CacheD: Identifying Cache-Based Timing Channels in Production Software

Shuai Wang, Pei Wang, Xiao Liu, Danfeng Zhang, and Dinghao Wu,
The Pennsylvania State University

https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/wang-shuai

This paper is included in the Proceedings of the 26th USENIX Security Symposium

August 16–18, 2017 • Vancouver, BC, Canada

ISBN 978-1-931971-40-9

Open access to the Proceedings of the 26th USENIX Security Symposium is sponsored by USENIX

CacheD: Identifying Cache-Based Timing Channels in Production Software

Shuai Wang, Pei Wang, Xiao Liu, Danfeng Zhang, and Dinghao Wu
The Pennsylvania State University
{szw175, pxw172, xvl5190}@ist.psu.edu, zhang@cse.psu.edu, dwu@ist.psu.edu

Abstract

Side-channel attacks recover secret information by analyzing the physical implementation of cryptosystems based on non-functional computational characteristics, e.g. time, power, and memory usage. Among all well-known side channels, cache-based timing channels are notoriously severe, leading to practical attacks against certain implementations of theoretically secure crypto algorithms, such as RSA, ElGamal and AES. Such attacks target the hierarchical design of the modern computer memory system, where different memory access patterns of a program can bring observable timing difference.

In this work, we propose a novel technique to help software developers identify potential vulnerabilities that can lead to cache-based timing attacks. Our technique leverages symbolic execution and constraint solving to detect potential cache differences at each program point. We adopt a cache model that is general enough to capture various threat models that are employed in practical timing attacks. Our modeling and analysis are based on the formulation of cache access at different program locations along execution traces. We have implemented the proposed technique as a practical tool named CacheD (Cache Difference), and evaluated CacheD towards multiple real-world cryptosystems. CacheD takes less than 17 CPU hours to analyze 9 widely used cryptographic algorithm implementations with over 120 million instructions in total. The evaluation results show that our technique can accurately identify vulnerabilities reported by previous research. Moreover, we have successfully discovered previously unknown issues in two widely used cryptosystems, OpenSSL and Botan.

1 Introduction

Side-channel attacks recover secret information by analyzing the physical implementation of crypto and other systems based on non-functional computational characteristics. Typical attributes exploited in such attacks include time [30], power [37], memory consumption [28], network traffic [16], and electromagnetic [46].

Among all side-channel attacks, cache-based timing attacks steal confidential information based on the program's runtime cache behaviors. Cache-based timing attacks are perhaps the most practical and important ones, since those attacks does not require any physical access to the confidential computation, yet the timing signal carries enough information to break RSA [3, 45, 59], AES [8, 11, 42, 53, 27] and ElGamal [63, 34]. Other than cryptosystems, research has also shown that cache-based timing channels may leak other confidential information [47, 57, 62, 58].

The mitigation mechanisms towards cache-based timing channels can be categorized into hardware and software based solutions. Hardware-based solutions focus on new cache designs such as partitioned cache [43, 54, 31, 61], randomized/remapping cache [54, 55, 33], and line-locking cache [54]. But such secure hardware assumes that crucial memory accesses are identified (by security experts) in the first place. Most software-based solutions only consider cache-based timing channels due to secret-dependent control flow [4, 25, 38, 7, 17, 44] and hence, cannot prevent subtle leakage found in source code without any secret-dependent control flow (see §2.2.2). More advanced program analyses [6, 19, 20, 60] can detect the subtle leakage missed by those solutions, but they only provide an upper-bound on timing-based information leakage; it is unclear what/where the vulnerability is when those tools report a non-zero upper bound.

We focus on cache-based timing analysis. Cache attacks can be categorized into three models [51], time-driven, access-driven, and trace-driven attacks, each of which leverages a different approach to monitor the cache behavior. Time-driven attacks [8] observe the overall execution time of the cryptosystems and require many measure samples. Existing work has demonstrated the feasibility to launch the cache-based attack locally or

remotely towards the AES encryption algorithm [8, 42]. In contrast, access-driven attacks [24, 53] and trace-driven attacks [2] exploit more fine-grained cache behavior and require fewer measurement samples, but they are based on more sophisticated threat models and require deep knowledge about the hardware and software system under attack [53, 39, 45].

Given the complexity of the memory hierarchy in modern computer systems, it is difficult for developers to reason about the cache access behavior of a program or a particular memory access. For example, the Appendix A shows a large and complicated symbolic formula of a memory access address found in our experiment. It is quite obvious how complicated it is to reason its cache behavior, let alone take the context into consideration. Developers may be able to come up with better abstractions and reasoning, but it is easy to miss nuances and corner cases as demonstrated in our findings (see §7). Thus it is of great practical value to develop an automated tool that can help developers reason about the cache behavior of a memory access.

In this paper, we propose a general trace-based method with symbolic execution and constraint solving to detect potential cache variations at each program location. Our theory and cache modeling are independent of threat models that are employed in attacks to utilize the potential vulnerabilities detected. Our modeling and analysis are based on formulations of cache access at different program locations along the execution trace. More specifically, we record the execution trace, and use symbolic execution (with the secret as symbols) to formulate the cache access variations at each memory access. In other words, for each memory access in an execution trace, we check whether it is possible that this memory access can touch different cache lines given different secret inputs. Moreover, our method also provides two values that will cause such cache access variations at one memory access using a constraint solver. Once confirmed, such cache access variations can be leveraged, with various threat models, for cache-based side-channel attacks.

We have implemented the proposed technique as a practical tool named CacheD (Cache Difference), and evaluated CacheD towards multiple real-world cryptosystems. The evaluation results show that our technique can accurately identify vulnerabilities reported by previous research. Moreover, we have successfully discovered previously unknown issues in two widely used cryptosystems, OpenSSL (version 0.9.7c and 1.0.2f) and Botan (version 1.10.13).

We make the following contributions.

- We propose a novel trace-based analysis method that models the cache variations on every memory access. Our modeling is conceptually simple yet general enough to capture most adopted threat models. While existing research is designed to infer an "upper-bound" on timing-based information leakage, our technique can accurately point out what/where the vulnerability is, and provide concrete examples to trigger the issue. It becomes much simpler for developers to identify potential timing channels in their code.
- We have developed a practical tool called CacheD, which is precise and scalable enough to assist developers in identifying vulnerable program points in production cryptosystems.
- We applied CacheD to a set of widely used cryptosystems to search for timing channels in the implementations of well-known cryptographic algorithms. Within 17 CPU hours, CacheD identified 156 vulnerable program points along the analyzed execution traces of over 120 million instructions.
- By monitoring cache traffic of the test cases using a hardware simulator, we have confirmed the identified vulnerabilities as true positives: different secrets provided by CacheD lead to observable cache behavior difference, which further reveals potential timing channels.

2 Background

2.1 Memory Hierarchy and Set-Associative Cache

The storage system of modern computers adopts a hierarchical design. In the hierarchy, storage hardware in higher layers has faster response time but lower capacity due to hardware cost. When the CPU needs to retrieve the data, it will access the layers from the top to the bottom. In this way, the CPU can speed up data retrieval with limited hardware resources, based on the observation that memory accesses in computer programs are usually temporarily and spatially coalesced.

The topmost three layers of the hierarchy are processor registers, caches, and the main memory, the latter two of which share the same address space. Since caches are built with costly and fast on-chip devices, their latency is much lower than that of the main memory. When a data read misses the cache, the CPU will have to retrieve the data from the main memory, thus leading to a significant delay up to hundreds of CPU cycles. Therefore, minimizing cache misses is one of the most important objectives in processor design.

The organization of a cache refers to the policy that decides how the data are stored and replaced based on their addresses in the memory space. Modern processors usually have multiple levels of caches that form a structure isomorphic to the whole memory hierarchy. In most

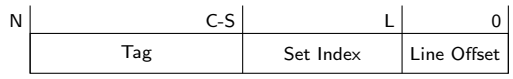

Figure 1: Cache indexing of S-way set-associative caches with the line size of 2^L bytes the capacity of 2^C bytes for a 2^N-byte address space.

cases, regardless of the levels on which the caches serve in the hierarchy, all of them are similarly organized, in a way called set-associative organization.

The minimal storage unit of set-associative caches is called a line and the cache is divided into sets consisting of the same number of lines. For set-associative caches, the organization is fully decided by three factors, i.e., the cache size, the line size, and the set size, where the set size is usually referred to as the number of ways the cache is associated in.

For a 2^K-way associative cache with the line size of 2^L bytes and a total capacity of 2^C bytes, it will be divided into 2^{C-K} sets. Bytes at each address can reside in any of lines of one particular set. Given an N-bit memory address, the cache looks up the data by dissecting the address into three parts, as shown in Fig. 1. The set index is used to locate the set in which the data may be stored, and the tag is used to confirm that the data is currently present in one of the lines in that set. If the tag matches, the line offset is used to locate the data inside the cache line; otherwise, accessing memory at that address leads to a cache miss, and the processor will have to retrieve the data from the lower layers of the memory hierarchy.

The low L bits of the address used to access the cache is irrelevant to timing, since the CPU always retrieve a whole cache line at a time. Only the high $N-L$ bits of the address indicate whether the memory access may hit the cache or not, based on the status of the cache. For most CPUs, one cache line holds 64 bytes of data, meaning the value of L is 6. Without losing generality, we will adopt this configuration in the rest of the paper unless noted otherwise.

2.2 Cache-Based Timing Channels

The cache is highly optimized, sophisticated, and inherently shared, in modern multicore and manycore architectures; even a small change in confidential data may bring drastic and subtle changes to the traffic to the cache. The consequence is that a dedicated attacker may reveal confidential data by measuring the latency introduce by the cache.

2.2.1 Leakage due to secret-dependent control flow

The confidential data may affect the traffic to the cache when there is secret-dependent control flow in the source code. Such leakage is best illustrated in the square-and-multiply implementation (shown below) of modular exponentiation, the core computation of cyphers such as RSA and ElGamal.

The pseudo-code computes $b^e \mod m$, where we assume the n-bit secret exponent e is in its binary representation and $e[i]$ is the i-th bit of e. Note that in this implementation, the branch condition at line 4 depends on one bit of e. Moreover, the real implementation of $r = r * b \mod m$ involves memory reads since b and e consist of multiple words in RSA and ElGamal. Consequently, an attacker that probes the cache usage of this victim program can recover the exponent, and hence, reveals the private key of RSA [1, 59] and ElGamal [63, 34].

```
1  r = 1;
2  for i from n–1 to 0 {
3      r = r*r mod m;
4      if (e[i] == 1) {
5          r = r*b mod m;
6      }
7  }
```

Previous work shows that secret-dependent control flow can be identified via information flow analysis, and be mitigated by removing secret-dependent branches or balancing branches conditioned on confidential data [4, 25, 38, 7, 17]. But these mechanisms cannot prevent leakages due to memory traffic as we discuss next. In fact, some tricks of removing secret-dependent control flow may introduce those new leakages in a program.

