Cache Refinement Type for Side-Channel Detection of Cryptographic Software

Ke Jiang
Nanyang Technological University
Singapore, Singapore
ke006@e.ntu.edu.sg

Yuyan Bao
University of Waterloo
Waterloo, Ontario, Canada
yuyan.bao@uwaterloo.ca

Shuai Wang*
Hong Kong University of Science and Technology
Hong Kong, China
shuaiw@cse.ust.hk

Zhibo Liu
Hong Kong University of Science and Technology
Hong Kong, China
zliudc@cse.ust.hk

Tianwei Zhang*
Nanyang Technological University
Singapore, Singapore
tianwei.zhang@ntu.edu.sg

Abstract

Cache side-channel attacks exhibit severe threats to software security and privacy, especially for cryptosystems. In this paper, we propose CaType, a novel refinement type-based tool for detecting cache side channels in crypto software. Compared to previous works, CaType provides the following advantages: (1) For the first time CaType analyzes cache side channels using refinement type over x86 assembly code. It reveals several significant and effective enhancements with refined types, including bit-level granularity tracking, distinguishing different effects of variables, precise type inferences, and high scalability. (2) CaType is the first static analyzer for crypto libraries in consideration of blinding-based defenses. (3) From the perspective of implementation, CaType uses cache layouts of potential vulnerable control-flow branches rather than cache states to suppress false positives. We evaluate CaType in identifying side channel vulnerabilities in real-world crypto software, including RSA, ElGamal, and (EC)DSA from OpenSSL and Libgcrypt. CaType captures all known defects, detects previously-unknown vulnerabilities, and reveals several false positives of previous tools. In terms of performance, CaType is 16× faster than CacheD and 131× faster than CacheS when analyzing the same libraries. These evaluation results confirm the capability of CaType in identifying side channel defects with great precision, efficiency, and scalability.

Keywords

cryptography; cache side-channel; static analysis; refinement type inference

1 Introduction

Cache-based side channels have demonstrated serious threats to crypto algorithms, such as the symmetric cipher AES [44, 46], the asymmetric cipher RSA [34, 70, 73], and the digital signature (EC)DSA [2, 50, 69]. The essence of these cache attacks is the interference of program memory accesses toward cache units, where secret-dependent memory accesses or program branches leave distinguishable footprints in cache units. Thus, identifying and removing cache interference can eliminate side channel leakage.

Designing novel security-aware cache architectures may eliminate adversarial interference. Prior research relies mostly on two strategies, namely partitioning-based and randomization-based approaches. Strong isolation is achieved in partition isolated caches [20, 62] by physically partitioning the shared cache into multiple zones for applications of various security levels. In contrast, [49, 62, 63, 67] obscure adversary observations by randomizing the cache states. Although it is envisaged that these architectures will eliminate interference and secure programs that run on top of them, recent works show that these randomization-based caches may be still vulnerable to cache side channels [48, 54]. Also, these new cache designs achieve security promise at the expense of performance. Besides, they are not yet ready for commercial use due to extra cost in chip circuit manufacturing.

Software-based mitigation of cache side channels appears increasingly viable. However, manually detecting vulnerable crypto code takes specialized knowledge, which drastically restricts normal developers from analyzing and patching their crypto software. With the fast development of more efficient crypto software under various usage scenarios, launching timely side channel analysis becomes even more challenging. With this regard, developing a general, automated, and efficient analytic tool for detecting cache side channels is receiving broad attention from both academics and industry. Recent works [13, 21, 22, 59, 60, 66] serve as examples of this. In general, these works construct constraints through symbolic modeling of program states and cache accesses. Then, constraint solving techniques (e.g., Z3 [38]) are employed to check the satisfiability of constraints and decide whether the program is vulnerable to cache side channels. While these automated methods have made concrete progress in discovering cache side channels in real-world cryptosystems, they still face a number of obstacles.

Challenge 1: *Software-based analysis needs to address precision issues and be scalable to production crypto libraries.* CacheAudit [21] and its extension [22] calculate the upper bound of information leakage by counting all possible final cache states via abstract interpretation [18]. However, estimating worst-case leakage bound may not reflect the reality. Moreover, CacheAudit cannot pinpoint what/where the vulnerability is, prohibiting the debugging/fixing of analyzed code. Using symbolic execution, CaSym [13] distinguishes two different cache states resulting from secret variants. Though

*Corresponding authors

CaSym covers multiple paths, it suffers from path explosion and is less scalable. CacheS [59], likely the most scalable static tool in this field, also uses abstract interpretation. It achieves higher scalability due to modeling secret/non-secret semantics with symbolic formulas of different granularity. Dynamic approaches, in contrast, analyze concrete execution traces to track program states and pinpoint side channels. CacheD [60] detects secret-dependent memory accesses via symbolic execution, while not considering secret-dependent branches. DATA [66] considers both memory access leaks and branch leaks through differentiating address traces. Existing dynamic methods, though manifest relatively improved scalability, may still be slow to analyze production crypto libraries (due to the usage of constraint solving) or require many well-chosen inputs to induce distinct observations.

Challenge 2: *Cache models adopted by software analyzers have an effect on the scalability and detection granularity.* Relying on concrete cache replacement policies (e.g., LRU, FIFO, and RLRU), CacheAudit precisely describes a program been executed on the expected architecture, at the cost of scalability due to architectural complexity. CaSym uses high-level abstract cache models (i.e., infinite and age models) to achieve higher analysis scalability. It uses the array index to compute the accessed cache locations. However, these abstract models have granularity issues: there is a gap between the array index and the cache location in realistic architectures. At the other extreme, a much simplified cache model is shared by [3, 22, 59, 60, 66], where an architectural-independent model is used to detect cache side channels. Though this model is realistic and efficient, performing analysis at such granularity results in false positives, as will be discussed in this paper.

Challenge 3: *Supporting a comprehensive analysis of crypto software rather than some specific defects in sensitive code fragments.* For instance, CacheD omits the analysis of secret-dependent program branches. Moreover, modern crypto libraries extensively use randomization schemes like binding to mitigate side channels, whose effectiveness (and remaining leaks) have not been analyzed by previous tools. Supporting randomization is inherently hard for previous static (abstract interpretation-based) tools [21, 22, 59], requiring new abstract domains, new abstract operators, and soundness proofs. Meanwhile, modeling randomization is also costly for approaches that use constraint solvers, as it demands to iterate blinding quantifiers [13, 59, 60]. [66] conceptually differentiates traces derived from blinding-involved computations, but it overlooks the complex computations involving blinding in production cryptosystems, which may contain new attack vectors.

The aforementioned obstacles incentive the design of CaType, an automated, precise, and efficient cache side-channel analysis tool. CaType is scalable and capable of analyzing large-scale, complex crypto software. CaType follows [60, 66] to log execution traces of crypto software and performs trace-based type inference on the logged traces. It features a novel refinement type system that enables tracking program variables in the bit-level representation. Different from previous constraint solving-based approaches that are inherently costly, our sound type system guarantees fine-grained secret tracking and side channel detection with largely improved efficiency. Lastly, CaType comprehensively models randomization-based mitigation schemes adopted in modern crypto software. It allocates specific refined types for differentiating the responsibilities of (secret or randomized) variables, enabling precise information flow tracking under the presence of randomization. In sum, we make the following contributions:

- Conceptually, for the first time, cache side channels are analyzed using refinement type techniques. We establish our novel refinement type system directly over x86 assembly code, and formulate cache side channels over refined types.
- Technically, CaType features several important and effective enhancements compared with prior tools on the basis of refinement type system, including bit-level granularity tracking, distinguishing different effects of variables, precise type inferences, and much higher scalability. CaType takes into account randomization-based defenses using specific refined types, and uses novel cache layouts to suppress potential false positives.
- Empirically, we evaluate CaType to uncover side channel vulnerabilities among real-world crypto libraries. CaType captures all known design flaws, identifies unknown flaws, and reveals several false positives in existing tools. CaType is 16× faster than CacheD and 131× faster than CacheS, demonstrating its high applicability toward production crypto software.

2 Preliminaries

2.1 Refinement Type Systems

A type system is a well-established formal system comprising a set of rules that assigns types to terms in a programming language [15, 47]. For example, C language contains a basic type system, where types (e.g., `int`, `double`, and `int*`) give *meaning* to data in the memory or registers. Modern C compilers can feature basic type checking rules to detect invalid operations, e.g., when a variable of double is used as `int*` (for pointer dereference), an error is thrown at the compilation time.

Type systems are widely-used in language-based security research [71] like tracking secure information flow. In those systems, the types of variables and expressions are attached with annotations that specify confidentiality policies enforcing the use of the typed data. For instance, two type annotations H and L are used to denote high and low security sensitivity of data. To detect the violation of confidentiality policy, a set of type rules is defined to check if the two classified sets of data interfere with each other.

Refinement types [30] extend standard type annotations with predicates that confine the use of the values described by the type. Typically, a variable x's refinement type can be defined in the form of $x : T\{v : P\}$, where T is a basic type and P is the associated predicate. For example, a non-negative integer variable x is represented as $x : int\{v : 0 \leq v\}$, where predicate $0 \leq v$ refines the basic type *int* by specifying that the integer must be greater than or equal to zero. With well-defined predicates, the refinement types can provide stronger guarantees. For example, the zero-division errors can be alerted at the compilation time when the predicate $N \geq 0$ indicates that the divisor may be zero. Meanwhile, one can elaborately specify security policies over the refinement types to verify software security vulnerabilities. [6, 8–10] are successful examples of adopting refinement type systems in high-level languages (e.g., F*) to provide security guarantees in crypto infrastructures. To our best knowledge, CaType is the first to employ refinement types over assembly code and for cache side channel detection.

2.2 Cache Hierarchy

Caches are incorporated into CPUs to accelerate process execution due to the locality principle. In modern CPUs, each core (i.e., a processing unit on a CPU chip) monopolizes an L1 cache and a L2 cache. All cores share a megabyte-size LLC (Last-Level Cache). The access time for a cache hit is around tens of cycles. In contrast, the latency will become much higher (usually hundreds of cycles) when a cache miss occurs and the main memory has to be accessed. Modern CPUs use a W-way set-associative cache. Different memory blocks may reside on the same cache set, and each cache set is further divided into W cache lines. Given an N-bit memory address, S-set cache with L byte-size cache line, the lowest $log_2 L$ bits of the address represent the offset since continuous memory blocks are cached together within one `load` instruction. The middle $log_2 S$ bits starting from bit $log_2 L$ are used to locate the cache set index. The upper part represents cache hit/miss tag bits.

2.3 Cache Side Channels

Cache poses threats of secret leakage, as program cache accesses may be leveraged by adversaries to reconstruct confidential information. In this section, we introduce two representative vulnerable code patterns, *secret-dependent branch condition (SDBC)* and *secret-dependent memory access (SDMA)*, via classic examples in RSA.

Secret-Dependent Branch Condition (SDBC). Fig. 1a shows a simplified view of the square-and-multiply implementation of modular exponentiation in RSA. e_i (line 4) denotes a private key and decides if line 5 is executed. By monitoring the L1 instruction cache (I-cache), attackers are aware of the execution of line 5, and further reconstruct e_i using well-established cache attacks [34, 70].

Secret-Dependent Memory Access (SDMA). Besides SDBC, SDMA also leads to exploitations. Consider Fig. 1b, where the sliding window modular exponentiation algorithm initializes a precomputed array $g[i]$ (lines 1–3) to accelerate the computation. When performing decryption, a window size key w_i (line 8) is used as the index to query the precomputed table $g[i]$. For each for-loop (line 8), monitoring the accessed data cache (D-cache) line can reveal certain bits in w_i and gradually reconstruct the private key [34].

```
1 : x ← 1
2 : for i ← |e| − 1 downto 0
3 :     x ← x^2 mod m
4 :     if e_i = 1 then
5 :         x ← x · b mod m
6 : return x
```

(a) Square-and-Multiply Exp.

```
1 : g[0] ← b mod m
2 : for j ← 1 to 2^(S−1) − 1
3 :     g[j] ← b^(2j+1) mod m
4 : x ← g[(w_(n−1) − 1)/2] mod m
5 : for i ← n − 2 downto 0
6 :     x ← x^(2^L(w_i)) mod m
7 :     if w_i ≠ 0 then
8 :         x ← x · g[(w_i − 1)/2] mod m
9 : return x
```

(b) Sliding-window Exp.

Figure 1: Cache Side-channel Examples.

2.4 Cache Side Channel Mitigation

[35] surveys software-level countermeasures of cache side channels. Overall, two code patterns can remove secret-dependent cache access patterns: *AlwaysAccess-BitwiseSelect* permits programs to access secret-dependent data within each loop iteration in a constant manner, while deciding whether or not to accept it via bitwise operations. Moreover, if the calculation is inexpensive and free of secret-dependent branches, *On-the-fly Calculation* avoids using lookup tables, which eliminates leakage shown in Fig. 1b. Similarly, to remove secret-dependent branches, *AlwaysExecute-ConditionalSelect* enables covering all branches regardless of the `if` conditions. *AlwaysExecute-BitwiseSelect* eliminates secret-dependent branches by selecting correct results through bitwise operations.

The aforementioned code patterns can frequently introduce high overhead. They are thus less frequently used to only secure several core code fragments, which may miss subtle usage of secrets [22, 60]. *Blinding* introduces extra randomness in crypto computations to obscure the inference of secrets. Depending on the blinding target, there are two distinct usages of blinding masks.

Key Blinding. With this scheme enabled, the attacker obtains blinded secrets without knowing the blinding mask r. As r is randomly generated before each cipher process, attacker cannot exploit the cryptosystem. For example, exponent blinding in RSA adds a random multiple of Euler's ϕ function, i.e., $r \cdot \phi(n)$, to the secret exponent. Then, RSA decryption performs $c^{d+r\cdot\phi(n)} \ mod \ n$, which equals $c^d \ mod \ n$. Though some known attacks [52] exploit this scheme, the exponent blinding still impedes the attacker at large.

Plaintext/Ciphertext Blinding. Blinding can also be applied to plaintext/ciphertext. For instance, when enforcing blinding, RSA converts the ciphertext m into $m \cdot r^e$, where r is the random factor. The original result $m^d \ mod \ n$ can be obtained by multiplying the new result $(m \cdot r^e)^d \ mod \ n$ by r^{-1} due to $r^{ed} \cdot r^{-1} mod \ n \equiv 1 \ mod \ n$. The plaintext/ciphertext blinding defeats known-input attacks that leverage timing side channels.

