SCOUT-CT: Sound Constant-Time Outcome with Uncertainty Tracking using multi-taint analysis

Damien Maier*†, Jean-François Pasche*, Maxim Golay*†, Alexandre Duc*
* *School of Engineering and Management Vaud*
HES-SO University of Applied Sciences and Arts Western Switzerland
Yverdon-les-Bains, Switzerland
alexandre.duc@heig-vd.ch
† *School of Computer and Communication Sciences*
EPFL
Lausanne, Switzerland
damien.maier@epfl.ch, maxim.golay@epfl.ch

***Abstract*—Side-channel attacks are an important class of security exploits, in which an attacker gains access to confidential data by observing information inadvertently leaked by a system. Writing constant-time code is a common defense against time-based and microarchitectural side-channel attacks.**

Many approaches have been proposed to automatically verify that a program is constant-time. Sound methods can detect all information leaks but, to efficiently analyze large programs, most of them rely on overapproximation which can yield false alarms (i.e., reports of non-existent information leaks). Each finding produced by such analyses therefore requires manual inspection. Additionally, most existing approaches do not perform binary-level analysis and thus miss vulnerabilities introduced by compilation.

In this paper, we present a novel sound analysis for detecting information leaks under the constant-time threat model. Compared with existing work, our technique improves taint analysis by systematically tracking precision loss to determine whether a detected information leak could be caused by overapproximation. Findings for which no precision loss is detected are reported as *confirmed*; as long as they do not arise from dead code, confirmed findings are guaranteed to be true and thus do not require significant manual inspection. Only findings with detected precision loss need classical human verification. Our analysis operates directly on binary executables. We instantiate our technique within the abstract interpretation framework and provide a proof of correctness.

We implemented our approach in a prototype tool, SCOUT-CT, and evaluated it on a benchmark of constant-time and non-constant-time programs, including real-world cryptographic implementations. Our results show that SCOUT-CT is effective: our tool detected all 98 timing leaks in the benchmark and automatically classified 97 as confirmed findings that do not require significant manual inspection.

***Index Terms*—timing attacks, constant-time verification, abstract interpretation, binary analysis**

Note (added 10th July 2026). This revision corrects an error in our evaluation of Binsec/Rel (Tables 2 and 3, Section 8). The original evaluation used a misconfigured Binsec/Rel setup, identified by the Binsec authors during an independent replication attempt. With a corrected configuration, Binsec/Rel achieves exhaustive analysis on all real-world benchmarks in our suite, with substantially faster runtime than previously reported. We have corrected Tables 2 and 3, and revised our comparative claims in Sections 1, 8, and 9 accordingly. The core technical contribution of this paper—a multi-taint system for classifying constant-time violations as proven versus requiring manual inspection—is unaffected by this correction. We thank the Binsec team for identifying this issue.

1. Introduction

Computer systems can leak secret information through subtle and unintended channels. The data processed by a system can influence various externally visible behaviors such as its execution time or its memory access patterns. These so-called *side-channels* can be monitored by an attacker to gain knowledge about operations performed by the system, and ultimately about the secret data it manipulates. Ensuring that a system does not leak information through side-channels is an inherently difficult task.

1.1. Timing attacks

Timing attacks are a particularly prominent class of side-channel attacks, where the attacker infers secret information through timing measurements. Numerous timing attacks have been demonstrated in

This work was partly supported by armasuisse Science and Technology, Cyber-Defense Campus, Switzerland and partly supported by the IICT institute of HEIG-VD/HES-SO, Switzerland.

```
1 r ← 1
2 for b ← bits of e from MSB to LSB do:
3   r ← (r · r) mod n
4   if b = 1 then:
5     r ← (r · a) mod n
```

Listing 1. Square-and-multiply algorithm. Computes $r = a^e \bmod n$.

practice against real-world software. Timing attacks can be classified into two different — but related — categories, depending on whether the attacker (i) measures the execution time of the victim system, or (ii) measures the execution time of *its own* operations running on hardware shared with the victim system [1].

Attacks belonging to the first category target programs whose execution time depends on secret data. A well-known example is the timing attack presented in 1996 by Kocher [2], which targets modular exponentiation implementations. As an illustration, Listing 1 shows a simple implementation of modular exponentiation that is vulnerable to such an attack: at line 4, the `if` statement is only entered when the exponent bit is 1, causing the duration of a loop iteration to depend on the (typically secret) exponent. Various effective attacks leveraging this technique remotely against real-world software were demonstrated [3], [4].

Other attacks exploit the fact that the execution time of some instructions depends on the value of their inputs. A well-known example is the KyberSlash Attack [5] by Bernstein et al.

The second category of attacks exploits the fact that two programs that use the same resources concurrently influence each other's execution time. By measuring timing variations of its own program, an attacker can gain knowledge about the usage of shared resources by a victim program, which reveals information about the computations performed by the victim, and ultimately about secret data it manipulates. This type of attack is particularly relevant in multi-tenant environments such as cloud computing platforms, where an attacker can easily run its own code on hardware shared with a victim program [6].

Prominent examples of such attacks are *cache attacks* [7], [8], where the attacker uses timing measurements to probe the state of a cache shared with the victim program to learn about its memory access patterns. A well-known instance of this attack targets implementations of cryptographic algorithms such as AES, which use substitution boxes stored in memory [9]. Listing 2 presents a typical vulnerable substitution box implementation. At line 6, the address accessed in the `sbox` array depends on the value of `input`; this address can be inferred by an attacker monitoring the cache state.

```
1 char sbox[256] = {
2   /* ... */
3 };
4
5 char sub_byte(char input) {
6   return sbox[input];
7 }
```

Listing 2. S-box implementation.

Although the attack illustrated above targets explicit data memory accesses (i.e., memory accesses caused by load instructions in the victim program), other cache attacks monitor memory accesses to the code segment of the victim program itself, and thus enable tracing of execution paths [7].

1.2. Automated constant-time verification

To protect their software against timing attacks, developers strive to ensure that secret data does not influence attacker-observable behaviors. The dominant technique to achieve this goal is constant-time (CT) programming [10]–[19]. A constant-time program is one where secret data influences neither

- the condition of any jump instruction, nor
- the address of any memory access.

Such programs produce execution paths and memory access patterns that are independent of secret data, and are therefore immune to the different types of timing attacks described above. Some authors [11], [14], [20] extend this threat-model to also require that secret data does not influence the inputs of variable-time instructions.

Writing constant-time programs is however a notoriously difficult task [20], and therefore having a tool that automatically detects CT violations, or proves their absence, is desirable.

Many approaches have been proposed for automated constant-time verification; these approaches differ in the reliability guarantees they offer and their ability to scale to real-world software. We discuss the various techniques used by automatic CT verification tools in Section 3.

Key characteristics of CT analyses are *soundness* and *completeness*.

- Sound approaches reason about *all* possible executions of the tested program, and thus guarantee that no actual information leak goes undetected (no false negatives).
- Complete approaches only take into account executions that can actually happen, and thus guarantee that all reported information leaks are actual leaks (no false positives).

Those two properties are often mutually exclusive: due to undecidability, no terminating automated tool that verifies programs of arbitrary size can be both sound and complete [21]. Consequently, existing CT analysis tools can be classified into three categories:

1) tools that trade soundness for scalability (i.e., perform underapproximation) and thus may miss existing leaks [15]–[19], [22]–[29];
2) tools that trade completeness for scalability (i.e., perform overapproximation) and thus may report non-existing leaks [11]–[13], [30]–[33];

3) tools that are both sound and complete, but are inherently more computationally expensive and thus have a lower ability to scale to real-world software [10], [14], [34].

Approaches belonging to the first category can be useful to discover information leaks, but they cannot guarantee detection of all of them. Developers who want to ensure that all leaks are discovered are left with no satisfactory solution: they must either resort to tools from the third category, which suffer from scalability issues, or to tools from the second category, which require manual inspection of *all* reported findings to discard false positives.

Another important property of tools for automated CT verification is the ability to analyze programs provided as compiled binaries. Analyzing binaries is typically challenging, especially for static approaches, and thus many existing tools [11]–[14], [30]–[32] operate on intermediate representations (IRs). However, it has been shown that timing leaks can be introduced during compilation [10], so analyzing binaries instead of IRs is necessary to ensure that all CT violations are found.

1.3. Our contribution

In this paper, we propose a novel approach for automated constant-time verification that improves the state of the art with respect to these issues.

We present a binary-level analysis that is sound (cannot miss violations) and scalable through overapproximation, but whose key novelty is that it provides guarantees that reduce the manual effort required to discard false positives.

Our analysis automatically classifies findings into one of two categories:

- *Potential* CT violations that can be spurious alarms and require manual verification.
- *Confirmed* CT violations that can only be spurious alarms if they are located in unreachable code (dead code). As long as there exists at least one feasible execution reaching the program point associated with a confirmed violation, the violation is guaranteed to be a true positive.

Consequently, contrary to existing overapproximation-based tools that require manual verification of all findings, the user of our tool only needs to manually verify the findings classified as potential. Confirmed findings simply require checking that they are not located in dead code, which is a much easier task.

Our approach builds on top of the "static taint analysis" idea used in VirtualCert [12], CacheS [33], and the work of Blazy et al. [13]: we follow the control-flow graph of the program, propagating a taint whose source is the secret input across assignments and operations; tainted jump conditions and memory accesses are reported as CT violations. We extend this approach with the following features:

- We use a *multi-taint system* that enables overapproximation tracking for distinguishing confirmed findings from potential ones. Initially, the analysis propagates a *red* taint whose source is the secret input. Red-tainted data is known to be secret-dependent. When overapproximation is performed, data in the resulting state is marked with an *orange* taint, to denote that it may be secret-dependent. CT violations caused by red-tainted data are confirmed leaks, while those caused by orange-tainted data are potential leaks.
- We use a *path-sensitive* analysis, that is, for each program point we maintain separate state information for each feasible execution path reaching that point. This enables precise analysis of loops, where each iteration is analyzed separately. For scalability, we limit the number of paths tracked to a user-defined threshold and compute a sound overapproximation of possible states when this threshold is exceeded.
- We track the concrete values of input-independent data during the analysis, by simulating instruction executions on them. This enables precise handling of indirect memory accesses.

We implement our approach in a prototype tool named SCOUT-CT, and make it publicly available.

We evaluate SCOUT-CT on a benchmark suite composed of various CT and non-CT programs, including real-world cryptographic implementations. Our results confirm that our approach effectively scales to real-world implementations. SCOUT-CT successfully analyzes all programs of our benchmark in a reasonable time, reports all known leaks, and classifies almost all of them as confirmed CT violations.

We compare the performance of SCOUT-CT against two state-of-the-art tools: Microwalk [16] (complete, not sound) and Binsec/Rel [10], [34] (sound and complete). The results confirm the usefulness of the new trade-off we propose: while our tool offers weaker guarantees regarding false positives, it scales soundly to large programs, with execution times comparable to Microwalk.

Overall, our contributions are the following:

- We propose a novel approach for automated constant-time verification that is sound, binary-level, path-sensitive, and tracks overapproximation to reduce the manual verification effort.
- We instantiate our approach in the abstract interpretation framework, and prove its correctness.
- We implement our approach in a prototype tool named SCOUT-CT, and make it publicly available.
- We evaluate SCOUT-CT on a benchmark suite, demonstrating the effectiveness of our approach on real-world implementations, and we compare its performance against two state-of-the-art tools.