2.2.2 Leakage due to secret-dependent memory traffic

Cache-based timing attacks also work on implementations without any secret-dependent control flow, as illustrated in our running example (Fig. 2). This simple program iteratively accesses a table of size 128 stored in the memory. Note that the execution of this program follows the same control flow regardless of the secret since the loop body is executed exactly 200 times. Nevertheless, the memory address accessed at line 9 is key dependent, and hence, may result in memory requests to secret-dependent cache lines. Hence, a cache-probing attack (e.g., prime-and-probe attack [42, 53, 11, 63, 34]) may peek which cache line is accessed, and consequently, infer at least some bits of the secret in this example.

We emphasize that although we use this contrived running example in this paper for its simplicity, variants of the vulnerability illustrated in this example are found and exploited in real-world implementations of crypto systems, such as the AES implementation in OpenSSL, exploited in [24, 42, 53], the RSA and ElGamal implementations in Libgcrypt, exploited in [59, 34]. For example, the *sliding-window*-based modular exponential implementation [12] is vulnerable to cache-based timing attacks. With a parameter L (window size), the sliding-window implementation splits the secret expo-

```
1  void foo(int secret)
2  {
3    int table[128] = {0};
4    int i, t;
5    int index = 0;
6    for (i=0; i<200; i++)
7    {
8      index = (index+secret) % 128;
9      t = table[index];
10     t = table[(index) % 4];
11   }
12 }
```

Figure 2: CacheD running example.

nent e into a couple of *windows*, where each window holds a value (with at most L-bits) that is either a sequence of 0's, or bits that starts with a 1 and ends with a 1 (hence, an odd number). Given a non-zero window value, say v, this implementation computes b^v via a table lookup: $T[(v-1)/2]$, where T is a precomputed table such that $T[i] = b^{i*2+1} \mod N$. Note computing b^v involves no secret-dependent branch, but different cache lines are accessed given different values of v, hence, leading to practical cache-based timing attacks (e.g., [34]). CacheD successfully detects such vulnerabilities in Libgcrypt (§7.3).

2.3 Threat Model

We consider an attacker who shares the same hardware platform with the victim, a common scenario in the era of cloud computing. Hence, the attacker may observe cache accesses at different program locations along a program execution trace. That is, we assume an attacker can either directly or indirectly learn the trace of cache lines being accessed during the execution of the victim program. This strong threat model captures most cache-based timing attacks in the literature, such as an attacker who observes cache accesses by measuring the latency of the victim program (e.g., cache reuse attacks [43, 11, 10, 24] and evict-and-time attack [42]), or the latency of the attacker's program (e.g., prime-and-probe attacks [42, 53, 11, 63, 34]).

Compared with previously categorized threat models based on the abstraction of cache hit and miss (namely, the time-based, trace-based and access-based models [19, 51]), our more detailed model using the abstraction of cache lines has a couple of benefits. Firstly, our threat model is stronger than those based on cache hit/miss, since in most architectures, a trace of cache lines being accessed uniquely determines cache hit/miss at any program point. Secondly, working on the cache line abstraction makes the vulnerability analysis more general, since unlike cache hit/miss, the abstraction is independent of cache implementation details, such as cache-replacement policies, cache associativity and so on.

3 Method

3.1 Overview

In modern multicore and manycore architectures, the cache behavior may bring drastic difference in the latency of memory accesses (§2.1). Based on this observation, we propose a technique that detects potential timing channels caused by variant cache behavior. More specifically, we model cache lines being accessed as symbolic formulas where sensitive program data are treated as free variables during symbolic execution. In practice, sensitive data are typically the private keys used in cryptosystems and any data derived from those keys. With the help of constraint solvers, we can logically deduce whether sensitive data would affect the cache behavior of the program and hence, reveal potential timing channels.

Operationally, given a program point where a memory access occurs, we can model the memory address being accessed as a symbolic formula $F(\vec{k})$, where $\vec{k}$, as the only free variables in F, stands for program secrets. By substituting all occurrences of $\vec{k}$ in F with new free variables $\vec{k'}$, we can obtain another formula $F(\vec{k'})$. A satisfiable formula $F(\vec{k}) \neq F(\vec{k'})$ indicates that at this particular program point, the address used to access the memory depends on the values of the secrets.

We further refine the formulation above regarding two aspects. First, a difference in the memory address does not imply a difference in the cache line being accessed. That is, the low L bits (the line offset part in Fig. 1) of the address are irrelevant to cache behavior. Therefore, instead of trying to solve $F(\vec{k}) \neq F(\vec{k'})$, we construct F as a bit vector and solve $F(\vec{k}) \gg L \neq F(\vec{k'}) \gg L$, where $\gg$ is the right shift operation on bit vectors. Second, a solution of the refined formula may not be feasible along the trace under examination. For better precision, we augment the formula with the path condition (C) collected along the already processed trace. The path condition is the conjunction of all the branch conditions along the trace before this memory access (assuming an SSA transformation on the trace). The final formula for satisfiability checking is then $(F(\vec{k}) \gg L \neq F(\vec{k'}) \gg L) \wedge C$.

3.2 Example

Consider the running example shown in Fig. 2, in which the secret is used as the index of a table. By symbolizing the secret as k, a memory access formula can be build which presents the first table query (line 9) in the first iteration of the loop as:[1]

$$F(k) \equiv 10 + 4 \cdot k \mod 128$$

[1] Variable `index` is accumulated in the loop so further memory access formulas are different.

where 10 is assumed the base address of the table. This formula can be further refined into a cache line access formula as

$$F(k) \gg L \equiv 10 + 4 \cdot k \bmod 128 \gg L$$

where L equals to 6 regarding the cache configuration of most CPUs on the market (discussed in §2.1).

To find two secrets that result in different cache behaviors, we further replace symbol k in formula $F(k)$ with a fresh symbol, and check the satisfiability of $F(k) \gg 6 \neq F(k') \gg 6$ using a theory prover; a reasonable solver will answer yes, meaning the constraint is satisfiable, with a solution such as

$$[k = 1, k' = 30]$$

Hence, we have successfully identified that different secrets (e.g., 1 and 30) can lead to the access of different cache lines at line 9 of the sample code. Actually, by feeding different secrets (1 or 30) to this function, memory access in the above case hits cache line 0 or 1, which enables attackers to launch cache probing attacks (e.g., prime-and-probe attacks [42, 53, 11, 63, 34]) to infer the value of the secret.

Another obvious secret dependent memory access is at line 10, which holds the memory access formula (also for the first iteration of the loop) as:

$$G(k) \equiv 10 + 4 \cdot (k \bmod 128) \bmod 4$$

According to constraint solving, $G(k) \gg 6 \neq G(k') \gg 6$ is unsatisfiable at this time. That means, memory access at line 10 always access the same cache line, and hence, is immune to cache-probing attacks.

3.3 Scope and Limitations

Trace-based Analysis. CacheD is designed to analyze execution traces of program executables. In general, low-level analysis (such as analysis towards the execution trace) is capable of capturing pitfalls or vulnerabilities that are mostly ignored by analyzing the source code [5]. In addition, since the inputs to CacheD are execution traces generated from program executables, CacheD is also capable of identifying vulnerabilities introduced by compiler optimizations or even commonly used obfuscations without additional efforts. We take execution traces as the input for CacheD because whole-binary symbolic execution is mostly considered unscalable, even through trace-based analysis loses some generality for only analyzing one or several execution paths. Moreover, since we only keep symbols derived from the secret, pointers which do not contain symbols can be updated with concrete values acquired from the execution trace.

Main Audiences. The main audiences of our work are software developers: developers can use CacheD to "debug" their software (through execution traces) and identify vulnerable program points that may lead to cache-based timing attacks. Previously, finding such vulnerabilities are challenging—if possible at all—towards industrial-strength cryptosystems.

The trace-based analysis is usually unable to cover all program points; in other words, to produce execution traces that can cover the vulnerable code, it might require deliberate selection of proper program inputs to trigger the vulnerability. Although this coverage issue is unavoidable in general, we assume developers themselves would be able to construct proper program inputs and provide critical execution traces to CacheD. There are also techniques, such as concolic testing [49, 22, 23], developed in the software testing and verification community that can be leveraged.

On the other hand, considering the research objective in this paper (i.e., cryptosystems), most critical procedures (where vulnerabilities could exist) can indeed be triggered by following the standard routines defined by the cryptographic libraries. Evaluation details of our work are presented in §7.

Adoption of Constraint Solver. In practice, searching for different secret values that lead to different cache behaviors is very complex and thus difficult for developers without resort to rigorous tools. For example, the big and complex formula shown in Appendix A is almost impossible for developers to deduce a solution. Symbolic execution is considerably more precise than traditional data-flow analysis, and when constraint solver finds a solution for memory accessing formula, it naturally provides counter examples that lead to the variant cache accesses, making it easier for developers to reveal underling issues in their software.

Soundness vs. Precision. CacheD is not sound in the terminology of program analysis; that is, when CacheD reports no vulnerability, it does not mean the program under examination is free of cache-based side-channel attacks. On the other hand, CacheD is quite precise with few false positives. According to our threat model, false positives only occur in scenarios such as if the symbolic memory model is not precise enough. Constraint solving will not introduce false positives as a positive solution is really satisfiable for the formula, but it might miss positives. In practice, our evaluation also reports consistent findings that positive cases studied in the hardware simulator can surely lead to cache line access variance (details are given in §7.3). We actually have not encountered any false positives

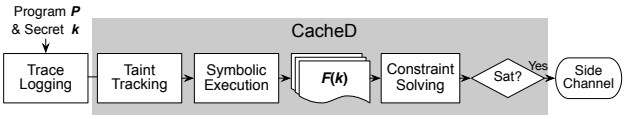

Figure 3: The architecture of CacheD.

in our evaluation. As previously discussed, existing research [19, 20] aims at reasoning the "upper-bound" of information leakage through abstract interpretation, but may not be precise enough due to over-approximation. Moreover, CacheD distinguishes itself by being able to point out where/what the vulnerability is, and provide examples that are likely to trigger the issue. Considering CacheD as a "debugging" or vulnerability detection tool, it is equally important to adopt its precise and practical techniques on side-channel detection.

