages from cryptography benchmarks are independent under the chi-squared test. We think it is probably due to the avalanche effect in cryptography.

5.8.2 Non-cryptography Applications

We evaluate Quincunx on TinyDNN, a deep learning framework in C++. We construct a LeNet-5 model [138] and train it with the MINIST dataset. LeNet-5 network has six layers, including three convolutional layers, one fully connected layer, and two pooling layers. The batch size in our experiment is 10, and the number of epochs is 30. After the training, the accuracy of the neural networks on test set images is 99.5%. Then we use the model to infer the images. The inference dataset consists of $10,000$ images. During the inference, we record the memory access information. The evaluation shows all three types of layers have different memory accesses under different inputs. We find that the leakages primarily come from the activation function and the image convert function.

Chapter 6 | Discussion and Limitations

6.1 Fast and Precise Side-channel Vulnerability Detection

Code Coverage

Lorem analyzes one x86 execution trace each time. The design is very precise in terms of true leakages compared to many static source code methods [57, 139], but shares common limitations of dynamic approaches. Lorem may only cover part of the code. Each time we only get one single execution trace. Therefore, we may miss some side-channel vulnerabilities not covered by the analyzed trace. However, this is not a crucial problem for analyzing cryptographic libraries, because cryptographic libraries are designed to have the same code coverage under various inputs. The evaluation result also confirms the above intuition. For symmetric encryption, there is no secret-dependent control-flow transfers during our evaluation. RSA implementations have several secret-dependent control-flow transfers. Many of them are bound checks, which do not leak much information and have negligible effects on the whole code coverage as well.

Design Choices

At the early stage of the project, we used SMT solvers to get the solutions of constraints. However, we found for some ciphers (RSA), Lorem timed out after several days. The profiling result showed that the tool spent most its time in Z3. Later, we used a sampling method to detect side-channel leakages. The sampling method seems simple and may miss some leakages in theory. However, the evaluation result shows Lorem can identify all leakages found by the previous work [13, 21, 22].

Cache Model

We use a simple cache model in this work. Recent works such as CacheAudit [23] and CaSym [21] use a Least Recently Used (LRU) cache. First, there is no fundamental difficulties that prevent us from adopting a more realistic cache models. Second, the LRU model is still a substantial simplification of a real cache. For example, Intel Sandy Bridge CPUs use an undisclosed hash function to map memory addresses to L3 cache lines [140]. However, the choice of LRU cache models can bring expensive overheads. For example, CaSym can only analyze small code snippets.

Leakages

While recent works have reported lots of potential side-channel vulnerabilities, most of them are not patched by developers. The DES implementation of OpenSSL has a few side-channel leakages, but given the end of life status of DES, it is still unpatched for the worth of engineering effort. Lorem also detects several leakages in asymmetric ciphers such as RSA. After we manually analyze these leakage sites, we find many of them leak very little or useless information (e.g., the length of big numbers), which may partially explain why developers are not interested in fixing all the side-channel leakages.

6.2 Precise Analysis on Single-trace Attacks

While recent work found many side-channel vulnerabilities, we note that many of them are not patched by developers. Side-channels are ubiquitous in software, and it is difficult to fix all of them. We need a tool that estimates the sensitivity of each vulnerability so software engineers can focus on “severe” leakages. For example, Abacus reports that the modular exponentiation using square and multiply algorithms can leak more information than a key validation function.

Software developers can use Abacus to find severe vulnerabilities and reason about countermeasures. Abacus estimates the amount of leaked information for each side-channel leakage in one execution trace. Abacus is useful for software engineers to test programs and fix vulnerabilities. The design, which is more precise in reporting true leakages as compared with other static methods [57, 139], obviously misses leakages on unexplored traces. The amount of leaked information quantified by Abacus also depends on the value of secret keys. However, as the tool is intended for debugging and testing, we think it is a software engineer's responsibility to select the input key and trigger the path in which they are interested. It is not a problem for crypto software since virtually all keys follow similar computational paths.

We use the amount of leaked information to represent the severity level of each side-channel vulnerability. Although imperfect, Abacus produces a reasonable measurement for each leak. For example, the simple modular exponentiation is notoriously famous for multiple side-channel attacks [5]. During the execution, a single leak point may execute multiple times and each time leak a different bit. In this case, Abacus reports that the vulnerability can leak the whole key. However, not every leak point inside a loop is severe. If a site in the loop leaks the same bit from the original key, and these leaks are not independent. Abacus captures most fine-grained information by modeling each leak during the execution as a formula and the conjunction of the formulas to describe its total effect. Some leakage sites (e.g., square and multiply in RSA) can leak one particular bit of the original key, but some leakage sites leak one bit from several bytes in the original key. Abacus can capture the dependency among the leaks and reports more precise leakage information.

Abacus reaches full precision if the number of estimated leaked bits equals to Definition 1 in Chapter 4. Abacus may lose precision from the memory model it uses in theory. However, we did not find false positives caused by the imprecise memory model during our evaluation. Sampling introduces imprecision but with a probabilistic guarantee. However, during the evaluation, we find that Abacus cannot estimate the amount of leakage for some leakage sites in a reasonable time, which means the number of K^o is very small. According to Definition 1 in Chapter 4, it means the leakage is very severe.

6.3 Precise Analysis on Multiple-trace Attacks

In this work, we present an approach that tries to give an estimation of the amount of leaked information by access-based side-channel attacks. These attacks exploit the data-flow from secrets to load address and the data-flow from the data-flow from secrets to branch conditions to retrieve secrets based on the observation on the memory accesses. Although these kinds of side-channel leakages have been discovered for decades, the up-to-date software still has some side-channel leakages. For some of these vulnerabilities, developers do not fix them because they think these side-channel vulnerabilities are not important. To show the importance of these side-channel leakages, one way is to demonstrate an end-to-end attack based on the vulnerability. However, demonstrating an end-to-end attack often need a lot of manual effort and the domain knowledge of the victim program, which is not often the case in practice. It is good to have a tool that automatically assesses the severity level of these side-channel leakages. So it would be better to have a proper metric to quantify the side-channel leakage. However, we find previous side-channel quantification tools are developed to ensure the noninterference of the program. They use the over-approximation heuristics method to quantify the leakage. For example, CacheAudit estimates that a 128-bit AES encryption can leak more than 128 bits. As a result, even these tools report severe leakage, it does not mean the program has a truly severe leakage.

Quincunx defines the amount of leaked information by the empirical entropy of the observations. For a determinism program, channel capacity is the tight upper bound of the entropy of the observation. The quantification result is always smaller than the channel capacity and can approximate the value of the channel capacity in some cases. Quincunx can give a conservative estimation of the side-channel leakage based on Channel Capacity. The channel capacity measures the information flow between the source and the destination. One useful characteristic of channel capacity is not affected by the input, which is useful because we cannot assume the input secrets of the software in practice. In Chapter 5, we use the sampling result to approximate the true value of the channel capacity. We show that the sampling method presented in the chapter can only give the conservative estimation of the amount of the true leakage.