Roadmap. In Section 2, we introduce static taint analysis and abstract interpretation. We discuss related work in Section 3. We motivate our work in

Section 4 using meaningful examples. We formalize our analysis in Section 5 and prove its correctness in Section 6. We present the implementation of SCOUT-CT in Section 7. We evaluate our tool on a benchmark and compare it to existing work in Section 8. Finally, we conclude in Section 9.

2. Background

In this section we present abstract interpretation, the mathematical framework in which our analysis is instantiated. We then introduce static taint analysis, the core approach used in existing abstract-interpretation-based tools for the automatic detection of CT violations.

2.1. Abstract interpretation

Abstract interpretation is a framework for constructing static analyses, introduced by Cousot and Cousot [35], [36]. Its key idea is to reason not about all concrete program states explicitly (which is typically not feasible), but about a simpler overapproximation that can be computed effectively.

A common goal of abstract interpretation is to overapproximate the *collecting semantics* that computes all reachable program states at each program point.

Formally, define a *concrete state* as a mapping from each program variable to some property of interest (for instance, the concrete value of the variable). Let $\Sigma^\flat$ denote the set of concrete states. A *concrete states assignment* is a mapping $\mu^\flat : \mathsf{points} \to \mathcal{P}(\Sigma^\flat)$ from each program point to a set of program states.

The *concrete transformer* for the collecting semantics is the function $\mathcal{F}^\flat$ that receives a concrete states assignment $\mu_1^\flat$, simulates one execution step of the program for each state, and returns the updated concrete states assignment $\mu_2^\flat$. Repeatedly applying the concrete transformer on an initial states assignment $\mathcal{F}^\flat(\mathcal{F}^\flat(\dots(\mu^\flat)\dots))$ until no new states are discovered is equivalent to performing an exhaustive exploration of all program executions. That is, the set of reachable states is the *least fixed point* of $\mathcal{F}^\flat$.

As explicitly computing the least fixed point of $\mathcal{F}^\flat$ is typically intractable, abstract interpretation consists instead in performing the analysis on a simplified representation of program states, called an *abstract domain*. Each abstract state in the abstract domain represents a set of concrete states. The analysis is performed on a new abstract semantics that operates on abstract states.

Overall, the key components of an abstract interpretation are:

- a set $\Sigma^\sharp$ of possible abstract states,
- a concretization function $\gamma : \Sigma^\sharp \to \mathcal{P}(\Sigma^\flat)$ that determines the set of concrete states represented by an abstract state,
- an abstraction function $\alpha : \mathcal{P}(\Sigma^\flat) \to \Sigma^\sharp$ that maps a set of concrete states to the minimal abstract state representing them,
- an abstract transformer $\mathcal{F}^\sharp$ that receives a collection of abstract states and simulates one execution step for each of them, according to the abstract semantics.

If the abstract domain satisfies some mathematical conditions (namely, α and γ form a Galois connection, $\mathcal{F}^\sharp$ is a sound approximation of $\mathcal{F}^\flat$), then the least fixed point of $\mathcal{F}^\sharp$ overapproximates the least fixed point of $\mathcal{F}^\flat$ (see, for instance Propositions 24 and 25 of [36]).

Concretely, computing the least fixed point of $\mathcal{F}^\sharp$ and applying γ to the result yields a superset of all reachable program states. The least fixed point of $\mathcal{F}^\sharp$ can be computed using Kildall's worklist algorithm [37].

We will define the concrete domain used for our analysis in Section 5.2, its abstract domain in Section 5.3, and the abstraction and concretization functions in Section 6.

2.2. Static taint analysis for CT

We now present a minimal abstract-interpretation-based analysis for finding CT violations. Existing tools based on abstract interpretation like CacheS [33], VirtualCert [12] and the work of Blazy et al. [13] all follow this fundamental approach.

For this analysis, a concrete state maps each variable to:

- a concrete value, and
- whether the variable is secret-dependent or not.

The concrete semantics follows an intuitive definition: concrete values are computed according to the meaning of program instructions, and a variable becomes secret-dependent if it is assigned the result of an operation that itself depends on a secret-dependent variable.

Note that this notion of secret-dependency is syntactic and not based on whether the final value actually varies with the secret. A value computed from secret-dependent data is always treated as secret-dependent even when it is constant; for example, if a is secret-dependent, then $b = a \oplus a$ is always 0, but we still consider b to be secret-dependent.

The goal of the analysis is to overapproximate the secret-dependency property of the variables. To this end, the analysis propagates a taint whose source is the secret program input.

An abstract state maps each variable to either *tainted* or *untainted*. An untainted variable represents a non-secret-dependent variable, while a tainted variable represents a variable that may be secret-dependent. At the beginning of the analysis, all variables are untainted, except for the secret input variable in the entry point.

The abstract semantics of non-jump operations is straightforward: if any operand is tainted, the result is tainted; otherwise, it is untainted. The abstract execution of a conditional jump simply consists of propagating the abstract state to both successors. When a program point can be reached through multiple paths

(i.e., has multiple predecessors), the abstract states from each incoming path are soundly merged: if a variable is tainted in any of the incoming abstract states, it is tainted in the merged abstract state.

As an example, consider the program depicted in Listing 3. The variable `secret` denotes a secret input of the program, while `public` is a public input. Both memory accesses at lines 7 and 8 are CT violations, as the address being accessed depends on the secret value, at least for some values of `public`.

```
1 var1 = secret;
2 if (public) {
3   var2 = 0;
4 } else {
5   var2 = secret;
6 }
7 /* ... */ = array[var1];
8 /* ... */ = array[var2];
```

Listing 3. Tested program example.

The result of performing the static taint analysis on this program is represented in Listing 4. At line 1, `var1` becomes tainted. Similarly, at line 5, `var2` becomes tainted. At line 6, the two branches of the if-else merge. As `var1` is tainted in both the then and else branches, it naturally remains tainted after the merge. Regarding `var2`, it is untainted in the then branch but tainted in the else branch; to ensure a sound overapproximation, it is marked as tainted after the merge.

```
1 var1 = secret;
2 if (public) {
3   var2 = 0;
4 } else {
5   var2 = secret;
6 }
7 /* ... */ = array[var1];
8 /* ... */ = array[var2];
```

Listing 4. Simple taint analysis result. Red, underlined names denote tainted variables.

The analysis finally reports all tainted memory accesses or jump conditions as CT violations. In this example, both memory accesses at lines 7 and 8 are reported.

3. Related work

Previous work [20], [38], [39] lists and compares tools for automated constant-time verification.

Existing tools vary in the threat model they consider. We first discuss those threat models and then specifically compare analysis techniques for the constant-time threat model.

3.1. Threat models

Some tools [40]–[42] simply follow a black-box approach that consists in measuring the execution time of many runs of the tested program and using statistical methods to determine whether those measurements correlate with the secret data. Others [43]–[48] ensure that whenever a conditional jump depends on secret data, both branches result in the same execution times. Those approaches do not account for attack scenarios presented in Section 1.1 where the attacker measures its own execution time on shared hardware.

Most tools [10]–[19] use the constant-time threat model defined in Section 1.2 that forbids secret-dependent jump conditions and memory accesses.

Some approaches specifically focus on cache attacks [22], [25], [32], [33] and use various relaxed versions of the constant-time threat model that allow some secret-dependent memory accesses which are believed to be safe from cache attacks. For instance, as memory accesses targeting the same cache line typically have the same effect on the cache state, those tools may allow secret-dependent memory accesses as long as only the lower bits of the accessed address depend on secret data.

Finally, some tools [49]–[53] extend the constant-time threat model to speculative execution.

3.2. Approaches for the constant-time threat model

We focus our discussion on tools that forbid secret-dependent jump conditions or memory accesses, that is, tools for the constant-time threat model or its variants that only target cache attacks.

The common problem all those tools aim to solve is the automatic detection of information flows from a source — the secret data — to sinks — the attacker-observable behaviors. We classify existing approaches based on how they tackle this problem. We briefly describe these approaches below. The results of our classification are presented in Table 1.

Static taint analysis. Those tools use the technique presented in Section 2.2.

Off-the-shelf verifier. This approach consists in building a product-program whose correctness is equivalent to the constant-time property of the tested program, and using an off-the-shelf software verifier to prove the correctness of the product program.

Symbolic execution. This approach consists in executing the program on symbolic inputs, building symbolic expressions to represent results of input-dependent computations, and systematically exploring execution paths. Whenever a symbolic expression e reaches an information sink, the tool solves an SMT formula to check whether e can take different values depending on secret inputs.

Dynamic taint analysis. This approach consists in performing real executions of the tested program while propagating a taint whose source is the secret data.

Comparison of execution traces. This approach consists in performing real executions of the tested program on different secret inputs, producing execution traces capturing the attacker-observable behaviors, and comparing those traces to find evidence of information leaks.

TABLE 1. CONSTANT-TIME VERIFICATION APPROACHES.

Ref.	Tool	Input
Static taint analysis		
[12]	VirtualCert	Mach[a]
[13]	Blazy et al.	C#minor[b]
[33]	CacheS	Binary
Off-the-shelf verifier		
[14]	ct-verif	LLVM
Symbolic execution		
[32]	CaSym	LLVM
[11]	Pitchfork	LLVM
[10]	Binsec/rel	Binary
Dynamic taint analysis		
[18]	ctgrind	Binary
[22]	CacheD	Binary
[24]	TimeCop	Binary
[19]	Abacus	Binary
[25]	Catype	Binary
[26]	MemSan	Binary
Comparison of execution traces		
[27]	Stacco	Binary
[15]	DATA	Binary
[16]	Microwalk	Binary
[28]	Anableps	Binary
[17]	ct-fuzz	Binary
[29]	CacheQL	Binary

[a] Last IR used by CompCert [54].
[b] Second IR used by CompCert [54].

4. Motivation

This section presents a high-level, intuitive description of the novel aspects of our analysis.

We proceed incrementally, building on top of the minimal taint analysis approach presented in Section 2.2. First, we introduce our multi-taint system and show how it enables overapproximation tracking. Then, we add the second key feature of our approach: path sensitivity.

Note that, although for the sake of clarity we present our approach on high-level pseudo-code, our analysis is performed on compiled code. What are referred to as "variables" in this section actually are registers and memory bytes in the real analysis.

4.1. Multi-Taint System for Overapproximation Tracking

The main downside of the classic taint analysis presented in Section 2.2 is that it may produce false positives due to overapproximation. Consequently, all reported findings require manual inspection.

To address this problem, we augment the analysis with a double-taint system that enables precision loss tracking. In this new abstract domain, tainted variables are either associated with a *red taint* or an *orange taint*. The source of the red taint is the secret input; this taint represents data that is guaranteed to be secret-dependent. When state merges are performed, the *orange taint* is used to represent variables that may depend on secret data due to overapproximation.

Listing 5 illustrates the results of this enhanced analysis on the program of Listing 3. Variables `var1` and `var2` become red-tainted at lines 1 and 5, respectively. At line 6, during the merge of the two branches, `var1` remains red-tainted, as it is red-tainted in both branches and thus guaranteed to depend on secret data. However, `var2` is red-tainted in the else branch but untainted in the then branch; consequently, it is marked as orange-tainted after the merge, indicating that it may depend on secret data due to overapproximation.