4 Design

We present CacheD, a tool that delivers scalable detection of cache-based timing channels in real-world cryptosystems. Fig. 3 shows the architecture of CacheD. In general, given a binary executable with secrets as inputs, we first get a concrete execution trace by monitoring its execution (§4.1). The trace is then fed into CacheD to perform taint analysis; we mark the secret as the taint seed (§4.2) and propagate the taint information along the trace to identify instructions related to the usage of the secret.

CacheD then symbolizes the secret into one or several symbols (each symbol represents one word), and performs symbolic execution along the tainted instructions on the trace (§4.3). During the symbolic interpretation, CacheD builds symbolic formulas for each memory access along the trace. Symbolic memory access formulas are further analyzed using a constraint solver to check whether cache behavior variations exist. As aforementioned, we check the satisfiability of $(F(\vec{k}) \gg L \neq F(\vec{k'}) \gg L) \ \wedge \ C$; if satisfiable, the solution to $\vec{k}$ and $\vec{k'}$ represent different secret values that can lead to different cache behavior of this program point. The only architecture-specific parameter to CacheD is the cache line size. As discussed in §2.1, we set L to be 6 throughout this paper since most CPUs on the market sets have a cache-line size of 64. Next, we elaborate on challenges and design of each step in the following sections.

4.1 Execution Trace Generation

CacheD takes a concrete program execution trace as its input. In general, the execution trace can be generated by employing dynamic instrumentation tools to monitor the execution of the target program and dump the execution trace. We assume the instrumentation tools also dump the context information (including the value of every register) of *every* executed instruction as well.

Locating Secrets in the Trace. Besides the dumped execution trace and the context information, another input of CacheD is the locations (e.g., a memory location or a register) of the secrets in a program. This information serves as the seed for the taint analysis and symbolic execution in later stages.

While the secrets (e.g., the private key or a random number) are usually obvious in the source code, it may not be straightforward to identify the location of the secret in an execution trace, since variable names are absent in the assembly code. Treating this as a typical (manual) reverse engineering task, our approach to searching for the secrets in the assembly is to "correlate" memory reads with the usage of the key in the source code. To do so, we identify the critical function in the source code where the key is initialized and then search for the function in the assembly code. The search space can be further reduced by cutting the assembly code into small regions according to the conditional jumps in the context. With further reverse engineering effort in small regions, we can eventually recognize the location of the secret in the assembly code, as a register, or a sequence of memory cells in the memory.

Although currently this step is largely manual, it is likely that it can be automated by a secret-aware compiler, which tracks the location of secrets throughout the compilation; however, we leave this as future work.

4.2 Taint Analysis

CacheD leverages symbolic execution to interpret each instruction along a trace to reason about memory accesses that are dependent on secrets. Our tentative tests show that the symbolic-level interpretation is one performance bottleneck of CacheD. However, we notice that only a subset of instructions in a trace is dependent on the secrets. Thus, a natural optimization in our context is to leverage taint analysis to rule out instructions that are irrelevant to the secret; the remaining instructions are the focus of the more heavy-weight symbolic execution.

After reading the execution trace, CacheD first parses the instructions into its internal representations. It then starts the taint analysis from the first usage of the secret. Following existing taint analyses (e.g., [48, 52]), we propagate the taint information along the trace following pre-defined tainting rules that we discuss shortly. After the taint analysis, we keep the instructions whose operands are tainted.

Taint propagation rules define how tainted information flows through instructions, memories and CPU flags, as

well as what operations introduce new taint or remove existing taint. Well-defined propagation rules should not miss needed taint propagation, while keeping the set of tainted memory cells as small as possible to reduce the overhead of further heavy-weight analysis (i.e., symbolic execution in our context). Considering the context of cryptosystems, we now propose our taint propagation rules as follows.

Taint Propagation for Registers. The propagation rules for register-level operations are very straightforward. In general, if a tainted variable flows to an untainted one, then the latter will be tainted. On the other hand, we delete the taint label on the information flow destination if the source is not tainted.

Taint Propagation for Memory-related Operations. We now define the tainting rules for memory operations. CacheD tracks the taint information for each memory cell. More specifically, the taint module of CacheD keeps a set, where each element is the address of tainted memory cell. Taint operation inserts new elements into this list, while untaint operation deletes the corresponding element. Recall that we dump the context information for each executed instruction (§4.1). For each memory access, we compute the address through the concrete register values recorded in the context information. Hence, the memory address is always a *concrete* value and memory-related taint propagation is considered accurate.

Memory addressing defined in the x86 instruction set can be divided into the base address and the memory offset, each of which is maintained by a register or a concrete value. Our tainting rule defines that whenever the registers representing either the base address or the offset are tainted, we would propagate the taint to the contents of the accessed memory cells. Our tainting rules are reasonable and avoid under-tainting, since in general the secret content can be used as memory pointers (representing base addresses) as well as memory offsets.

Taint Propagation for CPU Flags. In x86 instruction set, CPU flags participant in the computation of many instructions and are also used to select branches. To precisely track the secret information flow, CacheD record taint propagations towards CPU flags.

In general, CPU flags could be modified according to the computation results of certain instructions, for example, flag `ZF` will be set to one if the result of an `SUB` (subtraction) operation is zero. In case any operand of a given instruction is tainted, we taint all CPU flags that can be affected by the current computation. In addition, taint label can also be propagated from CPU flags to registers or memory cells; we taint registers or memory cells that hold the computation result of an instruction whenever tainted CPU flags participant the computation.

4.3 Symbolic Execution

We now introduce how we build the symbolic execution module of CacheD. As previously mentioned (§4.2), tainted instructions (i.e., instructions whose operands are tainted) are kept after taint analysis. These instructions, together with their associated context information, are passed to the symbolic execution module; the location of the secret is another input of symbolic execution. The symbolic execution engine starts the interpretation at the beginning of the first tainted instruction (i.e., the first usage of the secret) and interprets each instruction until the trace end.

Symbolization of the Secret. In general, secrets (e.g., private key) can be maintained as a variable (as shown in Fig. 2), an array, or a compound data structure. Note that only the *content* of the secret (e.g., the value of a private key) is considered as "secret" in our context.

If the secret is maintained as one variable (e.g., one register or a memory cell on the stack), it is straightforward for symbolization. On the other hand, if the secret is stored in a sequence of memory cells (e.g., one array, structure, or class instance), CacheD assigns the base address (provided by programmers in previous stage §4.1) to a special symbol. Further memory reads using this special symbol as the base address is considered to access the secret content. CacheD generates a fresh symbol (for simplicity's sake, we name such symbol as *key symbol*) each time when the memory read has a different offset (since it indicates a different part of the secret memory region is visited).

Design of the Symbolic Execution Engine. As aforementioned (§3.1), we collect all the conditions (i.e., some formulas evolving CPU flags) of the branches along the trace and conjunct them into the path condition. Since the program execution trace can be effectively viewed as the static single assignment (SSA) format, the path condition is accumulated along the trace and it must be always true at any execution point (otherwise the execution trace is invalid). Our side-channel checking is performed at every memory access. When encountering a memory access, CacheD pauses the symbolic execution engine and sends the memory access formula as well as the currently-collected path condition to the solver. Consistent with our taint propagation rules which captures information flow through memory accesses (§4.2), for a memory load operation whose addressing formula containing key symbols (i.e., either the base

address or the memory offset is computed from secrets), we would symbolize the memory cell with a fresh key symbol if it is visited for the first time.

Symbolic Execution Memory Model. Symbolic execution interprets programs with logical formulas instead of concrete values so that the semantics captured are not specific to a single input. However, some program semantics are difficult to analyze when the information flow is encoded symbolically, such as dereferencing a symbolic pointer. In general, when a symbolically executed program reads from the memory using an abstract (symbolized) address, the execution engine needs to decide the content read from the address. On the other hand, when the program writes to the memory using an abstract address, the engine needs to decide how to update the memory status. The policy that specifies those aspects is called a memory model.

When designing a symbolic execution engine, the trade-off between scalability and precision should be carefully considered. That is, we cannot employ a full-fledged memory model that features abstract memory chunks, since our tentative test shows that such memory model does not scale for the real-world applications. Instead, our current design develop a memory model that reasons symbolic pointers with their concrete values on the trace, which is conceptually the same as other commonly used binary analysis platforms (e.g., the trace-based analysis of BAP [14]).

5 Optimization

While taint analysis is efficient, symbolic execution and constraint solving are time consuming in general. Here we discuss several optimizations in CacheD.

Identify Independent Vulnerabilities. To capture information flow through memory operations in symbolic execution, we create a fresh key symbol for a memory load of unknown positions whenever the base or memory offset is computed from key symbols. In this section, we propose a finer-grained policy, which reveals "independent" vulnerable program points. The key motivation is that, by studying the underlying memory layout, attackers would be able to learn relations between the newly-created key symbol and the memory addressing formula (which contains one or several key symbols). Hence, we assume further vulnerabilities revealed through the usage of this new key symbol would mostly leak the same piece (or a subset) of secret information (we elaborate on this design choice shortly).

We now present an example to motivate this optimization. In general, for a memory load operation

$$load\ reg,\ [F(\vec{k})]$$

where $F(\vec{k})$ is the memory addressing formula through the secret $\vec{k}$, and *reg* stores the loaded content from the memory. There exist three different cases regarding the solution of our constraint solver:

- To test whether the array index, and hence the fetched content, may differ in two executions with different keys, CacheD checks the formula $(F(\vec{k}) \neq F(\vec{k'})) \wedge C$. If there is no satisfiable solution for this formula, we interpret this memory access is *independent* of the key. Thus, there is no need to create a fresh key symbol; we update the memory load output (i.e., *reg* in the above case) with concrete value from the trace.
- If there exist satisfiable solutions for $(F(\vec{k}) \gg L \neq F(\vec{k'}) \gg L) \wedge C$, it means we find an *independent vulnerable program point*. As discussed above, further vulnerable program points discovered through the newly created key symbol (stored in *reg*) would likely leak the same piece of secret information as this vulnerability, thus "depending" on this point.
- The remaining case is that there is no satisfiable solution for $(F(\vec{k}) \gg L \neq F(\vec{k'}) \gg L) \wedge C$ while there exists solutions for $(F(\vec{k}) \neq F(\vec{k'})) \wedge C$. In other words, while the current memory access does not reveal a vulnerability, still, different secrets would lead to the access of different memory cells, which constructs an information flow. Hence, we create a fresh key symbol and use it to update the memory load output.

In general, we consider independent vulnerabilities are highly informative to attackers; independent vulnerabilities probably indicate the most-likely attack surface of the victim, because stealing secret through "dependent" vulnerabilities need additional efforts to learn the program memory layout. On the other hand, memory layouts are feasible and likely to be learned as precomputed data structures are widely deployed in real-world cryptosystems to speed up the computation. Overall, "dependent" vulnerabilities reveal an additional attack surface which are commonly ignored by previous research.