Quincunx requires the random input variables to be independent and identically distributed (i.i.d.). That's why we apply a black box fuzzing instead of a coverage-guided fuzzing. As long as each time we get a random input from the whole input space (Population) that is independent of the previous input, the input variables should satisfy the i.i.d. assumption. For AES-128 encryption, each time we randomly generate a 128 bit key from 0 to 2^{128}, the sequence of key values follows the i.i.d. assumption. The situation is a little different for machine learning libraries. In this dissertation, we evaluated Quincunx on a deep learning model. Each time, we randomly select an item from a dataset (Sample). If the dataset is biased, then we cannot get correct results. The good news is that the most well-known data sets in machine learning are independent and identically distributed. The assumption of i.i.d. is required by many machine learning techniques (e.g., cross-validation, SVM, back-propagation). The i.i.d. assumption simplifies many methods by assuming the data will not change over the time. In this dissertation, we used MINIST, a well-known handwriting digit dataset. We believe the i.i.d. assumption holds under the circumstance, although the assumption can be violated in some real-world scenarios.

6.4 Limitations

The dissertation research has a few limitations.

1. The methods in this dissertation cannot find all kinds of side-channel leakages. No tool can find all forms of leakages. Even for address-based side-channel leakages, Abacus and Quincunx cannot find all the leakages due to the code coverage problem. Some verification tools can prove the absence of side-channel leakages for some small code snippets. However, to the best of my knowledge, no tools as yet can prove the absence of side-channel leakages in production software.

2. The quantification result only applies to specific threat models. For example, Abacus quantifies the number of leaked bits during one real execution. So it is possible that one leakage site leaks little information in one execution but leaks much more information in another execution.

3. Abacus and Quincunx can only handle address-based side-channel attacks. Many side-channel attacks infer secret data based on other side-channel signals (e.g., timing, EM signals). However, the root cause of many of these attacks is still the same. That is, the program accesses different addresses when it processes different input secrets. Under the circumstance, our methods can still detect these leakage sites.

4. Quincunx is a dynamic approach. So it bears the same limitations of dynamic approaches as well. Quincunx cannot cover all the paths and may miss some side-channel vulnerabilities. It is usually not a crucial problem for cryptography libraries as cryptography libraries are designed to have the same control flow under various inputs. For other libraries such as graphic rendering and machine learning, Quincunx is very likely to miss some side-channel vulnerabilities. However, Quincunx is not designed to find

every side-channel vulnerability. The goal of Quincunx is to identify severe side-channel vulnerabilities from many less severe vulnerabilities. So we do not think it is the main limitation of Quincunx. However, users of Quincunx should be aware of the code coverage problem.

Chapter 7 | Conclusion

This dissertation research studies address-based side-channel leakages in two aspects: detection and quantification. In the side-channel vulnerability detection research, we explore and solve the two bottlenecks in today's automated side-channel vulnerability detection tool: performance and precision. In the side-channel vulnerability quantification research, we propose two different side-channel leakage quantification methods based on information theory.

7.1 Summary

In Chapter 3, we present a practical address-based side-channel detection tool, Lorem, that is capable of detecting secret-dependent control-flows and data accesses at the same time. We model side-channel leakage sites as math constraints and use dynamic symbolic execution to generate constraints for possible leakages. Using statistical testing, Lorem can identify the true leakages from these constraints. Lorem analyzes the side-channel leakages directly from X86 execution traces. Existing binary analysis tools usually transfer x86 instructions to intermediate languages (IR) simplify the implementation. However, such designs have expensive overheads and imprecise analysis results. To tackle these problems, we take the engineering efforts and implement the analysis from scratch based on the Intel developer's manual. We evaluate Lorem on popular cryptographic libraries including OpenSSL, mbedTLS, Libgcrypt, and Monocyper. The evaluation results show that Lorem is three times to one hundred times faster than the state-of-art tools while finding all the leakages reported by the previous tools. In addition, Lorem identifies new leakages. The new leakages were later confirmed by other researchers and software developers.

In Chapter 4, we present a novel method to quantify address-based side-channel leakage.

We quantify the amount of leakage information for a single trace attack. The amount of the leaked information is based on the search space reduced by side-channel attacks. We rely on Lorem to detect side-channel leakages. After that, we use approximate model counting to quantify the leakage sites. The method is implemented in a prototype tool called Abacus. We show its effectiveness in quantifying side-channel leakage. With a new definition of information leakage that models actual side-channel attackers, quantifying the number of leaked bits helps understand the severity level of side-channel vulnerabilities. The evaluation confirms that Abacus is useful in estimating the amount of leaked information in real-world applications.

In Chapter 5, we propose a fuzzing method to automatically detect and quantify the address-based side-channel leakages automatically. The proposed method can produce an estimation of the side-channel leakage of the deterministic program. As a result, any severe leakage reported by Quincunx is a true severe leakage. We also study the total effect of two leakages. Using statistical hypothesis testing, we are able to determine if the two leakages are independent. If so, we can calculate the sum of the amount of leakage by adding them. Our evaluation result on both several libraries and benchmarks show that Quincunx is effective and accurate in detecting the side-channel leakage.

In conclusion, this dissertation research develops practical techniques for precise side-channel analysis in software systems. For each proposed method, we implement and evaluate it on real-world software. The evaluation results show the effectiveness and generality of our proposed method. We also release the prototypes to facilitate future research in the area.

7.2 Future Directions

There are a few potential directions to extend the dissertation research.

Kernel Space Side-channel Vulnerability Detection. There are several side-channel vulnerability detection tools. Most of the tools are designed to analyze the side-channel leakages in user-level applications such as cryptographic libraries. It is reasonable since many side-channel attacks are used to break cryptographic applications. Recently, researchers also started to focus on side-channel attacks [48] inside the kernel space. However, it is non-trivial to apply existing side-channel detection tools in the Linux kernel. Take the method in the dissertation, for example. We will have the following challenges. First, our proposed tools in the dissertation rely on dynamic binary instrumentation frameworks to collect the runtime information. For kernel programs, a similar task can be achieved by some whole system emulators such as BitBlaze [111] and S2E [141]. S2E uses a modified Linux kernel and QEMU to trace the kernel components. S2E uses QEMU 1.0, which does not support the latest Linux kernel. The engineering efforts of extending S2E to work on the latest kernel are significant. Second, the kernel has very complicated control-flow graphs compared to cryptography libraries. Some vulnerabilities need to be triggered by specific input from the user space applications or devices. Our tool cannot handle interruptions and exceptions. But they are common inside the kernel. On the positive side, we expect there are plenty of leakage sites inside the kernel. The dissertation provides methods to identify "severe" leakages automatically.