At line 7, the memory access is flagged as a *confirmed* CT violation, since the address being accessed is red-tainted. At line 8, however, the memory access is reported as a *potential* CT violation, as the address is only orange-tainted. Only the second violation requires manual inspection, thereby reducing the analyst's workload.

```
1 var1 = secret;
2 if (public) {
3   var2 = 0;
4 } else {
5   var2 = secret;
6 }
7 /* ... */ = array[var1];
8 /* ... */ = array[var2];
```

Listing 5. Multi-taint analysis result. Red, underlined names denote red-tainted variables. Orange, dotted-underlined names denote orange-tainted variables.

4.2. Path Sensitivity

The second key feature of our analysis is path sensitivity.

As a motivational example, consider Listing 6, which represents the result of an analysis with the multi-taint system but *without* path sensitivity. When the body of the loop is analyzed for the first time, `array1[0]` becomes red-tainted at line 2. However, the analysis then goes back to line 1 and merges the status coming from the first iteration with the initial status before the loop. As `array1[0]` is untainted before the loop and red-tainted after the first iteration, it becomes orange-tainted after the merge.

Consequently, the memory access at line 4 is reported as only a potential CT violation, even though it is an actual violation.

```
1 for(i = 0; i < 3; i++) {
2   array1[i] = secret;
3 }
4 /* ... */ = array2[array1[0]];
```

Listing 6. Multi-taint analysis result, without path sensitivity.

To improve precision, we introduce two enhancements to the analysis.

- First, instead of mapping each program point to a single abstract state, the abstract domain now maps each program point to a *set* of abstract states. When a program point has multiple predecessors, the abstract semantics associates to it the union of states coming from each predecessor. Only when the number of states

associated to a program point exceeds a user-defined threshold are states soundly merged.
- Second, the analysis keeps track of concrete values of variables that do not depend on the program inputs. An abstract state now maps each variable to either the red taint (secret dependent, unknown value), the orange taint (possibly secret dependent, unknown value), the new green taint (not secret dependent, unknown value), or a concrete value (not secret dependent, known value).

Tracking the concrete values enables correct handling of indirect memory accesses. It also improves precision for conditional jumps: when the condition evaluates to a concrete value, the abstract state is only propagated to the taken branch; when it evaluates to a taint, both branches are taken as in the non-path-sensitive analysis.

Capping the number of abstract states per program point prevents an exponential blow-up of the analysis cost and thereby ensures its scalability, at the inevitable cost of some precision loss.

With path sensitivity, analyzing the program of Listing 7 results in line 3 (end of the loop body, after increment of `i`) being associated with three abstract states:

- One where `i` is 1, and `array1[0]` is red-tainted.
- One where `i` is 2, and `array1[0]` and `array1[1]` are red-tainted.
- One where `i` is 3, and `array1[0]`, `array1[1]`, and `array1[2]` are red-tainted.

Finally, the last state is propagated to line 4, providing a precise description of the concrete state at this point, and a confirmed CT violation is reported.

```
1 for(i = 0; i < 3; i++) {
2   array1[i] = secret;
3 }
4 /* ... */ = array2[array1[0]];
```

Listing 7. Multi-taint analysis result, with path sensitivity.

4.3. Confirmed and potential leaks with path sensitivity

Whenever an attacker-observable (condition of a jump instruction, address of a memory access, or operand of a variable-time instruction) is red- or orange-tainted, the analysis reports a CT violation. Additionally, when such an observable is red-tainted in *all* abstract states associated with the program point, the violation is reported as *confirmed*.

CT violations reported by our analysis are guaranteed to be a superset of all actual leaks in the program. Moreover, when a CT violation is reported as confirmed it is guaranteed to be an actual leak, *assuming that there exists at least one feasible execution reaching the violation.*

The complete proof is given in Section 6.3. Intuitively, the argument goes as follows:

1) The analysis produces a sound overapproximation of secret-dependency, and thus if a variable is secret-dependent in the concrete domain, it is tainted (red or orange) in at least one abstract state.
2) If a variable is red-tainted in all abstract states associated with a program point, then it is secret-dependent in all concrete states represented by those abstract states. As the abstract states overapproximate the concrete states, either there exists no concrete state for this program point, or the variable is secret-dependent in all concrete states.

5. Formalization

In this section, we formalize our algorithm. Section 5.1 introduces the program model. Section 5.2 presents the concrete domain; this part formalizes the conventional secret-dependency notion introduced in Section 2.2 and is inspired from previous work [13]. The core novelty is presented in Section 5.3, where we define the abstract domain of our analysis.

5.1. Notation and program model

This section formalizes the program representation used to define our analysis. Our goal here is to describe a collection of mathematical objects for modelling the code (syntax) of any program. Our model is general enough to represent programs written in any assembly language. We also introduce notation that will be used in the next sections.

5.1.1. Miscellaneous notation. For some set S, we use $\mathcal{P}(S)$ to denote the power set of S. We use $\mathbb{B} \triangleq \{0, \ldots, 255\}$ to denote the set of possible values that can be stored in a byte.

5.1.2. Program states. We use $\mathcal{B}$ to denote the set of *architectural bytes* of the machine being modeled. The set $\mathcal{B}$ contains any memory unit whose value can be read or written by a program instruction. In particular, it includes each byte of the main memory and of each register.

For some set V of possible values, a *state* $\sigma : \mathcal{B} \to V$ maps each architectural byte to an element of V. The set V will be instantiated in the next sections, when defining concrete and abstract states.

Consider some architectural byte $b \in \mathcal{B}$, some tuple of architectural bytes $B \in \mathcal{B}^n$, and some value $v \in V$. We use

- $\sigma[b]$ to denote the element of V associated to architectural byte b by state σ;
- $\sigma[B]$ to denote the tuple $(\sigma[b_1], \ldots, \sigma[b_n])$ where $B = (b_1, \ldots, b_n)$;
- $\sigma[b \mapsto v]$ to denote the state identical to σ except that b is now mapped to v.

5.1.3. Instructions. We use $\mathcal{I}$ to denote the set of all *instructions* that can appear in a program. An element of $\mathcal{I}$ is one of the following forms:

- $\langle \mathsf{compute}, f, I, o \rangle$ where $f : \mathbb{B}^n \to \mathbb{B}$ performs some computation, $I \in \mathcal{B}^n$ are the input bytes, and $o \in \mathcal{B}$ is the output byte.

- $\langle \mathsf{load}, f, S, d \rangle$ where $f : \mathbb{B}^n \to \mathcal{B}$ describes how to compute the source address, $S \in \mathcal{B}^n$ are the bytes involved in computing the source address, and $d \in \mathcal{B}$ is the destination byte.
- $\langle \mathsf{store}, f, D, s \rangle$ where $f : \mathbb{B}^n \to \mathcal{B}$ describes how to compute the destination address, $D \in \mathcal{B}^n$ are the bytes involved in computing the destination address, and $s \in \mathcal{B}$ is the source byte.
- $\langle \mathsf{branch}, g, I, T, F \rangle$ where $g : \mathbb{B}^n \to \{\text{true}, \text{false}\}$ is the branch condition, $I \in \mathcal{B}^n$ are the bytes involved in the branch condition, and $T, F \in \mathsf{points}$ are the target program points for true and false branches respectively.

Clearly, any real assembly instruction can be encoded in this schema, possibly by breaking it down into multiple instructions.

Assignments, arithmetic operations or logical operations can be represented using the compute instruction form. Indirect memory accesses can be represented using the load and store instruction forms. Jumps and conditional branches can be represented using the branch instruction form.

5.1.4. Program. A *program* is a tuple $(\mathsf{points}, \text{instr})$ where:

- points is the set of *program points* (i.e., instruction addresses),
- $\text{instr} : \mathsf{points} \to \mathcal{I}$ is a mapping from each program point to an instruction.

Let $\mathsf{flow} \subseteq \mathsf{points} \times \mathsf{points}$ denote the *control-flow relation* of the program. We have $(\ell, \ell') \in \mathsf{flow}$ iff

- $\text{instr}(\ell) = \langle \mathsf{branch}, g, I, T, F \rangle$ and $\ell' \in \{T, F\}$, or
- $\text{instr}(\ell) \neq \langle \mathsf{branch}, g, I, T, F \rangle$ and ℓ' is the next program point in the program after ℓ.

The set points together with the relation flow form the *control-flow graph* (CFG).

5.2. Concrete domain

This section formalizes the concrete semantics of analyzed programs, that is, the actual behavior that we will later approximate using abstract interpretation.

As the goal of our analysis is to track how secret data propagates through program execution, we need to incorporate the notion of secret-dependency in the concrete semantics. To this end, we use the approach proposed in Section 3.3 of [13], where each value in the concrete domain is paired with a taint indicating whether it was influenced by secret data or not.

The taint used in the present section should not be confused with the multi-taint system used in the next section for the abstract domain.

5.2.1. Concrete state. A *concrete state* $\sigma^\flat : \mathcal{B} \to \mathbb{B} \times \{\mathsf{Sec}, \mathsf{Pub}\}$ maps each architectural byte to

- a concrete value, and
- a taint indicating whether the value of the byte is influenced by secret data or not.

We use $\Sigma^\flat$ to denote the set of all such concrete states.

Let $a \in \mathbb{B} \times \{\mathsf{Sec}, \mathsf{Pub}\}$ and $A \in (\mathbb{B} \times \{\mathsf{Sec}, \mathsf{Pub}\})^n$. We use

- $a.\mathsf{val}$ to denote the concrete value component of a;
- $a.\mathsf{taint}$ to denote the taint component of a;
- $A.\mathsf{val}$ to denote the concrete value components in A;
- $A.\mathsf{taint}$ to denote the taint components in A.

5.2.2. Concrete semantics. We now define the effect of applying an instruction to a single program state.

For any instruction $i \in \mathcal{I}$ that is not a branching instruction, the concrete semantics of i is the function $[\![i]\!]^\flat : \Sigma^\flat \to \Sigma^\flat$ that receives a program state and applies the instruction i to it. We define it separately for each type of instruction.

First, consider a compute instruction $\langle \mathsf{compute}, f, I, o \rangle$. Its concrete semantics is to read the values stored at bytes I, apply the function f to them, and write the result to byte o. Regarding secret-dependency, o becomes secret-dependent if any of the input bytes I is secret-dependent.

Formally, we have:

$$[\![\langle \mathsf{compute}, f, I, o \rangle]\!]^\flat(\sigma^\flat) \triangleq \sigma^\flat[o \mapsto (v, t)]$$

where

$$v = f(\sigma^\flat[I].\mathsf{val})$$
$$t = \begin{cases} \mathsf{Sec} & \text{if } \mathsf{Sec} \in \sigma^\flat[I].\mathsf{taint} \\ \mathsf{Pub} & \text{otherwise} \end{cases}$$

Now, consider a load instruction $\langle \mathsf{load}, f, S, d \rangle$. Its concrete semantics is to read the values stored at bytes S, apply the function f to them to obtain an address a, read the value stored at address a, and write it to byte d. Regarding secret-dependency, d becomes secret-dependent if either the source address S or the loaded value at address a is secret-dependent.

Formally, we have:

$$[\![\langle \mathsf{load}, f, S, d \rangle]\!]^\flat(\sigma^\flat) \triangleq \sigma^\flat[d \mapsto (v, t)]$$

where

$$a = f(\sigma^\flat[S].\mathsf{val})$$
$$v = \sigma^\flat[a].\mathsf{val}$$
$$t = \begin{cases} \mathsf{Sec} & \text{if } \mathsf{Sec} \in \sigma^\flat[S].\mathsf{taint} \cup \{\sigma^\flat[a].\mathsf{taint}\} \\ \mathsf{Pub} & \text{otherwise} \end{cases}$$

Finally, consider a store instruction $\langle \mathsf{store}, f, D, s \rangle$. Its concrete semantics is to read the values stored at bytes D, apply the function f to them to obtain an address a, read the value stored at byte s, and write it at address a. Regarding secret-dependency, the value stored at address a becomes secret-dependent if either any of the address bytes D or the source byte s is secret-dependent.