Early Stop Criterion of Symbolic Execution. One vulnerable program point (e.g., a table query) can be executed for one or more times during the runtime (thus appearing more than once on the execution trace). On the other hand, a program point can be considered as "vulnerable" as long as one of its usage is

confirmed vulnerable. Hence, while in general CacheD symbolically interpret all the tainted instructions, one early stop criterion adopted in CacheD is that we have already identified at least one vulnerable memory access for *any* tainted access relating to the same program point.

Domain knowledge of RSA and ElGamal implementation. As previously discussed, our taint propagation rule would taint the visited memory cells whenever registers hold the base address or memory offset are tainted. While this rule reasonably captures the information flow through memory accesses, we terminate the taint propagation for one specific case, given the domain knowledge of cryptosystems being analyzed.

To speed up processing, the sliding-window based modular exponentiation algorithm leverages a two-level "precomputed table" to store the modular exponentiation values of the base regarding some predefined window-sized value. Additionally, a precomputed size table is deployed to store the length of each precomputed modular exponentiation value. Hence, most of the computation are substituted into two table lookups towards the precomputed table and the size table through the window size key. Appendix B presents the structure of a two-level precomputed table used in Libgcrypt 1.6.1. Each element in the first-level array stores a pointer referring to the second level, and each second level array stores a big integer (b^v for some window-sized value v).

Our study of such tables shows that for non-trivial decryptions (e.g., decrypt an encrypted message of one character), the lengths of all the second-level arrays are equal to N (we observed that N is fixed to 32 for RSA while 64 for ElGamal implementations evaluated in §7). Hence, elements in the precomputed size table are identical and the attacker would observe the same output regardless of the secret input. In other words, it is reasonable to assume secrets can hardly be leaked by observing the table query outputs. Given such observation, CacheD is optimized to terminate the taint propagation towards the precomputed size table.

Trace Cut. CacheD is designed to analyze any fragment of program execution trace, with a tradeoff of performance and coverage. Ideally, we would like to analyze the entire trace from the program entry point until the end. With taint analysis (§4.2), the analysis effectively starts from the beginning of the function where the key is used for the first time. Besides, for the RSA and ElGamal decryption, where the secret key is used for multiple operations, we choose only critical procedures (i.e., functions implemented the modular exponentiation operation and their callees) that have been the target for various timing attacks. Analyzing the same procedure which has been well-studied from different angles in the literature makes it easier to compare our experiment results (in terms of re-discover existing issue and identify unknown issue) with existing work. On the other hand, there is no issue for CacheD to analyze other standard computation procedures.

Table 1: Cryptosystems analyzed by CacheD.

Algorithm	Implementation	Versions
RSA	Libgcrypt [32]	1.6.1, 1.7.3
	OpenSSL [40]	0.9.7c, 1.0.2f
	Botan [35]	1.10.13
ElGamal	Libgcrypt [32]	1.6.1, 1.7.3
AES	OpenSSL [40]	0.9.7c, 1.0.2f

6 Implementation

CacheD is implemented in Scala, with over 4,800 lines of code. The program execution trace is generated by Pin [36], a widely-used dynamic binary instrumentation tool. Pin provides infrastructures to intercept and instrument the execution of a binary. During execution, Pin inserts the instrumentation code into the original code and recompiles the output leveraging a Just-In-Time (JIT) compiler. We develop a plugin of Pin (162 lines of C++ code) to log the executed instructions as well as the context information during the execution. While our current implementation (including CacheD and the Pin plugin) analyzes binaries on the 32-bit Linux platforms (i.e., binaries with the ELF format), we emphasize that the proposed technique is mostly independent with the underlying architecture details, and hence not difficult to port to other platforms (e.g., Windows or 64-bit Linux).

CacheD leverages the widely-used constraint solver Z3 [18] for constraint solving (Z3 provides Java API, which bridges Z3 solver to our Scala code). In addition, we leverage bit vectors provided by Z3 to represent the taint label of each general-purpose register as well as symbols used in symbolic execution. Note that x86 instructions can manipulate the subset of each register, and benefit from bit vectors, arbitrary operations on the subset of each general-purpose register are supported without additional effort. As aforementioned, we track the taint towards CPU flags; a vector of one bit is created to represent each CPU flag.

7 Evaluation

We evaluate CacheD on several real-world cryptographic libraries. The cryptosystems used in our evaluation are listed in Table 1. In sum, we evaluated CacheD on five real-world cryptosystems in total, including nine different implementations of three cryptographic algorithms, RSA, AES, and ElGamal.

Table 2: Evaluation results of different cryptographic algorithm implementations.

Algorithm	Implementation	Adopt the "Domain Knowledge of RSA and Elgamal" Optimization (§6)	Vulnerable Program Points (known/unknown)	Independent Vulnerable Points (known/unknown)	# of Instructions on the Traces	Processing Time (CPU Seconds)
RSA	Libgcrypt 1.6.1	✓	2/20	2/0	26,848,103	11542.3
RSA	Libgcrypt 1.7.3	✓	0/0	NA	27,775,053	10788.9
ElGamal	Libgcrypt 1.6.1	✓	2/19	2/0	31,077,760	17044.8
ElGamal	Libgcrypt 1.7.3	✓	0/0	NA	31,407,882	12463.1
RSA	OpenSSL 0.9.7c	×	0/2	0/1	674,797	199.3
RSA	OpenSSL 1.0.2f	×	0/2	0/1	473,392	165.6
AES	OpenSSL 0.9.7c	×	48/0	48/0	791	43.4
AES	OpenSSL 1.0.2f	×	32/0	32/0	2,410	48.5
RSA	Botan 1.10.13	✓	0/29	0/2	2,005,124	7527.0
Total			84/72	84/4	120,265,312	59822.9

Experiment setup. All the cryptosystems are C/C++ libraries. We write simple programs to invoke the test libraries for key generation, encryption as well as decryption. We generate keys of 128 bits for AES experiments, and keys of 2048 bits for other experiments. After generating the keys, for all test cryptographic algorithms, we first use their encryption routines to encrypt a plain text "hello world". The encrypted message is then fed into the decryption procedures. As previously introduced, the execution traces of those decryption procedures are logged for analysis. The programs are compiled into binary code on 32-bit Ubuntu 12.04, with gcc/g++ compiler (version 4.6.3).

7.1 Evaluation Result Overview

Vulnerability Identification. We present the breakdown of the positives reported by CacheD in Table 2. As shown in the table, most of the evaluated implementations are reported to contain vulnerabilities that can lead to cache-based side-channel attacks. Overall, CacheD reveal 156 (84 known and 72 unknown) vulnerable program points, among which 88 (84 known and 4 unknown) program points are independent. Considering the large number of issues discovered by CacheD, we interpret the evaluation result as promising.

In general, existing research has pointed out potential issues that can lead to the cache based side-channel attacks on the implementation of sliding-window based modular exponentiation [20], and such implementation is leveraged by both RSA and ElGamal decryption procedures. In this research CacheD has successfully confirmed such already-reported issues. We present a detailed study of two independent vulnerable program points found in RSA implementation of Libgcrypt 1.6.1 in §7.3, and also compare our findings of RSA and ElGamal with existing literatures in §7.4.1. Besides, considering its multiple rounds of table lookup, AES has also been pointed out as vulnerable in terms of cache-based side channel attacks by previous work [15]. CacheD reports consistent findings in §7.4.2.

Moreover, CacheD has also successfully identified a number of vulnerable program points in two widely-used cryptosystems (Botan and OpenSSL). Those vulnerabilities, to the best of our knowledge, are unknown to existing research (the "unknown" issues). We elaborate on these identified issues in §7.5.

We also evaluate CacheD towards the RSA and ElGamal implementations of Libgcrypt 1.7.3, which are considered as safe from information leakage since there is no secret-dependent memory access. CacheD reports no vulnerable program point in both the ElGamal and RSA implementations. Indeed, taint analysis of CacheD identifies **zero** secret-dependent memory access in both implementations. Although trace-based analysis is in general not sufficient to "prove" a cryptosystem as free of information leakage, considering related research as less scalable ([19]), CacheD presents a scalable and practical way to study such industrial strength cryptosystems.

Processing Time. We also report the processing time of CacheD in Table 2. The experiments use a machine with a 2.90GHz Intel Xeon(R) E5-2690 CPU and 128GB memory. Table 2 presents the number of processed instructions and the processing time for each experiment. In general, we evaluate CacheD regarding five industrial-strength cryptosystems, with over **120 million** instructions in total. Table 2 shows that all the experiments can be finished within 5 CPU hours. We interpret the processing time of CacheD as very promising, and this evaluation faithfully demonstrates the high scalability of CacheD in terms of real-world cryptosystems.

Our evaluation also shows the proposed optimizations (§5) are effective, which surely improve the overall scalability of CacheD. Indeed tentative implementation of CacheD (without optimizations) times out after 20 hours to process the ElGamal Libgcrypt 1.6.1 test case. On the other hand, we observed that CacheD becomes slower largely due to the nature of symbolic execution; with more symbolic variables and formulas carried on, every further reasoning can take more time. In addition, since symbolic formulas can grow larger during interpretation (e.g., variables are manipulated for iterations in a loop), solver could probably encounter more challenging problems during further constraint solving. Also, branch con-

Table 3: Gem5 configurations.

ISA	x86
Processor type	single core, out-of-order
L1 Cache	4-way, 32KB, 2-cycle latency
L2 Cache	8-way, 1MB, 50-cycle latency
Cache line size	64 Bytes
Cache replacement policy	LRU

Table 4: Results of executing test cases under gem5.

Algorithm	Implementation	Observe the Access of Different Cache Lines	Observe Different Cache Status (hit vs. miss)
RSA	Libgcrypt 1.6.1	✓	✓
ElGamal	Libgcrypt 1.6.1	✓	✓
RSA	OpenSSL 0.9.7c	✓	✓
RSA	OpenSSL 1.0.2f	✓	
AES	OpenSSL 0.9.7c	✓	✓
AES	OpenSSL 1.0.2f	✓	
RSA	Botan 1.10.13	✓	✓

ditions are accumulated along the trace; more constraints need to be solved, which leads to performance penalties as well.

7.2 Exploring the Positives

To study whether the positives can lead to real cache difference during execution, we employ a commonly-used computer architecture simulator—gem5 [9]—to check the identified vulnerable program points. As previously discussed (§5), independent vulnerable program points (88 in total) are considered as mostly informative; "dependent" vulnerable program points would mostly leak the same piece of information as independent ones. Hence in this step, we focus on the check of independent vulnerable program points. While CacheD identifies 8 independent vulnerabilities in the RSA and ElGamal implementations, 80 program points are reported as vulnerable in two AES implementations. Without losing generality, we check all the independent vulnerabilities for RSA and ElGamal, while only checking the first four vulnerabilities for these two AES implementations.

