Identifying Cache-Based Side Channels through Secret-Augmented Abstract Interpretation

Shuai Wang, *HKUST;* Yuyan Bao and Xiao Liu, *Penn State University;* Pei Wang, *Baidu X-Lab;*
Danfeng Zhang and Dinghao Wu, *Penn State University*

https://www.usenix.org/conference/usenixsecurity19/presentation/wang-shuai

This paper is included in the Proceedings of the
28th USENIX Security Symposium.

August 14–16, 2019 • Santa Clara, CA, USA

978-1-939133-06-9

Open access to the Proceedings of the
28th USENIX Security Symposium
is sponsored by USENIX.

Identifying Cache-Based Side Channels through Secret-Augmented Abstract Interpretation

Shuai Wang*[1], Yuyan Bao[2], Xiao Liu[2], Pei Wang*[3], Danfeng Zhang[2], and Dinghao Wu[2]

[1]The Hong Kong University of Science and Technology
[2]The Pennsylvania State University
[3]Baidu X-Lab
shuaiw@cse.ust.hk, {yxb88, xvl5190}@ist.psu.edu, wangpei10@baidu.com, zhang@cse.psu.edu, dwu@ist.psu.edu

Abstract

Cache-based side channels enable a dedicated attacker to reveal program secrets by measuring the cache access patterns. Practical attacks have been shown against real-world crypto algorithm implementations such as RSA, AES, and ElGamal. By far, identifying information leaks due to cache-based side channels, either in a static or dynamic manner, remains a challenge: the existing approaches fail to offer high precision, full coverage, and good scalability simultaneously, thus impeding their practical use in real-world scenarios.

In this paper, we propose a novel static analysis method on binaries to detect cache-based side channels. We use abstract interpretation to reason on program states with respect to abstract values at each program point. To make such abstract interpretation scalable to real-world cryptosystems while offering high precision and full coverage, we propose a novel abstract domain called the Secret-Augmented Symbolic domain (**SAS**). **SAS** tracks program secrets and dependencies on them for precision, while it tracks only coarse-grained public information for scalability.

We have implemented the proposed technique into a practical tool named CacheS and evaluated it on the implementations of widely-used cryptographic algorithms in real-world crypto libraries, including Libgcrypt, OpenSSL, and mbedTLS. CacheS successfully confirmed a total of 154 information leaks reported by previous research and 54 leaks that were previously unknown. We have reported our findings to the developers. And they confirmed that many of those unknown information leaks do lead to potential side channels.

1 Introduction

Cache-based timing channels enable attackers to reveal secret program information, such as private keys, by measuring the runtime cache behavior of the victim program. Practical attacks have been executed with different attack scenarios, such as time-based [16, 44], access-based [37, 60, 62], and trace-based [5], each of which exploits a victim program through either coarse-grained or fine-grained monitoring of cache behavior. Additionally, previous research has successfully launched attacks on commonly used cryptographic algorithm implementations, for example, AES [37, 60, 74, 16], RSA [23, 44, 7, 62, 86], and ElGamal [90].

Pinpointing cache-based side channels from production cryptosystems remains a challenge. Existing research employs either static or dynamic methods to detect underlying issues [77, 32, 33, 41, 82, 22, 81]. However, the methods are limited to low detection coverage, low precision, and poor scalability, which impede their usage in analyzing real-world cryptosystems in the wild.

Abstract interpretation is a well-established framework that can be tuned to balance precision and scalability for static analysis. It models program execution within one or several carefully-designed *abstract domains*, which abstract program concrete semantics by tracking certain program states of interest in a concise representation. Usually, the elements in an abstract domain form a complete lattice of finite height, and the operations of the program concrete semantics are mapped to the abstract transfer functions over the abstract domain. A well-designed abstract interpretation framework can correctly approximate program execution and usually yields a terminating analysis within a finite step of computations. Nevertheless, the art is to carefully design an abstraction domain that fits the problem under consideration, while over-approximating others to bound the analysis to a controllable size; this enables the analysis of non-trivial cases.

We propose a novel abstract domain named the Secret-Augmented Symbolic domain (**SAS**), which is specifically designed to perform abstract interpretation on *large-scale* secret-aware software, such as real-world cryptosystems. **SAS** is designed to perform fine-grained tracking of program secrets (e.g., private keys) and dependencies on them, while coarsely approximating non-secret information to speed up the convergence of the analysis.

We implement the proposed technique as a practical tool named CacheS, which models program execution within the **SAS** and pinpoints cache-based side channels with constraint solving techniques. Like many bug finding tech-

*Most of this work is done while Shuai Wang and Pei Wang were working at PSU.

niques [55, 84, 54], CacheS is soundy [53]; the implementation is unsound for speeding up analysis and optimizing memory usage, due to its lightweight but unsound treatment of memory. However, in contrast to previous studies that analyze only small-size programs, single procedure or single execution trace [32, 33, 77, 22, 81], CacheS is scalable enough to deliver whole program static analysis of real-world cryptosystems without sacrificing much accuracy. We have evaluated CacheS on multiple popular crypto libraries. Although most libraries have been checked by many previous tools, CacheS is able to detect 54 unknown information leakage sites from the implementations of RSA/ElGamal algorithms in three real-world cryptosystems: Libgcrypt (ver. 1.6.3), OpenSSL (ver. 1.0.2k and 1.0.2f), and mbedTLS (ver. 2.5.1). We show that CacheS has good scalability as it largely outperforms previous research regarding coverage; it is able to complete context-sensitive interprocedural analysis of over 295 K lines of instructions within 0.5 CPU hour. In summary, we make the following contributions:

- We propose a novel abstract interpretation-based analysis to pinpoint information leakage sites that may lead to cache-based side channels. We propose a novel abstract domain named **SAS**, which performs fine-grained tracking of program secrets and dependencies, while over-approximating non-secret values to enable precise reasoning in a scalable way.
- Enabled by the "symbolic" representation of abstract values in **SAS**, we facilitate information leak checking in this research with constraint solving techniques. Compared with previous abstract interpretation-based methods, which only reason on the information leakage upper-bound, our technique adequately simplifies the process of debugging and fixing side channels.
- We implement the proposed technique into a practical tool named CacheS and apply it to detect cache-based side channels in real-world cryptosystems. From five popular crypto library implementations, CacheS successfully identified 208 information leakage sites (with only one false positive), among which 54 are unknown to previous research, to the best of our knowledge.

2 Background

Abstract Interpretation. Abstract interpretation is a well-established framework to perform sound approximation of program semantics [28]. Considering that program concrete semantics forms a value domain **C**, abstract interpretation maps **C** to an abstract (and usually more concise) representation, namely, an abstract domain **A**. The design of the abstraction is usually based on certain program properties of interest, and (possibly infinite) sets of concrete program states are usually represented by one abstract state in **A**. To ensure termination, abstract states could form a lattice with a finite height, and computations of program concrete semantics are mapped into operators over the abstract elements in **A**.

The abstract function (α) and concretization function (γ) need to be defined jointly with an abstract domain **A**. Function α lifts the elements in **C** to their corresponding abstract elements in **A**, while γ casts an abstract value to a set of values in **C**. To establish the correctness of an abstract interpretation, the abstract domain and the concrete domain need to form a Galois connection, and operators defined upon elements in an abstract domain are required to form the local and global soundness notions [28].

Cache Structure and Cache-Based Timing Channels. A cache is a fast on-CPU data storage unit with a very limited capacity compared to the main memory. Caches are usually organized to be set-associative, meaning that the storage is partitioned into several disjoint sets while each set exclusively stores data of a particular part of the memory space. Each cache set can be further divided into smaller storage units of equal size, namely cache lines. Given the size of each cache line as 2^L bytes, usually the upper $N-L$ bits of a N-bit memory address uniquely locate a cache line where the data from that address will be temporally held.

When the requested data is not found in the cache, the CPU will have to fetch them from the main memory. This is called a cache miss and causes a significant delay in execution, compared with fetching data directly from the cache. Therefore, an attacker may utilize the timing difference to reveal the cache access pattern and further infer any information on which this pattern may depend.

Threat Model. As mentioned above, some bits of a memory address can be directly mapped to cache lines being visited, which potentially enables information leakage via *secret-dependent memory traffic*. In this research, attackers are assumed to share the same hardware platform with the victim program, and therefore are able to "probe" the shared cache state and infer cache lines being accessed by the victim. As illustrated in Fig. 2, our threat model assumes that the attacker can observe the address of every memory access, expect for the low bits of addresses that distinguish locations in the same cache line. Overall, by tracking the secret-dependent cache access of the victim, several bits of program secrets (w.r.t. entropy) could be leaked to the attacker.

We note that this threat model indeed captures most infamous and practical side channel attacks [39], including prime-and-probe [60], flush-and-reload [86], and prime-and-abort [31], which are designed to infer the cache line access by measuring the latency of the victim program or attacker's program at different scales and for different attack scenarios. Additionally, while this threat model is aligned with many existing side channel detection works [77, 33, 41, 82, 22], novel techniques proposed in this work enable us to perform scalable static analysis and reveal much more information leaks of real-world cryptosystems.[1] In addition, while this

[1]Consistent with this line of research, CacheS pinpoints information leaks in cryptosystems where cache access depends on secrets. Cryptosystem developers can fix the code with information provided by CacheS. Contrarily, the exploitability of the leaks (e.g., reconstruct the entire key by recovering half bits of the RSA private key [19]) is beyond the scope of this work.

	(a)		(b)		(c)
1	`foo:`	1	$\{\texttt{ebx} = \{k_1\}\}$	1	$\{\texttt{ebx} = \{s_1\}\}$
2	`mov eax, ebx`	2	$\{\texttt{ebx} = \{k_1\},\ \texttt{eax} = \{k_1\}\}$	2	$\{\texttt{ebx} = \{s_1\},\ \texttt{eax} = \{s_1\}\}$
3	`add eax, 0x1`	3	$\{\texttt{ebx} = \{k_1\},\ \texttt{eax} = \{k_1+1\}\}$	3	$\{\texttt{ebx} = \{s_1\},\ \texttt{eax} = \{s_1+1\}\}$
4	`load ecx, esi`	4	$\{\texttt{ebx} = \{k_1\},\ \texttt{eax} = \{k_1+1\},\ \texttt{ecx} = \{m_1\}\}$	4	$\{\texttt{ebx} = \{s_1\},\ \texttt{eax} = \{s_1+1\},\ \texttt{ecx} = \{p\}\}$
5	`add ecx, 0x12`	5	$\{\texttt{ebx} = \{k_1\},\ \texttt{eax} = \{k_1+1\},\ \texttt{ecx} = \{m_1+12\}\}$	5	$\{\texttt{ebx} = \{s_1\},\ \texttt{eax} = \{s_1+1\},\ \texttt{ecx} = \{p\}\}$
6	`mov edx, edi`	6	$\{\texttt{ebx} = \{k_1\},\ \texttt{eax} = \{k_1+1\},\ \texttt{ecx} = \{m_1+12\},\ \texttt{edx} = \{\texttt{edi}_0\}\}$	6	$\{\texttt{ebx} = \{s_1\},\ \texttt{eax} = \{s_1+1\},\ \texttt{ecx} = \{p\},\ \texttt{edx} = \{p\}\}$
7	`add eax, ecx`	7	$\{\texttt{ebx} = \{k_1\},\ \texttt{eax} = \{k_1+m_1+13\},\ \texttt{ecx} = \{m_1+12\},\ \texttt{edx} = \{\texttt{edi}_0\}\}$	7	$\{\texttt{ebx} = \{s_1\},\ \texttt{eax} = \{\top\},\ \texttt{ecx} = \{p\},\ \texttt{edx} = \{p\}\}$

(a) Sample Code. (b) Modeling program states with logic formulas $l \in L$. (c) Modeling program states with **SAS**.