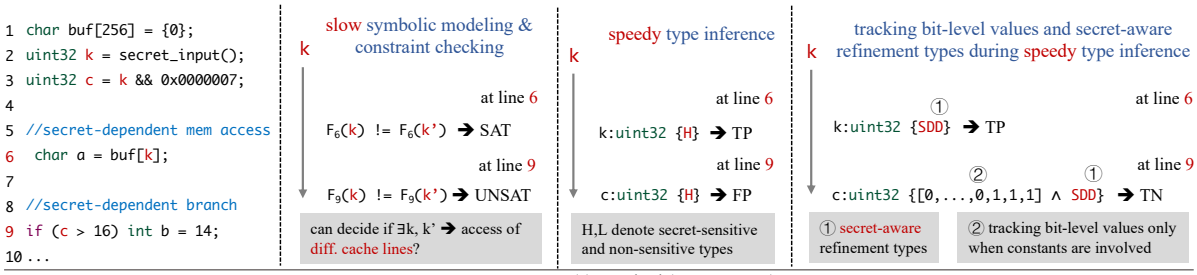
Blinding can usually provide more comprehensive protection as once key/ciphertext is blinded, all their follow-up usages and their (subtle) influence on other variables should be protected. However, their effectiveness in mitigating cache side channels are not yet comprehensively analyzed, given the difficulty of modeling them automatically in previous methods (noted in **Challenge 3** in Sec. 1).

3 Research Overview

3.1 Assumptions

Threat Model. CaType follows an identical threat model as most current cache side channel detectors [3, 13, 59, 60, 68]. We assume that an adversary shares the same hardware platform as the victim, a typical and practical assumption in cloud computing systems. Thus, while the adversary cannot directly monitor the victim's memory accesses, he can probe the shared cache states to determine if certain cache lines have been visited by the victim software. This threat model covers the majority of cache side channel attacks in the literature. For example, adversaries infer cache accesses by measuring the latency of the victim program in EVICT-TIME attack [44], or the latency of the attacker program in PRIME-PROBE [34, 44, 46], FLUSH-RELOAD [70], and FLUSH-FLUSH attacks [29].

Existing works [13, 21] commonly refer to the attackers in our threat model as "trace-based attackers" since they are able to probe the cache state after the execution of each program statement in the victim software. It is also worth noting that the attackers can distinguish cache layouts of instructions inside the program branches of shared libraries. This is due to the fact that modern OSes adopt aggressive memory deduplication techniques, allowing shared libraries to be mapped to copy-on-write pages. As a result, the probing granularity of attackers is precisely reduced to cache lines.



(a) sample code. (b) constraint solving-based approaches. (c) standard (secret-aware) type system. (d) refinement type system in CaType.

Figure 2: Comparison of constraint solving-based techniques (b), type inference-based approach (c), and CaType (d). TP, FP, and TN denotes true positive, false positive, and true negative, respectively.

Main Audience. Consistent with previous works [3, 13, 16, 19, 21, 55, 59, 60, 66, 68], CaType is primarily designed for crypto software developers who have sufficient knowledge about their own software. Before release, CaType serves as a "vulnerability debugger" for the developers to detect attack vectors in their software. CaType provides fully automated and speedy analysis to flag program points that leak secrets via cache side channels. Developers can accordingly patch CaType's findings to mitigate leakage. Nevertheless, we clarify that CaType is *not* an attack tool; the exploitability of its findings (e.g., whether RSA private keys can be reconstructed via CaType's findings) is beyond the scope of this paper.

3.2 Methodology Overview

This section illustrates the high-level methodology overview and compares with existing efforts in Fig. 2. Recall that we have introduced two typical cache side channel patterns in Sec. 2: SDMA and SDBC. Fig. 2(a) presents a sample code that is vulnerable to SDMA (line 6) whereas the condition at line 9 is *not* vulnerable to SDBC, given that the `else` branch will always be executed.

Symbolic Execution-Based Approaches. De facto side channel detectors perform heavyweight symbolic execution, where program (secret-related) data facts are modeled using symbolic formulas. Then, at each memory access and branch condition, they check if different secrets can lead to the access of different cache lines using constraint solving. For instance, let symbol k represent the secret read in line 2 of Fig. 2(a), existing side channel detectors [3, 13, 59, 60] primarily check the following constraint to decide SDMA/SDBC:

$$\exists k \neq k',\ F(k) \neq F(k') \tag{1}$$

where F denotes the memory access constraint formed at line 6, or branch condition constraint formed at line 9. The symbolic engine forms $F(k) = b + k \times 4$ at line 6, where b is the base address of `buf`. The satisfiability (SAT) of Constraint 1 checks the existence of two secrets that lead to the access of different cache lines, such that certain amount of secrets will be leaked to the attacker. Moreover, the symbolic engine will track computations using symbolic formulas, and at line 9, the constraint solver yields unsatisfiable (UNSAT) for Constraint 1, thereby proving the safety of line 9.

The primary obscurity of such detectors is *scalability*. Overall, existing symbolic execution (or abstract interpretation)-based side channel detectors need to maintain complex symbolic states for each program statement to encode program semantics. As symbolic execution continues, the symbolic constraints (encoding program states) will steadily accumulate and grow in size, filling a vast amount of memory. Even worse, existing tools need to perform constraint solving for each suspicious memory access and conditional branch instruction, and constraint solving is generally slow. With this regard, we notice that existing static analysis tools are often limited to analyzing small programs, or fail to consider the effect of side channel mitigation techniques like blinding.

Conventional Type-Based Analysis. Sec. 2.1 has introduced basic mechanisms of type systems and the extensions to track high/low secret-sensitive data with type annotations H and L. As illustrated in Fig. 2(c), performing type inference can easily establish that the types of `k` and `c` (in lines 6 and 9, respectively) are `uint32`. Moreover, by assigning a high security sensitivity type H to `k` at line 2, the type system identifies two usage of sensitive data at line 6 and line 9. These two statements are deemed as "vulnerable", leading to secret-dependent memory access and branch condition. Nevertheless, we underlie that while the statement at line 6 is a true positive (TP) finding, statement at line 9 is a false positive (FP), as `c` can never exceed 7 (see line 3 in Fig. 2(a)). Overall, conventional type-based analysis delivers speedy tracking of (secret-related) data through type annotations. They, however, lack of tracking values and are less expressive than constraint solving-based methods. Indeed, Sec. 7 compares taint analysis, *conceptually similar* to type systems enforcing information-flow security (e.g., [51]), with refinement type system implemented in CaType. We show that taint analysis yields considerably more false positives than CaType.

Refinement Type System in CaType. Recall the refinement type of a variable x can be expressed as $x : T\{v : P\}$ (Sec. 2.1), where T and P are basic types and predicates, respectively. Fig. 2(d) illustrates the usage of the refinement type system in CaType, where the refinement formalizes the concerned (secret-related) program properties as predicates. In particular, we use type SDD to denote secret-dependent values, and the refinement type system infers that in line 6, `k` is of type uint32$\{v$: SDD$\}$, revealing a potential SDMA case. Similarly, the refinement type of `c` in line 9 also has type SDD, revealing a potential SDBC case (which is *not* vulnerable; see below for clarification). CaType defines in total five predicates, systematically considering secret-dependent, secret-independent, as well as blinding operations. In this way, CaType can benefit from refinement type techniques to keep track of secret propagations and identify SDMA/SDBC in a speedy manner while

correctly considering randomization mechanisms like blinding (see **Blinding** later this section for further discussion).

Moreover, CaType explores an important improvement, by tracking bit values directly in refinement types, in the form of value predicates. A value predicate is defined as $v = b$, where b is either 0 or 1. CaType is carefully designed to deliver a "mild tracking" of bit-level values. That is, only the refinement types of constants are initialized to comprise bit-level predicates. Then, CaType tracks the bit-level predicates via type inference in a correct yet conservative manner. For instance, when a constant, 0x0000007, is used as the mask over the secret (line 3), the type of the output means that it is a bitvector with all secret bits (except the three least significant bits) set to 0. Note that value predicates in refinement types can be absent, indicating that the precise bit-level values are unknown.

By tracking of bit values from constants, CaType can exclude the majority, if not all, cases where different secret values at a suspicious SDMA/SDBC case result in visiting the *same* cache line (i.e., a safe program site). For instance, when `k` is masked by 0x0000007 before being used in the `if` condition at line 9 of Fig. 2(a), the refinement type of `c` has all bits set to 0 except the lowest three bits, and CaType can simply decide that the branch condition will always be evaluated as "false" with an arithmetic comparison over two bitvectors. Therefore, when analyzing the statement at line 9 of Fig. 2(a), CaType yields a true negative (TN) finding, as shown in Fig. 2(d). Overall, we view that the refinement type system designed in CaType manifests comparable capability with constraint solving-based methods to analyze cache side channels. Moreover, CaType avoids the use of constraint solving, and is therefore dramatically faster; see Table 4 in Sec. 6.1.

Potential False Positives. We clarify that the refinement type system in CaType may not always know the precise bit values: the absence of value predicates means the value could be 0 or 1. Overall, CaType tracks the bit values introduced by constants using refinement types at "its best effort". Thus, we may encounter false positives, e.g., due to constants that are however not tracked by CaType. Nevertheless, cache side channels are rare in practice, and we confirm that all findings of CaType over production cryptosystems are true positives. Also, the refinement type system is sound without introducing false negatives, as benchmarked in Sec. 7.

Blinding. As introduced in Sec. 2.4, modern cryptosystems use randomness mechanisms like blinding to impede side channels. To capture the security property of blinding, our refinement type system facilitates a smooth and accurate modeling of blinding, by adding specific predicates in type refinement to denote uniformly random data (i.e., the blinding mask). We also define type inference rules and propagation rules for blinding involved computations, so that we can capture sufficient information used to infer potential leaks. For example, uniformly random factors can perfectly mask the result through logic xor operation, eliminating the effects of a secret if it is a source operand. See details in Sec. 4.2 and Sec. 4.3.

In contrast, adding support for blinding presumably increases the search space of constraint solving-based methods to a great extent. Consequently, finding a SAT solution for Constraint 1 is highly expensive, especially when both secrets and blinding masks are present. Though an "optimal solution" is not yet clear, inspired by relevant research in perfect masking analysis [24–26], we expect to fix two different secrets k, k' and then iterate the quantifiers of all involved masks $r_1, \ldots, r_n$ to count the ranges under k, k'. This process may take a dramatically longer time or timeout.

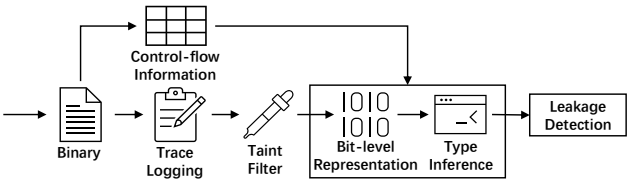


Figure 3: Workflow of CaType.

4 Design

Overview. Fig. 3 depicts the workflow of CaType. Given the crypto software in executable format, we first run the executable using Intel Pin [36] to perform concerned crypto computation (e.g., RSA decryption) and log an execution trace. Then, we require users of CaType to mark the program secrets and random factors on the execution trace, and perform taint analysis by tainting those secrets/randomness and extract a tainted sub-trace depicting how tainted variables are propagated and used. Meanwhile, we also disassemble the executable and extract control flow information into a lookup table from the disassembled assembly code, which will be used later in checking SDBC (see Sec. 4.4).

CaType then performs type inference over the tainted sub-trace, by first annotating variables with bit-level types of initialized refinements (Sec. 4.1). It tracks the propagation and usage of secure-sensitive values in refined types during type inference (Sec. 4.2 and Sec. 4.3). When encountering memory accesses or branch conditions, CaType uses the refined types of involved variables to check if SDBC/SDMA exists (Sec. 4.4). Once a side channel flaw is discovered, it reports the detected instruction's address to users for confirmation, debugging, and patching. We now discuss each step in detail.

Design Consideration: Binary vs. Source. CaType is designed to directly analyze x86 binary code compiled from crypto software. Thus, the refinement type system is defined over x86 assembly code, and CaType's analysis depends on the specific memory layout. Overall, side channels are sensitive to the low-level architecture and system details. We clarify that prior works in this field are consistently analyzing software in executable format. This enables the analysis of legacy code and third-party libraries without accessing source code. More importantly, by analyzing low-level assembly instructions, it is possible to take into account low-level details, such as memory allocation. Recent works [53] have shown that compiler optimizations could introduce extra side channel opportunities that are not visible at the high-level code representation level.

Design Consideration: Information Flow Tracking. When illustrating cache side channels in Fig. 1 and Fig. 2, we depict how the use of secrets result in side channels. Nevertheless, in addition to side channels induced via the *direct* usage of secrets, it is crucial to treat data derived from the secrets as "sensitive". *CaType tracks both explicit and implicit information flows propagated from secrets.* When a variable x is of SDD type, and the data is loaded from memory address formed by x, the destination variable has type SDD. Similarly, when x is used to form branch conditions, the

result type is SDD as well. By modeling information flows, CaType comprehensively uncovers attack surface of cryptosystems.

4.1 Bit-level Representation and Types

We first clarify that in analyzing x86 assembly code, registers, CPU flags, and memory cells are all considered as *variables* in CaType. We use bit-level representation for variables encountered on the execution trace, allowing us to track variables with fine-grained precision. Considering the instruction syntax in Fig. 4, where an expression e can be a constant bit b, a variable x, a constant bitvector $[b, \cdots, b]$, or computations over expressions. Concatenation $e_1 \sharp\ e_2$ uses e_1 and e_2 to form the highest and lowest several bits, respectively. Extracting several bits from the designated position of a bitvector expression produces a fragment, dubbed as $[n_1 : n_2]/e$. Other operations include negation ($\neg$), arithmetic and logic operations ($\bowtie$) over two expressions, and the conditional expression with three operands (the syntax mimics conditional selection in the C language). A statement s is an assignment, a memory load/store, or a sequence of statements. We clarify that execution trace forms a typical straight-line code of instructions, omitting branch merges.

Types and Hierarchy. As introduced in Sec. 3.2, a type ρ has the form of $\{v : T \mid P\}$, where T is a basic type and predicate P is the refinement. We define basic type T as primitive types of bit representations, i.e., one bit B or a bitvector of n bits $\text{Vec}\langle n\rangle$. A refinement type P is either a security type predicate τ or a conjunction with a value predicate. A security type predicate τ can be any of the five types, i.e., SDD, URA, SID, WRA and CST, denoting *secret-dependent*, *uniformly random*, *secret-independent*, *weakly random*, and *constant* values. A value predicate is termed as $v = b$ (where b is 1 or 0), meaning that v has value b. The expression typing judgment, $\Gamma \vdash e : \rho$, states that expression e has type ρ, where Γ is the typing environment mapping from variables to types.