Formally, we have:

$$[\![\langle \mathsf{store}, f, D, s \rangle]\!]^\flat(\sigma^\flat) \triangleq \sigma^\flat[a \mapsto (v, t)]$$

where

$$a = f(\sigma^\flat[D].\mathsf{val})$$
$$v = \sigma^\flat[s].\mathsf{val}$$
$$t = \begin{cases} \mathsf{Sec} & \text{if } \mathsf{Sec} \in \sigma^\flat[D].\mathsf{taint} \cup \{\sigma^\flat[s].\mathsf{taint}\} \\ \mathsf{Pub} & \text{otherwise} \end{cases}$$

5.2.3. Concrete states propagation. The next step is to define how sets of reachable states propagate along edges of the control-flow graph.

For any pair of program points (ℓ, ℓ'), the concrete semantics of the transition from ℓ to ℓ' is the function $[\![(\ell, \ell')]\!]^\flat : \mathcal{P}(\Sigma^\flat) \to \mathcal{P}(\Sigma^\flat)$ that receives a set of reachable states at program point ℓ, applies the instruction at ℓ to each of them, and returns the resulting set of states to be propagated to program point ℓ'.

If ℓ contains a compute, load, or store instruction, then the instruction is simply applied to each state. If ℓ contains a branching instruction, then the states are filtered according to the branch condition.

Formally, let $A \in \mathcal{P}(\Sigma^\flat)$. Then:

- if $(\ell, \ell') \notin \mathsf{flow}$, then $[\![(\ell, \ell')]\!]^\flat(A) \triangleq \emptyset$;
- if $\mathrm{instr}(\ell) = \langle \mathsf{branch}, g, I, T, F \rangle$ where $\ell' = T$, then $[\![(\ell, \ell')]\!]^\flat(A) \triangleq \{\sigma^\flat \in A \mid g(\sigma^\flat[I].\mathsf{val})\}$;
- if $\mathrm{instr}(\ell) = \langle \mathsf{branch}, g, I, T, F \rangle$ where $\ell' = F$, then $[\![(\ell, \ell')]\!]^\flat(A) \triangleq \{\sigma^\flat \in A \mid \neg g(\sigma^\flat[I].\mathsf{val})\}$;
- otherwise, $[\![(\ell, \ell')]\!]^\flat(A) \triangleq \{[\![\mathrm{instr}(\ell)]\!]^\flat(\sigma^\flat) \mid \sigma^\flat \in A\}$.

5.2.4. Concrete transformer. The last step is to define how we discover the set of all reachable states at each program point.

A *concrete states assignment* is a mapping $\mu^\flat : \mathsf{points} \to \mathcal{P}(\Sigma^\flat)$ from each program point to a set of concrete states. We use $\mathcal{M}^\flat$ to denote the set of all such concrete states assignments.

We define the *concrete transformer* $\mathcal{F}^\flat : \mathcal{M}^\flat \to \mathcal{M}^\flat$ that receives a mapping from each program point to a set of previously discovered states, and returns an updated mapping where a single execution step has been performed for each state.

Formally, let $\mu^\flat \in \mathcal{M}^\flat$ and $\ell \in \mathsf{points}$. Then:

$$\mathcal{F}^\flat(\mu^\flat)[\ell] \triangleq \mu^\flat[\ell] \cup \bigcup_{\ell' \in \mathsf{points}} [\![(\ell', \ell)]\!]^\flat(\mu^\flat[\ell'])$$

For any program point ℓ, $\mathrm{lfp}(\mathcal{F}^\flat)[\ell]$ gives the set of all concrete states that can arise at program point ℓ, where lfp denotes the least fixed point. That is, it describes, for all possible program executions, the values that each architectural byte can take and whether those values depend on secret data.

Our goal in the next section will be to compute an overapproximation of $\mathrm{lfp}(\mathcal{F}^\flat)$.

5.3. Abstract domain

This section formalizes the abstract domain of our analysis. It implements the multi-taint system for overapproximation tracking and the path-sensitive analysis introduced in Section 4.

5.3.1. Abstract byte values. In the abstract domain, the value of an architectural byte can be one of the following:

- a concrete, non-secret-dependent value $v \in \mathbb{B}$, or
- the taint green, representing an unknown, non-secret-dependent value, or
- the taint orange, representing an unknown value that may depend on the secret, or
- the taint red, representing an unknown value that depends on the secret.

Formally, we define the set of *abstract byte values*:

$$\mathbb{B}^\sharp \triangleq \mathbb{B} \cup \{\mathsf{green}, \mathsf{orange}, \mathsf{red}\}$$

The relation "can be represented by" on $\mathbb{B}^\sharp$ induces a partial order whose Hasse diagram is depicted in Fig. 1. For instance, any element of $\mathbb{B}^\sharp$ can be represented by orange since orange captures any possible byte value that does or does not depend on the secret.

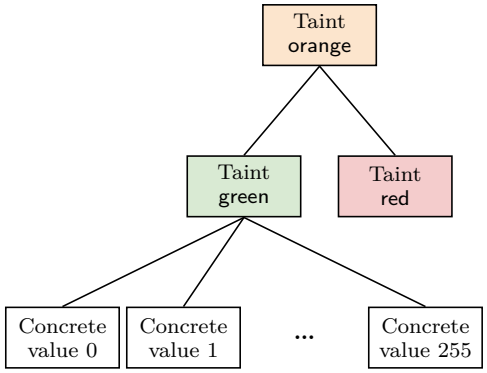

Figure 1. Hasse diagram for the set $\mathbb{B}^\sharp$ of abstract byte values.

We introduce a join operation $\wedge$ on $\mathbb{B}^\sharp$ defined as the least upper bound with respect to the partial order depicted in Fig. 1. For instance, we have $1 \wedge 2 = \mathsf{green}$.

5.3.2. Abstract states. An *abstract state* $\sigma^\sharp : \mathcal{B} \to \mathbb{B}^\sharp$ maps each architectural byte to an abstract byte value. We use $\Sigma^\sharp$ to denote the set of all such abstract states.

The state $\sigma^\sharp_\top$ is defined as the abstract state mapping each architectural byte to taint orange. It is the abstract state that can represent any possible concrete state.

We lift the join operation $\wedge$ to abstract states. Let $\sigma^\sharp_1, \sigma^\sharp_2 \in \Sigma^\sharp$ and $b \in \mathcal{B}$. Then:

$$(\sigma^\sharp_1 \wedge \sigma^\sharp_2)[b] \triangleq \sigma^\sharp_1[b] \wedge \sigma^\sharp_2[b]$$

Intuitively, the operation $\sigma^\sharp_1 \wedge \sigma^\sharp_2$ produces an abstract state that represents all concrete states represented by either $\sigma^\sharp_1$ or $\sigma^\sharp_2$. This operation will be used later to consolidate multiple abstract states into a single one when their number exceeds a given threshold.

5.3.3. Abstract semantics. We now define the effect of applying an instruction on a single abstract state.

For any instruction $i \in \mathcal{I}$ that is not a branching instruction, the abstract semantics of i is the function

$[\![i]\!]^\sharp : \Sigma^\sharp \to \Sigma^\sharp$ that receives an abstract state and applies the instruction i to it. We define it separately for each type of instruction.

First, consider a compute instruction $\langle \mathsf{compute}, f, I, o\rangle$. If any input value is red, then the output is guaranteed to be secret-dependent, so we set it to red. Otherwise, if any input is orange, then the output may be secret-dependent, so we set it to orange. Otherwise, if any input is green, then the output is non-secret-dependent but unknown, so we set it to green. Finally, if all inputs are known concrete values, we execute the instruction as in the concrete semantics.

Formally, we have:

$$[\![\langle \mathsf{compute}, f, I, o\rangle]\!]^\sharp(\sigma^\sharp) \triangleq \sigma^\sharp[o \mapsto v]$$

where

$$v = \begin{cases} \mathsf{red} & \text{if } \mathsf{red} \in \sigma^\sharp[I] \\ \mathsf{orange} & \text{else if } \mathsf{orange} \in \sigma^\sharp[I] \\ \mathsf{green} & \text{else if } \mathsf{green} \in \sigma^\sharp[I] \\ f(\sigma^\sharp[I]) & \text{otherwise} \end{cases}$$

Now, consider a load instruction $\langle \mathsf{load}, f, S, d\rangle$. If any address byte is red, then the loaded value is guaranteed to be secret-dependent, so we set it to red. Otherwise, if any address byte is tainted, then the load may target any location in memory, so we set it to orange. Finally, if all address bytes are known concrete values, we execute the instruction as in the concrete semantics.

Formally, we have:

$$[\![\langle \mathsf{load}, f, S, d\rangle]\!]^\sharp(\sigma^\sharp) \triangleq \sigma^\sharp[d \mapsto v]$$

where

$$v = \begin{cases} \mathsf{red} & \text{if } \mathsf{red} \in \sigma^\sharp[S] \\ \mathsf{orange} & \text{else if } \mathsf{orange} \in \sigma^\sharp[S] \\ \mathsf{orange} & \text{else if } \mathsf{green} \in \sigma^\sharp[S] \\ \sigma^\sharp[f(\sigma^\sharp[S])] & \text{otherwise} \end{cases}$$

Consider finally a store instruction $\langle \mathsf{store}, f, D, s\rangle$. If all address bytes are known concrete values, we execute the instruction as in the concrete semantics. If any address byte is tainted, then the store may target any location in memory, so we conservatively set the entire abstract state to $\sigma^\sharp_\top$ to represent all possible outcomes.

Formally, we have:

$$[\![\langle \mathsf{store}, f, D, s\rangle]\!]^\sharp(\sigma^\sharp) \triangleq \begin{cases} \sigma^\sharp_\top & \text{if } \sigma^\sharp[D] \notin \mathbb{B}^n \\ \sigma^\sharp[a \mapsto \sigma^\sharp[s]] & \text{otherwise} \end{cases}$$

where $a = f(\sigma^\sharp[D])$.

5.3.4. Abstract semantics of jump conditions. When a jump condition only depends on concrete values, it is evaluated as in the concrete semantics and the state is only propagated to the feasible successor. When a jump condition depends on a tainted value, its outcome is unknown to the analysis, so the state is soundly propagated to both successors.

Formally, for any guard function $g : \mathbb{B}^n \to \{\text{true}, \text{false}\}$, we define its abstract counterpart $g^\sharp : \mathbb{B}^{\sharp n} \to \{\text{true}, \text{false}\}$ as follows.

$$g^\sharp(v_1, \ldots, v_n) \triangleq \begin{cases} \text{true} & \text{if } (v_1, \ldots, v_n) \notin \mathbb{B}^n \\ g(v_1, \ldots, v_n) & \text{otherwise} \end{cases}$$

We also define $\neg g^\sharp : \mathbb{B}^{\sharp n} \to \{\text{true}, \text{false}\}$, the abstract counterpart of the negation of g:

$$\neg g^\sharp(v_1, \ldots, v_n) \triangleq \begin{cases} \text{true} & \text{if } (v_1, \ldots, v_n) \notin \mathbb{B}^n \\ \neg g(v_1, \ldots, v_n) & \text{otherwise} \end{cases}$$

5.3.5. Abstract states propagation. We now define how sets of abstract states are propagated along the edges of the control-flow graph. We proceed analogously to the concrete semantics.