Figure 1: Execute assembly code with different program representations. Program secrets and all the affected registers are marked as red in Fig. 1a. Program states at line 1 of Fig. 1b and Fig. 1c represent the initial state. Here $\texttt{k}_1$ is a symbol exhibiting one piece of program secrets (e.g., the first element in a key array), and $\texttt{m}_1$ is a free symbol representing non-secret content of unknown memory cells. $\texttt{edi}_0$ is a symbol representing the initial value of register `edi`. Symbol s_1, p, and $\top$ defined in **SAS** stand for one piece of secret, entire non-secret information and all the program information, respectively (see Sec. 4).

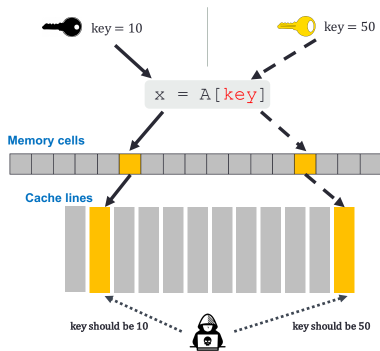


Figure 2: The threat model. Different secrets lead to the access of different cache lines at one particular program point, which may leak secret information to the attackers by indirectly observing cache line access variants. At least one bit information (w.r.t. entropy) could be leaked in this example.

model is relatively stronger than those based on cache status [32], cache status at any point can be determined by analyzing the accessed cache units in execution.

3 Motivation

In general, capturing cache-based side channels requires modeling program secret-dependent semantics (we will discuss the connection between program semantics and cache access in Sec. 5). In this section we begin by discussing two baseline approaches to modeling program semantics; the limitations of both approaches naturally motivate the design of our novel abstract domain.

Modeling Program Semantics with Logic Formulas. An intuitive way is to represent program concrete semantics with logic formulas (as in a typical symbolic execution approach [77]), and perform whole-program static reasoning until a fixed point is reached. The overall workflow exhibits a typical dataflow analysis procedure, and upon termination, each program point maintains a program state that maps variables (i.e., registers, CPU flags, and memory cells) to sets of formulas representing the possible values each variable may hold regarding any execution paths and inputs. For ease of presentation, we name the value domain formed by logic formulas l as logic domain L.

An example is given in Fig. 1, where we model the execution of instructions with logic formulas (Fig. 1b). While the overall approach will precisely model program semantics, some tentative studies indicate its low scalability. Indeed, we implement this approach and evaluate it with two real-world cases: the AES and RSA implementations of OpenSSL. We report that both tests are unable to terminate (evaluation results are given in Sec. 8). In summary, the analysis is impeded for the following reasons:

- Typically, more and more memory cells would be modeled throughout the analysis, and for each variable, its value set (i.e., set of formulas) would also continue to increase. Therefore, the memory usage could become significant to even unrealistic for real-world cases.
- Program states could be continuously updated within loop iterations. In addition, "loops" on the call graph (e.g., recursive calls) could exist in cryptosystems as well and complicate the analysis.

We implement algorithms to detect loop induction variables [11] considering both registers and stack memories. Identified induction variables are lifted into a linear function of symbolic loop iterators; operations on induction variables are "merged" into the linear function, thereby leading to a stable stage. While the simpler AES case terminated when we re-ran the test, the RSA case still yielded a "timeout" due to the practical challenges mentioned above (see results in Sec. 8.1).

Modeling Program Semantics with Free Symbols. Another "baseline" approach is to model program semantics in a permissive way. That is, we introduce two free symbols: one for any public information and the other for secrets. Any secret-related computation outputs the same secret symbol, while others preserve the same public symbol. Note that this is comparable to static taint tracking, where each value is either "tainted" or "untainted". Despite its simplicity, our tentative study reveals new hurdles as follows:

- Memory tracking becomes pointless. Every memory address becomes (syntactically) identical because it

holds the same public or secret symbol. Therefore, a memory store could overturn the entire memory space.
- Even if memory addresses are tracked in a more precise way, representing any secret value and their dependencies coarsely as one free secret symbol yields many false positives (since secret-dependent memory accesses do not necessarily lead to vulnerable cache accesses; see our cache modeling in Sec. 5). Tentative tests of the AES case report a false positive rate of 20% (8 out of 40) due to such modeling. In contrast, our novel program modeling yields no false positive when testing this case (see Sec. 8).

Motivation of Our Approach. This paper presents a novel abstract domain that enables abstract interpretation of large-scale cryptosystems in the wild. Our observation is that imprecise tracking of secrets impedes the accurate modeling of cache behaviors (cache access modeling is discussed in Sec. 5). Nevertheless, tracking too much information, such as modeling whole-program semantics with logic formulas, could face scalability issues when analyzing real-world cryptosystems due to various practical challenges.

Our study of real-world cryptosystems actually reveals an interesting and intuitive finding. That is, program secrets and their dependencies usually exhibit at a very *small portion* of program points, and even in such secret-carrying points, most variables maintain *only public information*. It should be noted that in common scenarios non-secret information is not critical for modeling cache-based timing channels. Hence, based on our observation, we promote a novel abstract domain that is particularly designed to *model the secret-dependent semantics of real-world crypto systems*. Our abstract domain delivers fine-grained tracking of program secrets and their dependencies with different identifiers for each piece of secret information, while performing coarse-grained tracking of other public values to effectively enhance scalability.

4 Secret-Augmented Symbolic Domain

This section presents the definition of our abstract domain **SAS**. We formally define each component following convention, including the concrete semantics, the abstract domain, and the abstract transfer functions. We also prove that the computations specified in **SAS** correctly over-approximate concrete semantics. Due to space limitations, we highlight only certain necessary components to make the paper self-contained. We refer readers to the extended version of this paper for more details [76] .

4.1 Abstract Values

We start by defining abstract values $f \in \mathbf{AV}$ (soon we will show that **SAS** is defined as the powerset of **AV**). Comparable to "symbolic formulas" in symbolic execution, f combines symbols and constants via operators. Elementary symbols in each abstract value are defined as follows:

- p: a unique symbol representing all the program public information.
- s_i: a symbol representing a piece of program secrets; for instance, the i-th element of a secret array.
- e: a unique symbol representing the initial value of the x86 stack register `esp`.

Literal	n	$\in \mathbb{Z}$
OP_1	$\oplus$	$::= + \mid -$
OP_2	$\otimes$	$::= \times \mid \div \mid \% \mid \text{AND} \mid \text{OR} \mid \text{XOR} \mid \text{SHIFT}$
Atom	t	$::= \top \mid p \mid s_i \mid n$
Expression	exp	$::= t \mid t \oplus exp \mid t \otimes exp$
Formula	f	$::= e \mid exp \mid e \oplus exp$

Figure 3: Syntax of abstract value.

While only one free symbol p is used to represent any and all unknown non-secret information (e.g., initial value $\mathtt{edi}_0$ of register `edi` in Fig. 1b), we retain finer-grained information about program secrets. Multiple s_i are generated, and are mapped to different pieces of program secrets (e.g., a symbol s_1 representing $\mathtt{k}_1$ in Fig. 1c). Therefore, different s_i symbols are *semantically different*, meaning each of them stands for different secrets.

Syntax. The syntax of a core of abstract values $f \in \mathbf{AV}$ is defined in Fig. 3. Literal specifies that concrete data is preserved in **AV**. OP_1 and OP_2 explain typical operators in **AV**. Atom includes symbols and literals, among which $\top$ (top) is the abstraction of any concrete value. Expression and Formula additionally define expressions and formulas. Note that stack memory expands linearly in the process address space, and stack register `esp` at any program point shall hold a value which adds or subtracts an offset from the initial value of `esp` (i.e., e). In the syntax definition, stack memory offsets could be a constant or an exp.

Since the symbol $\{s_i\}$ represents the secrets, which our analysis intends to keep track of, the formulas that contain these symbols usually need to be specially treated. We denote this infinite set of special formulas by $\mathbf{AV}_s$, where $\mathbf{AV}_s = \{f \in \mathbf{AV} \mid \exists s \in \{s_i\} \text{ s.t. } s \text{ occurs in } f\}$.

Reduction of Abstract Formulas. We now define the operator semantics of abstract value $f \in \mathbf{AV}$. For any operator $\odot \in \{\oplus\} \cup \{\otimes\}$, we define a reduction rule $T_\odot : \mathbf{AV} \times \mathbf{AV} \to \mathbf{AV}$ such that $[\![a_1 \odot a_2]\!] = T_\odot([\![a_1]\!], [\![a_2]\!])$ for any $a_1, a_2 \in \mathbf{AV}$, where $[\![\cdot]\!]$ denotes the semantics. We then define $T_\odot(a_1, a_2)$ as follows:

$$T_\odot(a_1,a_2) = \begin{cases} \top & \text{if } a_1 = \top \text{ or } a_2 = \top \\ \top & \text{else if } a_1 = p \wedge a_2 \in \mathbf{AV}_s \text{ or} \\ & a_2 = p \wedge a_1 \in \mathbf{AV}_s \\ p & \text{else if } a_1 = p \wedge a_2 \notin \mathbf{AV}_s \text{ or} \\ & a_2 = p \wedge a_1 \notin \mathbf{AV}_s \\ a_1 \odot a_2 & \text{otherwise} \end{cases}$$

Essentially, the first three cases perform reasonable over-approximation on $f \in \mathbf{AV}$ with different degrees of abstraction. The last case would apply if no other case can be matched; indeed similar to symbolic execution, most operations on $f \in \mathbf{AV}$ "concatenates" abstract values via abstract

operators following this rule. For the implementation, we also implement "constant folding" rules for operands of concrete data; such rules help the reduction of stack increment and decrement operations.

Since abstract interpretation typically needs to process sets of facts, we extend $T_\odot$ so that it can be applied to pairs of subsets of abstract values $f \in \mathbf{AV}$, where

$$\forall X, Y \in \mathscr{P}(\mathbf{AV}), \forall \odot \in \{\oplus\} \cup \{\otimes\},$$
$$T_\odot(X,Y) = \{T_\odot(a,b) \mid a \in X, b \in Y\}$$

4.2 Abstract Domain

Naturally, each element in **SAS** represents the possible values that a program variable may hold; therefore each element in **SAS** forms a set of abstract values. That is,

Definition 1. *Let* **AV** *be the set of abstract values. Then*

$$\mathbf{SAS} = \mathscr{P}(\mathbf{AV})$$

forms a domain whose elements are subsets of all valid abstract values.

Claim 1. ***SAS*** *forms a lattice, with the top element* $\top_{\mathbf{SAS}}$*, bottom element* $\bot_{\mathbf{SAS}}$ *and a join operator* $\sqcup$ *defined over* ***SAS***.

We specify the $\top$, $\bot$, and join operator $\sqcup$ in Appendix A. We bound the size of each element in **SAS** with a maximal number N (therefore the lattice has a finite height) and give corresponding evaluations in Appendix B. For further discussion, see the extended version [76].

Example. Fig. 1 explains typical computations within **SAS**. We present a set of abstract values for each register in Fig. 1c. While the computations over secret symbol s_i are precisely tracked (line 3 in Fig. 1c), the computations over p preserve this symbol (line 5 in Fig. 1c), and the computations between abstract value $a \in \mathbf{AV}_s$ and p lead to $\top$ (line 7 in Fig. 1c).