Compiler-assisted Side-channel Vulnerability Analysis and Mitigation. It is promising to mitigate the side-channel leakages in a compiler backend. For example, the conditional execution can be eliminated by using masks. Eliminating side-channel leakages from compilers is not a brand new idea. Coppens et al. [54] propose several methods to remove all potential leakages that are related to data flow and control flow. GNU toolchain adds new options (-mlfence-after-load, -mlfence-before-indirect-branch) to mitigate the Load Value Injection (LVI) attack during the compilation. Coppens et al.'s approach has a slowdown with a factor from 2 to 24. The LVI harden binaries generated from GNU toolchains takes 6-10x longer to run. For example, GNU toolchains insert LFENCE before any possible vulnerable instructions, including all load instructions, indirect branches, and return instructions. We suggest only changing points that are likely to leak sensitive information. Finally, with the tool presented in the dissertation, we can fix only part of the "severe" leakage sites.

Side-channel Leakage Quantification from Noisy Observations. It is possible to extend the quantification part to include other side-channel signals (e.g., sound, CPU usages, EM signals). The key is to model the noises properly. In this dissertation, we assume that an attacker can have a noise-free observation. While a noise-free observation is possible for some address-based side-channel attacks, the assumption is still too ideal for many real side-channel attacks. The good news is that there are some mature methods in information theory to model some types of noises (e.g., Gaussian noise). The noisy-channel coding theorem [116] provides a computable method to estimate the information that can be reliably transferred over a channel between the secret data and attackers' observations.

Appendix A
Abacus Usage Manual[1]

Introduction

Abacus [1] is an address-based side-channel vulnerability analysis tool. Different from previous tools [13, 22, 23], it can also give a precise estimation of the amount of the leaked information for each leakage site. Abacus is open source under the MIT License.

Abacus takes a binary executable as the input. It uses dynamic binary instrumentation tools to collect execution traces. After that, Abacus analyzes these traces and produces the vulnerability report. While Abacus can work on the stripped binary executable, Abacus can also read the symbol and debugging information to give a more fine-grained (e.g., line numbers) report. Table A.1 shows an example of the report. If you are interested in how Abacus works, please refer to the technical paper [1].

Table A.1: A sample leakage report generated by Abacus

File	Line No.	Function	# Leaked Bits	Type
set_key.c	350	DES_set_key_unchecked	5.8	DA
set_key.c	350	DES_set_key_unchecked	6.6	DA
set_key.c	350	DES_set_key_unchecked	7.5	DA
set_key.c	350	DES_set_key_unchecked	6.4	DA
set_key.c	355	DES_set_key_unchecked	1.9	DA
set_key.c	355	DES_set_key_unchecked	3.1	DA

[1] The original manual was published in [142].

Requirements

We have tested Abacus on both macOS and Ubuntu. You can refer to the continuous integration scripts to build Abacus on your operating system. However, it is strongly recommended to build Abacus inside a container to avoid any dependency problems. To simplify the illustration, we only include the instructions of installations within the docker in this chapter.

- Supported OS: Ubuntu 18.04
- Memory: 32 GB (If you want to run experiments concurrently, update the size of RAM accordingly. Otherwise the program may be terminated by the system.)

Installation

Abacus can be built within a docker, simply run the following command:

```
$ git clone https://github.com/s3team/Abacus.git
$ cd Abacus
$ ./docker.sh
```

The “docker.sh” script creates a docker image automatically and enters the container that includes all dependencies. After that, run the following command to build Abacus:

```
$ ./build.sh
```

Run the hello world example

In this section, we walk you through the steps to test a simple function with Abacus.

As shown in Figure A.1, the function takes a 32-bit integer as the secret input and checks the last digit of the integer. An attacker can know the last bit of the input integer by observing which branch is actually executed. So in the above function, we think the code has one secret-dependent control-flow vulnerability, and it can leak one bit of the secret information.

```c
#include <stdio.h>

void is_odd(uint16_t secret) {
  int res = secret % 2;
  if (res) {
    printf("Odd␣Number\n");
  } else {
    printf("Even␣Number\n");
  }
}
```

Figure A.1: A sample function that leaks one bit information

Mark secret data as symbolic

In order to test this function with Abacus, we need to mark the variable that representing the secret data as a symbolic variable. We use the function abacus_make_symbolic. The function takes three arguments: the type of the symbol, the address of the secret, and the length of the secret input. In the below example, the secret input is the variable secret, and its length is two bytes. We add the main function in Figure A.2 and compile the source code into an executable.

```c
int main() {
  uint16_t secret = 6;
  char *type = "1";
  abacus_make_symbolic(type, &secret, 2);
  is_odd(secret);
  return 0;
}
```

Figure A.2: A simple function that marks a variable secret as symbolic

Build the example

Abacus analyzes vulnerabilities on the binary executable. Here we build it into a 32-bit ELF executable. Note that while Abacus can work on stripped binaries without the source code, we use debug information to get a more detailed result (e.g., the line number in the source code) in this example.

```
$ cd examples
$ gcc -m32 -g example1.c
```

Collect the trace

We use the pin tool to collect the execution trace. The tool can automatically collect the trace and other necessary runtime information.

```
$ cd /abacus/Pintools
$ make PIN_ROOT=/abacus/Intel-Pin-Archive/ TARGET=ia32
$ cd /abacus
$ /abacus/Intel-Pin-Archive/pin \
  -t Pintools/obj-ia32/MyPinToolLinux.so \
  -- ./examples/a.out
```

You will get two files Function.txt and Inst_data.txt. Inst_data.txt is the mandatory input of Abacus. Function.txt is optional.

Quantify the leakage

To analyze the execution trace and generate the report, run the following command:

```
$ ./build/App/QIF/QIF ./Inst_data.txt -f Function.txt \
  -d ./examples/a.out -o result.txt
```

You should get the following output:

```
Start Computing Constraints
Total Constraints: 1
Control Transfer: 1
Data Access: 0
Information Leak for each address:
Address: 5664259b Leaked:1.0 bits Type: CF
Source code: example1.c line number: 3
Function Name:  is_odd Module Name:  a.out Offset: 30
...
```

As expected, it shows that the function is_odd has one secret-dependent control-flow vulnerability at line 5. Also, Abacus shows the vulnerability leaks 1 bit information.

Analyze Cryptography Function

We have applied Abacus on the following libraries:

- OpenSSL: 0.9.7, 1.0.2f, 1.0.2k, 1.1.0f, 1.1.1, 1.1.1g
- MbedTLS: 2.5, 2.15
- Libgcrypt: 1.8.5
- Monocyper: 3.0

The results can be reproduced by running the simple command after you build Abacus successfully inside the container. We have prepared scripts to analyze each cryptography algorithm automatically. For example, if you want to test AES in mbedTLS 2.5, you can simply run the following command.

```
$ cd /abacus/script/AES_MBEDTLS_2.5
$ ./start.sh
```

Command-line Options

Abacus takes the trace file as the input (Inst_data.txt). Besides the trace file, Abacus has the following command-line options:

-d <executable file>

Read an elf *executable file*. Abacus can parse the debug information inside the file. With the optional input, Abacus is able to output which line in the original source code actually leaks the sensitive information.