For any pair of program points (ℓ, ℓ'), the abstract semantics of the transition from ℓ to ℓ' is the function $[\![(\ell, \ell')]\!]^\sharp : \mathcal{P}(\Sigma^\sharp) \to \mathcal{P}(\Sigma^\sharp)$ defined as follows.

Let $A \in \mathcal{P}(\Sigma^\sharp)$. Then:

- if $(\ell, \ell') \notin \mathsf{flow}$, then $[\![(\ell, \ell')]\!]^\sharp(A) \triangleq \emptyset$;
- if $\mathrm{instr}(\ell) = \langle \mathsf{branch}, g, I, T, F\rangle$ where $\ell' = T$, then $[\![(\ell, \ell')]\!]^\sharp(A) \triangleq \{\sigma^\sharp \in A \mid g^\sharp(\sigma^\sharp[I])\}$;
- if $\mathrm{instr}(\ell) = \langle \mathsf{branch}, g, I, T, F\rangle$ where $\ell' = F$, then $[\![(\ell, \ell')]\!]^\sharp(A) \triangleq \{\sigma^\sharp \in A \mid \neg g^\sharp(\sigma^\sharp[I])\}$;
- otherwise, $[\![(\ell, \ell')]\!]^\sharp(A) \triangleq \{[\![\mathrm{instr}(\ell)]\!]^\sharp(\sigma^\sharp) \mid \sigma^\sharp \in A\}$.

5.3.6. Abstract transformer. We can now define states assignments and the transformer function. This part is straightforward and is analogous to the equivalent concrete definitions.

An *abstract states assignment* is a mapping $\mu^\sharp : \mathsf{points} \to \mathcal{P}(\Sigma^\sharp)$ from each program point to a set of abstract states. We use $\mathcal{M}^\sharp$ to denote the set of all such abstract states assignments.

We define the *abstract transformer* $\mathcal{F}^\sharp : \mathcal{M}^\sharp \to \mathcal{M}^\sharp$ as follows. Let $\mu^\sharp \in \mathcal{M}^\sharp$ and $\ell \in \mathsf{points}$. Then:

$$\mathcal{F}^\sharp(\mu^\sharp)[\ell] \triangleq \mu^\sharp[\ell] \cup \bigcup_{\ell' \in \mathsf{points}} [\![(\ell', \ell)]\!]^\sharp(\mu^\sharp[\ell'])$$

5.3.7. Abstract states consolidation. The abstract transformer simply accumulates newly computed abstract states at each program point. If used as is, the analysis would typically be intractable, as the number of abstract states at each program point could grow unboundedly.

To ensure scalability, we cap the number of abstract states stored at each program point by a threshold τ. When τ is exceeded, we soundly consolidate all abstract states into a single one. This is implemented in the form of a *widening operator* (see section 4.3 of [36]).

We define the widening operator $\nabla_\tau : \mathcal{M}^\sharp \times \mathcal{M}^\sharp \to \mathcal{M}^\sharp$ that receives the current abstract states assignment and the result of applying the abstract transformer, and, at each program point, consolidates the abstract states if their number exceeds the consolidation threshold, using the join operation $\wedge$ defined in Section 5.3.2.

Formally, let $\mu_1^\sharp, \mu_2^\sharp \in \mathcal{M}^\sharp$ and $\ell \in \mathsf{points}$. Then:

$$(\mu_1^\sharp \nabla_\tau \mu_2^\sharp)[\ell] \triangleq \begin{cases} A & \text{if } |A| < \tau \\ \bigwedge_{\sigma^\sharp \in A} \sigma^\sharp & \text{otherwise} \end{cases}$$

where $A = \mu_1^\sharp[\ell] \cup \mu_2^\sharp[\ell]$.

Our analysis consists of executing the abstract transformer to simulate one execution step, applying the widening operator to consolidate abstract states when necessary, and repeating this process until a fixpoint is reached.

Formally, we define the chain of abstract states assignments $(\mu_i^\sharp)_{i \geq 0}$ as follows.

$$\mu_0^\sharp \triangleq \bot^\sharp$$
$$\mu_{i+1}^\sharp \triangleq \mu_i^\sharp \nabla_\tau \mathcal{F}^\sharp(\mu_i^\sharp)$$

where $\bot^\sharp$ is the abstract states assignment corresponding to the initial state of the program (entry program point mapped to an abstract state representing the initial program state, all other program points mapped to the empty set).

The analysis computes the limit $\mu_*^\sharp \triangleq \lim_{i \to \infty} \mu_i^\sharp$. We will see in the next section that $\mu_*^\sharp$ is an overapproximation of all possible program behaviors.

6. Analysis correctness

In this section we define the abstraction and concretization functions, that is, the formal relationship between the concrete and abstract domain. We then state and prove the correctness of our analysis.

6.1. Concretization function

The concretization function maps abstract elements to the set of concrete elements they represent. We proceed incrementally, starting by describing how a single abstract state is concretized, then lifting this to sets of states, and finally defining how a mapping from program points to sets of abstract states is concretized.

We define the concretization function on states $\bar{\gamma} : \Sigma^\sharp \to \mathcal{P}(\Sigma^\flat)$. This function receives an abstract state, and computes all the concrete states that it can represent. The taint orange represents any concrete value, regardless of its secret-dependency status. The taint red only represents concrete values that are secret-dependent, while the taint green only represents concrete values that are non-secret-dependent. A concrete value in the abstract domain represents the same, non-secret-dependent value in the concrete domain.

Formally, we have:

$$\bar{\gamma}(\sigma^\sharp) \triangleq \{\sigma^\flat \in \Sigma^\flat \mid \forall b \in \mathcal{B}, V(\sigma^\sharp[b], \sigma^\flat[b]) \wedge T(\sigma^\sharp[b], \sigma^\flat[b])\}$$

where V and T are predicates defined as follows:

$$V(x, y) \equiv \begin{cases} y.\mathsf{val} = x & \text{if } x \in \mathbb{B} \\ \text{true} & \text{if } x \in \{\mathsf{green}, \mathsf{orange}, \mathsf{red}\} \end{cases}$$

$$T(x, y) \equiv \begin{cases} y.\mathsf{taint} = \mathsf{Sec} & \text{if } x = \mathsf{red} \\ \text{true} & \text{if } x = \mathsf{orange} \\ y.\mathsf{taint} = \mathsf{Pub} & \text{if } x = \mathsf{green} \\ y.\mathsf{taint} = \mathsf{Pub} & \text{if } x \in \mathbb{B} \end{cases}$$

We can now define the concretization function $\gamma : \mathcal{P}(\Sigma^\sharp) \to \mathcal{P}(\Sigma^\flat)$ on sets of states by lifting the concretization function on states. The function γ receives a set of abstract states and simply applies the concretization function $\bar{\gamma}$ to each of them.

Formally, let $\{\sigma_1^\sharp, \ldots, \sigma_n^\sharp\} \in \mathcal{P}(\Sigma^\sharp)$. Then:

$$\gamma(\{\sigma_1^\sharp, \ldots, \sigma_n^\sharp\}) \triangleq \bar{\gamma}(\sigma_1^\sharp) \cup \cdots \cup \bar{\gamma}(\sigma_n^\sharp)$$

Finally, we define the concretization function $\Gamma : \mathcal{M}^\sharp \to \mathcal{M}^\flat$ on states assignments by lifting the concretization function on sets of states. The function Γ receives a mapping from program points to sets of abstract states, and applies the concretization function γ at each program point.

Formally, let $\mu^\sharp \in \mathcal{M}^\sharp$ and $\ell \in \mathsf{points}$. Then:

$$\Gamma(\mu^\sharp)[\ell] \triangleq \gamma(\mu^\sharp[\ell])$$

6.2. Abstraction function

The abstraction function maps a set of concrete elements to the minimal abstract element that represents them. As with the concretization function, we first define the abstraction of a single state, then lift this to sets of states, and finally define the abstraction of a mapping from program points to sets of concrete states.

We define the abstraction function on states $\bar{\alpha} : \Sigma^\flat \to \Sigma^\sharp$. This function receives a concrete state, and computes the minimal abstract state that represents it. Non-secret-dependent values are mapped to the equivalent concrete value, while secret-dependent values are mapped to the taint red.

Formally, let $\sigma^\flat \in \Sigma^\flat$ and $b \in \mathcal{B}$. Then:

$$\bar{\alpha}(\sigma^\flat)[b] \triangleq \begin{cases} \mathsf{red} & \text{if } \sigma^\flat[b].\mathsf{taint} = \mathsf{Sec} \\ \sigma^\flat[b].\mathsf{val} & \text{if } \sigma^\flat[b].\mathsf{taint} = \mathsf{Pub} \end{cases}$$

We now define the abstraction function $\alpha : \mathcal{P}(\Sigma^\flat) \to \mathcal{P}(\Sigma^\sharp)$ on sets of states by lifting the abstraction function on states. This function receives a set of concrete states, and applies the abstraction function $\bar{\alpha}$ to each of them.

Formally, let $\{\sigma_1^\flat, \ldots, \sigma_n^\flat\} \in \mathcal{P}(\Sigma^\flat)$. Then:

$$\alpha(\{\sigma_1^\flat, \ldots, \sigma_n^\flat\}) \triangleq \{\bar{\alpha}(\sigma_1^\flat), \ldots, \bar{\alpha}(\sigma_n^\flat)\}$$

Finally, we define the abstraction function $\mathcal{A} : \mathcal{M}^\flat \to \mathcal{M}^\sharp$ on states assignments by lifting the abstraction function on sets of states. Let $\mu^\flat \in \mathcal{M}^\flat$ and $\ell \in \mathsf{points}$. Then:

$$\mathcal{A}(\mu^\flat)[\ell] \triangleq \alpha(\mu^\flat[\ell])$$

6.3. Correctness

We can now state the theorem of soundness of our analysis. It ensures that the analysis described in Section 5.3 produces a result that correctly overapproximates all possible behaviors of the analyzed program, as defined in Section 5.2.

Theorem 1 (Soundness of the analysis)**.** *The result $\mu_*^\sharp$ of our analysis (defined in Section 5.3.7) overapproximates the least fixed point of the concrete transformer $\mathcal{F}^\flat$ (defined in Section 5.2.4).*

That is, for every program point ℓ, we have:

$$lfp(\mathcal{F}^\flat)[\ell] \subseteq \Gamma(\mu^\sharp_*)[\ell]$$

Proof. The complete proof is given in Appendix A.1. It can be sketched as follows.

- The functions α and γ form a Galois connection between $\mathcal{P}(\Sigma^\flat)$ and $\mathcal{P}(\Sigma^\sharp)$.
- By pointwise extension, the functions $\mathcal{A}$ and Γ form a Galois connection between $\mathcal{M}^\flat$ and $\mathcal{M}^\sharp$.
- The abstract semantics of instructions (defined in Section 5.3) soundly overapproximates their concrete semantics (defined in Section 5.2).
- The least fixed point of the concrete transformer is thus overapproximated by the least fixed point of the abstract transformer (see Propositions 24 and 25 of [36]).
- The use of the widening operator in the computation of $\mu^\sharp_*$ preserves the overapproximation (see Proposition 31 of [36]).

□

Our analysis reports a CT violation whenever an attacker-observable (address of a memory access, condition of a branch instruction, or operand of a variable-time instruction) depends on red-tainted or orange-tainted data. The next result ensures that all CT violations present in the tested program are reported by our analysis.