5 Pinpointing Information Leakage Sites

Upon the termination of static analysis, we check abstract memory addresses of each memory load and store instruction. When a secret-dependent address $a \in \mathbf{AV}_s$ is identified, its corresponding memory access instruction is considered to be "secret-dependent." We then translate each secret-dependent address a into an SMT formula f for constraint checking (this translation is discussed in Sec. 6.4).

In this research, we adopt a cache model proposed by the existing work to check each secret-dependent memory access [77]. Given an SMT formula f translated from $a \in \mathbf{AV}_s$ that represents a memory address, CacheS checks potential cache line access variants by solving the satisfiability of the following predicate:

$$f \gg L \neq f[s'_i/s_i] \gg L \tag{1}$$

As discussed in Sec. 2, assuming the cache has the line size of 2^L bytes, for a memory address of N bits, the upper $N-L$ bits map a memory access to its corresponding cache line access. In other words, the upper $N-L$ bits decide which cache line the upcoming memory access would visit. Therefore, for an SMT formula f derived from $a \in \mathbf{AV}_s$, we right shift f by L bits, and the result $f \gg L$ indicates the cache line being accessed. Furthermore, by replacing each s_i with a fresh secret symbol s'_i, we obtain $f[s'_i/s_i] \gg L$. As a standard setting, the cache line size is assumed to be 64 (2^6) in this work; therefore, we set L as 6.

The constructed constraint checks whether different secrets (s_i and s'_i) can lead to the access of different cache lines at this memory access. Recall the threat model shown in Fig. 2, the existence of at least one satisfiable solution reveals potential side channels at this point. From an attacker's perspective, by (indirectly) observing the access of different cache lines, a certain number of secrets could be leaked to adversaries. In addition, while this constraint assumes that accesses to different offsets within cache lines are indistinguishable, Constraint 1 can be extended to detect related issues. For example, information leaks which enable cache bank attacks can be detected by changing L from 6 to 2 [87].

6 Design of CacheS

We now present CacheS, a tool that uses precise and scalable static analysis to detect cache-based timing channels in real-world cryptosystems. Fig. 4 presents the workflow of CacheS. Given a binary as the input, CacheS first leverages a reverse engineering tool to recover the assembly code and the control flow structures from the input. The assembly instructions are further lifted into *platform-independent* representations before analysis. Technical details on reverse engineering are discussed in Sec. 7.

Given all the recovered program information, we initialize the abstract program state at each program point. In particular, we update the initial state of certain program points with one or several "secret" symbols to represent program secrets (e.g., a sequence of memory cells) when the analysis starts. We then perform abstract interpretation on the whole program until the fixed point in **SAS** is reached.

Abstract interpretation reasons the program execution within **SAS** (Sec. 6.1), and as mentioned, the proposed abstract domain performs fine-grained tracking of program secret-related semantics while maintaining only coarse-grained public information for scalability. The entire analysis framework forms a standard worklist algorithm, where each program point maintains its own program state mapping variables to sets of abstract values (Sec. 6.1.2).

We define information flow rules to propagate secret information (Sec. 6.2) in our context-sensitive and interprocedural analysis (Sec. 6.3). Upon the termination of analyzing one function, we identify secret-dependent memory accesses and translate corresponding memory addressing formulas into SMT formulas (Sec. 6.4) and check for side channels (Sec. 5).

Application Scope. In this research we design our abstract domain **SAS** to analyze assembly code: program memory access can be accurately uncovered by analyzing assembly

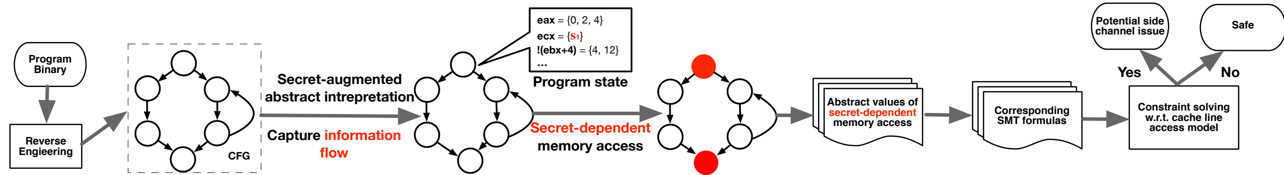


Figure 4: The overall workflow of CacheS.

code, thus supporting a “down-to-earth” modeling of cache behavior (see Sec. 5).

To assist the analysis of off-the-shelf cryptosystems and capture information leaks in the wild, we designed CacheS to directly process binary executables, including stripped executables with no debug or relocation information. We rely on reverse engineering tools to recover program control structures from the input binary, and further build our analysis framework on top of that (see Sec. 7).

6.1 Abstract Interpretation

In this section, we discuss how the proposed abstract domain **SAS** is adopted in our tool, and elaborate on several key points to deliver a practical and scalable analysis.

6.1.1 Initialization

Before the analysis, we first initialize certain program points with $\{s_i\}$ to represent the initial secret program information; for the rest their corresponding initial states are naturally defined as $\{\}$, or $\{e\}$ for the stack register `esp`.

Program secrets are maintained in registers or memory cells (e.g., on the stack) during execution. Since CacheS is designed to directly analyze binary code, we must first recognize the location of program secrets. We reverse-engineer the input binary and mark the location of secrets manually. Once the locations of secrets are flagged, we update the initial value set of corresponding variables (i.e., registers or memory cells) with a secret symbol s_i. Additionally, while “manual reverse engineering” is sufficient for studies in this research, it is always feasible to leverage automatic techniques [26] to search for secrets directly from executables or secret-aware compilers to track secret locations when source code is available. We leave this to future work.

In addition, since program secrets may be stored in a region of sequential memory cells (e.g., in an array), we create another identifier named u to represent the base address of the secret memory region. While u itself is treated as *public information*, we specify that memory loading from u will obtain program secrets; that is, we introduce one s_i for each memory loading via u.

6.1.2 Program State

At each program point, CacheS maintains a lookup table that maps variables to value sets; each value set $S \in \mathbf{SAS}$ consists of abstract values $f \in \mathbf{AV}$ representing possible values of a variable at the current program point. While the “lookup table” is an essential piece of any non-trivial analysis framework, our study has shown that naively-designed program state representations in CacheS could consume significant

Program State Lookup Table

new key	key	value	new value
ebx	ebx	{14+esi₀*4}	{ p }
ecx	ecx	{ 12 }	{ 12 }
eax	eax	{8+k₂*4}	{8+s₂*4}
esp	esp	{esp₀-120}	{e-120}
!(ebx)	!(14+esi₀*4)	{ k₂ }	{ s₂ }
!(ecx-4)	!(8)	{eax₀+4}	{ p }
!(eax)	!(8+k₂*4)	{ 14 }	{ 14 }
!(e-120)	!(esp₀-120)	{ 33 }	{ 33 }

Figure 5: A sample program state lookup table. $\texttt{esp}_0$, $\texttt{eax}_0$ and $\texttt{esi}_0$ in the “key” and “value” entries are symbols representing the register initial values. Symbol ! means pointer dereference, for example !(`eax`) means memory loading from address stored in `eax`. Lookup tables at each program point are the major factor for memory usage, and we optimize the design by replacing “key” and “value” columns with “new key” and “new value” columns, respectively (see Sec. 6.1.2). Hence, shaded boxes are eliminated in CacheS.

amounts of computing resources and impede the analysis of non-trivial programs. Thus, at this step we seek to design a *concise* and *practical* representation of program states. For the rest of this section, we first explain a “baseline” implementation of the lookup table, and further discuss two refinements.

The “Baseline” Approach. A sample lookup table is shown in Fig. 5 (the “key” and “value” columns), where each table maps registers and memory addressing formulas to their corresponding sets for logic formulas $l \in \mathbf{L}$. When it encounters a memory access instruction, CacheS computes the memory addressing formula and searches for its existence in the lookup table. (This requires some “equivalence checking”; the details will be explained in Sec. 6.1.4). If the search identifies an entry in the table, CacheS extracts or updates the content of that entry accordingly. Consider the example in Listing 1, where we first store concrete data 14 into memory via address stored in `eax`, and further load it out into `ebx`.

Listing 1: Sample instructions.

```
store eax , 14
load  ebx , eax
```

Knowing that value set of `eax` is 8+$\texttt{k}_2$*4 (third entry in Fig. 5), the first instruction creates an entry from address 8+$\texttt{k}_2$*4 to 14 (Fig. 5 shows program states after executing the first instruction). Further memory loading would acquire the value set in `eax`, and then reset the entry of `ebx` with 14 in the state lookup table of the second instruction.

Reading from unknown registers and memory locations would introduce symbols of different credentials regarding our information flow policy (see Sec. 6.2 for details).

Optimization of Table Values. While the precisely tracked logic formulas $l \in \mathbf{L}$ result in notable computing resource usage (Sec. 3), the proposed abstract domain **SAS** (Sec. 4) enables succinct representation of abstract values. As shown in Fig. 5, the "value" column of the lookup table is now replaced by the "new value" column. Consequently, memory consumption is considerably reduced (details are reported in our evaluation section).

Optimization of Table Keys. Since only abstract values are traced in **SAS**, the "key" column can be updated into a compact representation as well. However, using symbols such as p as the key will result in an imprecise modeling of memory addresses.

CacheS optimizes the "key" column in the following way. For most memory related entries, instead of using abstract memory addressing formulas, memory access expressions (expressions of registers and constant offsets) are used as keys. For example, the first instruction in Listing 1 uses memory access expression (i.e., "eax") instead of its abstract value $8 + s_2 * 4$ for memory lookup. Hence, when analyzing the store instruction, CacheS creates (or updates) an entry in the lookup table, which uses !(eax) as the key (here symbol "!" means pointer dereference). Likewise, for memory load, !(eax) will be used to look up the program state table. To safely preserve lookup entries via expressions, whenever the value set of a register is reset in the analysis, entries in the table are deleted if their keys are memory access expressions via the newly-updated register.

Nevertheless, since stack register `esp` is frequently manipulated to access stack memory, we preserve abstract addressing formulas via e to keep track of stack memory access precisely (e.g., the last entry in the "new key" column of Fig. 5).

6.1.3 Order of Program State

When multiple program states are possible for a program point, it is important to define the "merge" operation in abstract interpretation. Such an operation can be defined based on the least upper bound operation $\sqcup$ of **SAS** (recall that **SAS** forms a lattice (Sec. 4.2)).

Given lookup tables T_1 and T_2 representing two program states, $T_1 \sqcup T_2$ is defined as the following table, say T_3:

- T_3's key set is the union of the key sets of T_1 and T_2;
- For each key k in T_3, $T_3[k] = T_1[k] \sqcup T_2[k]$ (assuming $T_1[k]$ or $T_2[k]$ is an empty set if k is not in the table).

Moreover, the least upper bound of program states entails the partial order of any two program states: $T_1 \sqcup T_2 = T_1 \leftrightarrow T_2 \subseteq T_1$.

6.1.4 Memory Model

When encountering a memory load and store operation, we must decide which memory cell is accessed by tracing the memory address. However, considering CacheS models program semantics with abstract values, a memory address can usually contain one or several symbols instead of only concrete data. Therefore, policies (i.e., a "memory model") are usually required to determine the location of an accessed memory cell given a symbolic pointer.

When defining the abstract semantics within **SAS** (see our technical report [76]), we assume the assistance of a sound points-to analysis module as pre-knowledge. Nevertheless, finding such a convenient tool for assembly code of large-scale cryptosystems is quite difficult in practice. We have tried several popular "end-to-end" binary analysis platforms that take an executable as the input and perform various reverse engineering campaigns including points-to analysis; nevertheless, so far we cannot find a practical and robust solution to our scenario.