As aforementioned, for each vulnerable program point, the constraint solver provides at least one satisfiable solution (i.e., a pair of $\vec{k}$ and $\vec{k}'$) that leads to the access to different cache lines. Hence, for *each* vulnerable program point, we instrument the source code of the corresponding test program to modify secrets with $\vec{k}$ and $\vec{k}'$; we then compile the source code into two binaries. We monitor the execution of instrumented binaries using the full-system simulation mode of gem5, and intercept cache access from CPU to L1 Data Cache. The full-system simulation uses Ubuntu 12.04 with kernel version 3.2.1.[2] Table 3 presents the configurations.

[2]The full-system simulation mode of gem5 only supports 64-bit kernels. Also, we compiled the instrumented source code into 64-bit binaries since the simulated OS threw some TLB translation exceptions when executing 32-bit binaries.

Results. When executing each vulnerable program point (i.e., a memory access), we record the visited cache line as well as the cache status of this cache line. Table 4 present the results. By comparing cache traffic of executing binaries with secret $\vec{k}$ or $\vec{k}'$, we have confirmed that memory accesses at **all** vulnerable program points indeed visited different cache lines. We have also confirmed that cache statuses are different at the vulnerable program points for most of the test cases.

There are two test cases (row 5 and 7 in Table 4) that show identical cache status at the vulnerable program points. Note that we only record cache status at the memory access of these vulnerable points (some examples are given shortly in Fig. 4c); it is likely that the accesses to different cache lines actually lead to cache behavior variations during further program execution. On the other hand, given our current conservative observation, still, most of the test cases reveal noticeable cache difference. We will present detailed study of the RSA Libgcrypt 1.6.1 (row 2 in Table 4) in §7.3.

In general, we consider the evaluation results as quite promising; while previous work (e.g., [19]) performs overall reasoning of the program information leakage upper bound and lack of information on what/where the vulnerability is, CacheD fills the gap by providing concrete examples to trigger cache behavior variations at its discovered program points.

7.3 Case Study of RSA Vulnerabilities

In this section, we present a case study of two identified vulnerable program points, with detailed explanation in terms of the source code patterns as well as hardware simulation results.

As presented in Table 2, we identified two independent vulnerable program points in the RSA implementation of Libgcrypt 1.6.1. Source code shown in Fig. 4a is found in the sliding-window implementation of the modular exponentiation algorithm; we have confirmed that two identified independent vulnerable program points represent table queries at line 13 and 14. Indeed, `e` is an element of the secret array (line 5; thus `e` is secret), and `e0` is a sliding window of `e` (line 9). `e0` is used to access the first level of the precomputed table (line 13) and precomputed size table (line 14). Intuitively, different `e0` accesses different table entries, which potentially leads to different cache line and eventually leaks the secret (i.e., `e`).

When analyzing the execution trace, CacheD successfully identified two secret-dependent memory accesses (line 4-5 in Fig. 4b), and by inquiring the constraint solver, CacheD finds two pairs of `e` that can lead to the access of different cache lines for the first and second memory accesses, respectively (the "solutions" in

```
1  void _gcry_mpi_powm (gcry_mpi_t res,
2    gcry_mpi_t base,
3    gcry_mpi_t expo, gcry_mpi_t mod) {
4    ...
5    e = ep[i];
6    count_leading_zeros (c, e);
7    e = (e << c) << 1;
8    ...
9    e0 = (e >> (BITS_PER_MPI_LIMB - W));
10   count_trailing_zeros (c0, e0);
11   e0 = (e0 >> c0) >> 1;
12   ...
13   base_u = b_2i3[e0 - 1];
14   base_u_size = b_2i3size[e0 - 1];
15   ...
16 }
```

(a) Source code.

```
1  ...
2  mov edx, [esp+0x38]
3  sub edx, 0x1
4  mov ecx, [esp+edx*4+0xb8]
5  mov edx, [esp+edx*4+0xf4]
6  mov esi, [esp+0x24]
7  ...

1  Solution of the first mem access (line 4):
2  e = 0x10000016
3  e'= 0x1000003e
4
5  Solution of the second mem access (line 5):
6  e = 0x400500d
7  e'= 0x1007c
```

(b) Instructions on the dumped trace and solutions provided by the solver.

```
1  0x44156a: 0x7fffffffe440 hit
2  0x441572: 0x7fffffffe3c8 hit
3  0x44156a: 0x7fffffffe438 hit
4  0x441572: 0x7fffffffe3c4 hit
5  0x44156a: 0x7fffffffe3d8 hit
6  0x441572: 0x7fffffffe394 hit
7  0x44156a: 0x7fffffffe408 hit
8  0x441572: 0x7fffffffe3ac hit

1  0x44156a: 0x7fffffffe3f0 hit
2  0x441572: 0x7fffffffe3a0 hit
3  0x44156a: 0x7fffffffe438 hit
4  0x441572: 0x7fffffffe3c4 hit
5  0x44156a: 0x7fffffffe3d8 miss
6  0x441572: 0x7fffffffe394 miss
7  0x44156a: 0x7fffffffe408 miss
8  0x441572: 0x7fffffffe3ac hit
```

(c) Hardware simulation results for the first memory access (e is 0x10000016 and e' is 0x1000003e).

Figure 4: Case study of two independent RSA vulnerable program points in Libgcrypt 1.6.1. The vulnerable program points and the corresponding memory access instructions on the trace are **bold**. The tainted variable in source code and trace are red, and e is secret (line 5 in Fig. 4a). Note that base_u_size is not tainted regarding the optimization of RSA precomputed size table (§5).

Fig. 4b). To confirm the findings, we compile four program binaries with modified e regarding the solutions.

Fig. 4c shows the simulation outputs using gem5. Due to the limited space, we only provide the first eight records for the first vulnerable program point (there are 604 records in total). The first column of each output represents the program counters; the second column shows the accessed memory addresses and the last column is the cache statuses of the accessed cache lines. Note that program counter 0x44156a and 0x441572 represent the first and second vulnerable program points, respectively. Comparing these two results, we can observe that different cache lines are accessed (corresponding memory addresses are marked as red at line 1-2), which further leads to timing difference of three cache hit vs. miss (corresponding cache statuses are marked as red at line 5-7). Simulation results for the second memory access are omitted due to the limited space. We report to have similar observations.

Consistent with the existing findings, these two table queries are also reported as vulnerable by previous work [20]. According to our taint policy, the table query output (base_u) would be tainted since e0 is tainted. Our study also shows that memory access through base_u would reveal another twenty vulnerable program points (row 2 in Table 2). Moreover, this modular exponentiation function is used by both RSA and ElGamal decryption procedures; two independent vulnerable program points found in the ElGamal implementation (row 4 in Table 2) are also due to these table queries.

7.4 Known Vulnerabilities

We also evaluate CacheD by confirming known side-channel vulnerabilities.

7.4.1 RSA and ElGamal in Libgcrypt

Function _gcry_mpi_powm (Fig. 4a) found in the Libgcrypt (1.6.1) sliding-window implementation of the modular exponentiation algorithm is vulnerable. Note that such implementation indeed is used by both RSA and ElGamal decryption procedures. We have already presented results and a case study in §7.3. Besides those two independent program points, CacheD finds 19 vulnerable program points in the ElGamal implementation and 20 points in the RSA implementation Table 2.

Consistent with previous work [20], CacheD confirms two vulnerable program points in the Libgcrypt (1.6.1) that can lead to cache-based timing attacks. On the other hand, while previous work [20] only reports potential timing channels through these two direct usage of secrets, CacheD can actually detect further unknown (to the best of our knowledge) pitfalls (around 20 unknown points for each). The results show that CacheD can provide developers with more comprehensive information regarding side-channel issues.

7.4.2 AES in OpenSSL

We also analyzed the positive results identified in AES implementations of OpenSSL (version 0.9.7c and 1.0.2f). In general, standard AES decryption undertakes

a three-step approach for decryption, in which the second and third steps consist of (multiple rounds) lookup table queries through blocks of secrets. Intuitively, such secret-dependent table queries could reveal considerable amount of timing-channel vulnerabilities.

Our evaluation has confirmed this intuition. CacheD successfully identifies 48 vulnerable program points for OpenSSL (0.9.7c). Indeed all of the identified program points are lookup table queries through secrets, which is consistent with previous research [15]. Analysis of another OpenSSL (1.0.2f) gave similar results: CacheD identifies 32 vulnerable program points of secret-dependent lookup table queries (Table 2).

7.5 Unknown Vulnerabilities

CacheD also successfully identifies several potential vulnerabilities that have not been reported in public, to the best of our knowledge.

7.5.1 RSA in OpenSSL

CacheD reported two positive results in each OpenSSL implementation (version 0.9.7c and version 1.0.2f) of the RSA decryption procedure. CacheD further identified one independent vulnerable point for each implementation. Appendix C presents the source code in which the independent positive is discovered. Before performing the modular exponentiation, function `BN_num_bits` calculates the length of the secret key by bit. The secret key information is represented by a `BIGNUM` structure pointed by `a`, with the key value stored in a byte buffer `a->d` and the length of the buffer stored in `a->top`, respectively. Since the key length by bit may not be a multiple of the key length by byte, the code uses a lookup table in `BN_num_bits_word` to determine the exact number of bits in the last entry of `a->d`. CacheD points out that accessing this lookup table will lead to a cache difference, thus leaking information about the most significant several bits of the secrete key which are stored in `a->d[a->top - 1]`. Results in §7.2 also support our finding. In addition, CacheD also identified another vulnerable program point which is derived from the output of this table query (row 6-7 in Table 2).

We also find the same implementation that could lead to timing channels in its most recent releases (released in late Sep. 2016): version 1.0.2j, version 1.1.0b, and version 1.0.1u.

7.5.2 RSA Implementation in Botan

Another vulnerability found by CacheD is in the Botan (1.10.13) implementation of RSA, whose source code is shown in Appendix D. The Montgomery exponentiator is an algorithm for modular exponentiation. Similar to the Libgcrypt (1.6.1) implementation of RSA (Fig. 4a), a precomputed table is employed to cache some intermediate results and a sliding window of the secret key is used to query the table (line 9). The queried output is maintained as a `BigInt` class instance and it is represented as a symbol of the key according to our taint propagation rules (§4.3). Later, when the class method `sig_words` is invoked (line 13), two memory accesses (line 19-20) through the key symbol are captured by CacheD (note that `reg` at line 19 is a private variable of class `BigInt`).