Corollary 2. *For every program point ℓ, if there exists a concrete state $\sigma^\flat \in lfp(\mathcal{F}^\flat)[\ell]$ and an architectural byte $b \in \mathcal{B}$ such that $\sigma^\flat[b].\mathsf{taint} = \mathsf{Sec}$, then there exists an abstract state $\sigma^\sharp \in \mu^\sharp_*[\ell]$ such that $\sigma^\sharp[b] \in \{\mathsf{red}, \mathsf{orange}\}$.*

The proof is given in Appendix A.2. It follows directly from Theorem 1 and the definition of the concretization function.

Our analysis reports a confirmed CT violation when an attacker-observable depends on red-tainted data in *all* abstract states at the corresponding program point. The next result ensures that all confirmed CT violations reported by our analysis correspond to actual CT violations, assuming that there exists at least one concrete execution reaching the corresponding program point.

Corollary 3. *Consider an arbitrary program point ℓ. If there exists an architectural byte $b \in \mathcal{B}$ such that for every abstract state $\sigma^\sharp \in \mu^\sharp_*[\ell]$ we have $\sigma^\sharp[b] = \mathsf{red}$, then for every concrete state $\sigma^\flat \in lfp(\mathcal{F}^\flat)[\ell]$ we have $\sigma^\flat[b].\mathsf{taint} = \mathsf{Sec}$.*

The proof is given in Appendix A.2. It follows directly from Theorem 1 and the definition of the concretization function.

7. Implementation

We implement our analysis in a prototype tool named SCOUT-CT and make it publicly available at https://github.com/SCOUT-CT.

SCOUT-CT is implemented on top of the angr [55] binary analysis framework. Although angr is primarily designed for symbolic execution, we repurpose it to implement our lightweight multi-taint analysis.

7.1. Program states in angr

In angr, a program state is represented as a mapping from each memory and register byte to either a *concrete value* or a *symbolic expression*. A symbolic expression is associated with a set of *symbols*.

From such a program state, angr can simulate the execution of a program instruction to produce a new state. If all inputs of the instruction are concrete values, angr simulates the operation as in a real execution and produces a concrete output. If some inputs of the instruction are symbolic expressions, the output is a new symbolic expression whose associated symbols are those of the input symbolic expressions.

7.2. Abstract states

As defined in Section 5.3.3, our analysis operates on abstract states that map memory bytes to either a concrete value or one of three taints (red, orange, or green).

In SCOUT-CT, an abstract state is implemented as an angr program state, where untainted data is represented as concrete values and tainted data is represented as symbolic expressions.

We use three distinct symbols to represent the three taints.

A symbolic expression may be associated with multiple symbols. We define the taint represented by a symbolic expression as red if at least one of its symbols is the red symbol; otherwise, it is orange if at least one of its symbols is the orange symbol; otherwise, it is green. This follows the abstract semantics defined in Section 5.3.3.

7.3. Basic block instructions semantics

The abstract semantics of non-branching instructions propagates taints from the input operands to the output operands, or produces a concrete value if all inputs are untainted.

We implement this by letting angr execute the instructions of a basic block on the corresponding program state. This makes angr generate symbolic outputs for instructions that operate on symbolic inputs, effectively propagating the taints.

7.4. Control flow graph traversal

SCOUT-CT associates the entry and exit of each basic block with a set of program states. It uses Kildall's algorithm to propagate program states across the control flow graph of the program until a fixed point is reached.

The exit states of a basic block are computed from its entry states as defined in Section 7.3.

When a basic block ends with a conditional branch, the guard of the branch is evaluated on each exit state of the basic block. If the guard has a concrete

value, the state is only propagated to the appropriate successor. If the guard is symbolic (i.e., tainted), the state is propagated to both successors. This follows the abstract semantics of conditional branching defined in Section 5.3.4.

Note that, unlike in symbolic execution, our analysis does not make any call to an SMT solver.

8. Evaluation

The main contribution of our tool is its ability to soundly scale to real-world implementations while automatically identifying a subset of *confirmed* violations that require little manual verification.

We aim to experimentally evaluate the performance of SCOUT-CT on realistic programs with respect to

- the proportion of confirmed violations among all reported violations, and
- its scalability compared to existing state-of-the-art solutions.

To this end, we create a benchmark of cryptographic software and run both SCOUT-CT and two existing CT analysis tools on it.

Our benchmark is made publicly available at https://github.com/SCOUT-CT.

8.1. Consolidation threshold

Our analysis is parametrized by a consolidation threshold τ (introduced in Section 4.2 and formally defined in Section 5.3.7) that controls the trade-off between precision and performance.

For the evaluation, the value of τ is selected dynamically. The analysis is first performed with $\tau = 8$ and, if an unconfirmed CT violation is reported, the value of τ is doubled and the analysis is re-executed, until either no unconfirmed CT violation is reported or τ reaches the value 16384.

8.2. Compared tools

We compare SCOUT-CT against Microwalk [16] and Binsec/Rel [10], [34].

Microwalk is a dynamic analysis tool that performs multiple executions of the tested program with different inputs, each time recording a trace of memory accesses and branches, and compares those traces to identify CT violations. Microwalk is not sound — it can miss leaks — but all reported leaks are guaranteed to be true positives.

As Microwalk cannot prove the absence of leaks, we only evaluate it on non-CT programs of our benchmark.

Binsec/Rel is an analysis tool based on symbolic execution. It executes the program on symbolic inputs, systematically explores execution paths, and solves SMT formulas to check if memory accesses or branches depend on secret data. Binsec/Rel performs bounded verification: the execution paths are explored up to a certain depth. All reported leaks are guaranteed to be true positives. Moreover, when the analysis is exhaustive, all leaks are reported.

Microwalk and Binsec/Rel test programs provided as compiled binaries. Both are recognized as effective tools for testing real-world software [38], [39].

8.3. Benchmark

Our benchmark is composed of:

- a non-CT AES implementation (tiny AES[1]),
- various CT and non-CT symmetric cryptographic primitives from BearSSL[2],
- a CT implementation of an elliptic-curve-based digital signature from libhydrogen[3],
- the reference implementation[4] of the K-PKE.Decrypt function of the post-quantum key encapsulation mechanism ML-KEM [56],
- a collection of CT and non-CT utility functions commonly found in cryptographic code,
- a program (named "overapproximation" in the results) similar to Listing 5 presented in Section 4.1, that is specifically crafted to make SCOUT-CT report a potential (unconfirmed) CT violation.

The cryptographic libraries are compiled without source code modification, and are used through their public API only. All programs are compiled with debug information (to enable locating the reported leaks in the source code) and without compiler optimizations.

Manual inspection of the source code was performed to identify lines that cause CT violations.

8.4. Metrics

For each program analysis we report:

- The analysis duration. For SCOUT-CT, this includes the time taken by all analysis iterations with increasing value of τ (see Section 8.1).
- Whether an exhaustive analysis was completed, that is, the analysis did not time out and (in the case of Binsec/Rel) did not reach the exploration depth limit. This metric is not applicable to Microwalk, as, by design, it only analyzes a subset of execution paths.
- The proportion of CT violations detected. For SCOUT-CT, the subset of confirmed violations is reported separately.

We set a timeout of 1 hour for each analysis. The evaluation was performed on a machine with an Intel Xeon Silver 4216 @ 2.10 GHz CPU and 64 GiB of memory.

1. github.com/kokke/tiny-AES-c
2. bearssl.org
3. github.com/jedisct1/libhydrogen
4. https://github.com/pq-crystals/kyber

TABLE 2. BENCHMARK RESULTS FOR NON-CT PROGRAMS.

	Microwalk		Binsec/Rel			SCOUT-CT			
Benchmark	**T**	**#V**	**T**	**E**	**#V**	**T**	**E**	**#V**	**#CV**
util-mult-nonct1	4	2 / 2	⧗	✗	1 / 2	11	✓	2 / 2	2
util-mult-nonct2	4	1 / 1	⧗	✗	1 / 1	9	✓	1 / 1	1
util-mult-nonct3	4	1 / 1	⧗	✗	1 / 1	5	✓	1 / 1	1
util-sqm-nonct	4	2 / 2	⧗	✗	1 / 2	13	✓	2 / 2	2
util-password-nonct	3	1 / 1	0.21	✓	1 / 1	5	✓	1 / 1	1
util-sbox	3	1 / 1	0.21	✓	1 / 1	3	✓	1 / 1	1
overapproximation	3	1 / 3	0.20	✓	3 / 3	3	✓	3 / 3	2
bearssl-aes-big-cbcdec	35	36 / 36	0.36	✓	36 / 36	89	✓	36 / 36	36
bearssl-aes-big-cbcenc	10	36 / 36	0.30	✓	36 / 36	19	✓	36 / 36	36
bearssl-aes-small-cbcdec	29	5 / 5	0.29	✓	5 / 5	50	✓	5 / 5	5
bearssl-aes-small-cbcenc	24	5 / 5	0.26	✓	5 / 5	35	✓	5 / 5	5
tinyaes-enc	20	5 / 5	0.27	✓	5 / 5	76	✓	5 / 5	5
Total	–	**96 / 98**	–	–	**96 / 98**	–	–	**98 / 98**	**97**

T: analysis time in seconds (⧗: timeout after 1 hour). E: whether an exhaustive analysis was performed. #V: number of violations reported. #CV: number of confirmed violations reported.

TABLE 3. BENCHMARK RESULTS FOR CT PROGRAMS.

	Binsec/Rel		SCOUT-CT	
Benchmark	**T**	**E**	**T**	**E**
util-mult-ct	0.19	✓	2	✓
util-sqm-ct	0.19	✓	3	✓
util-password-ct	0.21	✓	3	✓
bearssl-aes-ct-cbcdec	0.52	✓	49	✓
bearssl-aes-ct-cbcenc	0.44	✓	59	✓
hydrogen-sign	3	✓	2827	✓
ml-kem-indcpa-dec	0.76	✓	666	✓

T: analysis time in seconds. E: whether an exhaustive analysis was performed.

8.5. Results

The evaluation results on non-CT programs are presented in Table 2, and those on CT programs are presented in Table 3. On CT programs, an exhaustive analysis (E) indicates that the program was proven constant-time.

Microwalk finds almost all timing leaks and completes all analyses in at most 20 seconds. By design, it cannot prove the security of CT programs.

Binsec/Rel achieves impressive timings, struggling only on specially crafted examples. This shows the advantage of our method in this situation.

SCOUT-CT completes exhaustive analyses on all programs. Except for the digital signature and post-quantum KEM programs, whose analyses take around 45 minutes and 10 minutes respectively, all analyses complete in less than 2 minutes. SCOUT-CT successfully proves the security of all CT programs, and reports all leaks in non-CT programs. Remarkably, all leaks are reported as confirmed, except for the overapproximation program, which was specifically designed to trigger such behavior.

8.6. Discussion

Our evaluation confirms the effectiveness of the overapproximation tracking technique we propose. It demonstrates the ability of our tool to automatically classify most reported leaks as confirmed CT violations that require little manual verification.

The results also show that our approach, despite relying on overapproximation, produces few (in this benchmark, none) false positives.

The experiments demonstrate the ability of SCOUT-CT to scale to real-world cryptographic software, producing exhaustive analyses in reasonable time, and thus confirming its practical usefulness. The execution time of SCOUT-CT is comparable to Microwalk while, unlike Microwalk, our tool performs a sound analysis.