Therefore, we aim to implement a rigorous memory model by solving the *equality constraints* of two abstract formulas. However, tentative tests show that such a memory model may lose considerable precision in terms of reasoning symbolic pointers and may also not be scalable enough. On the other hand, since keys in the memory lookup table are formulas of e (for stack pointers; recall that e represents the initial value of `esp`) or memory access expressions (for other pointers), the current implementation of CacheS rigorously reasons on the equality constraints if abstract values are composed of e and concrete offsets, which is indeed often the case in analyzing assembly code. For the rest (e.g., e and symbolic offsets), we reason on the syntactical equivalence of memory access expressions. This design tradeoff may incorrectly deem equivalent symbolic pointers inequivalent (due to the symbolic "alias" issue) but not vice versa. Experiments show that this memory model is efficient enough to handle real-world cryptosystems while being promisingly accurate.

6.2 Information Flow

Considering that information leaks detected in this research are derived from secret-dependent memory accesses, CacheS keeps track of the secret program information flow throughout the analysis. In this section we elaborate on cases where the secret information can be propagated.

Variable-Level Information Flow. The explicit information flow is modeled in a straightforward way. Since variables (i.e., registers, memory cells, and CPU flags) are modeled as abstract formulas, high credential information (exhibited as abstract value $f \in \mathbf{AV}_s$) would naturally "flow" among variables during the computations. Moreover, reading from unknown variables (those with empty value sets) generates a symbol p as a proper over-approximation.

Information Flow via Memory Loading. By knowing the underlying memory layout, it could be feasible to infer table lookup indexes by observing the memory load outputs,

hence leaking table indexes of secrets to attackers. It should be noted that such cases are not rare in real-world cryptosystems, where many precomputed data structures are deployed in the memory to speed up computations. Thus, we define policies to capture information flow through memory loading. To do so, for a load operation, whenever the value sets of its base address or memory offset include formula $f \in \mathbf{AV}_s$, CacheS assigns the memory content to a fresh s_i, indicating secret information could have potentially propagated to the value being read. In contrast, when loading from unknown memory cells (memory cells of empty value sets) via non-secret addresses, we create a p to update the memory reader.

While most memory addressing formulas refer to specific locations in the memory, symbols p and $\top$ represent any program (public) information. To safely approximate memory read access via p and $\top$, CacheS assigns $\top$ to the memory reader. In case a memory storing is via symbol p or $\top$, we terminate the analysis since this would rewrite the whole memory space. Additionally, we note that memory loading and storing via $\top$ are considered to be information leaks as well since $\top$ implies that a variable has certain residual secrets (see Sec. 6.4).

6.3 Interprocedural Analysis

Our interprocedural analysis is context-sensitive. We build a classic function summary-based interprocedural analysis framework, where a summary $(\langle f', i\rangle, o)$ of a function call towards f maps the calling context $\langle f', i\rangle$ (f' is the caller name and i is the input) to the function call output o. CacheS maintains a set of summaries for each function f, and for an upcoming call of f, its calling context is first checked regarding the existing summaries of f. In case the context is a subset of any recorded entries (the partial order of calling context is derived from the order of program states defined in Sec. 6.1.2), the analysis will be skipped and we directly return the corresponding output.

To recover the function inputs, we inquire the employed reverse engineering platform (details are given in Sec. 7) to obtain the number of parameters the approaching function has. According to the calling convention of 32-bit x86 platforms, a memory stack is used to store function parameters; thus, we construct stack memory addresses of function parameters and acquire the value set of each parameter from the program state lookup table at the call site. If some memory cells of function parameters are absent, symbol p is used as an over-approximation. To compute the output information of a function, we join program states at every return instruction when the analysis of the target function terminates, which over-approximates the function return states.

6.4 Translating Abstract Values into SMT Formulas

As noted earlier (Sec. 5), cache-access side channels are summarized into SMT constraints. Upon the termination of analyzing each function, we identify secret-dependent memory addresses $a \in \mathbf{AV}_s$ and build the side channel constraints. SMT solvers are used to solve the constraint and check whether different secrets can lead to cache line access variants. Nevertheless, while many works to date leverage symbolic execution to construct SMT formulas, here we reason on program states within **SAS**. Therefore, before constraint checking, we first translate abstract formulas into SMT formulas.

Each abstract formula is maintained as a symbolic "tree" in CacheS, where tree leaves are symbols and concrete data while other nodes are operators. At this step, we translate each leaf on the tree into a bit vector implemented by a widely-used SMT solver—Z3 [30]; a bit vector would be instantialized with a numeric value if it was derived from a constant. In addition, we translate abstract operators on the tree into bit vector operations in Z3. Hence, an abstract formula tree would be reduced bottom-up into an SMT formula.

Translate Secret Symbols into Unique Bit Vectors. As noted earlier, s_i symbols are semantically different, each of which represents different pieces of secrets. For the implementation, we assign a unique id for each newly-created s_i symbol, which further leads to the creation of unique bit vectors at this step. In contrast, p (and e) symbols are transformed into identical bit vectors.

Memory Access via $\top$. It is easy to see that $\top$ implies that a variable has some residual secrets along with possibly public information. Hence, in addition to checking the constructed SMT constraints with Z3, memory accesses are flagged as vulnerable whenever their corresponding addressing formulas are $\top$.

7 Implementation

CacheS is mainly written in Scala (in 6,764 LOC; counted by CLOC [29]). The tentative implementation (in 7,163 LOC), which models program semantics with logic formulas (Sec. 3), is maintained as a separate "branch" of the code base.

Starting from an input binary code, the first step is to recover the assembly program as well as control flow and call graphs. Here we employ a popular reverse engineering tool, IDA-Pro (version 6.9) for the reverse engineering task [1]. We use the default configurations of IDA-Pro to recover assembly code and program control structures from the input executables.

Assembly Lifting. Many existing binary analysis infrastructures have provided facilities to lift x86 assembly code into a high-level intermediate representation. Without reinventing the wheel, here we employ a well-developed binary analysis platform BINNAVI [34] to transform x86 assembly code into a *platform-independent* intermediate language, REIL [72]. Our analysis procedures are built on top of the recovered representations. In addition, for a formal definition of program concrete semantics in terms of the REIL language, please refer to our technical report [76].

The current implementation of CacheS analyzes ELF binaries on the x86 platform. Nevertheless, since REIL language is designed as *platform-independent*, there is no fundamental limitation for CacheS to analyze binaries of other

Table 1: Cryptosystems analyzed by CacheS.

Implementation	Versions	Analysis Starting Function	Implement Which Algorithm
Libgcrypt [48]	1.6.1, 1.7.3	_gcry_mpi_powm	RSA/ElGamal
OpenSSL [59]	1.0.2f, 1.0.2k	BN_mod_exp_mont_consttime	
mbedTLS [57]	2.5.1	mbedtls_mpi_exp_mod	RSA
OpenSSL [59]	1.0.2f, 1.0.2k	_x86_AES_decrypt_compact	AES
mbedTLS [57]	2.5.1	mbedtls_internal_aes_decrypt	

formats or from other platforms (e.g., PE binaries on Windows) as long as the assembly instructions can be translated into REIL statements. As aforementioned, our current prototype focuses on 32-bit ELF binaries since the state-of-the-art REIL lifter (BinNavi [34]) does not have an official support for 64-bit binaries. However, the proposed technique shall be applicable to 64-bit binaries with no additional technical hurdles.

Recover x86 Memory Access Instructions from REIL Statements. As noted in Sec. 6.1.2, we use memory access expressions instead of address formulas as the key to simplify the memory lookup. While the memory access expressions can be acquired by checking assembly instructions, note that our analysis is launched on REIL IR; one memory access instruction is extended into multiple IR statements. Hence, we perform def-use analysis to "collapse" IR statements belonging to the same instruction and recover the corresponding memory access expression.

Critical Functions. CacheS is designed to perform both inter and intra-procedural analysis on any binary code component. For the evaluations in this research, instead of starting from the program entry point, analyses were launched on critical functions of cryptosystems that have become the target for many previous attacks. Such critical functions are the starting points of our interprocedural analysis, and we recursively discover all the reachable functions on the call graph. As reported in our evaluation (see Table 2), these recursively collected functions usually form a non-trivial subgraph on the program call graph. In addition, taking these critical functions as the starting points of CacheS makes it easier to compare our findings with existing work.

8 Evaluation

In contrast to many previous studies in which cache-based side channels are detected from only simple cases, CacheS is evaluated on several real-world cryptosystems. As reported in Table 1, three cryptosystems are evaluated in this research. OpenSSL and Libgcrypt are widely used cryptosystems on multi-purpose computers, while mbedTLS is commonly adopted by embedded devices. Eight critical functions are selected as the starting point of our analysis, which covers major security-sensitive components in three crypto algorithm implementations: RSA, AES, and ElGamal.

To prepare CacheS inputs, we compile test programs shipped in each cryptosystem and link with the corresponding libraries. All the crypto libraries are written in C. We build each library and test program into a 32-bit ELF binary on Ubuntu 12.04 with gcc compiler (version 4.6.3).

8.1 Evaluation Result Overview

Table 2 presents the evaluation result overview. In summary, 208 information leak points are reported from the real world cryptosystems evaluated in this research. We interpret the results as promising; most of the evaluated cryptosystems contain information leaks due to cache-based side channels, and CacheS helps to pinpoint these leaks with program-wide static analysis.

It is commonly acknowledged that the table lookup implementation of the AES decryption routine is vulnerable to various real-world cache attacks. CacheS identifies 32 information leaks from the AES implementations of OpenSSL (versions 1.0.2f and 1.0.2k), and 64 leaks from mbedTLS. Indeed, all of these issues are lookup table queries via direct usages of secrets, which is consistent with findings in existing research [25, 77].

Existing research has pinpointed multiple information leaks in the modular exponentiation implementation of OpenSSL and Libgcrypt [77, 52]; vulnerable functions are adopted by both RSA and ElGamal for decryption. CacheS confirmed these findings (see Sec. 8.4 for one false positive in OpenSSL). Furthermore, CacheS successfully revealed a much larger information leakage surface than existing trace and static analysis based techniques, because of its scalable modeling of program semantics. Table 2 shows that CacheS identifies more information leaks from Libgcrypt and OpenSSL in addition to confirming all issues reported by CacheD [77]. Moreover, CacheS identifies multiple information leakage sites from the modular exponentiation implementation of mbedTLS, which, to the best of our knowledge, is unknown to the research community.

While 40 information leakage sites are reported in Libgcrypt (version 1.6.1), our study shows that they have been fixed in version 1.7.3. Without secret-dependent memory accesses, the RSA/ElGamal implementation of Libgcrypt 1.7.3 is generally accepted as safe regarding our threat model. Our evaluation reports consistent findings that no leak is detected regarding our threat model on secret-dependent cache-line accesses (but we do find secret-dependent control flows, see Sec. 8.5).

Computing Resource. Our evaluation is launched on a machine with 2.90 GHz Intel Xeon(R) E5-2690 CPU and 128 GB memory. For each context-sensitive analysis campaign, Table 2 presents the covered functions, contexts, and processed IR instructions. We report that CacheS takes less than 1700 CPU seconds to process all the test cases, and on average the peak memory usage to evaluate one case is less than 5 GB. Overall, CacheS finished all the analysis campaigns with reasonable amount of computing resources, and we interpret that the promising results demonstrate the high scalability of CacheS in analyzing real-world cryptosystems.

Modeling Program Semantics with Logic Formulas. As noted in Sec. 3, we tentatively implement the idea of modeling program concrete semantics with logic formulas. Note that in addition to the semantics modeling, all the design and evaluation settings are unchanged.