-f <function file>

Read a function file that was generated by the Pin tool. The command is optional. With the optional input, Abacus is able to output which function leaks the information and call sites from the sensitive buffer to the leaked site.

-n <Monte Carlo times>

Set the times of Monte Carlo Sampling when Abacus estimates the amount of leakage information. If you do not specify the option, Abacus can automatically terminate the sampling when the tool has 95% confidence that the error of estimated leaked information is less than 1 bit. Refer to the paper [1] to learn more about the details.

-a <the address of the secret buffer>
-s <the size of the buffer>

The two input options must be used together. In the previous example, we use abacus_make_symbolic to mark the secret buffer. For a raw binary, we use the above two options to tell Abacus which buffer is the secret and its length.

Bibliography

[1] Bao, Q., Z. Wang, J. R. Larus, and D. Wu (2021) "Abacus: Precise Side-Channel Analysis," in *Proceedings of the 43rd International Conference on Software Engineering (ICSE)*, pp. 797–809.

[2] HAMILTON, M. H. (1969) "Apollo 11 Guidance Computer (AGC) Source Code for the Command and Lunar Modules," *URL https://github.com/chrislgarry/Apollo-11*.

[3] Telegram (2021) "Telegram for Android Source," *URL https://github.com/DrKLO/Telegram*.

[4] Potvin, R. and J. Levenberg (2016) "Why Google Stores Billions of Lines of Code in a Single Repository," *Communications of the ACM*, **59**(7), pp. 78–87.

[5] Kocher, P. C. (1996) "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems," in *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO)*, Springer, pp. 104–113.

[6] Yarom, Y. and K. Falkner (2014) "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack," in *Proceedings of the 23rd USENIX Security Symposium (USENIX Security)*, pp. 719–732.

[7] Disselkoen, C., D. Kohlbrenner, L. Porter, and D. Tullsen (2017) "Prime+Abort: A Timer-free High-precision L3 Cache Attack using Intel TSX," in *Proceedings of the 26th USENIX Security Symposium (USENIX Security)*, pp. 51–67.

[8] Liu, F., Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee (2015) "Last-Level Cache Side-Channel Attacks are Practical," in *Proceedings of the 36th IEEE Symposium on Security and Privacy (IEEE S&P)*, pp. 605–622.

[9] Kar, M., A. Singh, S. Mathew, A. Rajan, V. De, and S. Mukhopadhyay (2017) "Improved Power-side-channel-attack Resistance of an AES-128 Core via a Security-aware Integrated Buck Voltage Regulator," in *Proceedings of the 2017 IEEE International Solid-State Circuits Conference (ISSCC)*, IEEE, pp. 142–143.

[10] Agrawal, D., B. Archambeault, J. R. Rao, and P. Rohatgi (2002) "The EM Side-Channel(s)," in *International Workshop on Cryptographic Hardware and Embedded Systems*, Springer, pp. 29–45.

[11] ALAM, M., H. A. KHAN, M. DEY, N. SINHA, R. CALLAN, A. ZAJIC, and M. PRVULOVIC (2018) "One&Done: A Single-Decryption EM-Based Attack on OpenSSL's Constant-Time Blinded RSA," in *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, pp. 585–602.

[12] CHARI, S., C. S. JUTLA, J. R. RAO, and P. ROHATGI (1999) "Towards Sound Approaches to Counteract Power-analysis Attacks," in *Annual International Cryptology Conference*, Springer, pp. 398–412.

[13] WANG, S., P. WANG, X. LIU, D. ZHANG, and D. WU (2017) "CacheD: Identifying Cache-Based Timing Channels in Production Software," in *Proceedings of the 26th USENIX Security Symposium (USENIX Security)*, pp. 235–252.

[14] WEISER, S., A. ZANKL, R. SPREITZER, K. MILLER, S. MANGARD, and G. SIGL (2018) "DATA - Differential Address Trace Analysis: Finding Address-based Side-Channels in Binaries," in *Proceedings of the 27th SENIX Security Symposium (USENIX Security)*, pp. 603–620.

[15] SHIH, M.-W., S. LEE, T. KIM, and M. PEINADO (2017) "T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs." in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, pp. 1544–3566.

[16] ZHANG, D., Y. WANG, G. E. SUH, and A. C. MYERS (2015) "A Hardware Design Language for Timing-Sensitive Information-Flow Security," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 503–516.

[17] LIU, F., Y. YAROM, Q. GE, G. HEISER, and R. B. LEE (2015) "Last-level Cache Side-Channel Attacks are Practical," in *Proceedings of the 2015 IEEE Symposium on Security and Privacy (IEEE S&P)*, IEEE, pp. 605–622.

[18] YAROM, Y., D. GENKIN, and N. HENINGER (2017) "CacheBleed: a Timing Attack on OpenSSL Constant-time RSA," *Journal of Cryptographic Engineering*, **7**(2), pp. 99–112.

[19] XU, Y., W. CUI, and M. PEINADO (2015) "Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems," in *Proceedings of the 36th IEEE Symposium on Security and Privacy (IEEE S&P)*, pp. 640–656.

[20] WICHELMANN, J., A. MOGHIMI, T. EISENBARTH, and B. SUNAR (2018) "MicroWalk: A Framework for Finding Side Channels in Binaries," in *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC)*, pp. 161–173.

[21] BROTZMAN, R., S. LIU, D. ZHANG, G. TAN, and M. KANDEMIR (2019) "CaSym: Cache Aware Symbolic Execution for Side Channel Detection and Mitigation," in *Proceedings of the 40th IEEE Symposium on Security and Privacy (IEEE S&P)*, pp. 364–380.

[22] WANG, S., Y. BAO, X. LIU, P. WANG, D. ZHANG, and D. WU (2019) "Identifying Cache-Based Side Channels through Secret-Augmented Abstract Interpretation," in *Proceedings of the 28th USENIX Security Symposium (USENIX Security)*, pp. 657–674.

[23] DOYCHEV, G., D. FELD, B. KOPF, L. MAUBORGNE, and J. REINEKE (2013) "CacheAudit: A Tool for the Static Analysis of Cache Side Channels," in *Proceedings of the 22nd USENIX Security Symposium (USENIX Security)*, USENIX, pp. 431–446.

[24] AUMÜLLER, C., P. BIER, W. FISCHER, P. HOFREITER, and J.-P. SEIFERT (2002) "Fault attacks on RSA with CRT: Concrete results and practical countermeasures," in *International Workshop on Cryptographic Hardware and Embedded Systems*, Springer, pp. 260–275.

[25] WEI, W. and B. SELMAN (2005) "A New Approach to Model Counting," in *Theory and Applications of Satisfiability Testing* (F. Bacchus and T. Walsh, eds.), Springer Berlin Heidelberg, pp. 324–339.

[26] GE, Q., Y. YAROM, D. COCK, and G. HEISER (2018) "A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware," *Journal of Cryptographic Engineering*, **8**(1), pp. 1–27.