With a corrected configuration (see erratum), Binsec/Rel also completes exhaustive analyses on all real-world benchmarks in our suite, with substantially faster runtime than SCOUT-CT. The two tools represent different design points: Binsec/Rel's symbolic execution provides concrete counterexamples for every violation, while SCOUT-CT's multi-taint tracking distinguishes proven from potential violations without requiring SMT solving, which may scale differently as benchmark complexity grows beyond what we tested here.

Overall, the results support the usefulness of the new trade-off we propose for automatic program analysis in the constant-time threat model: while our tool offers weaker guarantees regarding false positives than complete tools such as Binsec/Rel and Microwalk, it offers a sound analysis with good scalability.

9. Conclusion

We presented SCOUT-CT, a sound, path-sensitive, binary-level analysis for verifying programs under the constant-time threat model. Our approach uses overapproximation to scale to real-world software, while systematically tracking precision loss to automatically identify a subset of reported violations as *confirmed*.

We formalized the analysis in the abstract interpretation framework and proved its correctness.

Our prototype implementation successfully analyzes a set of real-world cryptographic implementations in reasonable time and classifies almost all reported violations as *confirmed*. In our comparative

evaluation against state-of-the-art tools, SCOUT-CT was the only approach that completed an exhaustive analysis of all programs.

Acknowledgment

The authors express their gratitude towards the anonymous reviewers of the paper for their valuable comments. We thank Sébastien Bardin, Lesly-Ann Daniel, Tamara Rezk, and Frederic Recoules, authors of the BINSEC platform, for the care and rigor they brought to replicating our evaluation, and for working with us to identify and correct a configuration error in our use of BINSEC/Rel. GitHub Copilot and ChatGPT were used for identifying typos and assisting with LaTeX.

References

[1] Jakub Szefer. Survey of microarchitectural side and covert channels, attacks, and defenses. *Journal of Hardware and Systems Security*, 3(3):219–234, 2019.

[2] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In Neal Koblitz, editor, *Advances in Cryptology — CRYPTO '96*, pages 104–113, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[3] David Brumley and Dan Boneh. Remote timing attacks are practical. In *12th USENIX Security Symposium (USENIX Security 03)*, Washington, D.C., August 2003. USENIX Association.

[4] Nadhem J Al Fardan and Kenneth G Paterson. Lucky thirteen: Breaking the tls and dtls record protocols. In *2013 IEEE symposium on security and privacy*, pages 526–540. IEEE, 2013.

[5] Daniel J. Bernstein, Karthikeyan Bhargavan, Shivam Bhasin, Anupam Chattopadhyay, Tee Kiah Chia, Matthias J. Kannwischer, Franziskus Kiefer, Thales B. Paiva, Prasanna Ravi, and Goutam Tamvada. Kyberslash: Exploiting secret-dependent division timings in kyber implementations. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2025(2):209–234, 2025.

[6] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 199–212, 2009.

[7] Yuval Yarom and Katrina Falkner. {FLUSH+ RELOAD}: A high resolution, low noise, l3 cache {Side-Channel} attack. In *23rd USENIX security symposium (USENIX security 14)*, pages 719–732, 2014.

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

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

[10] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. Binsec/rel: Symbolic binary analyzer for security with applications to constant-time and secret-erasure, 2022.

[11] Craig Disselkoen, Sunjay Cauligi, Dean Tullsen, and Deian Stefan. Finding and eliminating timing side-channels in crypto code with pitchfork. In *TECHCON*, 2020.

[12] Gilles Barthe, Gustavo Betarte, Juan Campo, Carlos Luna, and David Pichardie. System-level non-interference for constant-time cryptography. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, page 1267–1279, New York, NY, USA, 2014. Association for Computing Machinery.

[13] Sandrine Blazy, David Pichardie, and Alix Trieu. Verifying constant-time implementations by abstract interpretation. *Journal of Computer Security*, 27(1):137–163, 2019.

[14] Jose Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying Constant-Time implementations. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 53–70, Austin, TX, August 2016. USENIX Association.

[15] Samuel Weiser, Andreas Zankl, Raphael Spreitzer, Katja Miller, Stefan Mangard, and Georg Sigl. DATA – differential address trace analysis: Finding address-based Side-Channels in binaries. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 603–620, Baltimore, MD, August 2018. USENIX Association.

[16] Jan Wichelmann, Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. Microwalk: A framework for finding side channels in binaries. In *Proceedings of the 34th Annual Computer Security Applications Conference*, ACSAC '18. ACM, December 2018.

[17] Shaobo He, Michael Emmi, and Gabriela Ciocarlie. ct-fuzz: Fuzzing for timing leaks, 2019.

[18] Adam Langley. ctgrind. https://github.com/agl/ctgrind, 2010.

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

[20] Jan Jancar, Marcel Fourné, Daniel De Almeida Braga, Mohamed Sabt, Peter Schwabe, Gilles Barthe, Pierre-Alain Fouque, and Yasemin Acar. "they're not that hard to mitigate": What cryptographic library developers think about timing attacks. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*, pages 632–649. IEEE, 2022.

[21] Isabella Mastroeni and Michele Pasqua. Statically analyzing information flows: an abstract interpretation-based hyperanalysis for non-interference. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, pages 2215–2223, 2019.

[22] Shuai Wang, Pei Wang, Xiao Liu, Danfeng Zhang, and Dinghao Wu. CacheD: Identifying Cache-Based timing channels in production software. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 235–252, Vancouver, BC, August 2017. USENIX Association.

[23] Gorka Irazoqui, Kai Cong, Xiaofei Guo, Hareesh Khattri, Arun Kanuparthi, Thomas Eisenbarth, and Berk Sunar. Did we learn from llc side channel attacks? a cache leakage detection tool for crypto libraries. *arXiv preprint arXiv:1709.01552*, 2017.

[24] Moritz Neikes. https://www.post-apocalyptic-crypto.org/timecop/, 2020.

[25] Ke Jiang, Yuyan Bao, Shuai Wang, Zhibo Liu, and Tianwei Zhang. Cache refinement type for side-channel detection of cryptographic software. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 1583–1597, New York, NY, USA, 2022. Association for Computing Machinery.

[26] https://crocs-muni.github.io/ct-tools/tutorials/memsan.

[27] Yuan Xiao, Mengyuan Li, Sanchuan Chen, and Yinqian Zhang. Stacco: Differentially analyzing side-channel traces for detecting ssl/tls vulnerabilities in secure enclaves. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 859–874, 2017.

[28] Wubing Wang, Yinqian Zhang, and Zhiqiang Lin. 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)*, pages 443–457, 2019.

[29] Yuanyuan Yuan, Zhibo Liu, and Shuai Wang. Cacheql: Quantifying and localizing cache side-channel vulnerabilities in production software, 2022.

[30] Bruno Rodrigues, Fernando Magno Quintão Pereira, and Diego F. Aranha. Sparse representation of implicit flows with applications to side-channel detection. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, page 110–120, New York, NY, USA, 2016. Association for Computing Machinery.

[31] Alexander Schaub. *Formal methods for the analysis of cache-timing leaks and key generation in cryptographic implementations*. PhD thesis, Institut Polytechnique de Paris, 2020.

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

[33] Shuai Wang, Yuyan Bao, Xiao Liu, Pei Wang, Danfeng Zhang, and Dinghao Wu. Identifying Cache-Based side channels through Secret-Augmented abstract interpretation. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 657–674, Santa Clara, CA, August 2019. USENIX Association.

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

[35] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, 1977.

[36] Patrick Cousot and Radhia Cousot. Abstract interpretation and application to logic programs. *The Journal of Logic Programming*, 13(2-3):103–179, 1992.

[37] Gary A Kildall. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 194–206, 1973.

[38] Antoine Geimer, Mathéo Vergnolle, Frédéric Recoules, Lesly-Ann Daniel, Sébastien Bardin, and Clémentine Maurice. A systematic evaluation of automated tools for side-channel vulnerabilities detection in cryptographic libraries. In Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda, editors, *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*, pages 1690–1704. ACM, 2023.

[39] Marcel Fourné, Daniel De Almeida Braga, Jan Jancar, Mohamed Sabt, Peter Schwabe, Gilles Barthe, Pierre-Alain Fouque, and Yasemin Acar. "these results must be false": A usability evaluation of constant-time analysis tools. In Davide Balzarotti and Wenyuan Xu, editors, *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*. USENIX Association, 2024.

[40] Martin Dunsche, Marcel Maehren, Nurullah Erinola, Robert Merget, Nicolai Bissantz, Juraj Somorovsky, and Jörg Schwenk. With great power come great side channels: Statistical timing Side-Channel analyses with bounded type-1 errors. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 6687–6704, Philadelphia, PA, August 2024. USENIX Association.

[41] Oscar Reparaz, Josep Balasch, and Ingrid Verbauwhede. Dude, is my code constant time? In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pages 1697–1702. IEEE, 2017.

[42] https://tlsfuzzer.readthedocs.io/en/latest/timing-analysis.html, 2020.

[43] Shirin Nilizadeh, Yannic Noller, and Corina S Pasareanu. Diffuzz: differential fuzzing for side-channel analysis. in 2019 ieee/acm 41st international conference on software engineering (icse). *IEEE, 176ś187*, 2019.

[44] Timos Antonopoulos, Paul Gazzillo, Michael Hicks, Eric Koskinen, Tachio Terauchi, and Shiyi Wei. Decomposition instead of self-composition for proving the absence of timing channels. *ACM SIGPLAN Notices*, 52(6):362–375, 2017.

[45] Jia Chen, Yu Feng, and Isil Dillig. Precise detection of side-channel vulnerabilities using quantitative cartesian hoare logic. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 875–890, New York, NY, USA, 2017. Association for Computing Machinery.

[46] Konstantinos Athanasiou, Byron Cook, Michael Emmi, Colm MacCarthaigh, Daniel Schwartz-Narbonne, and Serdar Tasiran. Sidetrail: Verifying time-balancing of cryptosystems. In Ruzica Piskac and Philipp Rümmer, editors, *Verified Software. Theories, Tools, and Experiments*, pages 215–228, Cham, 2018. Springer International Publishing.

[47] Tegan Brennan, Seemanta Saha, Tevfik Bultan, and Corina S. Păsăreanu. Symbolic path cost analysis for side-channel detection. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, page 27–37, New York, NY, USA, 2018. Association for Computing Machinery.

[48] Tuba Yavuz, Farhaan Fowze, Grant Hernandez, Ken Yihang Bai, Kevin RB Butler, and Dave Jing Tian. Encider: detecting timing and cache side channels in sgx enclaves and cryptographic apis. *IEEE Transactions on Dependable and Secure Computing*, 20(2):1577–1595, 2022.

[49] Guanhua Wang, Sudipta Chattopadhyay, Arnab Kumar Biswas, Tulika Mitra, and Abhik Roychoudhury. Kleespectre: Detecting information leakage through speculative cache attacks via symbolic execution. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 29(3):1–31, 2020.

[50] Marco Guarnieri, Boris Köpf, José F Morales, Jan Reineke, and Andrés Sánchez. Spectector: Principled detection of speculative information flows. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1–19. IEEE, 2020.

[51] Shengjian Guo, Yueqi Chen, Peng Li, Yueqiang Cheng, Huibo Wang, Meng Wu, and Zhiqiang Zuo. Specusym: Speculative symbolic execution for cache timing leak detection. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 1235–1247, 2020.

[52] Robert Brotzman, Danfeng Zhang, Mahmut Taylan Kandemir, and Gang Tan. Specsafe: detecting cache side channels in a speculative world. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–28, 2021.