Table 2: Evaluation result overview. We compare the identified information leakage sites by CacheS with a recent research (CacheD [77]), and we report CacheS can identify all the leakage sites reported by CacheD. A summary of all leaks can be found at the extended version of this paper [76].

Algorithm	Implementation	Information Leakage Sites (known/unknown)	# of Analyzed Procedures	# of Analyzed Contexts	Processing Time (CPU Seconds)	# of Processed REIL Instructions	Peak Memory Usage (MB)	Information Leakage Units	Results Reported in CacheD [77]		
									Leakage Sites	Processing Time	Leakage Units
RSA/ElGamal	Libgcrypt 1.6.1	22/18	60	81	228.8	50,436	7,749	11	22	14293.6	5
RSA/ElGamal	Libgcrypt 1.7.3	0/0	59	59	182.2	33,386	5,823	0	0	11626.0	0
RSA/ElGamal	OpenSSL 1.0.2k	2/3	71	81	179.2	83,183	6,134	2	N/A	N/A	N/A
RSA/ElGamal	OpenSSL 1.0.2f	2/4	68	72	169.5	80,096	6,113	3	2	165.6	2
RSA	mbedTLS 2.5.1	0/29	29	36	775.9	35,963	9,654	2	N/A	N/A	N/A
AES	OpenSSL 1.0.2k	32/0	1	1	33.2	3,748	620	1	N/A	N/A	N/A
AES	OpenSSL 1.0.2f	32/0	1	1	35.8	3,748	578	1	32	48.5	1
AES	mbedTLS 2.5.1	64/0	1	1	32.8	4,803	619	1	N/A	N/A	N/A
Total		154/54	290	332	1,637.4	295,363	37,290	21	56	26,133.7	8

Table 3: Model program semantics in the logic formulas $l \in$ **L** and **SAS** and test OpenSSL 1.0.2k. The second and third rows report the modeling results with logic formulas, while the last row reports results in **SAS**. The comparison of these two program modelings is given in Sec. 3.

Algorithm	Execution Time (CPU Second)	# of Processed Function	# of Processed Context	Peak Memory Usage (MB)	Detected Leaks
RSA/ElGamal	timeout (> 5 CPU hours)	15	28	7,283	N/A
AES	timeout (> 5 CPU hours)	1	1	47,798	N/A
RSA/ElGamal	timeout (> 5 CPU hours)	28	85	53,054	N/A
AES	115.8	1	1	621	32
RSA/ElGamal	179.2	71	81	6,134	5
AES	33.2	1	1	620	32

The first two rows of Table 3 give the evaluation results for the AES and RSA/ElGamal implementations in OpenSSL 1.0.2k, both of which report a "timeout" after 5 CPU hours. As explained in Sec. 3, we extend the prototype with loop induction variable detection, and the third row reports the results of the re-launched tests. Still, the RSA/ElGamal case throws a timeout (a reflection on this tentative evaluation is given in Sec. 3). In summary, we interpret that the **SAS** proposed in this research has largely improved the analysis scalability, which serves as an indispensable component to pinpoint cache-based timing channels in real-world cryptosystems.

Comparison with CacheAudit.[2] Besides CacheD [77], we also compare our results with CacheAudit [32]. CacheAudit failed on all of our test cases for two reasons. First, two of our cases contain some x86 instructions that are not handled by CacheAudit. Second, CacheAudit refuses to analyze indirect function calls when constructing the control flow graph. In addition, we also describe the key differences between CacheS and CacheAudit in Sec. 10.

Identifying Information Leakage Units. Considering some occurrences of information leaks are on adjacent lines of a code component (a summary of all leaks can be found at the extended version of this paper [76]), once a leak is flagged by CacheS, presumably any competent programmer shall spot and remove all the related defects. Therefore, we group the flagged information leaks to assess the utility of CacheS and also estimate the bug fixing effort. Though it can be slightly subjective, we propose a metric according to the source code locations of defects: information leaks will be grouped together as a "leakage unit" if they are within the same or adjacent C statements (e.g., within the same loop or adjacent if branches). Also, if a macro is expanded at different program points (e.g., the macro `MPN_COPY` which contains information leaks in Libgcrypt 1.6.1), we count it only once.

As reported in Table 2, CacheS identified 21 units of information leaks. We also grouped the findings of CacheD with the same metric. We have confirmed that CacheS covered all leakage units reported in CacheD, and further revealed new leakage units within statements or functions not covered by CacheD (e.g., 6 new leakage units in Libgcrypt 1.6.1). Overall, we interpret the evaluation results as promising; trace-based analysis, like CacheD, is incapable of modeling the program collecting semantics, and therefore underestimates the attack surface.

Confirmation with Library Authors. As shown in Table 2, we found unknown information leaks from OpenSSL (versions 1.0.2f and 1.0.2k) and mbedTLS (version 2.5.1). Our findings were reported and promptly confirmed by the OpenSSL developers [4]; the latest OpenSSL has been patched to eliminate these leaks (the leaks are discussed shortly in Sec. 8.4). At the time of writing, we are waiting for responses from the mbedTLS developers.

8.2 Exploring the Leaks in mbedTLS

Although mbedTLS developers have not confirmed our findings, we conduct further study of the 29 flagged information leakage sites from this library to check whether they can lead to cache-based side channels.

As mentioned above, the constraint solver provides at least one pair of satisfiable solutions (a pair of secrets k and k') to each leakage site (Sec. 5). To verify one leak, we instrument the program source code and modify secrets with k and k'. We then compile the instrumented programs into two binaries and monitor the execution of each binary executable via a widely-used hardware simulator (`gem5` [18]). The compiled code is fed with test cases shipped with the cryptosystems, and we use the full-system simulation mode of `gem5` to monitor the execution of the instrumented program. The full-system simulation mode uses 64-bit Ubuntu 12.04 (this mode only supports 64-bit OS) to host the application code. We compile the instrumented source code into 64-bit binaries since executing 32-bit binaries on the 64-bit OS throws some TLB translation exceptions (this issue is also reported in [77]). The configuration of `gem5` is reported in Table 4. At

[2] `https://github.com/cacheaudit/cacheaudit`

Table 4: 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 5: Hardware simulation results.

# of CacheS Detected Leakage Sites	# of Executed Leakage Sites	Cache Line Access Variants	Cache Status Variants
29	14	14	6

the leakage point, we intercept the cache access from CPU to L1 Data Cache; the accessed cache line and corresponding cache status (hit vs. miss) are recorded.

As shown in Table 5, among 29 information leakage sites found in mbedTLS, 14 sites are covered during simulation. We observe that different cache lines are accessed at these leakage points, when instrumenting the program with secrets k and k'. In other words, by observing the access of different cache lines, attackers will be able to infer a certain amount of secret information. In addition, we report that cache status variants (in terms of cache hit vs. miss) are observed in several cases. In summary, we interpret the verification results as highly promising; we have confirmed that all the executed information leakages are *true positives* since cache line access variants are observed.

Although the employed program inputs cannot lead to the full coverage of every leakage site, we manually checked all the uncovered cases, and we found that these cases share the same pattern as the covered leaks. For instance, the covered and uncovered leaks are the same inline assembly sequences residing within different paths. Overall, we interpret it as convincing to conclude that all the detected information leaks in mbedTLS are true positives.

8.3 Case Study of Leaks in mbedTLS

This section presents a thorough case study of several information leaks identified by our tool. As presented in Table 2, we identified 29 information leakage points in mbedTLS 2.5.1. In particular, the first four leaks were found in the function `mpi_montmul` (source code is given in Fig. 6(a)), which is a major component of the modular exponentiation implementation in mbedTLS. The value of function parameter `B` is derived from a window size of the secret key (line 2). In `mpi_montmul`, `B` is used as a pointer to access elements in a C struct (line 6, line 10, line 11). We envision that different program secrets would derive into different values of `B`, which further lead to the access of different cache lines in secret-dependent memory accesses.

The evaluation shows consistent findings. As shown in Fig. 6(b), CacheS identifies four suspicious memory accesses in `mpi_montmul` (two pointer dereferences at line 6 of Fig. 6(a) are optimized into one memory load at line 2 of Fig. 6(b)). By checking the constraint solver, we find a pair of program secrets that affect the value of `B` and further lead to the access of different cache lines at the first memory access (the solution is given in Fig. 6(c)).

We then instrument the program private key with the solver provided solutions in Fig. 6(c) and observe the runtime cache access within `gem5`. This secret pair is generated by analyzing the first leakage memory access, but since variants of `B` may affect the following memory traffic as well, we report the cache status at all the suspicious memory accesses in `mpi_montmul`. We note that while CacheS analyzes 32-bit binaries, at this step we compile the instrumented source code into 64-bit binaries since the simulated OS throws some exceptions when running 32-bit code. After compilation, the five leakage points in the source code actually produce three memory load instructions in the 64-bit assembly code. Cache behaviors, including the accessed cache line and the corresponding cache status, are recorded at these points. Fig. 6(d) presents the simulation results. Due to the limited space, we provide only the first seven records (59568 records in total). Program counters 0x40770a, 0x407744 and 0x40775d represent the three identified memory loads of information leaks. It is easy to see that different cache lines are accessed at each point. Additionally, a timing window of one cache hit vs. miss is found (this memory access represents a table lookup in the first element of `B->p`).

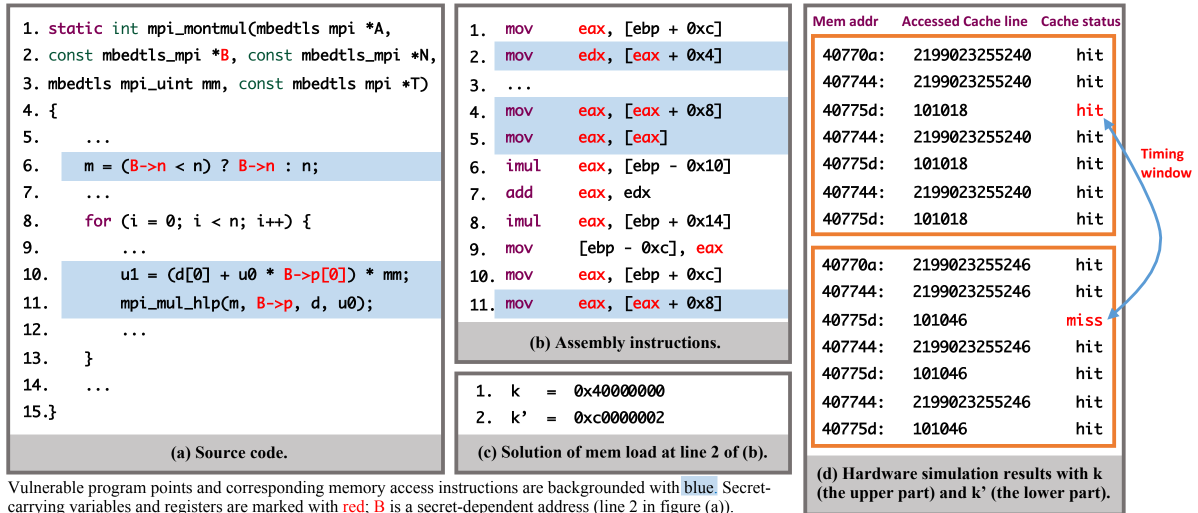
8.4 Information Leaks in the Modular Exponentiation Algorithm

Both RSA and ElGamal algorithms employ the modular exponentiation algorithm for decryption. Existing research has reported that such an algorithm is vulnerable to cache-based timing channel attacks [77, 52]. Here, we evaluate the corresponding implementations in OpenSSL, Libgcrypt, and mbedTLS. As reported in Table 2, CacheS successfully revealed a much larger leakage surface, including 80 (54 unknown and 26 known) information leaks, from our test cases.