The hierarchy of security types τ is CST $\leq$: URA $\leq$: WRA $\leq$: SID $\leq$: SDD. We clarify that among the five refined types, only SDD is related to secrets. We use WRA to denote a data of weakly random distribution, meaning it is not uniformly random (in other words, not perfect and secure blinding). URA means uniformly random data, representing perfect and secure masking. The join operator $\sqcup$ takes the least upper bound of two types; for instance, SID $\sqcup$ SDD = SDD, as SDD sits higher in the hierarchy.

Types Annotation. Before launching type inference, we first annotate variables with security types. Secrets, random factors, and constants are marked as SDD, URA, and CST, respectively. We mark other variables using SID, and type WRA may be generated during type inference. Given that we perform bit-level type annotation and inference, if variable x hosts a 32-bit secret, it is annotated as $\{v : \text{Vec}\langle 32\rangle \mid v : \text{SDD}\}$. This vector type implies that each bit in the vector has type SDD, i.e., $\forall b_i \in x.\ b_i : \{v : B \mid v : \text{SDD}\}$. For constants, we also explicitly annotate each bit (whether it equals 0 or 1) in the value predicate. Thus, each bit of a constant c is in the form of $b_i \in c.\ b_i : \{v : B \mid v = b \wedge v : \text{CST}\}$, where b is 0 or 1, depending on the value of c. Recall as noted in Sec. 3.2, our refinement type-based inference conducts a "best-effort" tracking of bit-level values derived from constants. The bit-level tracking updates value predicates during type inference. Nevertheless, when a bit value becomes unknown (could be either 0 or 1), we conservatively omit its value predicate and only retain the security type predicate.

$$
\begin{array}{rrll}
\text{Expr} & e & ::= & b \mid x \mid [b, \cdots, b] \mid \neg e \mid e_1 \bowtie e_2 \\
 & & \mid & e\ ?\ e_1 : e_2 \mid e_1 \sharp\ e_2 \mid [n_1 : n_2]/e \\
\text{Stmt} & s & ::= & x \leftarrow e \mid x \leftarrow e_1[e_2] \mid e_1[e_2] \leftarrow x \mid s_1; s_2 \\
\text{Basic Types} & \text{T} & ::= & \text{B} \mid \text{Vec}\langle n\rangle \\
\text{Security Types} & \tau & ::= & \text{SDD} \mid \text{URA} \mid \text{SID} \mid \text{WRA} \mid \text{CST} \\
\text{Refinements} & \text{P} & ::= & v : \tau \mid v = b \wedge v : \tau \\
\text{Type} & \rho & ::= & \{v : \text{T} \mid \text{P}\} \\
\text{Type Env} & \Gamma & ::= & \emptyset \mid \Gamma, x : \rho
\end{array}
$$

Figure 4: Syntax of bit-level representation.

$$
\|[b_{n-1}, \cdots, b_0]\|_t = \begin{cases}
\text{SDD} & \exists b_i.\ b_i : \{v : B \mid v : \text{SDD}\} \\
\text{URA} & (\nexists b_i.\ b_i : \{v : B \mid v : \text{SDD}\}) \wedge \\
 & (\exists b_i.\ b_i : \{v : B \mid v : \text{URA}\}) \\
\text{SID} & (\nexists b_i.\ b_i : \{v : B \mid v : \text{SDD}\}) \wedge \\
 & (\nexists b_i.\ b_i : \{v : B \mid v : \text{URA}\}) \wedge \\
 & (\exists b_i.\ b_i : \{v : B \mid v : \text{SID}\}) \\
\text{WRA} & (\nexists b_i.\ b_i : \{v : B \mid v : \text{SDD}\}) \wedge \\
 & (\nexists b_i.\ b_i : \{v : B \mid v : \text{URA}\}) \wedge \\
 & (\nexists b_i.\ b_i : \{v : B \mid v : \text{SID}\}) \wedge \\
 & (\exists b_i.\ b_i : \{v : B \mid v : \text{WRA}\}) \\
\text{CST} & \forall b_i.\ b_i : \{v : B \mid v : \text{CST}\}
\end{cases}
$$

Figure 5: Type propagation from single-bit to bitvector.

4.2 Type Inference for Bitvectors

Different bits in a bitvector may have varying security types. Consider register `eax`, which stores a 32-bit data, where the upper 16 bits are URA and the lower 16 bits are SID. Intuitively, the bitvector's type can be inferred by simply taking the least upper bound of all the constituent bits' types, i.e., SID in this case. However, the high 16 bits are URA, meaning that each bit has equal possibility of being 0 or 1. Thus, the intuitive approach would lose the information of randomness, leading to inaccuracy in subsequent analyses.

To precisely track bit-level security propagation, we define function $\|x\|_t$ in Fig. 5 to infer a bitvector's type from the types of its constituent bits based on a notion of structural priority. We give type SDD the highest priority, meaning that a bitvector is of type SDD if it contains at least one bit of type SDD. In the absence of SDD type, type URA is structurally preceding, i.e., if there is a bit in a vector whose type is URA, then the vector itself is URA. As seen in Fig. 5, SID is structurally superior to WRA and CST, whereas WRA is structurally superior to CST.

From a holistic view, sensitive data (specified in refinements) are "propagated" from single-bit to whole bitvector following type rules in Fig. 5. Therefore, information flow analysis is performed here to determine how sensitive data are propagated and influence program execution. To clarify, in addition to type rules, CaType also conducts taint analysis over the Pin-logged trace and collects a list of tainted instructions. This is a classic optimization to reduce trace length, also adopted in previous works [3, 59, 60]. Our type inference is performed on the tainted trace, as illustrated in Fig. 3.

4.3 Type Inference Rules

CaType implements a comprehensive set of type inference rules over each encountered x86 assembly instruction to track the propagation of secure-sensitive types and check cache side channels.

Type Rules for One Bit Logical Operations. Fig. 6 presents a representative list of type rules for one bit logical operations. First,

$$\textsc{Conj\&Disj.I}\quad \frac{\Gamma \vdash e_1 : \{v : B \mid v : \tau_1\} \qquad \Gamma \vdash e_2 : \{v : B \mid v : \tau_2\} \qquad \tau_1 \neq \text{CST} \qquad \tau_2 \neq \text{CST} \qquad \neg(\tau_1 = \text{URA} \wedge \tau_2 = \text{URA}) \qquad \bowtie \in \{\wedge, \vee\}}{\Gamma \vdash e_1 \bowtie e_2 : \{v : B \mid v : \tau_1 \sqcup \tau_2\}}$$

$$\textsc{Conj\&Disj.II}\quad \frac{\Gamma \vdash e_1 : \{v : B \mid v : \text{URA}\} \qquad \Gamma \vdash e_2 : \{v : B \mid v : \text{URA}\} \qquad \bowtie \in \{\wedge, \vee\}}{\Gamma \vdash e_1 \bowtie e_2 : \{v : B \mid v : \text{WRA}\}}$$

$$\textsc{XOR.I}\quad \frac{\Gamma \vdash e_1 : \{v : B \mid v : \tau_1\} \qquad \Gamma \vdash e_2 : \{v : B \mid v : \tau_2\} \qquad \tau_1 \neq \text{URA} \qquad \tau_2 \neq \text{URA} \qquad \neg(\tau_1 = \text{CST} \wedge \tau_2 = \text{CST})}{\Gamma \vdash e_1 \oplus e_2 : \{v : B \mid v : \tau_1 \sqcup \tau_2\}}$$

$$\textsc{XOR.II}\quad \frac{\Gamma \vdash e_1 : \{v : B \mid v : \text{URA}\} \qquad \Gamma \vdash e_2 : \{v : B \mid v : \tau\}}{\Gamma \vdash e_1 \oplus e_2 : \{v : B \mid v : \text{URA}\}}$$

$$\textsc{Neg.I}\quad \frac{\Gamma \vdash e : \{v : B \mid v : \tau\}}{\Gamma \vdash \neg e : \{v : B \mid v : \tau\}}$$

Figure 6: Selected one bit B type rules for logical operations. See Appendix A for the complete list of rules.

$$\textsc{Concat.I}\quad \frac{\Gamma \vdash e_1 : \{v : \text{Vec}\langle n_1\rangle \mid v : \tau_1\} \qquad \tau_1 \neq \text{URA} \qquad \Gamma \vdash e_2 : \{v : \text{Vec}\langle n_2\rangle \mid v : \tau_2\} \qquad \tau_2 \neq \text{URA}}{\Gamma \vdash e_1 \sharp e_2 : \{v : \text{Vec}\langle n_1 + n_2\rangle \mid v : \tau_1 \sqcup \tau_2\}}$$

$$\textsc{Concat.II-1}\quad \frac{\Gamma \vdash e_1 : \{v : \text{Vec}\langle n_1\rangle \mid v : \text{URA}\} \qquad \Gamma \vdash e_2 : \{v : \text{Vec}\langle n_2\rangle \mid v : \tau_2\} \qquad \tau_2 \neq \text{SDD}}{\Gamma \vdash e_1 \sharp e_2 : \{v : \text{Vec}\langle n_1 + n_2\rangle \mid v : \text{URA}\}}$$

$$\textsc{Concat.II-2}\quad \frac{\Gamma \vdash e_1 : \{v : \text{Vec}\langle n_1\rangle \mid v : \text{URA}\} \qquad \Gamma \vdash e_2 : \{v : \text{Vec}\langle n_2\rangle \mid v : \text{SDD}\}}{\Gamma \vdash e_1 \sharp e_2 : \{v : \text{Vec}\langle n_1 + n_2\rangle \mid v : \text{SDD}\}}$$

$$\textsc{Extraction}\quad \frac{\Gamma \vdash e : \{v : \text{Vec}\langle n\rangle \mid v : \tau_e\} \qquad m_1 \leq m_2 \qquad \|[m_1 : m_2]/e\|_t = \tau}{\Gamma \vdash [m_1 : m_2]/e : \{v : \text{Vec}\langle m_2 - m_1 + 1\rangle \mid v : \tau\}}$$

$$\textsc{Logic.I}\quad \frac{\Gamma \vdash e_1 : \{v : \text{Vec}\langle n\rangle \mid v : \tau_1\} \qquad \Gamma \vdash e_2 : \{v : \text{Vec}\langle n\rangle \mid v : \tau_2\} \qquad \bowtie \in \{\wedge, \vee, \oplus\} \qquad \|e_1 \bowtie e_2\|_t = \tau}{\Gamma \vdash e_1 \bowtie e_2 : \{v : \text{Vec}\langle n\rangle \mid v : \tau\}}$$

$$\textsc{Logic.II}\quad \frac{\Gamma \vdash e : \{v : \text{Vec}\langle n\rangle \mid v : \tau\}}{\Gamma \vdash \neg e : \{v : \text{Vec}\langle n\rangle \mid v : \tau\}}$$

$$\textsc{Arith.I}\quad \frac{\Gamma \vdash e_1 : \{v : \text{Vec}\langle n\rangle \mid v : \tau_1\} \qquad \Gamma \vdash e_2 : \{v : \text{Vec}\langle n\rangle \mid v : \tau_2\} \qquad \tau_1 \neq \text{URA} \qquad \tau_2 \neq \text{URA} \qquad \neg(\tau_1 = \text{CST} \wedge \tau_2 = \text{CST}) \qquad \bowtie \in \{+, -, \times, \div\}}{\Gamma \vdash e_1 \bowtie e_2 : \{v : \text{Vec}\langle n\rangle \mid v : \tau_1 \sqcup \tau_2\}}$$

$$\textsc{Arith.II-1}\quad \frac{\Gamma \vdash e_1 : \{v : \text{Vec}\langle n\rangle \mid v : \text{URA}\} \qquad \Gamma \vdash e_2 : \{v : \text{Vec}\langle n\rangle \mid v : \tau_2\} \qquad \tau_2 \neq \text{SDD} \qquad \bowtie \in \{+, -, \times, \div\}}{\Gamma \vdash e_1 \bowtie e_2 : \{v : \text{Vec}\langle n\rangle \mid v : \text{URA}\}}$$

$$\textsc{Arith.II-2}\quad \frac{\Gamma \vdash e_1 : \{v : \text{Vec}\langle n\rangle \mid v : \text{URA}\} \qquad \Gamma \vdash e_2 : \{v : \text{Vec}\langle n\rangle \mid v : \text{SDD}\} \qquad \bowtie \in \{+, -, \times, \div\}}{\Gamma \vdash e_1 \bowtie e_2 : \{v : \text{Vec}\langle n\rangle \mid v : \text{SDD}\}}$$

$$\textsc{Comp}\quad \frac{\Gamma \vdash e_1 : \{v : \text{Vec}\langle n\rangle \mid v : \tau_1\} \qquad \Gamma \vdash e_2 : \{v : \text{Vec}\langle n\rangle \mid v : \tau_2\} \qquad \neg(\tau_1 = \text{CST} \wedge \tau_2 = \text{CST}) \qquad \bowtie \in \{<, \leq, >, \geq, =, \neq\}}{\Gamma \vdash e_1 \bowtie e_2 : \{v : \text{Vec}\langle 1\rangle \mid v : \tau_1 \sqcup \tau_2\}}$$

$$\textsc{Cond.I}\quad \frac{\Gamma \vdash e : \{v : \text{Vec}\langle 1\rangle \mid v : \text{SDD}\} \qquad \Gamma \vdash e_1 : \{v : \text{Vec}\langle n\rangle \mid v : \tau_1\} \qquad \Gamma \vdash e_2 : \{v : \text{Vec}\langle n\rangle \mid v : \tau_2\}}{\Gamma \vdash e\ ?\ e_1 : e_2 : \{v : \text{Vec}\langle n\rangle \mid v : \text{SDD}\}}$$

$$\textsc{Cond.II}\quad \frac{\Gamma \vdash e : \{v : \text{Vec}\langle 1\rangle \mid v : \tau\} \qquad \tau \neq \text{SDD} \qquad \Gamma \vdash e_1 : \{v : \text{Vec}\langle n\rangle \mid v : \tau_1\} \qquad \Gamma \vdash e_2 : \{v : \text{Vec}\langle n\rangle \mid v : \tau_2\} \qquad \neg(\tau_1 = \text{CST} \wedge \tau_2 = \text{CST})}{\Gamma \vdash e\ ?\ e_1 : e_2 : \{v : \text{Vec}\langle n\rangle \mid v : \tau_1 \sqcup \tau_2\}}$$

Figure 7: Type rules for expressions involving bitvector Vec$\langle n\rangle$.

type rules that involve CST type are designed to propagate CST in a straightforward way; see Appendix A for other involved rules. Rule Conj&Disj.I states that if two operands are not both CST or URA, then the result type is the least upper bound of the two operands' types, which enables the tracking of secure-sensitive values in types. Rule Conj&Disj.II handles the circumstance in which both operands are URA. Since the value of the result is no longer distributed uniform-randomly under logic *AND* and *OR*, the result type is lifted on the type hierarchy to WRA.