[27] SZEFER, J. (2019) "Survey of Microarchitectural Side and Covert channels, Attacks, and Defenses," *Journal of Hardware and Systems Security*, **3**(3), pp. 219–234.

[28] MOGHIMI, D., J. VAN BULCK, N. HENINGER, F. PIESSENS, and B. SUNAR (2020) "CopyCat: Controlled Instruction-Level Attacks on Enclaves," in *Proceedings of the 29th USENIX Security Symposium (USENIX Security)*, pp. 469–486.

[29] YAROM, Y. and K. FALKNER (2014) "FLUSH+RELOAD: a High Resolution, Low Noise, L3 Cache Side-Channel Attack," in *Proceedings of the 23rd USENIX Security Symposium (USENIX Security)*, pp. 719–732.

[30] GRUSS, D., C. MAURICE, K. WAGNER, and S. MANGARD (2016) "Flush+Flush: A Fast and Stealthy Cache Attack," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, Springer, pp. 279–299.

[31] XU, Y., W. CUI, and M. PEINADO (2015) "Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems," in *Proceedings of the 2015 IEEE Symposium on Security and Privacy (IEEE S&P)*, IEEE, pp. 640–656.

[32] GRAS, B., K. RAZAVI, H. BOS, and C. GIUFFRIDA (2018) "Translation Leak-aside Buffer: Defeating Cache Side-Channel Protections with TLB Attacks," in *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, pp. 955–972.

[33] GRUSS, D., E. KRAFT, T. TIWARI, M. SCHWARZ, A. TRACHTENBERG, J. HENNESSEY, A. IONESCU, and A. FOGH (2019) "Page Cache Attacks," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pp. 167–180.

[34] Gruss, D., R. Spreitzer, and S. Mangard (2015) "Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches," in *Proceedings of the 24th USENIX Security Symposium (USENIX Security)*, pp. 897–912.

[35] Osvik, D. A., A. Shamir, and E. Tromer (2006) "Cache Attacks and Countermeasures: The Case of AES," in *Proceedings of the 2006 The Cryptographers' Track at the RSA Conference on Topics in Cryptology*, CT-RSA'06, Springer-Verlag, pp. 1–20.

[36] Brickell, E. (2011) "Technologies to Improve Platform Security," in *Workshop on Cryptographic Hardware and Embedded Systems*, vol. 11, p. 0.

[37] Brumley, D. and D. Boneh (2005) "Remote Timing Attacks are Practical," *Computer Networks*, **48**(5), pp. 701–716.

[38] Bernstein, D. J. (2005) *Cache-timing Attacks on AES*, *Tech. rep.*, The University of Illinois at Chicago.

[39] Bonneau, J. and I. Mironov (2006) "Cache-collision Timing Attacks against AES," in *International Workshop on Cryptographic Hardware and Embedded Systems*, Springer, pp. 201–215.

[40] Arnaud, C. and P.-A. Fouque (2013) "Timing Attack against Protected RSA-CRT Implementation used in PolarSSL," in *Cryptographers' Track at the RSA Conference*, Springer, pp. 18–33.

[41] Hua, W., Z. Zhang, and G. E. Suh (2018) "Reverse Engineering Convolutional Neural Networks through Side-Channel Information Leaks," in *Proceedings of the 2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, IEEE, pp. 1–6.

[42] Wei, L., B. Luo, Y. Li, Y. Liu, and Q. Xu (2018) "I Know What You See: Power Side-Channel Attack on Convolutional Neural Network Accelerators," in *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC)*, pp. 393–406.

[43] Batina, L., S. Bhasin, D. Jap, and S. Picek (2019) "CSI NN: Reverse Engineering of Neural Network Architectures Through Electromagnetic Side Channel," in *Proceedings of the 28th USENIX Security Symposium (USENIX Security)*, pp. 515–532.

[44] Yan, M., C. W. Fletcher, and J. Torrellas (2020) "Cache Telepathy: Leveraging Shared Resource Attacks to Learn DNN Architectures," in *Proceedings of the 29th USENIX Security Symposium (USENIX Security)*, pp. 2003–2020.

[45] Wang, D., A. Neupane, Z. Qian, N. B. Abu-Ghazaleh, S. V. Krishnamurthy, E. J. Colbert, and P. Yu (2019) "Unveiling Your Keystrokes: A Cache-based Side-Channel Attack on Graphics Libraries," in *Proceedings of the 26th Annual Network and Distributed System Security Symposium (NDSS)*, pp. –.

[46] JANG, Y., S. LEE, and T. KIM (2016) "Breaking Kernel Address Space Layout Randomization with Intel TSX," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pp. 380–392.

[47] CAO, Y., Z. WANG, Z. QIAN, C. SONG, S. V. KRISHNAMURTHY, and P. YU (2019) "Principled Unearthing of TCP Side Channel Vulnerabilities," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pp. 211–224.

[48] CAO, Y., Z. QIAN, Z. WANG, T. DAO, S. V. KRISHNAMURTHY, and L. M. MARVEL (2016) "Off-Path TCP Exploits: Global Rate Limit Considered Dangerous," in *Proceedings of the 25th USENIX Security Symposium (USENIX Security)*, pp. 209–225.

[49] PAGE, D. (2005) "Partitioned Cache Architecture as a Side-Channel Defence Mechanism," *IACR Cryptology ePrint Archive*, p. 280.

[50] WANG, Z. and R. B. LEE (2007) "New Cache Designs for Thwarting Software Cache-based Side Channel Attacks," in *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA)*, ACM, pp. 494–505.

[51] LI, X., V. KASHYAP, J. K. OBERG, M. TIWARI, V. R. RAJARATHINAM, R. KASTNER, T. SHERWOOD, B. HARDEKOPF, and F. T. CHONG (2014) "Sapper: A Language for Hardware-level Security Policy Enforcement," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, ACM, pp. 97–112.

[52] WERNER, M., T. UNTERLUGGAUER, L. GINER, M. SCHWARZ, D. GRUSS, and S. MANGARD (2019) "ScatterCache: Thwarting Cache Attacks via Cache Set Randomization," in *Proceedings of the 28th USENIX Security Symposium (USENIX Security)*, USENIX Association, pp. 675–692.

[53] V. GLEISSENTHALL, K., R. G. KICI, D. STEFAN, and R. JHALA (2019) "IODINE: Verifying Constant-Time Execution of Hardware," in *Proceedings of the 28th USENIX Security Symposium (USENIX Security)*, pp. 1411–1428.

[54] COPPENS, B., I. VERBAUWHEDE, K. D. BOSSCHERE, and B. D. SUTTER (2009) "Practical Mitigations for Timing-Based Side-Channel Attacks on Modern x86 Processors," in *Proceedings of the 30th IEEE Symposium on Security and Privacy (IEEE S&P)*, IEEE Computer Society, pp. 45–60.