Information Leaks in Libgcrypt. A large number of leakage points are reported from the sliding window-based modular exponentiation implementation in Libgcrypt 1.6.1. Existing research has pointed out the direct usages of (window-size) secret keys as *exploitable* [52], and CacheS pinpointed this issue. In addition to the 4 direct usages of secrets, we further uncovered 36 leaks due to the propagation of secret information flows, as CacheS keeps track of both variable-level and memory loading based information flows (Sec. 6.2).

While previous trace-based analysis also keeps track of information flow propagation (i.e., CacheD [77]), CacheS still outperforms CacheD because of its program-wide analysis. With the help of CacheD's authors, we confirmed that CacheS can detect all 22 leaks reported in CacheD [77], and further reveals 18 additional points.

Information Leaks in mbedTLS. CacheS has also identified leaks in another commonly used cryptosystem, mbedTLS. Appendix E presents several leaks found in the mbedTLS case. In general, function `mbedtls_mpi_exp_mod` implements a sliding window-



(a) Source code. (b) Assembly instructions. (c) Solution of mem load at line 2 of (b). (d) Hardware simulation results with k (the upper part) and k' (the lower part).

Vulnerable program points and corresponding memory access instructions are backgrounded with blue. Secret-carrying variables and registers are marked with red; B is a secret-dependent address (line 2 in figure (a)).

Figure 6: Case study of information leaks in mbedTLS. The constraint solver finds a pair of secrets (`k` and `k'`) which leads to the access of different cache lines at line 2 of (b).

based modular exponentiation, which leads to secret-dependent memory accesses (precomputed table lookup). The table lookup statement (line 10) does not generate a leak point since it only gets a pointer referring to an array element, however, further memory dereferences on the acquired pointer reveal 4 direct usages of secrets (discussed in Sec. 8.3). We also find 25 leaks due to the propagation of secret information flows (Sec. 6.2).

We note that mbedTLS uses RSA exponent blinding as a countermeasure [43], which practically introduces noise and mitigates cache side channels (but is still exploitable with enough collisions or if the attacker can derive the exponent from a single trace). We leave it as future work to model the feasibility of exploitations, given the identified leaks and also taking program randomness (e.g., exponent blinding) into consideration.

Information Leaks in OpenSSL. Information leaks in OpenSSL are within or derived from functions counting the length of a secret array. Fig. 7 presents a function that contains 4 memory accesses which engender information leaks: `BIGNUM` maintains an array of 32-bit elements, and `BN_num_bits` counts the number of bits within a given big number. Since the last element of the array may be less than 32 bits, a lookup table is used to determine the exact bits in the last element of the secret array (function call at line 7 in Fig. 7). By storing secrets within a big number structure, table queries in `BN_num_bits_word` could lead to secret-dependent memory accesses.

While CacheD [77] flags only one information leak (line 26 in Fig. 7) covered by its execution trace, CacheS detects more leaks. As shown in Fig. 7, four table queries are analyzed by CacheS, and all of them are flagged as leaks. In addition to these four direct usages of secrets, CacheS also finds one more leak in OpenSSL 1.0.2k, and two in OpenSSL 1.0.2f, both are due to the propagation of secret information flows.

False Positive. In addition to the issues in Appendix C that have been confirmed and fixed by the OpenSSL developers, we also find one *false positive* when analyzing OpenSSL 1.0.2f. To defeat side channel attacks against the precomputed table lookup, OpenSSL forces the cache access at the table lookup point in a constant order [33]. This constant order table lookup is demonstrated with a sample C code in Appendix D. The base address of the lookup table is aligned to zero the least-significant bits, and scatter and gather methods are employed to mimic the Fortran-style memory allocation to access the table in a constant order and remove timing channels.

Ideally, with the base address being aligned, the table access should not produce an information leak regarding the cache line access model (but scatter-gather implementation can also be exploited with cache bank attacks [87]. As discussed in Sec. 5, our cache access constraint can be further extended to capture cache bank side channels). However, since public information (e.g., base address of the table) is abstracted as a symbol p in CacheS, the alignment is not modeled. Therefore, CacheS incorrectly flags the table lookup as a leak point, which leads to a false positive.

8.5 Flag Secret-Dependent Control Flow

To reduce false negatives and also show the versatility of CacheS, we extend CacheS to search for secret-dependent branch conditions. Similar to the detection of secret-dependent memory accesses (Sec. 5), we check each conditional jump and flag secret-dependent jump conditions. The conditional jump in REIL IR is `jcc`, and the value of its first operand specifies whether the jump is taken or not. We trans-

late each secret-dependent condition c into an SMT formula f and solve the following constraint:

$$f \neq f[s_i'/s_i] \quad (2)$$

where a satisfiable solution indicates that different secrets lead to the execution of different branches. In addition, since REIL IR creates *additional* `jcc` statements to model certain x86 instructions (e.g., the shift arithmetic right `sar` and bit scan forward `bsf`), we rule out `jcc` statements if their corresponding x86 instructions are *not* conditional jumps. In this research we do not take `jcc` into consideration if it does not represent an x86 conditional jump, since in general the silicon implementations of x86 instructions on mainstream CPUs have *fixed latency* [2].

Table 6 presents the evaluation results, including the information leakage units produced by the same metric used in Table 2. While secret-dependent control flow is absent in all the AES cases, CacheS pinpoints multiple instances in every RSA/ElGamal implementation. We further manually studied each of them, and we report that besides 4 false positives (explained later in this section), all the other cases represent secret-dependent branch conditions. In the example given in Appendix F, the value of `bits`, which is derived from the private key, is used to construct several conditions. Similar patterns are also found in other cases.

False Negative. Bernstein et al. exploited the secret-dependent control flows in Libgcrypt 1.7.6 [17], where the leading and trailing zeros of a window-size secret are used to compute a branch condition. While the corresponding vulnerable branches also exist in Libgcrypt 1.7.3, they are not detected by CacheS. In general, 32-bit x86 opcode `bsr` and `bsf` are used to count the leading and trailing zeros of a given operand, and both opcodes are lifted into a while loop implemented by a `jcc` statement (for the definition of their semantics, see the `bsr` and `bsf` sections of the x86 developer manual [3]). Consider a proof-of-concept pseudo-code below:

```
1  t = 0;
2  while (getBit(t, src) == 0) //src could be a secret
3  {
4    t += 1;
5  }
6  return t; // the number of trailing zeros in src
```

where the lifted while loop entails implicit information flow, which is not supported (see Sec. 6.2 for the information flow policy). In addition, although we disable the checking of `jcc` regarding Constraint 2 if its corresponding x86 instruction is *not* a conditional jump (like the `bsr` and `bsf` cases), we report that once enabling the checking of such `jcc` statements, secret-dependent control flows (e.g., line 3 of the pseudo-code) are detected for both cases.

False Positive. We find 4 false positives when analyzing Libgcrypt 1.6.1. This is due to the imprecise modeling of interprocedural call sites. Consider a sample pseudo-code below:

```
1  foo(k, p) { // k is {⊤} and p is {12}
2    if (...) {
3      r = bar(k); // r is {⊤}
4    } else {
5      r = bar(p); // r is {⊤} since ⟨foo,{12}⟩ ⊆ ⟨foo,{⊤}⟩
6      if (r)  // false positive
7        ...
8  }
9
10 bar(i){return i;}
```

where `foo` performs two function calls to `bar` with different parameters. The summary of the first call (line 3) is represented as $(\langle foo, \{\top\}\rangle, \{\top\})$, where $\langle foo, \{\top\}\rangle$ forms the calling context (as explained in Sec. 6.3, a calling context includes the caller name and the input), and the second $\{\top\}$ is the function call output. Then the following function call (line 5) with $\langle foo, \{12\}\rangle$ as the calling context will directly return $\{\top\}$ and cause a false positive (line 6) according to the recorded summary, since $\langle foo, \{12\}\rangle \subseteq \langle foo, \{\top\}\rangle$. Our study shows that such sound albeit imprecise modeling caused 4 false positives when analyzing Libgcrypt 1.6.1.

Table 6: Secret-dependent control branches. We found no issue in the AES implementations. A summary of all leakage points can be found at the extended version of this paper [76].

Implementation	Algorithm	# of Secret-dependent conditions	False Positive	Information Leakage Unit
Libgcrypt 1.6.1	RSA/ElGamal	21	4	9
Libgcrypt 1.7.3	RSA/ElGamal	6	0	4
mbedTLS 2.5.1	RSA	8	0	4
OpenSSL 1.0.2f	RSA/ElGamal	12	0	5
OpenSSL 1.0.2k	RSA/ElGamal	12	0	5
Total		59	4	27

9 Discussion

Soundness. Our abstraction is sound (see our technical report for the proof [76]), but the CacheS implementation is soundy [53] as it roots the same assumption as previous techniques that aim to find bugs rather than performing rigorous verification [55, 84, 54].

CacheS adopts a lightweight but unsound memory model implementation; program state representations are optimized to reduce the memory usage and speed up the analysis. There is a line of research aiming to deliver a (nearly) sound memory model when analyzing x86 assembly [13, 64, 65, 21]. We leave it to future work to explore practical methods to improve CacheS with a sound model without undermining the strength of CacheS in terms of scalability and precision.

Reduce False Positives. Our abstract domain **SAS** models public program information with free public symbols. To further improve the analysis precision and eliminate false positives, such as in the case discussed in Sec. 8.4, one approach is to perform a finer-grained modeling of public program information. To this end, so-called "lazy abstraction" can be adopted to postpone abstraction until necessary [71]. In contrast to our current approach where analyses are performed directly over **SAS**, lazy abstraction provides a flexible abstraction strategy on demand, where different program points can exhibit distinct levels of precision. Well-selected program points for lazy abstraction are critical to achieve scalability. For example, abstraction can be performed at ev-

ery loop merge point or whenever abstract formulas become too large and exhaust the memory resource. We leave it to future work to explore practical strategies for lazy abstraction.

10 Related Work

Timing Attacks. Kocher's seminal paper [44] identifies timing attacks as a potential threat to crypto system. Later work finds that timing information reveals the victim program's usage of data/instruction cache, leading to efficient timing attacks against real world cryptography software, including AES [37, 60, 74, 16, 20, 6], DES [75], RSA [7, 62, 86], ElGamal [90], and ECDSA [15]. Recent work shows that such cache-based timing attacks are possible on emerging platforms, such as cloud computing, VM environments, trusted computing environments, and mobile platforms [66, 85, 83, 89, 52, 49, 56, 69, 24, 36].

Detect Cache-Based Timing Channels. CacheAudit leverages static analysis techniques (i.e., abstract interpretation) to reason information leakage due to cache side channels [32, 33]. CacheS outperforms CacheAudit due to our novel abstract domain. CacheAudit uses relational and numerical abstract domains to only infer the information leakage bound, while our abstract domain models semantics with symbolic formulas, pinpoints information leaks with constraint solving, and enables the generation of counter examples to promote debugging. In addition, we propose a principled way to improve the scalability by tracking secrets and public information with *different granularities*. This enables a context-sensitive interprocedural analysis of real-world cryptosystems for which CacheAudit is not capable of handling. Brotzman et al. [22] propose a static symbolic reasoning technique that also covers multiple program paths. However, their analysis lacks abstraction of public values, and can analyze only small-size programs.