Our constraint solver has proved that there are multiple satisfiable solutions for both the first and the second memory access (line 19-20). Moreover, by employing different secrets provided by the solver, we report to observe cache behavior variations in the hardware simulator (§7.2). In addition, the memory query output (`x` at line 19) is used to access memories later, which results into 27 "dependent" vulnerable program points (row 10 in Table 2).

Besides the implementations evaluated in this work (version 1.10.13), we notice that this vulnerability affects several other versions of Botan, including 1.10.12, 1.10.11, and 1.11.33.

8 Related Work

8.1 Timing attacks

One major motivation for controlling timing channels is the protection of cryptographic keys against side-channels arising from timing attacks. Since the seminal paper of Kocher [30], attacks that exploit timing channels have been demonstrated on RSA [13, 30, 3, 45, 59], AES [24, 42, 53] and ElGamal [63].

Shared data cache is shown to be a rich source of timing channels. The potential risk of cache-based timing channels was first identified by Hu [26]. Around 2005, real cache-based timing attacks are demonstrated on AES [8, 42], and RSA [45]. Since then, more practical timing attacks are emerging. Previous work shows the practicality of various timing attacks utilizing the shared data: among VMs in multi-tenant cloud [50]. Timing attacks are shown to be a potential risk across VMs [57, 56, 47], and more evidence is emerging showing practical timing attacks that break crypto systems [63, 59, 34]. Recent work [41] presents a successful cache attack where the victim merely has to access a website owned by the attacker.

8.2 Mitigation of cache-based timing channels

Much prior application-level mitigation only handles timing leakage due to secret-dependent control flows [4, 25, 38, 7, 17, 44]. However, as shown in recent cache-based timing attacks [24, 42, 53, 59, 34], subtle timing leakage survives even with the absence of secret-dependent control flows. Recently, advanced program analyses are proposed to identify those subtle cache-based timing channels [6, 19, 20, 60], but they only provide an upper-bound on timing-based information leakage; it is unclear what/where the vulnerability is when those tools report a non-zero upper bound.

At the system level, Düppel [64] clears L1 and L2 cache before context switching; but it cannot mitigate the last-level cache-based attack, such as [34]. StealthMem [21, 29] manages a set of locked cache lines per core, which are never evicted from the cache. But its security relies on the assumption that "crucial" data was identified in the first place. But doing so nontrivial. For instance, the crucial data in AES is the lookup table, which only stores public data.

At the hardware level, one direction of mitigating cache-based timing channels is to either physically or logically partition the data cache [43, 54, 31, 61]. Line-locking cache was also implemented in hardware [54]. New hardware designs, such as RPCache [54], NewCache [55], and random fill cache [33], inject random noises to cache delay to confuse attackers. Common to those hardware-based mitigation mechanisms is the assumption that "crucial" data was identified by the software, where CacheD can be helpful.

9 Conclusion

To help developers improve the implementations of software that is sensitive to information leakage, we have developed a tool called CacheD to detect potential timing channels caused by the differences of cache behavior. With the help of symbolic execution techniques, CacheD models the memory addresses at each program point as logical formulas so that constraint solvers can check whether sensitive program data affects cache behavior, thus revealing potential timing channels. CacheD is scalable enough for analyzing real-world cryptosystems with decent accuracy. We have evaluated a prototype of CacheD with a set of widely used cryptographic algorithm implementations. CacheD is able to detect a considerable number of side-channel vulnerabilities, some of which are previously unknown to the public.

10 Acknowledgments

We thank the Usenix Security anonymous reviewers and Robert Smith for their valuable feedback. This research was supported in part by the National Science Foundation (NSF) under grants CNS-1223710 and CNS-1652790, and the Office of Naval Research (ONR) under grants N00014-16-1-2265 and N00014-16-1-2912.

References

[1] Aciiçmez, O., and Schindler, W. A vulnerability in RSA implementations due to instruction cache analysis and its demonstration on OpenSSL. In *Proceedings of the 2008 The Cryptopgraphers' Track at the RSA Conference on Topics in Cryptology* (2008), CT-RSA'08, pp. 256–273.

[2] Aciicmez, O., and Koc, C. K. Trace-driven cache attacks on AES. In *Proceedings of the 8th International Conference on Information and Communications Security* (2006), ICICS'06, pp. 112–121.

[3] Aciicmez, O., and Seifert, J.-P. Cheap hardware parallelism implies cheap security. In *Proceedings of the Workshop on Fault Diagnosis and Tolerance in Cryptography* (2007), pp. 80–91.

[4] Agat, J. Transforming out timing leaks. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2000), POPL '00, pp. 40–53.

[5] Balakrishnan, G., and Reps, T. WYSINWYX: What you see is not what you eXecute. *ACM Trans. Program. Lang. Syst. 32*, 6 (Aug. 2010), 23:1–23:84.

[6] Barthe, G., Betarte, G., Campo, J., Luna, C., and Pichardie, D. System-level non-interference for constant-time cryptography. In *Proc. ACM Conf. on Computer and Communications Security (CCS)* (2014), pp. 1267–1279.

[7] Barthe, G., Rezk, T., and Warnier, M. Preventing timing leaks through transactional branching instructions. *Electronic Notes in Theoretical Computer Science 153*, 2 (2006), 33–55.

[8] Bernstein, D. J. Cache-timing attacks on AES. https://cr.yp.to/antiforgery/cachetiming-20050414.pdf, 2005.

[9] Binkert, N., Beckmann, B., Black, G., Reinhardt, S. K., Saidi, A., Basu, A., Hestness, J., Hower, D. R., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N., Hill, M. D., and Wood, D. A. The Gem5 simulator. *SIGARCH Comput. Archit. News 39*, 2 (2011), 1–7.

[10] Bogdanov, A., Eisenbarth, T., Paar, C., and Wienecke, M. Differential cache-collision timing attacks on AES with applications to embedded CPUs. In *Proceedings of the 2010 International Conference on Topics in Cryptology* (Berlin, Heidelberg, 2010), CT-RSA'10, Springer-Verlag, pp. 235–251.

[11] Bonneau, J., and Mironov, I. Cache-collision timing attacks against AES. In *Cryptographic Hardware and Embedded Systems - CHES 2006*, L. Goubin and M. Matsui, Eds., vol. 4249 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2006, pp. 201–215.

[12] Bos, J., and Coster, M. Addition chain heuristics. In *Conference on the Theory and Application of Cryptology* (1989), Springer, pp. 400–407.

[13] Brumley, D., and Boneh, D. Remote timing attacks are practical. *Computer Networks* (Jan. 2005).

[14] BRUMLEY, D., JAGER, I., AVGERINOS, T., AND SCHWARTZ, E. J. BAP: A binary analysis platform. In *Proceedings of the 23rd international conference on computer aided verification* (2011), CAV '11, pp. 463–469.

[15] C, A., GIRI, R. P., AND MENEZES, B. Highly efficient algorithms for AES key retrieval in cache access attacks. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)* (2016), pp. 261–275.

[16] CHEN, S., WANG, R., WANG, X., AND ZHANG, K. Side-channel leaks in web applications: A reality today, a challenge tomorrow. In *Proceedings of the IEEE Symposium on Security and Privacy* (2010), S&P '10, pp. 191–206.

[17] COPPENS, B., VERBAUWHEDE, I., BOSSCHERE, K. D., AND SUTTER, B. D. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *Proceedings of the 30th IEEE Symposium on Security and Privacy* (2009), S&P '09, pp. 45–60.

[18] DE MOURA, L., AND BJØRNER, N. Z3: An efficient SMT solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (2008), TACAS'08.

[19] DOYCHEV, G., FELD, D., KOPF, B., MAUBORGNE, L., AND REINEKE, J. CacheAudit: A tool for the static analysis of cache side channels. In *Proceedings of the 22nd USENIX Security Symposium* (2013), pp. 431–446.

[20] DOYCHEV, G., AND KÖPF, B. Rigorous analysis of software countermeasures against cache attacks. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2017), PLDI '17, pp. 406–421.

[21] ERLINGSSON, Ú., AND ABADI, M. Operating system protection against side-channel attacks that exploit memory latency. Tech. Rep. MSR-TR-2007-117, Microsoft Research, August 2007.

[22] GODEFROID, P., KLARLUND, N., AND SEN, K. DART: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation* (2005), PLDI '05, pp. 213–223.

[23] GODEFROID, P., LEVIN, M. Y., AND MOLNAR., D. Automated whitebox fuzz testing. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium* (2008), NDSS '08, pp. 151–166.

[24] GULLASCH, D., BANGERTER, E., AND KRENN, S. Cache games—bringing access-based cache attacks on AES to practice. In *Proc. IEEE Symp. on Security and Privacy (S&P)* (2011), pp. 490–505.

[25] HEDIN, D., AND SANDS, D. Timing aware information flow security for a JavaCard-like bytecode. *Electronic Notes in Theoretical Computer Science 141*, 1 (2005), 163–182.

[26] HU, W.-M. Lattice scheduling and covert channels. In *Proceedings of the 1992 IEEE Symposium on Security and Privacy* (1992), S&P '92.

[27] IRAZOQUI, G., EISENBARTH, T., AND SUNAR, B. S$A: A shared cache attack that works across cores and defies VM sandboxing—and its application to AES. In *Proc. IEEE Symp. on Security and Privacy (S&P)* (2015), S&P '15, pp. 591–604.

[28] JANA, S., AND SHMATIKOV, V. Memento: Learning secrets from process footprints. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy* (2012), S&P '12, pp. 143–157.

[29] KIM, T., PEINADO, M., AND MAINAR-RUIZ, G. StealthMem: System-level protection against cache-based side channel attacks in the cloud. In *Proceedings of the 21st USENIX Conference on Security Symposium* (2012), pp. 189–204.

[30] KOCHER, P. C. *Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1996, pp. 104–113.

[31] LI, X., KASHYAP, V., OBERG, J. K., TIWARI, M., RAJARATHINAM, V. R., KASTNER, R., SHERWOOD, T., HARDEKOPF, B., AND CHONG, F. T. Sapper: A language for hardware-level security policy enforcement. In *Proc. 19th Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2014), pp. 97–112.

[32] Libgcrypt. https://www.gnu.org/software/libgcrypt/.

[33] LIU, F., AND LEE, R. B. Random fill cache architecture. In *Proc. 47th Annual IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)* (2014), pp. 203–215.

[34] LIU, F., YAROM, Y., GE, Q., HEISER, G., AND LEE, R. B. Last-level cache side-channel attacks are practical. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy* (2015), S&P '15, pp. 605–622.

[35] Botan. https://github.com/randombit/botan/.