[55] BRICKELL, E., G. GRAUNKE, M. NEVE, and J.-P. SEIFERT (2006) "Software Mitigations to Hedge AES against Cache-based Software Side Channel Vulnerabilities." *IACR Cryptology ePrint Archive*, **2006**, p. 52.

[56] CRANE, S., A. HOMESCU, S. BRUNTHALER, P. LARSEN, and M. FRANZ (2015) "Thwarting Cache Side-Channel Attacks Through Dynamic Software Diversity," in *Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS)*, pp. 8–11.

[57] ALMEIDA, J. B., M. BARBOSA, G. BARTHE, F. DUPRESSOIR, and M. EMMI (2016) "Verifying Constant-Time Implementations," in *Proceedings of the 25th USENIX Security Symposium (USENIX Security)*, USENIX Association, pp. 53–70.

[58] TIWARI, M., J. K. OBERG, X. LI, J. VALAMEHR, T. LEVIN, B. HARDEKOPF, R. KASTNER, F. T. CHONG, and T. SHERWOOD (2011) "Crafting a Usable Microkernel, Processor, and I/O System with Strict and Provable Information Flow Security," in *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA)*, ACM, p. 189–200.

[59] KÖNIGHOFER, R. (2008) "A Fast and Cache-timing Resistant Implementation of the AES," in *Cryptographers' Track at the RSA Conference*, Springer, pp. 187–202.

[60] REBEIRO, C., D. SELVAKUMAR, and A. DEVI (2006) "Bitslice Implementation of AES," in *International Conference on Cryptology and Network Security*, Springer, pp. 203–212.

[61] LANGLEY, A. (2010) "ctgrind—checking that functions are constant time with Valgrind, 2010," *URL https://github.com/agl/ctgrind*, **84**.

[62] XIAO, Y., M. LI, S. CHEN, and Y. ZHANG (2017) "Stacco: Differentially Analyzing Side-Channel Traces for Detecting SSL/TLS Vulnerabilities in Secure Enclaves," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pp. 859–874.

[63] WANG, W., Y. ZHANG, and Z. LIN (2019) "Time and Order: Towards Automatically Identifying Side-Channel Vulnerabilities in Enclave Binaries," in *Proceedings of the 22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, pp. 443–457.

[64] CORON, J.-S. (1999) "Resistance against Differential Power Analysis for Elliptic Curve Cryptosystems," in *International Workshop on Cryptographic Hardware and Embedded Systems*, Springer, pp. 292–302.

[65] BERNSTEIN, D. J., T. LANGE, and P. SCHWABE (2012) "The Security Impact of a New Cryptographic Library," in *International Conference on Cryptology and Information Security in Latin America*, Springer, pp. 159–176.

[66] *LibSodium*.
URL `https://libsodium.org`

[67] ZINZINDOHOUÉ, J.-K., K. BHARGAVAN, J. PROTZENKO, and B. BEURDOUCHE (2017) "HACL: A Verified Modern Cryptographic Library," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pp. 1789–1806.

[68] *Monocypher*.
URL `https://monocypher.org/`

[69] BERNSTEIN, D. J. ET AL. (2008) "ChaCha, a Variant of Salsa20," in *Workshop record of SASC*, vol. 8, pp. 3–5.

[70] LANGLEY, A., W. CHANG, N. MAVROGIANNOPOULOS, J. STROMBERGSON, and S. JOSEFSSON (2016) "ChaCha20-Poly1305 Cipher Suites for Transport Layer Security (TLS)," *RFC 7905*, (10).

[71] BARBOSA, M., G. BARTHE, K. BHARGAVAN, B. BLANCHET, C. CREMERS, K. LIAO, and B. PARNO (2021) "SoK: Computer-aided Cryptography," in *Proceedings of the 42nd IEEE Symposium on Security and Privacy (IEEE S&P).*

[72] ELDIB, H., C. WANG, and P. SCHAUMONT (2014) "SMT-based Verification of Software Countermeasures against Side-Channel Attacks," in *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Springer, pp. 62–77.

[73] BAYRAK, A. G., F. REGAZZONI, D. NOVO, and P. IENNE (2013) "Sleuth: Automated Verification of Software Power Analysis Countermeasures," in *Proceedings of the International Conference on Cryptographic Hardware and Embedded Systems (CHES)*, Springer, pp. 293–310.

[74] SHI, J., X. SONG, H. CHEN, and B. ZANG (2011) "Limiting Cache-based Side-Channel in Multi-tenant Cloud using Dynamic Page Coloring," in *2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSN-W)*, IEEE, pp. 194–199.

[75] LIU, F., Q. GE, Y. YAROM, F. MCKEEN, C. ROZAS, G. HEISER, and R. B. LEE (2016) "CATalyst:Defeating Last-Level Cache Side Channel Attacks in Cloud Computing," in *Proceedings of the 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, pp. 406–418.

[76] WANG, Z. and R. B. LEE (2007) "New Cache Designs for Thwarting Software Cache-based Side Channel Attacks," in *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA)*, pp. 494–505.

[77] VATTIKONDA, B. C., S. DAS, and H. SHACHAM (2011) "Eliminating Fine Grained Timers in Xen," in *Proceedings of the 3rd ACM workshop on Cloud Computing Security Workshop*, pp. 41–46.

[78] FLETCHERY, C. W., L. REN, X. YU, M. VAN DIJK, O. KHAN, and S. DEVADAS (2014) "Suppressing the Oblivious RAM Timing Channel While Making Information Leakage and Program Efficiency Trade-offs," in *Proceedings of the 20th International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, pp. 213–224.

[79] ROBLING DENNING, D. E. (1982) *Cryptography and Data Security*, Addison-Wesley Longman Publishing Co., Inc.

[80] GRAY III, J. W. (1992) "Toward a Mathematical Foundation for Information Flow Security," *Journal of Computer Security*, **1**(3-4), pp. 255–294.

[81] MCCAMANT, S. and M. D. ERNST (2008) "Quantitative Information Flow as Network Flow Capacity," in *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI)*, pp. 193–205.

[82] BACKES, M., B. KÖPF, and A. RYBALCHENKO (2009) "Automatic Discovery and Quantification of Information Leaks," in *Proceedings of the 30th IEEE Symposium on Security and Privacy (IEEE S&P)*, pp. 141–153.

[83] PHAN, Q.-S., P. MALACARIA, O. TKACHUK, and C. S. PĂSĂREANU (2012) "Symbolic Quantitative Information Flow," *SIGSOFT Softw. Eng. Notes*, **37**(6), pp. 1–5.

[84] PASAREANU, C. S., Q.-S. PHAN, and P. MALACARIA (2016) "Multi-run Side-Channel Analysis using Symbolic Execution and Max-SMT," in *Proceedings of the IEEE 29th Computer Security Foundations Symposium (CSF) 2016*, IEEE, pp. 387–400.