Rule XOR.I is similar to rule Conj& Disj.I, where the result type is the least upper bound of the two operands' types, provided that neither bit expression is URA or CST simultaneously. Rule XOR.II states that if one of the operands is of type URA, the result type is URA. This refers to the fact that random factors can uniformly blind the results through exclusive or ($\oplus$) operations. Rule Neg.I keeps security types unchanged in front of the negation operation.

Type Rules for Bitvector Operations. Fig. 7 depicts the type rules for operations with bitvectors Vec$\langle n\rangle$. There are three rules applicable to concatenation expressions. Rules Concat.I states that the resultant's type takes the least upper bound of the two vectors' type, if both vectors are not URA. Rule Concat.II-1 states that type URA is structurally prior to other secret-free types, and Concat.II-2 specifies that a bitvector exhibits SDD type if at least one bit in expression e_2 is SDD. Rule Extraction is a well-demonstrated example that leverages function $\|x\|_t$ to determine the refined type of the segment extracted from the source operand. Note that shift operations do not have their own rules as they can be implemented by combining concatenation and extraction operations. Rule Logic.I infers a vector type from the types of its constituent bits, i.e., the type of the result is inferred by applying the structural priority defined in Fig. 5. Rule Logic.II is similar to Rule Neg.I.

For the arithmetic operations of two bitvectors, one difference lies in performing the calculation at the whole bitvector level as opposite to each bit. Specifically, we determine the security type of the result, and propagate it to each bit; this offers a sound estimation of each bit's security type. Similar to Concat rules, Arith rules conform to the security type propagation in bitvector structures.

As specified in x86 assembly code, the comparison operation only produces one-bit bitvector Vec$\langle 1\rangle$ to the result (i.e., the affected CPU flags). Rule Comp specifies that the resultant's type is the least upper bound of the two operands' types. We omit the case where two operands are both CST as it is straightforward. The last two rules are designed for conditional expressions. We specify two rules according to whether the condition expression e is related to the secret. Rule Cond.I states that if the refined security type of the condition expression e is SDD, the result type is SDD regardless of the type of two branch expressions. We clarify that this rule allows CaType to keep track of implicit information flow propagated from secret-dependent branch conditions to the instructions. Thus, it facilitates detecting potential cache side channels derived from

implicit information flow. In contrast, Rule Cond.II takes the least upper bound of two branch expressions' types.

Statement type rules are standard (see Appendix B), and we emphasize that CaType tracks secrets propagation through both explicit and implicit information flows.

Proposition 4.1. *Our type system guarantees security-safety statically: if an expression e is given the type* $\{v : T \mid v : \tau\}$*, then the type of its runtime value will be at least at level* τ *on the type hierarchy.*

That is, the type system in CaType is sound, and it does not make any false negatives in its analysis; see further discussions and empirical results about type system correctness in Sec. 7.

4.4 Cache Side Channel Detection

Sec. 2.3 has illustrated two representative forms of cache side channels, i.e., SDMA and SDBC. When performing type inference, CaType will check each encountered memory access or conditional jump instruction to see if cache side channels exist. Specifically, to check if a memory access leads to SDMA, we right shift the variable holding memory address by L bits, and decide if the resulting variable is of SDD type. Following a common setup [3, 59, 60], L equals 6, standing for 64-byte (2^6) cache line size on modern CPUs.

For SDBC, previous research [3, 13] merely checks if different secrets induce distinct executing branches. In contrast, CaType checks if the conditional expression is of SDD type, and further assures two branches are not within identical cache lines. Recall as shown in Fig. 3, we disassemble the crypto software executable and recover the control flow structure. At this step, we compute the covered cache units of two branches: a SDBC is confirmed, in case the condition is of SDD type, and two branches are placed within distinguishable (at least one non-overlapping) cache lines.

An Illustrative Example. We use an example from the OpenSSL library to visually demonstrate the type inference and detection of side channels. With respect to code in Fig. 8, we present the corresponding (simplified) type inference procedure launched by CaType in Table 1. The first and second columns report the applied type inference rules and the refinement types of relevant variables. The last column reports the relevant cache line layout: MA(a) represents a secret-dependent memory access, and we also report the accessed cache line. BC(a, b, c) indicates that for a conditional control transfer the `if` branch starts at virtual address a (ends at address b), whereas the `else` branch starts at b and ends at c. We also report the accessed cache lines in the last column ("c-line").

Before analysis, users mark `eax` as "secrets" (type SDD). With type inference applied, CaType identifies one SDMA and two SDBC (marked in red). As shown in the last column, for the memory address of the SDMA, CaType checks that the refinement type of highest $32 - L$ bits is of SDD type. As for those two SDBC cases, in addition to checking the branch condition's type is SDD, CaType further checks whether the `if` and `else` branches are located within distinguishable cache lines. CaType confirms all three cases as vulnerable to cache side channels, whose findings are aligned with [59, 60].

5 Implementation

CaType is implemented in Scala, and presently performs analysis on crypto software executables compiled on 32-bit x86 platforms. However, extending CaType to other platforms, e.g., 64-bit x86, is not complex. See discussion in Sec. 7. As a common practice for trace-based analysis, we use Pin [36] to log each covered instruction and its associated execution context, including all values in CPU registers. These logged contexts are used to compute the concrete values of pointers in the follow-up static analysis phase. In other words, our type inference phase employs a practical and common memory model [14, 60], such that we decide the addresses stored in a pointer using their concrete values logged on the trace.

We use *objdump* to disassemble executable files of crypto software, and recover the control flow graph over the disassembled assembly code. Currently, when encountering an indirect jump, we conservatively consider that it can jump to any legitimate control transfer destinations in the disassembled assembly code. For each conditional jump, we collect the memory address ranges of its `if/else` branches from the disassembled code. We build a lookup table over these control transfer information when checking if executing secret-dependent branches can visit different cache lines.

Usage of CaType. To use CaType, users need to manually identify the secrets and random factors like blinding in assembly code of crypto software. As noted in Sec. 3.1, CaType is designed primarily for crypto software developers, who have detailed knowledge of their own code. Note that the knowledge of sensitive data in crypto binary code is generally assumed by previous side channel detectors, as most of them analyze binary code [3, 59, 60, 66].

We clarify that, as existing works [3, 59, 60], flagging secret (e.g., RSA private key) only requires mundane reverse engineering of crypto executable and marking memory buffers that store keys. To date, disassemblers are mature for processing crypto executables. Moreover, to ease the localization of secrets/random factors in assembly code, we recommend developers to compile crypto software with debug information attached. We observe that it takes less than 30 minutes to flag the secrets for each of our evaluated crypto software. Other than manually localizing secrets, all follow-up analyses are done automatically by CaType, whose outputs would be localized vulnerable points in assembly code, as illustrated in Table 1. Then, developers will need to map those leakage assembly instructions to source code for diagnosis and patching. To ease mapping assembly instructions to source code, it is also suggested to compile binary code with debug information attached, thereby encoding source code line number into assembly instructions.

In addition, we do not particularly mark certain one-way functions on the execution trace, e.g., functions applying key blinding over secrets. Instead, we assign refined types (URA) to random data before the analysis, and whenever keys are used together with blinding, refined types for secrets and blinding will naturally fit their corresponding type inference rules (as defined in Fig. 6 and Fig. 7). Therefore, we should not miss any one-way function provided that random data has been marked correctly before the analysis.

6 Evaluation

Evaluation Setup. We evaluate CaType on production cryptosystems. Evaluations are conducted in Ubuntu 16.04 with Intel Xeon 3.50GHz CPU, 32GiB RAM. We collect execution traces of algorithms including RSA, Elgamal, and (EC)DSA from OpenSSL and Libgcrypt (see Table 2). $*$ represents using random factor on plaintext/ciphertext and $\star$ indicates using random factor on secrets. Besides, we evaluate the effectiveness of CaType on a constant-time

Figure 8: BN_num_bits_word.

```
804961d: mov eax, ptr [ebp+0x8]
8049620: and eax, 0xffff0000
8049625: test eax, eax
// secret-dependent condition
8049627: je 8049661
8049629: mov eax, ptr [ebp+0x8]
804962c: and eax, 0xff000000
8049631: test eax, eax
// secret-dependent condition
8049633: je 804964b
8049635: mov eax, ptr [ebp+0x8]
8049638: shr eax, 0x18
// secret-dependent mem access
804963b: mov al, ptr [eax+0x8110460]
8049641: and eax, 0xff
8049646: add eax,0x18
8049649: jmp 8049691
```

Table 1: Type Inference. "c-line" stands for cache line.

Involved refinment types	Applied rules	Control-flow & cache lines
$eax = \{K\}^{32} : SDD$		
$eax = \{K\}^{16}\{0\}^{16} : SDD,\ r_0 = \{1\}^{16}\{0\}^{16} : CST$	Logic.I, Conj&Disj.I, Const-Conj.I&II	
$eax = \{K\}^{16}\{0\}^{16} : SDD,\ r_0 = \{K\}^{16}\{0\}^{16} : SDD,$ $zf = \{K\} : SDD$	Logic.I, Conj&Disj.I, Const-Conj.I	
je condition (zf) ⟶ secret-dependent		
$eax = \{K\}^{32} : SDD$		BC(8049629,8049661,804968c)
$eax = \{K\}^{8}\{0\}^{24} : SDD,\ r_0 = \{1\}^{8}\{0\}^{24} : CST$	Logic.I, Conj&Disj.I, Const-Conj.I&II	true branch ⟶ c-line 201258
$eax = \{K\}^{8}\{0\}^{24} : SDD,\ r_0 = \{K\}^{8}\{0\}^{24} : SDD,$ $zf = \{K\} : SDD$	Logic.I, Conj&Disj.I, Const-Conj.I	false branch ⟶ c-line 201259 20125a
je condition (zf) ⟶ secret-dependent		BC(8049635,804964b,804965f)
$eax = \{K\}^{32} : SDD$		true branch ⟶ c-line 201258
$eax = \{0\}^{24}\{K\}^{8} : SDD,\ r_0 = 24 : CST$	Extraction, Concat.I	false branch ⟶ c-line 201259
$eax = \{0\}^{24}\{K\}^{8} : SDD,\ r_0 = 135332960 : CST,$ $r_1 = \{0\}^{4}\{1\}\{0\}^{6}\{1\}\{0\}^{3}\{1\}\{0\}^{5}\{1\}\{0\}^{2}\{K\}^{8} : SDD,$ memory address (r_1) ⟶ secret-dependent	Arith.I, Concat.I	MA(804963b) destination ⟶ c-line 0x201258 · · ·
$eax = \{0\}^{24}\{K\}^{8} : SDD,\ r_0 = \{0\}^{24}\{1\}^{8} : CST$	Logic.I, Conj&Disj.I, Const-Conj.I&II	
$eax = \{0\}^{24}\{K\}^{8} : SDD,\ r_0 = 24 : CST$	Arith.I, Concat.I	
		BR(8049649,8049691)

† r_0 and r_1 represent temporary variables. ‡ zf represents Zero Flag register.

Figure 9: Type inference over sample assembly code. To ease reading, we use K, I, W, and U to term refinement type predicates, corresponding to SDD, SID, WRA, and URA types. $\{K\}^{32}$ means bit K repeats 32 times, while $\{1\}^{16}$ means bit 1 repeats 16 times.

dataset offered in Binsec/Rel [19]. This will validate the correctness of our methodology to a reasonable extent.

The RSA/Elgamal algorithms from both libraries leverage the built-in secret generation function for generating 2048-bit secrets. The ECDSA algorithm adopts OpenSSL *sect571r1* curve. We initiate the plaintext or the message to be signed as "hello world". We use Intel Pin to log the execution traces when executing the crypto software for standard decryption/signature procedures, including the majority of asymmetric encryption functions such as modular exponentiation in RSA/ElGamal and point multiplication in the signature procedure of ECDSA.

Table 2: Cryptosystems analyzed by CaType.

Algorithms	Implementations	Versions
RSA	OpenSSL	1.0.2f^*, 1.1.0g^*, 1.1.0h^* 1.1.1n^*, 3.0.2*
	Libgcrypt	1.6.1*, 1.7.3*, 1.9.4**
ElGamal	Libgcrypt	1.6.1, 1.7.3*, 1.9.4**
(EC)DSA	OpenSSL	1.0.1e, 1.1.0g, 1.1.0$i^\star$ 1.1.1$n^\star$, 3.0.2*

6.1 Results Overview

Vulnerability Detection. We present the positives reported by CaType in Table 3. We report that CaType confirms all cache side channel vulnerabilities that have been found by CacheD/CacheS. Moreover, it identifies new defects that were neglected in previous analyses of the same crypto software. CaType detects precisely 485 information leakage sites, including 440 known sites and 45 newly found sites. To better characterize findings, we adhere to CacheD/-CacheS to group adjacent leakage sites (assembly instructions) into a unit and eliminate duplicated units. This way, 97 known units are confirmed and 14 unknown units are discovered. [64, 66] only report leakage units, which are compared here. We elaborate on the findings of CaType in the following two subsections.

Also, for the constant-time dataset offered by [19], CaType has *no* positive findings, meaning that CaType (over this dataset) does not produce false positives or false negatives. We notice that constant-time computations in this dataset (e.g., comparison and conditional selection) extensively use bitwise operations. Since CaType performs bit-level type inference, CaType manifests high accuracy without treating safe bitwise operations as vulnerable. Note that constant-time operations provided in this dataset are frequently used in modern crypto libraries; thus, experiments on this dataset verify the correctness of CaType to a reasonable extent.