[36] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (2005), PLDI '05, pp. 190–200.

[37] MESSERGES, T. S. *Using Second-Order Power Analysis to Attack DPA Resistant Software*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000, pp. 238–251.

[38] MOLNAR, D., PIOTROWSKI, M., SCHULTZ, D., AND WAGNER, D. The program counter security model: automatic detection and removal of control-flow side channel attacks. In *Proc. 8th International Conference on Information Security and Cryptology* (2006), pp. 156–168.

[39] NEVE, M., AND SEIFERT, J.-P. Advances on access-driven cache attacks on AES. In *Proceedings of the 13th International Conference on Selected Areas in Cryptography* (2006), SAC'06, pp. 147–162.

[40] OpenSSL. https://www.openssl.org/.

[41] OREN, Y., KEMERLIS, V. P., SETHUMADHAVAN, S., AND KEROMYTIS, A. D. The spy in the sandbox: Practical cache attacks in JavaScript and their implications. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security* (2015), pp. 1406–1418.

[42] OSVIK, D. A., SHAMIR, A., AND TROMER, E. Cache attacks and countermeasures: the case of AES. *Topics in Cryptology–CT-RSA 2006* (Jan. 2006), 1–20.

[43] PAGE, D. Partitioned cache architecture as a side-channel defense mechanism. In *Cryptology ePrint Archive, Report 2005/280* (2005).

[44] PASAREANU, C., PHAN, Q.-S., AND MALACARIA, P. Multi-run side-channel analysis using symbolic execution and Max-SMT. In *Proceedings of the 29th IEEE Computer Security Foundations Symposium* (2016), CSF' 16.

[45] PERCIVAL, C. Cache missing for fun and profit. In *BSDCan* (2005).

[46] QUISQUATER, J.-J., AND SAMYDE, D. Electromagnetic analysis (EMA): Measures and counter-measures for smart cards. In *Proceedings of Smart Card Programming and Security: International Conference on Research in Smart Cards, E-smart 2001* (2001), pp. 200–210.

[47] RISTENPART, T., TROMER, E., SHACHAM, H., AND SAVAGE, S. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security* (2009), CCS '09, pp. 199–212.

[48] SCHWARTZ, E. J., AVGERINOS, T., AND BRUMLEY, D. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the 2010 IEEE Symposium on Security and Privacy* (2010), S&P '10, pp. 317–331.

[49] SEN, K., MARINOV, D., AND AGHA, G. CUTE: a concolic unit testing engine for C. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering* (2005), ESEC/FSE '13, pp. 263–272.

[50] SHI, J., SONG, X., CHEN, H., AND ZANG, B. Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops* (2011), pp. 194–199.

[51] SPREITZER, R., AND PLOS, T. On the applicability of time-driven cache attacks on mobile devices. In *Proceedings of the 7th International Conference on Network and System Security* (2013), NSS '13.

[52] SUN, M., WEI, T., AND LUI, J. C. TaintART: A practical multi-level information-flow tracking system for android runtime. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), CCS '16, pp. 331–342.

[53] TROMER, E., OSVIK, D., AND SHAMIR, A. Efficient cache attacks on AES, and countermeasures. *Journal of Cryptology 23*, 1 (2010), 37–71.

[54] WANG, Z., AND LEE, R. B. New cache designs for thwarting software cache-based side channel attacks. In *Proc. Annual International Symp. on Computer Architecture (ISCA)* (2007), pp. 494–505.

[55] WANG, Z., AND LEE, R. B. A novel cache architecture with enhanced performance and security. In *Proc. 41st Annual IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)* (2008), pp. 83–93.

[56] WU, Z., XU, Z., AND WANG, H. Whispers in the hyper-space: High-speed covert channel attacks in the cloud. In *Proceedings of the 21st USENIX Security Symposium* (2012), pp. 159–173.

[57] XU, Y., BAILEY, M., JAHANIAN, F., JOSHI, K., HILTUNEN, M., AND SCHLICHTING, R. An exploration of L2 cache covert channels in virtualized environments. In *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop* (2011), CCSW '11, pp. 29–40.

[58] YAROM, Y., AND BENGER, N. Recovering OpenSSL ECDSA nonces using the flush+reload cache side-channel attack. Cryptology ePrint Archive, Report 2014/140, 2014.

[59] YAROM, Y., AND FALKNER, K. FLUSH+RELOAD: A high resolution, low noise, l3 cache side-channel attack. In *Proceedings of the 23rd USENIX Conference on Security Symposium* (2014), pp. 719–732.

[60] ZHANG, D., ASKAROV, A., AND MYERS, A. C. Language-based control and mitigation of timing channels. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)* (2012), pp. 99–110.

[61] ZHANG, D., WANG, Y., SUH, G. E., AND MYERS, A. C. A hardware design language for timing-sensitive information-flow security. In *Proc. 20th Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2015), pp. 503–516.

[62] ZHANG, Y., JUELS, A., OPREA, A., AND REITER, M. K. HomeAlone: Co-residency detection in the cloud via side-channel analysis. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy* (2011), S&P '11, pp. 313–328.

[63] ZHANG, Y., JUELS, A., REITER, M. K., AND RISTENPART, T. Cross-VM side channels and their use to extract private keys. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (2012), pp. 305–316.

[64] ZHANG, Y., AND REITER, M. K. Düppel: Retrofitting commodity operating systems to mitigate cache side channels in the cloud. In *Proc. ACM Conf. on Computer and Communications Security (CCS)* (2013), pp. 827–838.