[85] CHATTOPADHYAY, S., M. BECK, A. REZINE, and A. ZELLER (2017) "Quantifying the Information Leak in Cache Attacks via Symbolic Execution," in *Proceedings of the 15th International Conference on Formal Methods and Models for System Design (MEMOCODE)*, ACM, pp. 25–35.

[86] WEI, W. and B. SELMAN (2005) "A New Approach to Model Counting," in *International Conference on Theory and Applications of Satisfiability Testing*, Springer, pp. 324–339.

[87] GOMES, C. P., J. HOFFMANN, A. SABHARWAL, and B. SELMAN (2007) "From Sampling to Model Counting," in *Proceedings of the 20th International Joint Conference on Artifical Intelligence (AAAI)*, vol. 2007, pp. 2293–2299.

[88] GOMES, C. P., A. SABHARWAL, and B. SELMAN (2006) "Model Counting: A New Strategy for Obtaining Good Bounds," in *Proceedings of the 21st National Conference on Artificial Intelligence - Volume 1*, pp. 54–61.

[89] KROC, L., A. SABHARWAL, and B. SELMAN (2008) "Leveraging Belief Propagation, Backtrack Search, and Statistics for Model Counting," in *International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming*, Springer, pp. 127–141.

[90] CHISTIKOV, D., R. DIMITROVA, and R. MAJUMDAR (2017) "Approximate Counting in SMT and Value Estimation for Probabilistic Programs," *Acta Informatica*, **54**(8), pp. 729–764.

[91] PHAN, Q.-S. (2015) "Model Counting Modulo Theories," *arXiv preprint arXiv:1504.02796.*

[92] Biondi, F., M. A. Enescu, A. Heuser, A. Legay, K. S. Meel, and J. Quilbeuf (2018) “Scalable Approximation of Quantitative Information Flow in Programs,” in *Proceedings of the 19th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, Springer, pp. 71–93.

[93] Chakraborty, S., K. S. Meel, and M. Y. Vardi (2016) “Algorithmic Improvements in Approximate Counting for Probabilistic Inference: From Linear to Logarithmic SAT Calls,” in *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 3569—-3576.

[94] Luk, C.-K., R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood (2005) “Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation,” in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, ACM New York, NY, USA, pp. 190–200.

[95] Nethercote, N. and J. Seward (2007) “Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation,” in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, p. 89–100.

[96] Bellard, F. (2005) “QEMU, a Fast and Portable Dynamic Translator,” in *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, vol. 41, California, USA, pp. 46–56.

[97] Xu, D., J. Ming, and D. Wu (2017) “Cryptographic Function Detection in Obfuscated Binaries via Bit-Precise Symbolic Loop Mapping,” in *Proceedings of the 2017 IEEE Symposium on Security and Privacy (IEEE S&P)*, IEEE, pp. 921–937.

[98] Ming, J., D. Xu, Y. Jiang, and D. Wu (2017) “Binsim: Trace-based Semantic Binary Diffing via System Call Sliced Segment Equivalence Checking,” in *Proceedings of the 26th USENIX Security Symposium (USENIX Security)*, pp. 253–270.

[99] Jia, X., C. Zhang, P. Su, Y. Yang, H. Huang, and D. Feng (2017) “Towards Efficient Heap Overflow Discovery,” in *Proceedings of the 26th USENIX Security Symposium (USENIX Security)*, pp. 989–1006.

[100] Brumley, D., I. Jager, T. Avgerinos, and E. J. Schwartz (2011) “BAP: A Binary Analysis Platform,” in *International Conference on Computer Aided Verification*, Springer, pp. 463–469.

[101] ——— (2011) “BAP: A Binary Analysis Platform,” in *Computer Aided Verification* (G. Gopalakrishnan and S. Qadeer, eds.), pp. 463–469.

[102] Shoshitaishvili, Y., R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna (2016) “SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis,” in *Proceedings of the 2016 IEEE Symposium on Security and Privacy (IEEE S&P)*, pp. 138–157.

[103] GENKIN, D., A. SHAMIR, and E. TROMER (2014) “RSA Key Extraction via Low-Bandwidth Acoustic Cryptanalysis,” in *Annual Cryptology Conference*, Springer, pp. 444–461.

[104] BULCK, J. V., M. MINKIN, O. WEISSE, D. GENKIN, B. KASIKCI, F. PIESSENS, M. SILBERSTEIN, T. F. WENISCH, Y. YAROM, and R. STRACKX (2018) “Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution,” in *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, p. 991–1008.

[105] VAN SCHAIK, S., C. GIUFFRIDA, H. BOS, and K. RAZAVI (2018) “Malicious Management Unit: Why Stopping Cache Attacks in Software is Harder Than You Think,” in *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, pp. 937–954.

[106] LEE, S., M.-W. SHIH, P. GERA, T. KIM, H. KIM, and M. PEINADO (2017) “Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing,” in *Proceedings of the 26th USENIX Security Symposium (USENIX Security)*, pp. 557–574.

[107] GULLASCH, D., E. BANGERTER, and S. KRENN (2011) “Cache Games – Bringing Access-Based Cache Attacks on AES to Practice,” in *Proceedings of the 2011 IEEE Symposium on Security and Privacy (IEEE S&P)*, pp. 490–505.

[108] TSUNOO, Y., T. SAITO, T. SUZAKI, M. SHIGERI, and H. MIYAUCHI (2003) “Cryptanalysis of DES Implemented on Computers with Cache,” in *Proceedings of the Cryptographic Hardware and Embedded Systems (CHES)* (C. D. Walter, Ç. K. Koç, and C. Paar, eds.), Springer Berlin Heidelberg, pp. 62–76.

[109] ARNAUTOV, S., B. TRACH, F. GREGOR, T. KNAUTH, A. MARTIN, C. PRIEBE, J. LIND, D. MUTHUKUMARAN, D. O’KEEFFE, M. L. STILLWELL, ET AL. (2016) “SCONE: Secure Linux Containers with Intel SGX,” in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 689–703.

[110] SCHUSTER, F., M. COSTA, C. FOURNET, C. GKANTSIDIS, M. PEINADO, G. MAINAR-RUIZ, and M. RUSSINOVICH (2015) “VC3: Trustworthy Data Analytics in the Cloud using SGX,” in *Proceedings of the 2015 IEEE Symposium on Security and Privacy (IEEE S&P)*, IEEE, pp. 38–54.

[111] SONG, D., D. BRUMLEY, H. YIN, J. CABALLERO, I. JAGER, M. G. KANG, Z. LIANG, J. NEWSOME, P. POOSANKAM, and P. SAXENA (2008) “BitBlaze: A New Approach to Computer Security via Binary Analysis,” in *International Conference on Information Systems Security*, Springer, pp. 1–25.

[112] INTEL CORPORATION (2019) *Intel® 64 and IA-32 Architectures Software Developer’s Manual.*

[113] YUN, I., S. LEE, M. XU, Y. JANG, and T. KIM (2018) “QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing,” in *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, pp. 745–761.