Analysis Against Randomization. CaType is evaluated against blinding over plaintext/ciphertext and keys. CaType confirms that the secret leakage exists in OpenSSL-1.0.2f and Libgcrypt-1.6.1/1.7.3, notwithstanding the introduction of plaintext/ciphertext blinding. Note that secrets are still exposed to side channels without blinding in these cases. In contrast, key blinding mitigates most leakage sites. For instance, evaluations of RSA/ElGamal in Libgcrypt-1.9.4 reveal that secrets are now labeled as random data (with type URA) by CaType. However, this protection is at the cost of introducing extra (potentially vulnerable) procedures to perform blinding. CaType discovers five new leakage sites in RSA/Libgcrypt-1.9.4. These leakage units cover both the private key d and the prime p (recall in RSA, d and p are secrets). Therefore, we show that though key blinding obscures secrets, it introduces new leakage sites due to extra calculations. In sum, by considering random factors with specific refined types, CaType can analyze side channel mitigation techniques implemented in modern crypto software.

Performance Evaluation. We compare CaType with CacheD and CacheS by using the same crypto implementations, and report the comparison results in Table 4 (first five rows). For crypto libraries evaluated by CacheD/CacheS (with a total of 4.4M instructions), CaType finishes the analysis with around 120 CPU seconds, and exhibits promising speed across all evaluation settings with no time-out cases. To compare with CacheD/CacheS, we use the processing time per 10 thousand lines as an indicator. CaType handles per 10 thousand lines in 0.27 seconds on average, while CacheD and CacheS require 4.42 CUP and 35.41 CPU seconds, respectively. We also report performance statistics of other RSA evaluation settings in the next rows of Table 4. Their trace lengths range between thousands and millions. Fig. 10 illustrates the approximately linear correlations between trace length and time. Considering the complexity of analyzing real-world cryptosystems, CaType displays a highly promising performance and scalability.

The performance comparison results (Table 4) demonstrate the superiority of type inference as opposed to existing works (e.g., [3,

Table 3: Identified Information Leakage Sites/Units by CaType. We compare the results with recent works, including CacheD [60], CacheS [59] and DATA [64, 66].

Algorithms	Implementations	Information Leakage Sites (known/unknown)	Information Leakage Units (known/unknown)	CacheD reported [60] Leakage Sites/Units†	CacheS reported [59] Leakage Sites/Units†	DATA reported [64, 66] Leakage Units‡
RSA	OpenSSL 1.0.2f	30/0	6/0	2/2	6/3	4
RSA	OpenSSL 1.1.0g	30/4	8/1	-	-	5
RSA	OpenSSL 1.1.0h	22/0	5/0	-	-	5
RSA	OpenSSL 1.1.1n	9/0	5/0	-	-	3
RSA	OpenSSL 3.0.2	9/4	4/2	-	-	2
RSA	Libgcrypt 1.6.1	31/4	9/1	22/5	40/11	-
RSA	Libgcrypt 1.7.3	24/4	8/1	0/0	0/0	-
RSA	Libgcrypt 1.9.4	4/5	2/3	-	-	-
ElGamal	Libgcrypt 1.6.1	31/4	9/1	22/5	40/11	-
ElGamal	Libgcrypt 1.7.3	24/4	8/1	0/0	0/0	-
ElGamal	Libgcrypt 1.9.4	3/0	1/0	-	-	-
ECDSA	OpenSSL 1.0.1e	98/0	9/0	-	-	9
ECDSA	OpenSSL 1.1.0g	49/0	6/0	-	-	6
ECDSA	OpenSSL 1.1.0i	13/0	3/0	-	-	3
ECDSA	OpenSSL 1.1.1n	14/0	2/0	-	-	2
ECDSA	OpenSSL 3.0.2	14/0	2/0	-	-	3
DSA♮	OpenSSL 1.1.0i	0/4	0/1	-	-	-
DSA(swapped)♮	OpenSSL 1.1.0i	9/4	4/1	-	-	-
DSA	OpenSSL 1.1.1n	13/4	3/1	-	-	3
DSA	OpenSSL 3.0.2	13/4	3/1	-	-	3
total		440/45	97/14	46/12	86/25	48

† The RSA and Elgamal from Libgcrypt library are counted together in CacheD [60] and CacheS [59].
‡ We collect all leaky functions reported in DATA [64, 66] and locate whether these leaky functions appear in the corresponding OpenSSL version.
♮ DSA (OpenSSL-1.1.0i) and its swapped patch are only evaluated for the key blinding part.

Table 4: Performance comparison with CacheD/CacheS. We also list the analysis of eight RSA implementations for scalability assessment.

Crypto setup	Instructions on the Traces	Processing Time (CPU Seconds)	Time of Per 10^4 Lines	CacheD Per 10^4 Lines	CacheS Per 10^4 Lines
RSA & Elgamal OpenSSL-1.0.2f	1,620,404	35.58	0.22	3.49	21.16
RSA & Elgamal Libgcrypt-1.6.1	1,379,652	36.00	0.26	4.93	45.36
RSA & Elgamal Libgcrypt-1.7.3	1,411,081	48.40	0.34	3.92	54.57
total (first three rows)	4,411,137	119.98	0.27	4.42	35.41
RSA-OpenSSL 1.0.2f	1,620,404	35.58	0.22	-	-
RSA-OpenSSL 1.1.0g	822,151	18.58	0.22	-	-
RSA-OpenSSL 1.1.0h	28,874	4.88	1.69	-	-
RSA-OpenSSL 1.1.1n	1,763,970	39.29	0.22	-	-
RSA-OpenSSL 3.0.2	1,711,746	36.57	0.21	-	-
RSA-Libgcrypt 1.6.1	806,410	22.63	0.28	-	-
RSA-Libgcrypt 1.7.3	837,215	23.23	0.27	-	-
RSA-Libgcrypt 1.9.4	114,733	11.25	0.98	-	-

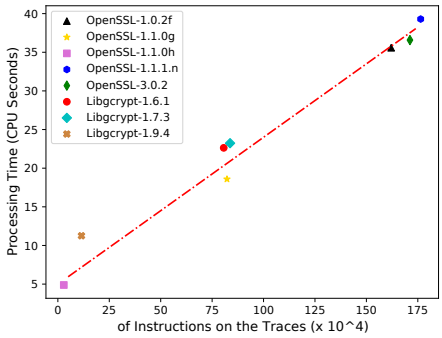


Figure 10: Trace lengths/processing time towards the analysis of RSA implementations.

13, 59, 60]) that use the constraint solver to decide the satisfiability of side channel constraints. Holistically, those works suffer from the accumulation of complex constraints when performing symbolic execution along the trace. In contrast, type inference ensures each deduction step has a straightforward result without huge search space. Overall, without using constraint solving, CaType maintains a comparable analysis capability as those of CacheD/CacheS. As noted in Sec. 4.4, by using bit-level secret tracking (SDD), deciding if secret-dependent memory access leads to cache side channels is recast to essentially recognize SDD in refined types. This pattern match operation is very efficient without undermining soundness.

6.2 Discussion of Known Vulnerabilities

CaType confirms all vulnerabilities reported by CacheD/CacheS in the RSA/Elgamal implementations from Libgcrypt-1.6.1, which adopts pre-computation tables for the sliding-window exponentiation (see Fig. 14 of Appendix C). Although Libgcrypt-1.7.3 employs a direct computation scheme rather than using pre-computation tables, CaType still finds 24 leakage sites that leak the secret length, which also exist in Libgcrypt-1.6.1. However, no leaks are reported in CacheD/CacheS about Libgcrypt-1.7.3. In the CacheS paper, they admit these leak points are false negatives of their tool. As Libgcrypt-1.9.4 adopts a new algorithm (i.e., left-to-right exponentiation), CaType reports a known secret length leakage in function _gcry_mpih_add_n, whereas prior leak operations are discontinued.

Concerning OpenSSL, CaType first confirms the existence of CVE-2018-0737, where RSA private key is leaked during key generation, in functions BN_gcd and BN_mod_inverse from OpenSSL-1.1.0g/1.1.0h. In analyzing modular inverse, CaType detects a new vulnerability in the function BN_rshift1 that discloses the length of the secret (see Sec. 6.3). A recently found vulnerability comes from function BN_num_bits_word, reported in CacheD/CacheS (See Fig. 15 of Appendix C). CaType performs the type deduction process in Table 1. The issue exists in OpenSSL-1.0.2f/1.1.0g/1.1.0h and has been fixed [43], hence disappears in the latest OpenSSL versions (OpenSSL-1.1.1n/3.0.2). CaType confirms a secret length leakage in function BN_window_bits_for_ctime_exponent_size in all analyzed OpenSSL versions, shown in Listing 1. The issue is also reported in CacheS, but is not fixed in the latest OpenSSL. CaType also detects a vulnerability reported in DATA, where constant-time flags of RSA secret primes p and q are not propagated to the temporary copies inside the function BN_MONT_CTX_set during the

Montgomery initialization for modular inverse. This issue exists in OpenSSL-1.0.2f, but the other four OpenSSL libraries resolve it.

Listing 1: Window size of modular exponentiation.

```
1 BN_window_bits_for_ctime_exponent_size(b) \
2       ((b) > 937 ? 6 : \
3       (b) > 306 ? 5 : \
4       (b) >  89 ? 4 : \
5       (b) >  22 ? 3 : 1)
```

When evaluating the (EC)DSA implementations, we mark the nonce used in Montgomery ladder as a secret. This is because the leaky nonce can result in the Hidden Number Problem (HNP) [11, 12], where collecting enough leaky nonce contributes to the recovery of private keys through constructing lattice [7, 39, 40]. CaType confirms a direct leakage of the nonce in the Montgomery ladder implementation from OpenSSL-1.0.1e. This vulnerability was reported in [69], and this flaw (CVE-2014-0076) has been fixed by the developers and implemented in a non-branch commit [41, 42]. Recently, Ryan [50] reports a vulnerability located in modular reduction of (EC)DSA implementations in OpenSSL that uses an early abort condition to estimate the range of private keys. CaType confirms this vulnerability comes from function BN_ucmp and BN_usub inside function BN_mod_add_quick (see Fig. 16 of Appendix C).

CaType is also evaluated on analyzing the lifetime of a nonce, including the generation, scalar multiplication, modular inversion, and main signing process. The leakage sites identified by CaType fully cover the findings reported in [64]. For example, by distinguishing whether an extra limb is used to expand the representation of nonce in BN_add, CaType confirms the padding resize vulnerabilities about the nonce reported in CVE-2018-0734 for DSA and CVE-2018-0735 for ECDSA, as shown in Listing 2. The vulnerability states that the result buffer resizes one more limb to hold the result. By distinguishing the resize operations, attackers can learn the range information of the nonce. Other known leakage sites of the nonce (e.g., skipping leading zero limbs through bn_correct_top, performing an early stop in BN_cmp, and conditional branches in BN_mul) are identified by CaType; they still exist in the latest versions. Individually, CaType reports non-constant-time vulnerabilities in OpenSSL-1.0.1e when performing ECDSA nonce modular inverse. This is because the constant-time flag was not set to the nonce. OpenSSL-1.1.0g/1.1.0i, on the other hand, implement Fermat's little theorem via constant-time modular exponentiation. Benefit to the cache layout checking, CaType finds four new leakage sites that reveal the secret key size through a series of `else/if` branches in DSA from OpenSSL-1.1.0i/1.1.1n/3.0.2 (see Sec. 6.3). Contrary to our expectations, CaType does not mark cases in the `switch` statement of BN_copy as vulnerable. Through rechecking the source code and its disassembly code, we confirm that CaType performs a correct inference because the trace on the cache cannot be distinguished (see Sec. 6.5).

Listing 2: Bignumber resize.

```
1 if (!BN_add(r, k, order)
2   || !BN_add(X, r, order)
3   || !BN_copy(k, BN_num_bits(r)>order_bits ? r:X))
4   goto err;
```

6.3 Unknown Vulnerabilities

CaType finds new vulnerable program points in Libgcrypt-1.6.1/1.7.3 that have been analyzed by existing tools. It finds that the size of secret exponentiation is leaked through the `if/else` statements at the beginning of function _gcry_mpi_powm, as shown in Listing 3. The sliding-window size `W` is determined by the size of secret exponent `esize`. Different execution traces of the `if/else` statements can be differentiated because it occupies multiple cache lines. However, we admit that the `if/else` statements are a moderate leakage because only line 1 and line 5 can be distinguished directly. CaType cannot distinguish execution between line 2 and line 4.

Listing 3: Window size selection.

```
1 if (esize * BITS_PER_MPI_LIMB > 512) W = 5;
2 else if (esize * BITS_PER_MPI_LIMB > 256) W = 4;
3 else if (esize * BITS_PER_MPI_LIMB > 128) W = 3;
4 else if (esize * BITS_PER_MPI_LIMB > 64) W = 2;
5 else  W = 1;
```

We find a new vulnerability in the OpenSSL function BN_rshift1 which performs GCD using the Euclid algorithm. Fig. 17 in Appendix D presents the source code from version 1.1.0g. We first demonstrate how this function leaks the length of the one-shifted-right operand. Function BN_rshift1 performs shifting to the right one-bit for each element of the BN_ULONG structure. The length of the source operand (i.e., `a->top`) is used as the while loop's condition. CaType confirms it as a secret-dependent branch, where the judgment of the while loops and part instructions inside the while loops (lines 11–13) are stored in one cache line and the subsequent instructions until the end of function BN_rshift1 (lines 14–17) are stored in another cache line. Therefore, the trace of the while loops can be distinguished. By probing the while loop condition, the value of `a->top` is inferred as one increment to the number of while loops.

[65] proposes a page-level attack to recover RSA primes p and q when performing prime testing using BN_gcd. CaType confirms the vulnerability in which four different branches are identified because BN_rshift1 of each branch is at different cache lines. Meanwhile, we argue the length information of the source operand leaked by BN_rshift1 accelerates the recovery in [65]. For example, between two adjacent loop operations ($a_{i+1} = a_i/2,\ a_{i+1} = (a_i - b_i)/2$ or $a_{i+1} = (a_i - b_i)/2,\ a_{i+1} = a_i/2$), one decrement in the latter a_{i+1}'s length indicates that the topmost bit of the former a_{i+1} is one. This deduction helps to reduce the range of intermediate results for each Euclid loop. In addition to BN_rshift1, CaType finds similar leakage in BN_lshift1 from OpenSSL-3.0.2.