A A Symbolic Memory Address Example

```
134526912+Concat(0,Extract(15,8,key22)^Conc
at(0,Extract(4,3,Concat(Extract(31,31,2*(Co
ncat(0,Extract(28,27,Concat(Extract(7,7,key
9),0,Extract(7,7,key8),0,Extract(7,7,key7),
0,Extract(7,7,key6),0)+4294967295*Concat(0,
Extract(7,7,key9),0,Extract(7,7,key8),0,Ext
ract(7,7,key7),0,Extract(7,7,key6))),0,Extr
act(25,24,Concat(Extract(7,7,key9),0,Extrac
t(7,7,key8),0,Extract(7,7,key7),0,Extract(7
,7,key6),0)+4294967295*Concat(0,Extract(7,7
,key9),0,Extract(7,7,key8),0,Extract(7,7,ke
y7),0,Extract(7,7,key6))),0,Extract(20,19,C
oncat(Extract(7,7,key9),0,Extract(7,7,key8)
,0,Extract(7,7,key7),0,Extract(7,7,key6),0)
+4294967295*Concat(0,Extract(7,7,key9),0,Ex
tract(7,7,key8),0,Extract(7,7,key7),0,Extra
ct(7,7,key6))),0,Extract(17,16,Concat(Extra
ct(7,7,key9),0,Extract(7,7,key8),0,Extract(
7,7,key7),0,Extract(7,7,key6),0)+4294967295
*Concat(0,Extract(7,7,key9),0,Extract(7,7,k
ey8),0,Extract(7,7,key7),0,Extract(7,7,key6
))),0,Extract(12,11,Concat(Extract(7,7,key9
),0,Extract(7,7,key8),0,Extract(7,7,key7),0
,Extract(7,7,key6),0)+4294967295*Concat(0,E
xtract(7,7,key9),0,Extract(7,7,key8),0,Extr
act(7,7,key7),0,Extract(7,7,key6))),0,Extra
ct(9,8,Concat(Extract(7,7,key9),0,Extract(7
,7,key8),0,Extract(7,7,key7),0,Extract(7,7,
key6),0)+4294967295*Concat(0,Extract(7,7,ke
y9),0,Extract(7,7,key8),0,Extract(7,7,key7)
,0,Extract(7,7,key6))),0,Extract(4,3,Concat
(Extract(7,7,key9),0,Extract(7,7,key8),0,Ex
tract(7,7,key7),0,Extract(7,7,key6),0)+4294
967295*Concat(0,Extract(7,7,key9),0,Extract
(7,7,key8),0,Extract(7,7,key7),0,Extract(7,
7,key6))),0,3*Concat(0,Extract(7,7,key6)))^
Concat(Extract(31,25,2*(Concat(key9,0)^Conc
at(0,key8,0)^Concat(0,key7,0)^Concat(0,key6
))),0,Extract(23,17,2*(Concat(key9,0)^Conca
t(0,key8,0)^Concat(0,key7,0)^Concat(0,key6)
)),0,Extract(15,9,2*(Concat(key9,0)^Concat(
0,key8,0)^Concat(0,key7,0)^Concat(0,key6)))
,0,Extract(7,1,2*(Concat(key9,0)^Concat(0,k
ey8,0)^Concat(0,key7,0)^Concat(0,key6))),0)
```

```
)),0,Extract(23,23,2*(Concat(0,Extract(28,2
7,Concat(Extract(7,7,key9),0,Extract(7,7,ke
y8),0,Extract(7,7,key7),0,Extract(7,7,key6)
,0)+4294967295*Concat(0,Extract(7,7,key9),0
,Extract(7,7,key8),0,Extract(7,7,key7),0,Ex
tract(7,7,key6))),0,Extract(25,24,Concat(Ex
tract(7,7,key9),0,Extract(7,7,key8),0,Extra
ct(7,7,key7),0,Extract(7,7,key6),0)+4294967
295*Concat(0,Extract(7,7,key9),0,Extract(7,
7,key8),0,Extract(7,7,key7),0,Extract(7,7,k
ey6))),0,Extract(20,19,Concat(Extract(7,7,k
ey9),0,Extract(7,7,key8),0,Extract(7,7,key7
),0,Extract(7,7,key6),0)+4294967295*Concat(
0,Extract(7,7,key9),0,Extract(7,7,key8),0,E
xtract(7,7,key7),0,Extract(7,7,key6))),0,Ex
tract(17,16,Concat(Extract(7,7,key9),0,Extr
act(7,7,key8),0,Extract(7,7,key7),0,Extract
(7,7,key6),0)+4294967295*Concat(0,Extract(7
,7,key9),0,Extract(7,7,key8),0,Extract(7,7,
key7),0,Extract(7,7,key6))),0,Extract(12,11
,Concat(Extract(7,7,key9),0,Extract(7,7,key
8),0,Extract(7,7,key7),0,Extract(7,7,key6),
0)+4294967295*Concat(0,Extract(7,7,key9),0,
Extract(7,7,key8),0,Extract(7,7,key7),0,Ext
ract(7,7,key6))),0,Extract(9,8,Concat(Extra
ct(7,7,key9),0,Extract(7,7,key8),0,Extract(
7,7,key7),0,Extract(7,7,key6),0)+4294967295
*Concat(0,Extract(7,7,key9),0,Extract(7,7,k
ey8),0,Extract(7,7,key7),0,Extract(7,7,key6
))),0,Extract(4,3,Concat(Extract(7,7,key9),
0,Extract(7,7,key8),0,Extract(7,7,key7),0,E
xtract(7,7,key6),0)+4294967295*Concat(0,Ext
ract(7,7,key9),0,Extract(7,7,key8),0,Extrac
t(7,7,key7),0,Extract(7,7,key6))),0,3*Conca
t(0,Extract(7,7,key6)))^Concat(Extract(31,2
5,2*(Concat(key9,0)^Concat(0,key8,0)^Concat
(0,key7,0)^Concat(0,key6))),0,Extract(23,17
,2*(Concat(key9,0)^Concat(0,key8,0)^Concat(
0,key7,0)^Concat(0,key6))),0,Extract(15,9,2
*(Concat(key9,0)^Concat(0,key8,0)^Concat(0,
key7,0)^Concat(0,key6))),0,Extract(7,1,2*(C
oncat(key9,0)^Concat(0,key8,0)^Concat(0,key
7,0)^Concat(0,key6))),0))),0,Extract(15,15,
2*(Concat(0,Extract(28,27,Concat(Extract(7,
7,key9),0,Extract(7,7,key8),0,Extract(7,7,k
ey7),0,Extract(7,7,key6),0)+4294967295*Conc
at(0,Extract(7,7,key9),0,Extract(7,7,key8),
0,Extract(7,7,key7),0,Extract(7,7,key6))),0
,Extract(25,24,Concat(Extract(7,7,key9),0,E
xtract(7,7,key8),0,Extract(7,7,key7),0,Extr
act(7,7,key6),0)+4294967295*Concat(0,Extrac
t(7,7,key9),0,Extract(7,7,key8),0,Extract(7
,7,key7),0,Extract(7,7,key6))),0,Extract(20
,19,Concat(Extract(7,7,key9),0,Extract(7,7,
key8),0,Extract(7,7,key7),0,Extract(7,7,key
6),0)+4294967295*Concat(0,Extract(7,7,key9)
,0,Extract(7,7,key8),0,Extract(7,7,key7),0,
Extract(7,7,key6))),0,Extract(17,16,Concat(
Extract(7,7,key9),0,Extract(7,7,key8),0,Ext
ract(7,7,key7),0,Extract(7,7,key6),0)+42949
67295*Concat(0,Extract(7,7,key9),0,Extract(
7,7,key8),0,Extract(7,7,key7),0,Extract(7,7
,key6))),0,Extract(12,11,Concat(Extract(7,7
,key9),0,Extract(7,7,key8),0,Extract(7,7,ke
y7),0,Extract(7,7,key6),0)+4294967295*Conca
t(0,Extract(7,7,key9),0,Extract(7,7,key8),0
,Extract(7,7,key7),0,Extract(7,7,key6))),0,
Extract(9,8,Concat(Extract(7,7,key9),0,Extr
act(7,7,key8),0,Extract(7,7,key7),0,Extract
(7,7,key6),0)+4294967295*Concat(0,Extract(7
,7,key9),0,Extract(7,7,key8),0,Extract(7,7,
key7),0,Extract(7,7,key6))),0,Extract(4,3,C
oncat(Extract(7,7,key9),0,Extract(7,7,key8)
,0,Extract(7,7,key7),0,Extract(7,7,key6),0)
+4294967295*Concat(0,Extract(7,7,key9),0,Ex
tract(7,7,key8),0,Extract(7,7,key7),0,Extra
ct(7,7,key6))),0,3*Concat(0,Extract(7,7,key
6)))^Concat(Extract(31,25,2*(Concat(key9,0)
^Concat(0,key8,0)^Concat(0,key7,0)^Concat(0
,key6))),0,Extract(23,17,2*(Concat(key9,0)^
Concat(0,key8,0)^Concat(0,key7,0)^Concat(0,
key6))),0,Extract(15,9,2*(Concat(key9,0)^Co
ncat(0,key8,0)^Concat(0,key7,0)^Concat(0,ke
y6))),0,Extract(7,1,2*(Concat(key9,0)^Conca
t(0,key8,0)^Concat(0,key7,0)^Concat(0,key6)
)),0))),0,Extract(7,7,2*(Concat(0,Extract(2
8,27,Concat(Extract(7,7,key9),0,Extract(7,7
,key8),0,Extract(7,7,key7),0,Extract(7,7,ke
y6),0)+4294967295*Concat(0,Extract(7,7,key9
),0,Extract(7,7,key8),0,Extract(7,7,key7),0
,Extract(7,7,key6))),0,Extract(25,24,Concat
(Extract(7,7,key9),0,Extract(7,7,key8),0,Ex
tract(7,7,key7),0,Extract(7,7,key6),0)+4294
967295*Concat(0,Extract(7,7,key9),0,Extract
(7,7,key8),0,Extract(7,7,key7),0,Extract(7,
7,key6))),0,Extract(20,19,Concat(Extract(7,
7,key9),0,Extract(7,7,key8),0,Extract(7,7,k
ey7),0,Extract(7,7,key6),0)+4294967295*Conc
at(0,Extract(7,7,key9),0,Extract(7,7,key8),
0,Extract(7,7,key7),0,Extract(7,7,key6))),0
,Extract(17,16,Concat(Extract(7,7,key9),0,E
xtract(7,7,key8),0,Extract(7,7,key7),0,Extr
act(7,7,key6),0)+4294967295*Concat(0,Extrac
t(7,7,key9),0,Extract(7,7,key8),0,Extract(7
,7,key7),0,Extract(7,7,key6))),0,Extract(12
,11,Concat(Extract(7,7,key9),0,Extract(7,7,
key8),0,Extract(7,7,key7),0,Extract(7,7,key
6),0)+4294967295*Concat(0,Extract(7,7,key9)
```

```
,0,Extract(7,7,key8),0,Extract(7,7,key7),0,
Extract(7,7,key6))),0,Extract(9,8,Concat(Ex
tract(7,7,key9),0,Extract(7,7,key8),0,Extra
ct(7,7,key7),0,Extract(7,7,key6),0)+4294967
295*Concat(0,Extract(7,7,key9),0,Extract(7,
7,key8),0,Extract(7,7,key7),0,Extract(7,7,k
ey6))),0,Extract(4,3,Concat(Extract(7,7,key
9),0,Extract(7,7,key8),0,Extract(7,7,key7),
0,Extract(7,7,key6),0)+4294967295*Concat(0,
Extract(7,7,key9),0,Extract(7,7,key8),0,Ext
ract(7,7,key7),0,Extract(7,7,key6))),0,3*Co
ncat(0,Extract(7,7,key6)))^Concat(Extract(3
1,25,2*(Concat(key9,0)^Concat(0,key8,0)^Con
cat(0,key7,0)^Concat(0,key6))),0,Extract(23
,17,2*(Concat(key9,0)^Concat(0,key8,0)^Conc
at(0,key7,0)^Concat(0,key6))),0,Extract(15,
9,2*(Concat(key9,0)^Concat(0,key8,0)^Concat
(0,key7,0)^Concat(0,key6))),0,Extract(7,1,2
*(Concat(key9,0)^Concat(0,key8,0)^Concat(0,
key7,0)^Concat(0,key6))),0))),0)+4294967295
*Concat(0,Extract(31,31,2*(Concat(0,Extract
(28,27,Concat(Extract(7,7,key9),0,Extract(7
,7,key8),0,Extract(7,7,key7),0,Extract(7,7,
key6),0)+4294967295*Concat(0,Extract(7,7,ke
y9),0,Extract(7,7,key8),0,Extract(7,7,key7)
,0,Extract(7,7,key6))),0,Extract(25,24,Conc
at(Extract(7,7,key9),0,...
```

B Structure of Two-level Precomputed Table and Precomputed Size Table

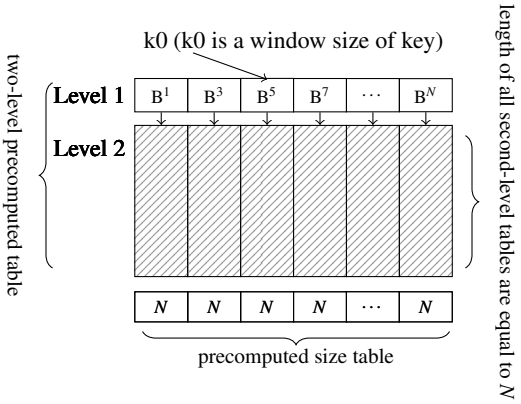

Figure 5: Two-level precomputed table and precomputed size table used in RSA and ElGamal. Our observation shows that the length of all the second-level precomputed tables are equal in non-trivial decryption processes of RSA and ElGamal. In other words, attackers can hardly infer k0 by observing query outputs of the precomputed size table.

C Unknown RSA Vulnerabilities in OpenSSL

```
1  int BN_num_bits(const BIGNUM *a) {
2    BN_ULONG l;
3    int i;
4
5    bn_check_top(a);
6
7    if (a->top == 0) return(0);
8    l=a->d[a->top-1];
9    assert(l != 0);
10   i=(a->top-1)*BN_BITS2;
11   return(i+BN_num_bits_word(l));
12 }
13
14 int BN_num_bits_word(BN_ULONG l) {
15   static const char bits[256]={
16     0,1,2,2,3,3,3,3,4,4,4,4,4,4,4,4,
17     5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,
18     ...
19     8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,
20     8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,
21   };
22   ....
23   return bits[l];
24 }
```

Figure 6: Unknown RSA vulnerabilities found in OpenSSL (version 0.9.7c and 1.0.2f). The tainted variable (i.e., secret) `l` is red and the vulnerable program point is **bold**.

D Unknown RSA Vulnerabilities in Botan

```
1  BigInt Montgomery_Exponentiator::execute() const {
2   ...
3   for(size_t i = exp_nibbles; i > 0; --i) {
4    ...
5    const u32bit nibble = exp.get_substring(
6    window_bits*(i-1), window_bits);
7
8    //note that the following code is not a mem access
9    const BigInt& y = g[nibble];
10
11   bigint_monty_mul(&z[0], z.size(),
12                    x.data(), x.size(), x.sig_words(),
13                    y.data(), y.size(), y.sig_words(),
14     ...
15   }
16 }
17
18 size_t sig_words() const {
19   const word* x = &reg[0];
20   size_t sig = reg.size();
21   ...
22 }
```

Figure 7: Unknown RSA vulnerabilities found in the Montgomery exponentiator of Botan (version 1.10.13). Tainted variables are marked as red and the vulnerable program points are **bold**. `sig` is not tainted according to the optimization of RSA precomputed size table (§5).