[114] DULLIEN, T. and S. PORST (2009), "REIL: A Platform-independent Intermediate Representation of Disassembled Code for Static Code Analysis," .

[115] DE MOURA, L. and N. BJØRNER (2008) "Z3: An Efficient SMT Solver," in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Springer-Verlag, pp. 337–340.

[116] SHANNON, C. E. (1948) "A Mathematical Theory of Communication," *Bell system technical journal*, **27**(3), pp. 379–423.

[117] CLARK, D., S. HUNT, and P. MALACARIA (2007) "A Static Analysis for Quantifying Information Flow in A Simple Imperative Language," *Journal of Computer Security*, **15**(3), pp. 321–371.

[118] SMITH, G. (2009) "On the Foundations of Quantitative Information Flow," in *Foundations of Software Science and Computational Structures* (L. de Alfaro, ed.), Springer Berlin Heidelberg, pp. 288–302.

[119] DOYCHEV, G. and B. KÖPF (2017) "Rigorous Analysis of Software Countermeasures Against Cache Attacks," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 406–421.

[120] ZHANG, K., Z. LI, R. WANG, X. WANG, and S. CHEN (2010) "Sidebuster: Automated Detection and Quantification of Side-Channel Leaks in Web Application Development," in *Proceedings of the 17th ACM conference on Computer and communications security (CCS)*, pp. 595–606.

[121] BANG, L., A. AYDIN, Q.-S. PHAN, C. S. PĂSĂREANU, and T. BULTAN (2016) "String Analysis for Side Channels with Segmented Oracles," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pp. 193–204.

[122] ASKAROV, A. and S. CHONG (2012) "Learning is Change in Knowledge: Knowledge-based Security for Dynamic Policies," in *Proceedings of the 25th Computer Security Foundations Symposium (CSF)*, IEEE, pp. 308–322.

[123] HOEFFDING, W. (1994) "Probability Inequalities for Sums of Bounded Random Variables," in *The Collected Works of Wassily Hoeffding*, Springer, pp. 409–426.

[124] WALPOLE, R. E., R. H. MYERS, S. L. MYERS, and K. YE (1993) *Probability and Statistics for Engineers and Scientists*, vol. 5, Macmillan New York.

[125] GODEFROID, P., M. Y. LEVIN, D. A. MOLNAR, ET AL. (2008) "Automated Whitebox Fuzz Testing," in *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS)*, vol. 8, pp. 151–166.

[126] BONNEAU, J. and I. MIRONOV (2006) "Cache-Collision Timing Attacks Against AES," in *Cryptographic Hardware and Embedded Systems - CHES 2006* (L. Goubin and M. Matsui, eds.), Springer Berlin Heidelberg, pp. 201–215.

[127] HUND, R., C. WILLEMS, and T. HOLZ (2013) "Practical Timing Side Channel Attacks against Kernel Space ASLR," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy (IEEE S&P)*, IEEE, pp. 191–205.

[128] HALEVI, T. and N. SAXENA (2015) "Keyboard Acoustic Side Channel Attacks: Exploring Realistic and Security-sensitive Scenarios," *International Journal of Information Security*, **14**(5), pp. 443–456.

[129] OPENSSL (2020) *OpenSSL 1.1.1 Series Release Notes*.
URL `https://www.openssl.org/news/openssl-1.1.1-notes.html`

[130] HÄHNEL, M., W. CUI, and M. PEINADO (2017) "High-resolution Side Channels for Untrusted Operating Systems," in *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC)*, pp. 299–312.

[131] HONG, S., M. DAVINROY, Y. KAYA, S. N. LOCKE, I. RACKOW, K. KULDA, D. DACHMAN-SOLED, and T. DUMITRAŞ (2018) "Security Analysis of Deep Neural Networks Operating in the Presence of Cache Side-Channel Attacks," *arXiv preprint arXiv:1810.03487*.

[132] WANG, E., Q. ZHANG, B. SHEN, G. ZHANG, X. LU, Q. WU, and Y. WANG (2014) "Intel Math Kernel Library," in *High-Performance Computing on the Intel® Xeon Phi™*, Springer, pp. 167–188.

[133] WANG, J., C. SUNG, M. RAGHOTHAMAN, and C. WANG (2021) "Data-Driven Synthesis of Provably Sound Side Channel Analyses," in *Proceedings of the 43rd International Conference on Software Engineering (ICSE)*, pp. 810–822.

[134] SUNG, C., B. PAULSEN, and C. WANG (2018) "CANAL: a Cache Timing Analysis Framework via LLVM Transformation," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*, pp. 904–907.

[135] CHATTOPADHYAY, S., M. BECK, A. REZINE, and A. ZELLER (2019) "Quantifying the Information Leakage in Cache Attacks via Symbolic Execution," *ACM Transactions on Embedded Computing Systems (TECS)*, **18**(1), pp. 1–27.

[136] PHAN, Q.-S., L. BANG, C. S. PASAREANU, P. MALACARIA, and T. BULTAN (2017) "Synthesis of Adaptive Side-Channel Attacks," in *Proceedings of the 2017 IEEE 30th Computer Security Foundations Symposium (CSF)*, IEEE, pp. 328–342.

[137] SUMNER, W. N. and X. ZHANG (2010) "Memory Indexing: Canonicalizing Addresses Across Executions," in *Proceedings of the 18th ACM SIGSOFT international symposium on Foundations of Software Engineering (FSE)*, pp. 217–226.

[138] LeCun, Y., L. Bottou, Y. Bengio, and P. Haffner (1998) "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, **86**(11), pp. 2278–2324.

[139] Bacelar Almeida, J., M. Barbosa, J. S. Pinto, and B. Vieira (2013) "Formal Verification of Side-Channel Countermeasures Using Self-composition," *Sci. Comput. Program.*, **78**(7), pp. 796–812.

[140] Farshin, A., A. Roozbeh, G. Q. Maguire Jr, and D. Kostić (2019) "Make the Most out of Last Level Cache in Intel Processors," in *Proceedings of the 14th EuroSys Conference (EuroSys)*, pp. 1–17.

[141] Chipounov, V., V. Kuznetsov, and G. Candea (2012) "The S2E platform: Design, Implementation, and Applications," *ACM Transactions on Computer Systems (TOCS)*, **30**(1), pp. 1–49.

[142] Bao, Q., Z. Wang, J. R. Larus, and D. Wu (2021) "Abacus: A Tool for Precise Side-Channel Analysis," in *Proceedings of the 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pp. 238–239.

Vita

Qinkun Bao

Qinkun Bao finished his Ph.D. in the College of Information Sciences and Technology, The Pennsylvania State University. He was a member of the Software System Security Research Lab, where he worked under the supervision of Dr. Dinghao Wu. His research interests span reverse engineering, binary analysis, and trusted computing. Before he went to Penn State, he received his bachelor's degree in Electronic Engineering and Information Science (EEIS) from University of Science and Technology of China (USTC) in 2016.