CaType also finds another vulnerability in the OpenSSL-1.1.0i implementation of DSA (Fig. 18 in Appendix D). The key blinding mechanism of DSA first multiplies the random factor `blind` and the DSA secret key `dsa->priv_key` by calling the function BN_mul, which calls the function bn_mul_normal to perform a classic multiplication if the length of both operands is less than BN_MULL_SIZE_NORMAL. Fig. 18 presents a for-loop, where four elements of the secret key as a group are multiplied by `blind`. Then, four branches, where `nb` is the length of the secret key, control whether to end the loop. When `nb` equals zero, the multiplication is complete and the function bn_mul_normal returns. CaType confirms that the secret-dependent branches leak the length of the

secret key. By probing different if-conditions present in distinct cache lines, the value of the secret length can be recovered. Such vulnerable operations are found in the latest OpenSSL-1.1.1n/3.0.2.

6.4 Discussion about Blinding

As stated in Sec. 6.1, CaType shows that the plaintext/ciphertext blinding cannot eliminate cache side channels, given that secrets themselves are still exposed (e.g., secret-dependent memory accesses and branches in modular exponentiation from Libgcrypt-1.6.1). However, key blinding impedes nearly all leakage. For example, CaType reports no vulnerability in the modular exponentiation from Libgcrypt-1.9.4. By inspecting the type inference outputs, we find that the secret exponent is marked as a random number (URA) through a series of blinding operations before conducting modular exponentiation. However, CaType finds new leakage sites in the blinding process. Considering key blinding in RSA/Libgcrypt-1.9.4, which uses $d\_blind = (d \bmod (p-1)) + (p-1) * r$ to mask the secret exponent d before performing modular exponentiation. Here, p represents one RSA prime number and r is the random factor. CaType newly discovers five leakage sites in the subtraction and division operations. They leak the length of the prime number p and secret exponent d. For instance, the function _gcry_mpi_sub_ui is invoked to perform $p-1$ on p. It leaks the length of p whenever the resize operation is performed on the result operand, as well as at other length-related branches.

Apart from the key blinding in Libgcrypt-1.9.4, CaType also explores the effect of different key blinding positions on mitigating cache side channels. For instance, DSA implementation from OpenSSL-1.1.0i applies key blinding b to avoid leaking the private key x as follows:

$$s = (bm + bxr) \bmod q \quad (2)$$

$$s = s \cdot k^{-1} \bmod q \quad (3)$$

$$s = s \cdot b^{-1} \bmod q \quad (4)$$

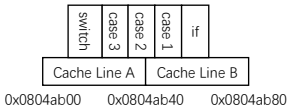
where statements 2, 3, and 4 are executed sequentially. Swapping statements 3 and 4 results in different key blinding use, which is applied in a LibreSSL patch [33]. CaType compares the original patch with the swapped one (we manually swap statements 3 and 4 in OpenSSL-1.1.0i DSA). We find nine additional leakage sites related to the length of the inverse nonce `kinv` in the swapped patch (see Table 3), although the statement 3 also leaks the inverse nonce length in the original patch. We argue when executing statement 4 first, s does not possess the property of randomization anymore due to $b(m+xr)b^{-1} \bmod q \equiv (m+xr) \bmod q$. Hence, the nonce inverse `kinv` is exposed to the attacker. The swapped practice is fix in a LibreSSL patch [32].

6.5 Reducing False Positives

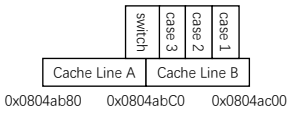
We explain how CaType reduces false positives by using cache layouts rather than cache states to detect side channels. Considering Fig. 11, function BN_copy is used by RSA and (EC)DSA. Take (EC)DSA as an example, whose secret nonce is copied from `b` to `a` via BN_copy. In particular, a `switch` statement at line 8 helps skipping the copy of leading zero in `b`. By manually reviewing this function, we would anticipate that certain information about the nonce is leaked by discriminating executed switch cases. However, CaType deems this case as safe.

```
1 BIGNUM * BN_copy
2   (BIGNUM * a,
3   BIGNUM * b){
4   ...
5   /* assign values
6     in groups of 4 */
7   ...
8   switch (b->top&3){
9   case 3 : A[2] = B[2];
10   case 2 : A[1] = B[1];
11   case 1 : A[0] = B[0];
12   case 0 :; }
13   ... }
```

(a) BN_copy function.



(b) of from OpenSSL-1.1.0g



(c) of from OpenSSL-1.1.0h

Figure 11: BN_copy from the OpenSSL Library.

We analyze the result released by CaType from the perspectives of both FLUSH-RELOAD and PRIME-PROBE attacks. We depict the cache layouts of BN_copy from OpenSSL-1.1.0g and OpenSSL-1.1.0h in Fig. 11(b) and Fig. 11(c). In these two libraries, the `switch` statement occupies two separate cache lines. Thus, the first cache line must be visited. Meanwhile, instructions after the statement are loaded into the second cache line and are also visited; in an extreme case, the whole `switch` statement is loaded into one cache line. In sum, different switch cases are not distinguishable (e.g., for the FLUSH-RELOAD attack). We further consider whether a PRIME-PROBE attack can distinguish the difference in cache layouts. First, the base addresses are loaded into the cache regardless of whether they correspond to the source array (`A[]`) or the destination array (`B[]`). Second, the largest offset for the element among the last group (both destination and source) is 8 bytes. In that sense, the address of any element is mapped to the same cache line (address $\gg 6$ for 64-byte cache lines). Therefore, PRIME-PROBE cannot collect a distinguishable observation and fails to extract secrets. However, CacheD/CacheS simply treats BN_copy as vulnerable, given that a secret-dependent branch condition (line 8) is (inaccurately) treated as "vulnerable" in the view of their cache state-based vulnerability pattern. However, it is indeed a false positive.

Robustness of Using Cache Layouts. The above experiments are conducted using OpenSSL's default compilation setting. The `switch` statement may be vulnerable, when the code chunk of each switch case occupies distinct cache lines. Overall, we anticipate that different optimization settings could result in placing instructions into different cache lines. To benchmark the robustness of using cache layouts instead of using cache state-based threat models, we measure how compiler optimizations may influence the results of CaType, whose results are given in Table 5. At this step, we only measure side channels due to SDBC, because we use the cache layout model to check SDBC. Also, given that we need to manually confirm and compare each finding across different optimizations, we only select a crypto library when its SDBC-related source code has visible changes across different versions. For instance, while we evaluate Libgcrypt 1.6.1, 1.7.3, and 1.9.4 in Table 3, we only evaluate versions 1.7.3 and 1.9.4, since version 1.6.1 appears to be identical with 1.7.3 in terms of those SDBC cases flagged by CaType.

Table 5 shows that optimizations affect the analysis results, as heavy optimizations tend to "condense" code into fewer cache lines. Similar to Table 3, we provide the discovered leakage sites as well as grouped leakage units. CaType can accurately capture the subtle leakage (without making false positives) with its employed cache

Table 5: Branch vulnerabilities identified by CaType under gcc -O0, -O2, and -O3 optimization settings.

Crypto setup	gcc-5.4		
	-O0	-O2	-O3
RSA-OpenSSL 1.1.0g	27/9	24/9	24/9
RSA-OpenSSL 1.1.0h	20/5	18/5	18/5
RSA/Elgamal-Libgcrypt 1.7.3	17/7	14/7	14/7
RSA/Elgamal-Libgcrypt 1.9.4	6/4	6/4	6/4
ECDSA-OpenSSL 1.1.0g	38/6	22/6	19/6
ECDSA-OpenSSL 1.1.0i	10/3	7/3	7/3
ECDSA-OpenSSL 3.0.2	9/2	9/2	9/2
DSA-OpenSSL 1.1.0i	4/1	3/1	3/1
DSA-OpenSSL 1.1.1n	14/4	12/4	12/4
total	145/41	115/41	112/41

layout threat model. With manual efforts, we confirm that *all* cases are true positives. Indeed, we report that all `-O2` findings are subsumed by those of `-O0`, and all `-O3` findings are subsumed by `-O2` findings. In contrast, we report that CacheD/CacheS yields *identical* findings across different optimization settings, meaning that they have a considerable number of false positives under `-O2` and `-O3`.

7 Discussion and Limitation

Type System Benchmarking. Scientifically, it would be ideal to benchmark our refinement type system against some "synthetic datasets" to determine their algorithmic effectiveness and efficiency before evaluating side channel detections, which is a "downstream" application of our type system. Nevertheless, it is practically hard to find a proper (synthetic) dataset to solely evaluate the type system, and using downstream applications to reflect the effectiveness of a type system is a common evaluation plan used by relevant works [56–58]. To avoid potential confusion, we revisit the effectiveness and efficiency of our type system as follows.

First, our type system is sound (per Proposition 4.1). All typing rules are intuitive, and there are no "tricky" ones implemented in CaType. Thus, the soundness is at ease. Second, in terms of efficiency, our implementation manifests approximately $O(n)$ complexity, where n is the number of instructions in a given trace. CaType is empirically very efficient. As demonstrated in Fig. 10, CaType manifests a mostly linear growth in terms of the trace length and processing time. Overall, the end-to-end evaluation on side channel analysis illustrates the accuracy of CaType, thereby reflecting the effectiveness of its underlying type systems at large.

Further to the above discussion, we empirically evaluate the type system by comparing it with taint analysis to check correctly-tagged variables. In general, taint analysis offers a holistic modelling of how secrets propagate through the program, while our type system is *more precise*. Most taint analysis implementation is performed at the syntax level (whose cost and accuracy is conceptually similar to conventional, syntax-level type inference). In contrast, as shown in Sec. 3.2 and Fig. 2, CaType's type system tracks bit-level values/secrets uniformly using refined types; thus, the type system captures stronger semantics properties, e.g., it models how blinding obscures secrets. Therefore, properly masked secrets are not treated as secrets in CaType (i.e., they do not have an SDD type), but taint analysis will "over-taint" them.

Recall CaType first conducts taint analysis over the Pin-logged trace before performing type inference. Thus, we compare the number of tainted registers/memory cells with the number of variables of type SDD over the same trace. Table 6 reports the evaluation results. As clarified above and observed in Table 6, the number of variables of SDD type is less than the number of tainted variables, as expected. Also, we confirm that *all* variables of type SDD exist in the tainted set, i.e., our type inference phase has no false negatives (when using tainted variables as the baseline). More importantly, we also manually study every "over-tainted" variable that does not have type SDD. As shown in the 7th column of Table 6, taint analysis finds considerably more tainted variables than type inference. Given the difficulty of manual inspection, for each evaluation setting, we randomly select 100 cases (if there are more than 100 cases). For each case, we comprehend the causality of how variable is tainted, and decide if this is a true positive (meaning that the tainted variable is carrying secrets correctly) or not.

We show the manual inspection results in the last column of Table 6. We find that all the "over-tainted" variables are *false positives* of the taint analysis. It is thus correct for our type system to neglect them. Among in total 1,905 randomly selected cases, the "over-tainted" variables belong to the following categories: ① variables of SDD type that have been appropriately masked with blinding, while they are still tainted, ② variables that are further tainted by variables belonging to ①, ③ variables of SDD type that have been zeroized by constants, whereas taint analysis retains the taint label over those variables, and ④ the base address of a secret buffer is deemed as a taint source, such that whenever loading from the base address, the output will be tainted. While ①, ②, and ③ are due to the inherent limitation of standard taint analysis technique, ④ is due to the "clumsy" implementation of our adopted taint analysis tool.[1] Out of 1,905 manually checked cases, we find that about 52% cases fall in ④, whereas the remaining 48% cases are due to ①, ②, or ③. Thus, we estimate that around 143K ($298525 \times 48\%$) false positives are due to the inherent limitation of taint analysis, which are correctly eliminated by our refinement type system.

Extension. We discuss the extension of CaType from both architectural and analysis target perspectives. First, the current implementation of CaType supports to analyze 32-bit x86 binaries. Given that the closely-related works (e.g., CacheD, CacheS, and CacheAudit) only support 32-bit x86 binaries, supporting the same binary format enables an "apple-to-apple" comparison. Moreover, CaType can be extended to 64-bit binaries with no extra research challenge. We expect to convert each refinement type, currently a 32-bit vector, to a 64-bit vector. We also need to handle new instructions. Nevertheless, these are engineering endeavors rather than open-ended research problems. We leave it as one future work to support other architectures including 64-bit x86.

Also, from the analysis target perspective, side channel analyzers in this field require to flag program secrets (or other sensitive data) specified by users, and then start to analyze their influence on cache. Detectors (including CaType) are *not* limited to crypto software. Analyzing crypto software targeted by previous analyzers, however, makes it easier to compare CaType with them. Given the scalability of CaType, it should be feasible to extend CaType to analyze production software running in trusted execution environments (TEEs) and detect their side channel leaks [1, 17, 61].

[1] We use the taint analysis tool provided by CacheD. Note that ④ eases the implementation of a taint engine, but overestimates secrets. Secrets (and their associated non-secret data) are often stored in a BIGNUM struct. By treating the base address of this struct as the taint source, non-secret data in the struct are all tainted due to ④.

Table 6: Checking the correctness of refinement type system in CaType by comparing with taint analysis. "FPs" denotes false positives of taint analysis. We randomly select 100 cases for each setting for confirmation except ElGamal/Libgcrypt 1.9.4.

Algorithms	Implementations	Instructions on the traces	Tainted instructions	Tainted registers and memory cells	Registers and memory cells with SDD types	"Over-tainted" registers and memory cells	#FPs in manually confirmed 100 "over-tainted" cases
RSA	OpenSSL 1.0.2f	1,620,404	4,127	3,271	3,165	106	100
RSA	OpenSSL 1.1.0g	822,151	4,092	3,568	3,452	116	100
RSA	OpenSSL 1.1.0h	28,874	9,672	7,939	5,977	1,962	100
RSA	OpenSSL 1.1.1n	1,763,970	51,933	34,518	30,276	4,242	100
RSA	OpenSSL 3.0.2	1,711,746	56,218	39,799	37,507	2,292	100
RSA	Libgcrypt 1.6.1	806,410	130,141	125,648	100,493	25,155	100
RSA	Libgcrypt 1.7.3	837,215	140,478	133,105	107,731	25,374	100
RSA	Libgcrypt 1.9.4	114,733	102,132	104,536	103,676	860	100
ElGamal	Libgcrypt 1.6.1	573,242	334,024	286,095	184,976	101,119	100
ElGamal	Libgcrypt 1.7.3	573,866	334,194	286,213	185,297	100,916	100
ElGamal	Libgcrypt 1.9.4	4,676	1,274	1,145	1,140	5	5
ECDSA	OpenSSL 1.0.1e	2,277,459	258,261	241,326	237,381	3,945	100
ECDSA	OpenSSL 1.1.0g	415,415	140,596	124,907	110,488	14,419	100
ECDSA	OpenSSL 1.1.0i	298,463	81,988	72,090	55,810	16,280	100
ECDSA	OpenSSL 1.1.1n	182,745	315	220	120	100	100
ECDSA	OpenSSL 3.0.2	164,613	315	220	120	100	100
DSA♮	OpenSSL 1.1.0i	18,516	4,766	3,970	3,371	599	100
DSA(swapped)♮	OpenSSL 1.1.0i	18,608	4,698	3,899	3,230	669	100
DSA	OpenSSL 1.1.1n	1678	578	435	302	133	100
DSA	OpenSSL 3.0.2	1678	578	435	302	133	100
total		12,236,462	1,660,380	1,473,339	1,174,814	298,525	1,905

♮ DSA (OpenSSL-1.1.0i) and its swapped patch are only evaluated for the key blinding part.