In contract, dynamic analysis-based approaches, such as taint analysis or trace-based symbolic execution, are incapable of analyzing the whole program [77, 82, 41, 81, 38]. CacheD [77] performs symbolic execution towards a single trace to detect side channels. In contrast, abstract interpretation framework approximates the program *collecting semantics*, which formalizes program abstract semantics at arbitrary program points regarding any path and any input. This is fundamentally different and much more comprehensive comparing to a path-based tool, like CacheD. Wichelmann et al. [82] log execution traces and perform differential analysis of various granularities to detect side channels. Weiser et al. [81] detect address-based side-channels by executing test programs under input variants and further compare traces to detect leakages.

Countermeasure. Existing countermeasures against cache side-channel attacks can be categorized into hardware-based and software-based approaches. Hardware-based solutions focus on randomizing the cache accesses with new cache design [79, 80, 45, 78, 51, 50], or enforcing fine-grained isolation with respect to cache usage [70, 42]. Wang et al. propose locking the cache lines and hiding cache access patterns [79], which further obfuscates cache accesses by diversifying the cache mappings [80]. Tiwari et al. [73] devise a novel micro architecture for information-flow tracking by design, where noninterference is deployed as the baseline confidentiality property. Another direction at the hardware level is based on contracts between software and hardware [91, 47, 88], where contracts are enforced by formal methods (e.g., type systems) on the hardware side. Furthermore, some advanced hardware extensions, like hardware transactional memory, have also been leveraged to prevent side channels even inside Intel SGX [35].

Analyses are also conducted on the software level to mitigate side channel attacks [27, 12, 63, 67, 68]. Program transformation techniques are leveraged to remove control-flow timing leaks by equalizing branches of conditionals with secret guards [8], together with a binary static checker [58], and its practicality is evaluated [27]. Constant time code defeats timing attacks by ensuring the control flow, memory accesses, and execution time of individual instruction is secret independent [10, 40, 61, 14, 9, 46].

11 Conclusion

In this paper, we have presented CacheS for cache-based timing channel detection. Based on a novel abstract domain **SAS**, CacheS does fine-grained tracking of sensitive information and its dependencies, while performing scalable analysis with over-approximated public information. We evaluated CacheS on multiple real-world cryptosystems. CacheS confirmed over 154 information leaks reported by previous research and pinpointed 54 leaks not known previously.

12 Acknowledgments

We thank the Usenix Security anonymous reviewers and Gary T. Leavens for their valuable feedback. The work was supported in part by the National Science Foundation (NSF) under grant CNS-1652790, and the Office of Naval Research (ONR) under grants N00014-16-12912, N00014-16-1-2265, and N00014-17-1-2894.

References

[1] IDAPro. https://goo.gl/snmrk3.

[2] Intel® 64 and IA-32 architectures optimization reference manual.

[3] Intel® 64 and IA-32 architectures software developers manual.

[4] Patched OpenSSL vulnerabilities. `https://git.io/fj0iz`, 2018.

[5] ACIICMEZ, O., AND KOC, C. K. Trace-driven cache attacks on AES. In *ICICS* (2006).

[6] ACIICMEZ, O., SCHINDLER, W., AND KOC, C. K. Cache based remote timing attack on the AES. In *CT-RSA* (2006).

[7] ACIICMEZ, O., AND SEIFERT, J. Cheap hardware parallelism implies cheap security. In *FDTC* (2007).

[8] AGAT, J. Transforming out timing leaks. In *POPL* (2000).

[9] ALMEIDA, J. B., BARBOSA, M., BARTHE, G., DUPRESSOIR, F., AND EMMI, M. Verifying constant-time implementations. In *USENIX Sec.* (2016).

[10] ALMEIDA, J. B., BARBOSA, M., PINTO, J. S., AND VIEIRA, B. Formal verification of side-channel countermeasures using self-composition. *Science of Computer Programming* (2013).

[11] APPEL, A. W. *Modern Compiler Implementation in ML.* Cambridge University Press, 2004.

[12] AVIRAM, A., HU, S., FORD, B., AND GUMMADI, R. Determinating timing channels in compute clouds. In *CCSW* (2010).

[13] BALAKRISHNAN, G., AND REPS, T. Analyzing memory accesses in x86 executables. In *CC* (2004).

[14] BARTHE, G., REZK, T., AND WARNIER, M. Preventing timing leaks through transactional branching instructions. *Electronic Notes in Theoretical Computer Science* (2006).

[15] BENGER, N., VAN DE POL, J., SMART, N. P., AND YAROM, Y. "Ooh aah... just a little bit" : A small amount of side channel can go a long way. In *CHES* (2014).

[16] BERNSTEIN, D. J. Cache-timing attacks on AES, 2005.

[17] BERNSTEIN, D. J., BREITNER, J., GENKIN, D., BRUINDERINK, L. G., HENINGER, N., LANGE, T., VAN VREDENDAAL, C., AND YAROM, Y. Sliding right into disaster: Left-to-right sliding windows leak. In *CHES* (2017).

[18] 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. *ACM SIGARCH Computer Architecture News* (2011).

[19] BONEH, D., DURFEE, G., AND FRANKEL, Y. An attack on RSA given a small fraction of the private key bits. In *ASIACRYPT* (1998).

[20] BONNEAU, J., AND MIRONOV, I. Cache-collision timing attacks against AES. In *CHES* (2006).

[21] BRADLEY, A. R., MANNA, Z., AND SIPMA, H. B. What's decidable about arrays? In *VMCAI* (2006).

[22] BROTZMAN, R., LIU, S., ZHANG, D., TAN, G., AND KANDEMIR, M. CaSym: Cache aware symbolic execution for side channel detection and mitigation. In *IEEE SP* (2018).

[23] BRUMLEY, D., AND BONEH, D. Remote timing attacks are practical. *Computer Networks* (2005).

[24] BULCK, V., MINKIN, M., WEISSE, O., GENKIN, D., KASIKCI, B., PIESSENS, F., SILBERSTEIN, M., WENISCH, T. F., YAROM, Y., AND STRACKX, R. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *USENIX Sec.* (2018).

[25] C, A., GIRI, R. P., AND MENEZES, B. Highly efficient algorithms for aes key retrieval in cache access attacks. In *EuroSP* (2016).

[26] CALVET, J., FERNANDEZ, J. M., AND MARION, J.-Y. Aligot: Cryptographic function identification in obfuscated binary programs. In *CCS* (2012).

[27] COPPENS, B., VERBAUWHEDE, I., BOSSCHERE, K. D., AND SUTTER, B. D. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *IEEE SP* (2009).

[28] COUSOT, P., AND COUSOT, R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL* (1977).

[29] DANIAL, A. CLOC. `https://goo.gl/3KFACB`.

[30] DE MOURA, L., AND BJØRNER, N. Z3: An efficient SMT solver. In *TACAS* (2008).

[31] DISSELKOEN, C., KOHLBRENNER, D., PORTER, L., AND TULLSEN, D. Prime+Abort: A timer-free high-precision L3 cache attack using Intel TSX. In *USENIX Sec.* (2017).

[32] DOYCHEV, G., FELD, D., KOPF, B., MAUBORGNE, L., AND REINEKE, J. CacheAudit: A tool for the static analysis of cache side channels. In *USENIX Sec.* (2013).

[33] DOYCHEV, G., AND KÖPF, B. Rigorous analysis of software countermeasures against cache attacks. In *PLDI* (2017).

[34] GOOGLE. BinNavi. `https://github.com/google/binnavi`, 2017.

[35] GRUSS, D., LETTNER, J., SCHUSTER, F., OHRIMENKO, O., HALLER, I., AND COSTA, M. Strong and efficient cache side-channel protection using hardware transactional memory. In *USENIX Sec.* (2017).

[36] GRUSS, D., MAURICE, C., FOGH, A., LIPP, M., AND MANGARD, S. Prefetch side-channel attacks: Bypassing smap and kernel aslr. In *CCS* (2016).

[37] GULLASCH, D., BANGERTER, E., AND KRENN, S. Cache games—bringing access-based cache attacks on AES to practice. In *IEEE SP* (2011).

[38] GUO, S., WU, M., AND WANG, C. Adversarial symbolic execution for detecting concurrency-related cache timing leaks. In *FSE* (2018).

[39] HE, Z., AND LEE, R. B. How secure is your cache against side-channel attacks? In *MICRO* (2017).

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

[41] IRAZOQUI, G., CONG, K., GUO, X., KHATTRI, H., KANUPARTHI, A. K., EISENBARTH, T., AND SUNAR, B. Did we learn from LLC side channel attacks? A cache leakage detection tool for crypto libraries. *CoRR* (2017).

[42] KIM, T., PEINADO, M., AND MAINAR-RUIZ, G. Stealthmem: System-level protection against cache-

based side channel attacks in the cloud. In *USENIX Sec.* (2012).

[43] KOCHER, P. C. Timing attacks on implementations of Diffie–Hellman, RSA, DSS, and other systems. In *CRYPTO* (1996).

[44] KOCHER, P. C. *Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems*. 1996.

[45] KONG, J., ACIICMEZ, O., SEIFERT, J. P., AND ZHOU, H. Hardware-software integrated approaches to defend against software cache-based side channel attacks. In *HPCA* (2009).

[46] KÖPF, B., AND RYBALCHENKO, A. Approximation and randomization for quantitative information-flow analysis. In *CSF* (2010).

[47] 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 *ASPLOS* (2014).

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

[49] LIPP, M., GRUSS, D., SPREITZER, R., MAURICE, C., AND MANGARD, S. Armageddon: Cache attacks on mobile devices. In *USENIX Sec.* (2016).

[50] LIU, F., GE, Q., YAROM, Y., MCKEEN, F., ROZAS, C., HEISER, G., AND LEE, R. B. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *HPCA* (2016).

[51] LIU, F., AND LEE, R. B. Random fill cache architecture. In *MICRO* (2014).

[52] LIU, F., YAROM, Y., GE, Q., HEISER, G., AND LEE, R. Last-level cache side-channel attacks are practical. In *IEEE S&P* (2015).

[53] LIVSHITS, B., SRIDHARAN, M., SMARAGDAKIS, Y., LHOTÁK, O., AMARAL, J. N., CHANG, B.-Y. E., GUYER, S. Z., KHEDKER, U. P., MØLLER, A., AND VARDOULAKIS, D. In defense of soundiness: A manifesto. *Commun. ACM* (2015).

[54] LIVSHITS, V. B., AND LAM, M. S. Tracking pointers with path and context sensitivity for bug detection in c programs. *SIGSOFT Softw. Eng. Notes* (2003).

[55] MACHIRY, A., SPENSKY, C., CORINA, J., STEPHENS, N., KRUEGEL, C., AND VIGNA, G. DR. CHECKER: A soundy analysis for linux kernel drivers. In *USENIX* (2017).

[56] MAURICE, C., WEBER, M., SCHWARZ, M., GINER, L., GRUSS, D., BOANO, C. A., MANGARD, S., AND RÖMER, K. Hello from the other side: SSH over robust cache covert channels in the cloud. In *NDSS* (2017).

[57] mbedtls. `https://tls.mbed.org/`.

[58] 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 *ICISC* (2005).

[59] Openssl. `https://www.openssl.org/`.

[60] OSVIK, D. A., SHAMIR, A., AND TROMER, E. Cache attacks and countermeasures: the case of AES. *CT-RSA* (2006).

[61] PASAREANU, C., PHAN, Q.-S., AND MALACARIA, P. Multi-run side-channel analysis using symbolic execution and Max-SMT. In *CSF* (2016).

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

[63] RAJ, H., NATHUJI, R., SINGH, A., AND ENGLAND, P. Resource management for isolation enhanced cloud services. In *CCSW* (2009).

[64] REPS, T., AND BALAKRISHNAN, G. Improved memory-access analysis for x86 executables. In *CC* (2008).

[65] REYNOLDS, J. C. Reasoning about arrays. *Commun. ACM* (1979).