8 Related Work

Perfect masking analysis conducted on power side channels is highly relevant to our work [31, 37]. In such analysis, all intermediate computation outputs are statistically examined for independence between secret data and power side channels. Recent efforts employ a type-based technique to deduce potentially leakage of program intermediate variables. Specifically, [4, 5, 23] use a syntactic type system that primarily relies on the variable structural information. [28, 72] extend the syntax-based approach to a semantic-based type system that refines inference rules for boolean masking scheme analysis. Two improvements [27, 45] add rules for additive and multiplicative masking. These works inspire the design of our refinement type system. However, crucial gaps exist in applying these rules to detect cache side channels. First, perfect masking analysis of software power side channel countermeasures targets specific masked programs (often bitwise operations), whose computation is usually straightforward (calculating and then assigning). Cache side channel analysis targets complicated production cryptosystems. Type systems proposed in prior works are primarily for bitvector logical operations, not general x86 assembly semantics. Second, our tentative exploration shows that earlier typing rules were often incomplete; they may need to use constraint solving when typing rules cannot be applied. Their performance is therefore downgraded. In contrast, CaType's type inference rules completely infer refined types for variables.

9 Conclusions

Detecting cache side channels in production cryptographic software is still an open problem. This paper presents CaType, a refinement type-based tool to deliver highly efficient and accurate analysis of cache side channels over x86 binary code. Evaluation over real-world cryptographic software shows that CaType identifies side channels with high precision, efficiency, and scalability.

Acknowledgments

We would like to thank the anonymous reviewers for their constructive feedback. This work has been supported in part by Singapore National Research Foundation under its National Cybersecurity R&D Programme (NCR Award NRF2018 NCR-NCR009-0001), Singapore Ministry of Education (MOE) AcRF Tier 1 RS02/19, NTU Start-up grant.

References

[1] Adil Ahmad, Byunggill Joe, Yuan Xiao, Yinqian Zhang, Insik Shin, and Byoungyoung Lee. 2019. Obfuscuro: A commodity obfuscation engine on intel sgx. In *Network and Distributed System Security Symposium*.

[2] Diego F Aranha, Felipe Rodrigues Novaes, Akira Takahashi, Mehdi Tibouchi, and Yuval Yarom. 2020. Ladderleak: Breaking ecdsa with less than one bit of nonce leakage. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 225–242.

[3] Qinkun Bao, Zihao Wang, Xiaoting Li, James R Larus, and Dinghao Wu. 2021. Abacus: Precise side-channel analysis. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 797–809.

[4] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, and Pierre-Yves Strub. 2015. Verified proofs of higher-order masking. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 457–485.

[5] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, Pierre-Yves Strub, and Rébecca Zucchini. 2016. Strong non-interference and type-directed higher-order masking. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 116–129.

[6] Gilles Barthe, Cédric Fournet, Benjamin Grégoire, Pierre-Yves Strub, Nikhil Swamy, and Santiago Zanella-Béguelin. 2014. Probabilistic relational verification for cryptographic implementations. *ACM SIGPLAN Notices* 49, 1 (2014), 193–205.

[7] Naomi Benger, Joop van de Pol, Nigel P Smart, and Yuval Yarom. 2014. ?Ooh Aah... Just a Little Bit?: a small amount of side channel can go a long way. In *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 75–92.

[8] Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D Gordon, and Sergio Maffeis. 2011. Refinement types for secure implementations. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 33, 2 (2011), 1–45.

[9] Karthikeyan Bhargavan, Cédric Fournet, and Andrew D Gordon. 2010. Modular verification of security protocol code by typing. *ACM Sigplan Notices* 45, 1 (2010), 445–456.

[10] Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, and Pierre-Yves Strub. 2013. Implementing TLS with verified cryptographic security. In *2013 IEEE Symposium on Security and Privacy*. IEEE, 445–459.

[11] Dan Boneh and Ramarathnam Venkatesan. 1996. Hardness of computing the most significant bits of secret keys in Diffie-Hellman and related schemes. In *Annual International Cryptology Conference*. Springer, 129–142.

[12] Dan Boneh and Ramarathnam Venkatesan. 1997. Rounding in Lattices and its Cryptographic Applications.. In *SODA*, Vol. 1997. Citeseer, 675–681.

[13] Robert Brotzman, Shen Liu, Danfeng Zhang, Gang Tan, and Mahmut Kandemir. 2019. CaSym: Cache aware symbolic execution for side channel detection and mitigation. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 505–521.

[14] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J Schwartz. 2011. BAP: A binary analysis platform. In *International Conference on Computer Aided Verification*. Springer, 463–469.

[15] Luca Cardelli. 1996. Type systems. *ACM Computing Surveys (CSUR)* 28, 1 (1996), 263–264.

[16] Sudipta Chattopadhyay, Moritz Beck, Ahmed Rezine, and Andreas Zeller. 2019. Quantifying information leakage in cache attacks via symbolic execution. *TECS* (2019).

[17] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. 2019. Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 142–157.

[18] P. Cousot and R. Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 238–252.

[19] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. 2020. Binsec/rel: Efficient relational symbolic execution for constant-time at binary-level. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1021–1038.

[20] Leonid Domnitser, Aamer Jaleel, Jason Loew, Nael Abu-Ghazaleh, and Dmitry Ponomarev. 2012. Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *ACM Transactions on Architecture and Code Optimization (TACO)* 8, 4 (2012), 1–21.

[21] Goran Doychev, Dominik Feld, Boris Kopf, Laurent Mauborgne, and Jan Reineke. 2013. {CacheAudit}: A Tool for the Static Analysis of Cache Side Channels. In *22nd USENIX Security Symposium (USENIX Security 13)*. 431–446.

[22] Goran Doychev and Boris Köpf. 2017. Rigorous analysis of software countermeasures against cache attacks. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 406–421.

[23] Inès Ben El Ouahma, Quentin L Meunier, Karine Heydemann, and Emmanuelle Encrenaz. 2017. Symbolic approach for side-channel resistance analysis of masked assembly codes. In *Security Proofs for Embedded Systems*.

[24] Hassan Eldib, Chao Wang, and Patrick Schaumont. 2014. Formal verification of software countermeasures against side-channel attacks. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 24, 2 (2014), 1–24.

[25] Hassan Eldib, Chao Wang, and Patrick Schaumont. 2014. SMT-based verification of software countermeasures against side-channel attacks. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 62–77.

[26] Hassan Eldib, Chao Wang, Mostafa Taha, and Patrick Schaumont. 2014. QMS: Evaluating the side-channel resistance of masked software from source code. In *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[27] Pengfei Gao, Hongyi Xie, Jun Zhang, Fu Song, and Taolue Chen. 2019. Quantitative verification of masked arithmetic programs against side-channel attacks. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 155–173.

[28] Pengfei Gao, Jun Zhang, Fu Song, and Chao Wang. 2019. Verifying and quantifying side-channel resistance of masked software implementations. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 28, 3 (2019), 1–32.

[29] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+ Flush: a fast and stealthy cache attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 279–299.

[30] Ranjit Jhala, Niki Vazou, et al. 2021. Refinement Types: A Tutorial. *Foundations and Trends in Programming Languages* 6, 3–4 (2021), 159–317.

[31] Paul Kocher, Joshua Jaffe, and Benjamin Jun. 1999. Differential power analysis. In *Annual international cryptology conference*. Springer, 388–397.

[32] Libressl-1f6b35b. 2019. Remove the blinding later to avoid leaking information on the length. https://github.com/libressl-portable/openbsd/commit/1f6b35b

[33] Libressl-2cd28f9. 2018. Use a blinding value when generating a DSA signature. https://github.com/libressl-portable/openbsd/commit/2cd28f9?diff=unified

[34] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. 2015. Last-level cache side-channel attacks are practical. In *2015 IEEE symposium on security and privacy*. IEEE, 605–622.

[35] Xiaoxuan Lou, Tianwei Zhang, Jun Jiang, and Yinqian Zhang. 2021. A Survey of Microarchitectural Side-channel Vulnerabilities, Attacks, and Defenses in Cryptography. *ACM Computing Surveys (CSUR)* 54, 6 (2021), 1–37.

[36] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices* 40, 6 (2005), 190–200.

[37] Amir Moradi, Alessandro Barenghi, Timo Kasper, and Christof Paar. 2011. On the vulnerability of FPGA bitstream encryption against power analysis attacks: Extracting keys from Xilinx Virtex-II FPGAs. In *Proceedings of the 18th ACM conference on Computer and communications security*. 111–124.

[38] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: an efficient SMT solver *(TACAS)*.

[39] Phong Q Nguyen and Igor E Shparlinski. 2002. The insecurity of the digital signature algorithm with partially known nonces. *Journal of Cryptology* 15, 3 (2002).

[40] Phong Q Nguyen and Igor E Shparlinski. 2003. The insecurity of the elliptic curve digital signature algorithm with partially known nonces. *Designs, codes and cryptography* 30, 2 (2003), 201–217.

[41] OpenSSL-2198be3. 2014. Fix for CVE-2014-0076. https://github.com/openssl/openssl/commit/2198be3483259de374f91e57d247d0fc667aef29

[42] OpenSSL-4b7a4ba. 2014. Fix for CVE-2014-0076. https://github.com/openssl/openssl/commit/4b7a4ba29cafa432fc4266fe6e59e60bc1c96332

[43] OpenSSL-972c87d. 2018. Make bn_num_bits_word constant-time. https://github.com/openssl/openssl/commit/972c87dfc7e765bd28a4964519c362f0d3a58ca4

[44] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache attacks and countermeasures: the case of AES. In *Cryptographers' track at the RSA conference*. Springer, 1–20.

[45] Gao Pengfei, Xie Hongyi, Pu Sun, Jun Zhang, Fu Song, and Taolue Chen. 2020. Formal verification of masking countermeasures for arithmetic programs. *IEEE Transactions on Software Engineering* (2020).

[46] Colin Percival. 2005. Cache missing for fun and profit.

[47] Benjamin C Pierce. 2002. *Types and programming languages*. MIT press.

[48] Antoon Purnal, Lukas Giner, Daniel Gruss, and Ingrid Verbauwhede. 2021. Systematic analysis of randomization-based protected cache architectures. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 987–1002.

[49] Moinuddin K Qureshi. 2018. CEASER: Mitigating conflict-based cache attacks via encrypted-address and remapping. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 775–787.

[50] Keegan Ryan. 2019. Return of the Hidden Number Problem. *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2019), 146–168.

[51] Andrei Sabelfeld and Andrew C. Myers. 2003. Language-based information-flow security. *IEEE J. Sel. Areas Commun.* 21, 1 (2003), 5–19. https://doi.org/10.1109/JSAC.2002.806121

[52] Werner Schindler. 2015. Exclusive exponent blinding may not suffice to prevent timing attacks on RSA. In *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 229–247.

[53] Laurent Simon, David Chisnall, and Ross Anderson. 2018. What you get is what you C: Controlling side effects in mainstream C compilers. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 1–15.

[54] Wei Song, Boya Li, Zihan Xue, Zhenzhen Li, Wenhao Wang, and Peng Liu. 2021. Randomized last-level caches are still vulnerable to cache side-channel attacks! But we can fix it. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 955–969.

[55] Chungha Sung, Brandon Paulsen, and Chao Wang. 2018. CANAL: a cache timing analysis framework via LLVM transformation *(ASE)*.

[56] John Toman, Ren Siqi, Kohei Suenaga, Atsushi Igarashi, and Naoki Kobayashi. 2020. ConSORT: Context- and Flow-Sensitive Ownership Refinement Types for Imperative Programs. In *ESOP (Lecture Notes in Computer Science, Vol. 12075)*. Springer, 684–714. https://doi.org/10.1007/978-3-030-44914-8_25

[57] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon L. Peyton Jones. 2014. Refinement types for Haskell. In *ICFP*. ACM, 269–282. https://doi.org/10.1145/2628136.2628161

[58] Panagiotis Vekris, Benjamin Cosman, and Ranjit Jhala. 2016. Refinement types for TypeScript. In *PLDI*. ACM, 310–325. https://doi.org/10.1145/2908080.2908110

[59] Shuai Wang, Yuyan Bao, Xiao Liu, Pei Wang, Danfeng Zhang, and Dinghao Wu. 2019. Identifying cache-based side channels through secret-augmented abstract interpretation. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 657–674.

[60] Shuai Wang, Pei Wang, Xiao Liu, Danfeng Zhang, and Dinghao Wu. 2017. Cached: Identifying cache-based timing channels in production software. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*. 235–252.

[61] Wubing Wang, Yinqian Zhang, and Zhiqiang Lin. 2019. Time and Order: Towards Automatically Identifying {Side-Channel} Vulnerabilities in Enclave Binaries. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. 443–457.

[62] Zhenghong Wang and Ruby B Lee. 2007. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 34th annual international symposium on Computer architecture*. 494–505.

[63] Zhenghong Wang and Ruby B Lee. 2008. A novel cache architecture with enhanced performance and security. In *2008 41st IEEE/ACM International Symposium on Microarchitecture*. IEEE, 83–93.

[64] Samuel Weiser, David Schrammel, Lukas Bodner, and Raphael Spreitzer. 2020. Big Numbers-Big Troubles: Systematically Analyzing Nonce Leakage in ({EC} DSA} Implementations. In *29th USENIX Security Symposium (USENIX Security 20)*. 1767–1784.

[65] Samuel Weiser, Raphael Spreitzer, and Lukas Bodner. 2018. Single trace attack against RSA key generation in Intel SGX SSL. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*. 575–586.

[66] Samuel Weiser, Andreas Zankl, Raphael Spreitzer, Katja Miller, Stefan Mangard, and Georg Sigl. 2018. {DATA}–Differential Address Trace Analysis: Finding Address-based {Side-Channels} in Binaries. In *27th USENIX Security Symposium (USENIX Security 18)*. 603–620.

[67] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. 2019. Scattercache: Thwarting cache attacks via cache set randomization. In *USENIX Security Symposium*.

[68] Jan Wichelmann, Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. 2018. MicroWalk: A Framework for Finding Side Channels in Binaries. In *ACSAC*.

[69] Yuval Yarom and Naomi Benger. 2014. Recovering OpenSSL ECDSA Nonces Using the FLUSH+ RELOAD Cache Side-channel Attack. *IACR Cryptol. ePrint Arch.* 2014 (2014), 140.

[70] Yuval Yarom and Katrina Falkner. 2014. FLUSH+ RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*. 719–732.

[71] Danfeng Zhang, Aslan Askarov, and Andrew C Myers. 2012. Language-based control and mitigation of timing channels. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*. 99–110.

[72] Jun Zhang, Pengfei Gao, Fu Song, and Chao Wang. 2018. SC Infer: refinement-based verification of software countermeasures against side-channel attacks. In *International Conference on Computer Aided Verification*. Springer, 157–177.

[73] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. 2012. Cross-VM side channels and their use to extract private keys. In *Proceedings of the 2012 ACM conference on Computer and communications security*. 305–316.

A One-bitvector Constant Type Rules for Logical Operations

Fig. 12 shows the one-bitvector type rules involving the CST type. Rule Const-Conj and Cons-Disj rules handle the situation when the refined type of one operand expression is CST. These four rules are straightforward. Rule XOR.III and XOR.IV describe the refined type CST cases, which are consistent with basic cognition, and the value predicates are given. Rules Neg.II and Neg.III keep security types unchanged while tracking values precisely in types.

B Type Rules for Statements

We extend the type environment Γ (defined in Sec. 4.1) to track the value and security type of each element in a vector, i.e., $\Gamma ::= \emptyset \mid \Gamma, x : \rho \mid \Gamma, e_1[e_2] : \rho$. We use the following rule to derive the type of vector indexing expression:

$$\text{T-Vec-Index}\ \frac{e_1[e_2] \in \Gamma}{\Gamma \vdash e_1[e_2] : \rho}$$

Fig. 13 shows the type rules for statements. It tracks values in a flow-sensitive way. The type judgment is in the form $\Gamma \vdash S \dashv \Gamma'$, meaning that statement S is type-checked under Γ, and produces a new type environment Γ'. The notation $\Gamma[x \mapsto \rho]$ overrides x's type with ρ in Γ if x is in Γ; otherwise extends Γ with $[x \mapsto \rho]$.

Rules Assign-I and Assign-II are for assignments, with and without the presence of a value predicate respectively. Rule Assign-I updates variable x's value predicate with the one on the right-hand side, enabling precise tracking values in types. Rules Load and Store are used for reading from and writing into memories, where memory access safety is assumed. Rules Load-I and Load-II retrieve the type of the vector e_1 at index e_2 and update x's type with it if $e_1[e_2]$ is already tracked in Γ. Otherwise, x's type is updated with $\tau_1 \sqcup \tau_2$ (rule Load-III), which is capable of tracking implicit information flow. Rules Store-I and Store-II update the type of the element $e_1[e_2]$ with x's type. Rule Seq checks the first instruction S_1 under Γ and produces a new type environment Γ'', under which, instruction S_2 is checked.

$$\text{Prim-I}\ \frac{}{\Gamma \vdash 0 : \{v : B \mid v = 0 \wedge v : \text{CST}\}} \qquad \text{Prim-II}\ \frac{}{\Gamma \vdash 1 : \{v : B \mid v = 1 \wedge v : \text{CST}\}}$$

$$\text{Const-Conj.I}\ \frac{\Gamma \vdash e_1 : \{v : B \mid v : \tau_1\} \quad \Gamma \vdash e_2 : \{v : B \mid v = 0 \wedge v : \text{CST}\}}{\Gamma \vdash e_1 \wedge e_2 : \{v : B \mid v = 0 \wedge v : \text{CST}\}} \qquad \text{Const-Conj.II}\ \frac{\Gamma \vdash e_1 : \{v : B \mid v : \tau_1\} \quad \Gamma \vdash e_2 : \{v : B \mid v = 1 \wedge v : \text{CST}\}}{\Gamma \vdash e_1 \wedge e_2 : \{v : B \mid v : \tau_1\}}$$

$$\text{Const-Disj.I}\ \frac{\Gamma \vdash e_1 : \{v : B \mid v : \tau_1\} \quad \Gamma \vdash e_2 : \{v : B \mid v = 0 \wedge v : \text{CST}\}}{\Gamma \vdash e_1 \vee e_2 : \{v : B \mid v : \tau_1\}} \qquad \text{Const-Disj.II}\ \frac{\Gamma \vdash e_1 : \{v : B \mid \tau_1\} \quad \Gamma \vdash e_2 : \{v : B \mid v = 1 \wedge v : \tau_2\}}{\Gamma \vdash e_1 \vee e_2 : \{v : B \mid v = 1 \wedge v : \text{CST}\}}$$

$$\text{XOR.III}\ \frac{\Gamma \vdash e_1 : \{v : B \mid v : \text{CST}\} \quad \Gamma \vdash e_2 : \{v : B \mid v : \text{CST}\} \quad e_1 \neq e_2}{\Gamma \vdash e_1 \oplus e_2 : \{v : B \mid v = 1 \wedge v : \text{CST}\}} \qquad \text{XOR.IV}\ \frac{\Gamma \vdash e : \{v : B \mid v : \text{CST}\}}{\Gamma \vdash e \oplus e : \{v : B \mid v = 0 \wedge v : \text{CST}\}}$$

$$\text{Neg.II}\ \frac{\Gamma \vdash e : \{v : B \mid v = 0 \wedge v : \text{CST}\}}{\Gamma \vdash \neg e : \{v : B \mid v = 1 \wedge v : \text{CST}\}} \qquad \text{Neg.III}\ \frac{\Gamma \vdash e : \{v : B \mid v = 1 \wedge v : \text{CST}\}}{\Gamma \vdash \neg e : \{v : B \mid v = 0 \wedge v : \text{CST}\}}$$

Figure 12: One-bitvector Constant Type Rules.

$$\text{Assign-I}\ \frac{\Gamma \vdash x : \{v : B \mid v : \tau_x\} \quad \Gamma \vdash e : \{v : B \mid v = b \wedge v : \tau_e\} \quad \Gamma' = \Gamma[x \mapsto \{v : B \mid v = b \wedge v : \tau_e\}]}{\Gamma \vdash x \leftarrow e \dashv \Gamma'}$$

$$\text{Assign-II}\ \frac{\Gamma \vdash x : \{v : B \mid v : \tau_x\} \quad \Gamma \vdash e : \{v : B \mid v : \tau_e\} \quad \Gamma' = \Gamma[x \mapsto \{v : B \mid v : \tau_e\}]}{\Gamma \vdash x \leftarrow e \dashv \Gamma'}$$

$$\text{Load-I}\ \frac{\Gamma \vdash x : \{v : B \mid v : \tau_x\} \quad \Gamma \vdash e_1[e_2] : \{v : B \mid v = b \wedge v : \tau_v\} \quad \Gamma' = \Gamma[x \mapsto \{v : B \mid v = b \wedge v : \tau_v\}]}{\Gamma \vdash x \leftarrow e_1[e_2] \dashv \Gamma'}$$

$$\text{Load-II}\ \frac{\Gamma \vdash x : \{v : B \mid v : \tau_x\} \quad \Gamma \vdash e_1[e_2] : \{v : B \mid v : \tau_v\} \quad \Gamma' = \Gamma[x \mapsto \{v : B \mid v : \tau_v\}]}{\Gamma \vdash x \leftarrow e_1[e_2] \dashv \Gamma'}$$

$$\text{Load-III}\ \frac{\Gamma \vdash x : \{v : B \mid v : \tau_x\} \quad \Gamma \vdash e_1 : \{v : \text{Vec}\langle n \rangle \mid v : \tau_1\} \quad \Gamma \vdash e_2 : \{v : \text{Vec}\langle n \rangle \mid v : \tau_2\} \quad \Gamma \nvdash e_1[e_2] \quad \tau_r = \tau_1 \sqcup \tau_2 \quad \Gamma' = \Gamma[x \mapsto \{v : B \mid v : \tau_r\}]}{\Gamma \vdash x \leftarrow e_1[e_2] \dashv \Gamma'}$$

$$\text{Store-I}\ \frac{\Gamma \vdash e_1 : \{v : \text{Vec}\langle n \rangle \mid v : \tau_1\} \quad \Gamma \vdash e_2 : \{v : \text{Vec}\langle n \rangle \mid v : \tau_2\} \quad \Gamma \vdash x : \{v : B \mid v = b \wedge v : \tau_x\} \quad \Gamma' = \Gamma[e_1[e_2] : \{v : B \mid v = b \wedge v : \tau_x\}]}{\Gamma \vdash e_1[e_2] \leftarrow x \dashv \Gamma'}$$

$$\text{Store-II}\ \frac{\Gamma \vdash e_1 : \{v : \text{Vec}\langle n \rangle \mid v : \tau_1\} \quad \Gamma \vdash e_2 : \{v : \text{Vec}\langle n \rangle \mid v : \tau_2\} \quad \Gamma \vdash x : \{v : B \mid v : \tau_x\} \quad \Gamma' = \Gamma[e_1[e_2] : \{v : B \mid v : \tau_x\}]}{\Gamma \vdash e_1[e_2] \leftarrow x \dashv \Gamma'}$$

$$\text{Seq}\ \frac{\Gamma \vdash s_1 \dashv \Gamma'' \quad \Gamma'' \vdash s_2 \dashv \Gamma'}{\Gamma \vdash s_1; s_2 \dashv \Gamma'}$$

Figure 13: Type Rules for Statements.

C Known Information Leaks

RSA/Elgamal-Libgcrypt. Fig. 14 demonstrates the sliding-window implementation. Before the sliding-window algorithm, two lookup tables are constructed. The first table stores the modular exponentiation values of various bases and the second one stores the length of the corresponding value. In the main loop of modular

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

Figure 14: RSA/Elgamal information leaks found in Libgcrypt-1.6.1.

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

Figure 15: RSA information leaks found in OpenSSL-1.0.2f.

exponentiation, symbol `e` represents the element of secret array and `e0` represents each sliding-window of `e`. Then `e0` is used to access the pre-computation tables. Intuitively, with a PRIME-PROBE attack, different cache sets are observed accessed under different sliding-window values, eventually leaking the secret `e`.

RSA-OpenSSL. The BN_num_bits_word is called by BN_num_bits that counts the number of bits of a secret (see Fig. 15). The secret is stored in a BIGNUM struct, where the key value is stored in a byte array `a->d` of 32-bit element, and the length of the array is stored in `a->top`. The number of bits of the last element requires determined separately because its valid bits may be less than 32 bits. Therefore, function BN_num_bits_word refers to a lookup table to determine the number of bits of the last element of the secret array. Different last elements lead to different entries of the lookup table being accessed. Meanwhile, the branches further narrow down the secret length. Thus, it is a combined vulnerability of memory access and branch.

ECDSA-OpenSSL. ECDSA performs the second step of signature as $s = (r \cdot priv\_key + m)\ mod\ order$ (see Fig. 16), where $r \cdot priv\_key$ represents the result of the first step that multiplies part of the signature r with the private key $priv\_key$. Before the addition operation of the second step, the value of $r \cdot priv\_key$ ensures to be reduced into the range [0, $order$-1]. By observing whether a reduction behaves after the addition operation inside the second step (function BN_mod_add_quick), an attacker can deduce the range information of the private key $priv\_key$.

```
1 ECDSA_SIG *ossl_ecdsa_sign_sig(...){
2   ...
3   do{
4     ...
5     if (!BN_mod_mul(tmp, priv_key, ret->r, order, ctx)){
6       ECerr(EC_F_OSSL_ECDSA_SIGN_SIG, ERR_R_BN_LIB);
7       goto err;
8     }
9     if (!BN_mod_add_quick(s, tmp, m, order)){
10       ECerr(EC_F_OSSL_ECDSA_SIGN_SIG, ERR_R_BN_LIB);
11       goto err;
12     }
13     ...
14   }
15   while(1);
16   ...
17 }
18 int BN_mod_add_quick(BIGNUM *r,
19   const BIGNUM *a, const BIGNUM *b,
20   const BIGNUM *m){
21   if (!BN_uadd(r, a, b))
22     return 0;
23   if (!BN_ucmp(r, m) >= 0)
24     return BN_usub(r, r, m);
25   return 1;
26 }
```

Figure 16: ECDSA information leaks found in OpenSSL-1.1.0g.

D Unknown Information Leaks in OpenSSL

```
1 int BN_rshift1(BIGNUM *r, const BIGNUM *a){
2   ...
3   i = a->top;
4   ap = a->d;
5   ...
6   rp = r->d;
7   t = ap[--i];
8   c = (t&1)? BN_TBIT : 0;
9   if (t >>= 1)
10     rp[i] = t;
11   while (i > 0){
12     t = ap[--i];
13     rp[i] = ((t >> 1)&BN_MASK2) | c;
14     c = (t&1)? BN_TBIT : 0;
15   }
16   ...
17 }
```

Figure 17: BN_rshift1 information leaks found in OpenSSL-1.1.0g.

```
1 int bn_mul_normal(BN_ULONG *r,
2   BN_ULONG *a, int na, BN_ULONG *b, int nb){
3   ...
4   for(;;){
5     if (--nb <= 0) return;
6     rr[1] = bn_mul_add_words(&(r[1]), a, na, b[1]);
7     if (--nb <= 0) return;
8     rr[2] = bn_mul_add_words(&(r[2]), a, na, b[2]);
9     if (--nb <= 0) return;
10     rr[3] = bn_mul_add_words(&(r[3]), a, na, b[3]);
11     if (--nb <= 0) return;
12     rr[4] = bn_mul_add_words(&(r[4]), a, na, b[4]);
13     ...
14   }
15 }
```

Figure 18: bn_mul_normal information leaks found in OpenSSL-1.1.0i.