[66] 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 *CCS* (2009), ACM.

[67] SCHWARZ, M., LIPP, M., AND GRUSS, D. Javascript zero: Real javascript and zero side-channel attacks. In *NDSS* (2018).

[68] SCHWARZ, M., LIPP, M., GRUSS, D., WEISER, S., MAURICE, C., SPREITZER, R., AND MANGARD, S. Keydrown: Eliminating software-based keystroke timing side-channel attacks. In *NDSS* (2018).

[69] SCHWARZ, M., WEISER, S., GRUSS, D., MAURICE, C., AND MANGARD, S. Malware guard extension: Using SGX to conceal cache attacks. In *DIMVA* (2017).

[70] SHI, J., SONG, X., CHEN, H., AND ZANG, B. Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring. In *DSNW* (2011).

[71] THAKUR, A. V., ELDER, M., AND REPS, T. W. Bilateral algorithms for symbolic abstraction. In *SAS* (2012).

[72] THOMAS, D., AND PORST, S. REIL: A platform-independent intermediate representation of disassembled code for static code analysis. In *CanSecWest* (2009).

[73] TIWARI, M., OBERG, J. K., LI, X., VALAMEHR, J., LEVIN, T., HARDEKOPF, B., KASTNER, R., CHONG, F. T., AND SHERWOOD, T. Crafting a usable microkernel, processor, and I/O system with strict and provable information flow security. In *ACM SIGARCH Computer Architecture News* (2011), ACM.

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

[75] TSUNOO, Y., SAITO, T., SUZAKI, T., SHIGERI, M., AND MIYAUCHI, H. Cryptanalysis of DES implemented on computers with cache. In *CHES* (2003).

[76] WANG, S., BAO, Y., LIU, X., WANG, P., ZHANG, D., AND WU, D. Identifying cache-based side chan-

nels through secret-agumented abstract interpretation. In *Arxiv* (2019).

[77] WANG, S., WANG, P., LIU, X., ZHANG, D., AND WU, D. CacheD: Identifying cache-based timing channels in production software. In *USENIX Sec.* (2017).

[78] WANG, Z., AND LEE, R. B. Covert and side channels due to processor architecture. In *ACSAC* (2006).

[79] WANG, Z., AND LEE, R. B. New cache designs for thwarting software cache-based side channel attacks. In *ISCA* (2007).

[80] WANG, Z., AND LEE, R. B. A novel cache architecture with enhanced performance and security. In *MICRO* (2008).

[81] WEISER, S., ZANKL, A., SPREITZER, R., MILLER, K., MANGARD, S., AND SIGL, G. DATA – differential address trace analysis: Finding address-based side-channels in binaries. In *USENIX Sec.* (2018).

[82] WICHELMANN, J., MOGHIMI, A., EISENBARTH, T., AND SUNAR, B. MicroWalk: A framework for finding side channels in binaries. In *ACSAC* (2018).

[83] WU, Z., XU, Z., AND WANG, H. Whispers in the hyper-space: High-speed covert channel attacks in the cloud. In *USENIX Sec.* (2012).

[84] XIE, Y., AND AIKEN, A. Scalable error detection using boolean satisfiability. In *POPL* (2005).

[85] XU, Y., BAILEY, M., JAHANIAN, F., JOSHI, K., HILTUNEN, M., AND SCHLICHTING, R. An exploration of L2 cache covert channels in virtualized environments. In *CCSW* (2011).

[86] YAROM, Y., AND FALKNER, K. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *USENIX Sec.* (2014).

[87] YAROM, Y., GENKIN, D., AND HENINGER, N. CacheBleed: A timing attack on OpenSSL constant time RSA. Tech. rep., Cryptology ePrint Archive, Report 2016/224, 2016.

[88] ZHANG, D., WANG, Y., SUH, G. E., AND MYERS, A. C. A hardware design language for timing-sensitive information-flow security. In *ASPLOS* (2015).

[89] ZHANG, Y., JUELS, A., OPREA, A., AND REITER, M. K. HomeAlone: Co-residency detection in the cloud via side-channel analysis. In *IEEE SP* (2011).

[90] ZHANG, Y., JUELS, A., REITER, M. K., AND RISTENPART, T. Cross-VM side channels and their use to extract private keys. In *CCS* (2012).

[91] ZHANG, Y., AND REITER, M. K. Düppel: Retrofitting commodity operating systems to mitigate cache side channels in the cloud. In *CCS* (2013).

A SAS as a Lattice

To further make **SAS** a lattice, we will need to specify a top element $\top \in \mathbf{SAS}$, a bottom element $\bot \in \mathbf{SAS}$, and a join operator $\sqcup$ over **SAS**.

Set Collapse and Bound. Each element in **SAS** is a set of abstract values $f \in \mathbf{AV}$. Considering f with different degrees of abstractions may exist in one set, here we define reasonable rules to "collapse" elements in a set. The "collapse" function $\text{COL} : \mathbf{SAS} \to \mathbf{SAS}$ is given by:

$$\text{COL}(X) = \begin{cases} \{\top\} & \text{if } \top \in X \\ \{\top\} & \text{else if } p \in X \wedge \mathbf{AV}_s \cap X \neq \varnothing \\ \{p\} & \text{else if } p \in X \wedge \mathbf{AV}_s \cap X = \varnothing \\ X & \text{otherwise} \end{cases}$$

While the first three rules introduce single symbols as a safe and concise approximation, the last rule preserve a set in **SAS**.

In addition, each set in **SAS** is also bounded with a maximum size of N through function BOU as follows:

$$\text{BOU}(X) = \begin{cases} \{\top\} & \text{if } |X| > N \wedge \mathbf{AV}_s \cap X \neq \varnothing \\ \{p\} & \text{else if } |X| > N \wedge \mathbf{AV}_s \cap X = \varnothing \\ X & \text{otherwise} \end{cases}$$

Hence, the abstract value set of any variable is bounded by N during computations within **SAS**, which practically speed ups the analysis convergence (N is set as 50 in this research, see Appendix B for a discussion of different configurations).

With COL and BOU defined, we can finally complete **SAS** as a lattice.

Claim 2. $\mathbf{SAS} = \mathscr{P}(\mathbf{AV})$ *forms a lattice with the top element*

$$\top_{\mathbf{SAS}} = \{\top\}$$

bottom element

$$\bot_{\mathbf{SAS}} = \{\}$$

and the join operator

$$\sqcup = \text{BOU} \circ \text{COL} \circ \cup$$

For further discussion of **SAS**, including the concrete and abstract semantics, soundness proof, etc., please refer to the extended version of this paper [76].

B Evaluating Different Configurations of the BOU Function

The definition of the BOU function includes a parameter N as the maximum size of each abstract value set. Table 7 reports the evaluation results of CacheS with respect to different N. As expected, with the increase of the allowed size, analyses took more time before reaching the fixed point. Also, when the allowed size is small (i.e., N is 1 or 10), the value set of certain registers is lifted into $\{p\}$ rapidly and terminates the analysis due to memory write accesses through p (see Sec. 6.2; we terminate the analysis for memory access of p since it rewrites the whole memory). The full evaluation data in terms of different configurations is available in the extended paper [76].

Table 7: Evaluating different configurations of BOU. When N is set as 1 and 10, several analyses terminated before reaching the fixed point due to memory write accesses through the public symbol p. The full evaluation data in terms of each configuration can be found at [76].

Value of N	True Positive	False Positive	Processing Time (CPU Seconds)
1	N/A	N/A	N/A
10	167	1	584.5
25	207	1	1,446.8
50 (the default config)	207	1	1,637.4
100	207	1	3,563.46

C Unknown Information Leaks in OpenSSL

```
1  int BN_num_bits(const BIGNUM *a) {
2   int i = a->top - 1;
3   bn_check_top(a);
4
5   if (BN_is_zero(a))
6     return 0;
7   return ((i * BN_BITS2) + BN_num_bits_word(a->d[i]));
8  }
9
10 int BN_num_bits_word(BN_ULONG l) {
11   static const char bits[256]={
12     0,1,2,2,3,3,3,3,4,4,4,4,4,4,4,4,
13     ...
14     8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,
15   };
16   if (l & 0xffff0000L) {
17     if (l & 0xff000000L)
18       return bits[l >> 24] + 24;
19     else
20       return bits[l >> 16] + 16;
21   }
22   else {
23     if (l & 0xff00L)
24       return bits[l >> 8] + 8;
25   else
26       return bits[l];
27   }
28 }
```

Figure 7: RSA information leaks found in OpenSSL (1.0.2f). Program secrets and their dependencies are marked as red and the leakage points are **boldfaced**.

D Scatter & Gather Methods in OpenSSL

```
1  char* align(char* buf) {
2    uintptr_t addr = (uintptr_t) buf;
3    return (char*)(addr - (addr&(BLOCK_SZ-1)) + BLOCK_SZ);
4  }
5
6  void scatter(char* buf, char p[][16], int k) {
7    for (int i = 0; i < N; i++) {
8      buf[k+i*spacing] = p[k][i];
9    }
10 }
11
12 void gather(char* r,char* buf,int k) {
13   for (int i = 0; i < N; i++) {
14     r[i] = buf[k+i*spacing];
15   }
16 }
```

Figure 8: Simple C program demonstrating the scatter & gather methods in OpenSSL to remove timing channels. This program should be secure regarding our threat model, but it would become insecure by skipping the alignment function.

E Unknown Information Leaks in mbedTLS

```
1  int mbedtls_mpi_exp_mod(mbedtls_mpi *X, mbedtls_mpi *A,
2    mbedtls_mpi *E, mbedtls_mpi *N, mbedtls_mpi *_RR)
3  {
4    ...
5    while (1) {
6      ei = (E->p[nblimbs] >> bufsize) & 1;
7      ...
8      wbits |= (ei << (wsize - nbits));
9      ...
10     mpi_montmul(X, &W[wbits], N, mm, &T);
11   }
12   ...
13 }
14
15 static int mpi_montmul(mbedtls_mpi *A, mbedtls_mpi *B,
16   mbedtls_mpi *N, mbedtls_mpi_uint mm, mbedtls_mpi *T)
17 {
18    ...
19    m = (B->n < n) ?  B->n :  n;
20    for(i = 0; i < n; i++)
21    {
22      u1 = (d[0] + u0 * B->p[0]) * mm;
23      mpi_mul_hlp(m, B->p, d, u0);
24    }
25    ...
26 }
```

Figure 9: RSA information leaks found in mbedTLS (2.5.1). Program secrets and their dependencies are marked as red and the leakage points are **boldfaced**.

F Secret-Dependent Branch Conditions in OpenSSL

```
1  int BN_mod_exp_mont_consttime(BIGNUM *rr,
2    const BIGNUM *a, const BIGNUM *p,
3    const BIGNUM *m, BN_CTX *ctx,
4    BN_MONT_CTX *in_mont) {
5   ...
6   bits = BN_num_bits(p);
7   if (bits == 0)
8     ...
9
10  window = BN_window_bits_for_exponent_size(bits);
11  for (wvalue = 0, i = bits%window; i>=0; i--,bits--)
12  {
13    ...
14    while (bits >= 0){
15      ...
16    }
17  }
18   ...
19 }
20
21 #define BN_window_bits_for_exponent_size(b) \
22        ((b) > 671 ? 6 : \
23         (b) > 239 ? 5 : \
24         (b) > 79 ? 4 : \
25         (b) > 23 ? 3 : 1)
```

Figure 10: Several secret-dependent branch conditions found in OpenSSL (1.0.2f). Program secrets and their dependencies are marked as red and the information leakage conditions are **boldfaced**. Note that the output of BN_num_bits depends on the private key.