[53] Oleksii Oleksenko, Bohdan Trach, Mark Silberstein, and Christof Fetzer. {SpecFuzz}: Bringing spectre-type vulnerabilities to the surface. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1481–1498, 2020.

[54] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. Compcert-a formally verified optimizing compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*, 2016.

[55] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Sok: (state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 138–157, 2016.

[56] National Institute of Standards and Technology. Module-Lattice-Based Key-Encapsulation Mechanism Standard. Federal Information Processing Standards Publication FIPS 203, National Institute of Standards and Technology, Gaithersburg, MD, August 2024.

Appendix A. Proofs

This section presents the proof of the soundness theorem and the two corollaries stated in Section 6.

A.1. Soundness theorem

Lemma 4. *Consider*

- *the powerset of concrete states* $\mathcal{P}(\Sigma^\flat)$*, as defined in Section 5.2*
- *the subset relation* $\subseteq$*,*
- *the union of sets operator* $\cup$*,*
- *and the intersection of sets operator* $\cap$*.*

Then, $\mathcal{P}(\Sigma^\flat)$ *together with* $\subseteq$ *is a complete lattice with join operator* $\cup$ *and meet operator* $\cap$*.*

Proof. Let $\mathcal{A} \subseteq \mathcal{P}(\Sigma^\flat)$ be an arbitrary subset. We need to show that (i) $\bigcup \mathcal{A}$ is the least upper bound on $\mathcal{A}$, and (ii) $\bigcap \mathcal{A}$ is the greatest lower bound on $\mathcal{A}$.

(i). Let U be an upper bound on $\mathcal{A}$. Then, for all $A \in \mathcal{A}$, we have $A \subseteq U$. Thus, $\bigcup \mathcal{A} \subseteq U$. Consequently, $\bigcup \mathcal{A}$ is the least upper bound on $\mathcal{A}$.

(ii). Let L be a lower bound on $\mathcal{A}$. Then, for all $A \in \mathcal{A}$, we have $L \subseteq A$. Thus, $L \subseteq \bigcap \mathcal{A}$. Consequently, $\bigcap \mathcal{A}$ is the greatest lower bound on $\mathcal{A}$. □

We introduce the equivalence relation $\sim$ on sets of abstract states, defined as follows. Let $A, B \in \mathcal{P}(\Sigma^\sharp)$. Then :

$$A \sim B \iff \gamma(A) = \gamma(B)$$

We define the quotient set $\mathcal{P}(\Sigma^\sharp)/\!\sim$ as the set of equivalence classes of $\mathcal{P}(\Sigma^\sharp)$ under the relation $\sim$.

In the following, we consider several functions on $\mathcal{P}(\Sigma^\sharp)$ and show that they *respect* the equivalence relation $\sim$, i.e., that they yield the same result for equivalent elements of $\mathcal{P}(\Sigma^\sharp)$. Consequently, these functions are well-defined on the quotient set $\mathcal{P}(\Sigma^\sharp)/\!\sim$.

We define the partial order $\sqsubseteq$ on $\mathcal{P}(\Sigma^\sharp)$ as follows. Let $A, B \in \mathcal{P}(\Sigma^\sharp)$. Then :

$$A \sqsubseteq B \iff \gamma(A) \subseteq \gamma(B)$$

Lemma 5. *The partial order* $\sqsubseteq$ *respects the equivalence relation* $\sim$*. That is, for all* $A, A', B, B' \in \mathcal{P}(\Sigma^\sharp)$ *satisfying* $A \sim A'$ *and* $B \sim B'$*, we have :*

$$A \sqsubseteq B \iff A' \sqsubseteq B'$$

Proof. By definition of $\sim$, we have $\gamma(A) = \gamma(A')$ and $\gamma(B) = \gamma(B')$. Thus, $\gamma(A) \subseteq \gamma(B)$ if and only if $\gamma(A') \subseteq \gamma(B')$. □

We define the join operator $\sqcup$ on $\mathcal{P}(\Sigma^\sharp)$ as follows. Let $A, B \in \mathcal{P}(\Sigma^\sharp)$. Then :

$$A \sqcup B \triangleq A \cup B$$

Lemma 6. *The join operator* $\sqcup$ *respects the equivalence relation* $\sim$*. That is, for all* $A, A', B, B' \in \mathcal{P}(\Sigma^\sharp)$ *satisfying* $A \sim A'$ *and* $B \sim B'$*, we have :*

$$A \sqcup B \sim A' \sqcup B'$$

Proof. By definition of $\sim$, we have $\gamma(A) = \gamma(A')$ and $\gamma(B) = \gamma(B')$. Thus, $\gamma(A \sqcup B) = \gamma(A \cup B) = \gamma(A) \cup \gamma(B) = \gamma(A') \cup \gamma(B') = \gamma(A' \cup B') = \gamma(A' \sqcup B')$. □

Lemma 7. *For any* $A, B \in \mathcal{P}(\Sigma^\sharp)$*, we have :*

$$\gamma(A \sqcup B) = \gamma(A) \cup \gamma(B)$$

Proof. By definition of $\sqcup$, we have : $\gamma(A \sqcup B) = \gamma(A \cup B)$. By definition of γ, we have : $\gamma(A \cup B) = \gamma(A) \cup \gamma(B)$. □

We define the meet operator $\sqcap$ on $\mathcal{P}(\Sigma^\sharp)$ as follows. Let $A, B \in \mathcal{P}(\Sigma^\sharp)$. Then :

$$A \sqcap B \triangleq \alpha(\gamma(A) \cap \gamma(B))$$

Lemma 8. *The meet operator* $\sqcap$ *respects the equivalence relation* $\sim$*. That is, for all* $A, A', B, B' \in \mathcal{P}(\Sigma^\sharp)$ *satisfying* $A \sim A'$ *and* $B \sim B'$*, we have :*

$$A \sqcap B \sim A' \sqcap B'$$

Proof. By definition of $\sim$, we have $\gamma(A) = \gamma(A')$ and $\gamma(B) = \gamma(B')$. Thus, $\gamma(A \sqcap B) = \gamma(\alpha(\gamma(A) \cap \gamma(B))) = \gamma(\alpha(\gamma(A') \cap \gamma(B'))) = \gamma(A' \sqcap B')$. □

Lemma 9. *For any* $A, B \in \mathcal{P}(\Sigma^\sharp)$*, we have :*

$$\gamma(A \sqcap B) = \gamma(A) \cap \gamma(B)$$

Proof. By definition of $\sqcap$, we have : $\gamma(A \sqcap B) = \gamma(\alpha(\gamma(A) \cap \gamma(B)))$. We show $\gamma(\alpha(\gamma(A) \cap \gamma(B))) = \gamma(A) \cap \gamma(B)$ by contradiction.

Suppose $\gamma(\alpha(\gamma(A) \cap \gamma(B))) \neq \gamma(A) \cap \gamma(B)$. This implies the existence of $b \in \mathcal{B}$ and $\sigma^\flat, \sigma^{\flat\prime} \in \Sigma^\flat$ such that (i) $\sigma^\flat$ and $\sigma^{\flat\prime}$ are identical except at byte b, (ii) $\sigma^\flat[b].\mathsf{taint} = \sigma^{\flat\prime}[b].\mathsf{taint} = \mathsf{Sec}$, (iii) $\sigma^\flat[b].\mathsf{val} \neq \sigma^{\flat\prime}[b].\mathsf{val}$, (iv) $\sigma^\flat \in \gamma(A) \cap \gamma(B)$, and (v) $\sigma^{\flat\prime} \notin \gamma(\alpha(\gamma(A) \cap \gamma(B)))$.

However, by definition of $\bar{\gamma}$, we have $\sigma^\flat \in \gamma(A) \iff \sigma^{\flat\prime} \in \gamma(A)$. Similarly, we have $\sigma^\flat \in \gamma(B) \iff \sigma^{\flat\prime} \in \gamma(B)$. Thus, $\sigma^\flat \in \gamma(A) \cap \gamma(B) \iff \sigma^{\flat\prime} \in \gamma(A) \cap \gamma(B)$, which is a contradiction with (iv) and (v).

□

Lemma 10. $\mathcal{P}(\Sigma^\sharp)/\!\sim$ *together with* $\sqsubseteq$ *is a complete lattice with join operator* $\sqcup$ *and meet operator* $\sqcap$*.*

Proof. First, note that, by Lemmas 5, 6, and 8, the partial order $\sqsubseteq$, the join operator $\sqcup$, and the meet operator $\sqcap$ are well-defined on the quotient set $\mathcal{P}(\Sigma^\sharp)/\!\sim$.

Additionally, consider an arbitrary subset $\mathcal{A} \subseteq \mathcal{P}(\Sigma^\sharp)/\!\sim$. We show that (i) $\bigsqcup \mathcal{A}$ is an upper bound on $\mathcal{A}$, (ii) it is the least such upper bound, (iii) $\bigsqcap \mathcal{A}$ is a lower bound on $\mathcal{A}$, and (iv) it is the greatest such lower bound.

(i). By Lemma 7, we have $\gamma(\bigsqcup \mathcal{A}) = \bigcup_{A \in \mathcal{A}} \gamma(A)$. Consequently, for all $A \in \mathcal{A}$, we have $\gamma(A) \subseteq \gamma(\bigsqcup \mathcal{A})$ and thus $A \sqsubseteq \bigsqcup \mathcal{A}$.

(ii). Let U be an arbitrary upper bound on $\mathcal{A}$. We show that $\bigsqcup \mathcal{A} \sqsubseteq U$. By definition of $\sqsubseteq$, for all $A \in \mathcal{A}$, we have $\gamma(A) \subseteq \gamma(U)$. Thus, $\gamma(\bigsqcup \mathcal{A}) = \bigcup_{A \in \mathcal{A}} \gamma(A) \subseteq \gamma(U)$ and $\bigsqcup \mathcal{A} \sqsubseteq U$.

(iii). By Lemma 9, we have $\gamma(\bigsqcap \mathcal{A}) = \bigcap_{A \in \mathcal{A}} \gamma(A)$. Consequently, for all $A \in \mathcal{A}$, we have $\gamma(\bigsqcap \mathcal{A}) \subseteq \gamma(A)$ and thus $\bigsqcap \mathcal{A} \sqsubseteq A$.

(iv). Let L be an arbitrary lower bound on $\mathcal{A}$. We show that $L \sqsubseteq \bigsqcap \mathcal{A}$. By definition of $\sqsubseteq$, for all $A \in \mathcal{A}$, we have $\gamma(L) \subseteq \gamma(A)$. Thus, $\gamma(L) \subseteq \bigcap_{A \in \mathcal{A}} \gamma(A) = \gamma(\bigsqcap \mathcal{A})$ and $L \sqsubseteq \bigsqcap \mathcal{A}$.

□

Lemma 11. *For any $A \in \mathcal{P}(\Sigma^\flat)$, we have :*

$$\gamma(\alpha(A)) \supseteq A$$

Proof. For an arbitrary $\sigma^\flat \in \Sigma^\flat$, we show that if $\sigma^\flat \in A$, then $\sigma^\flat \in \gamma(\alpha(A))$.

By definition of α, $\sigma^\flat \in A$ implies $\bar{\alpha}(\sigma^\flat) = \alpha(A)$. Recall that $\bar{\alpha}$ maps each non-secret byte value to the corresponding concrete byte value $v \in \mathbb{B}$, and each secret byte value to the taint red.

Now, by definition of γ, we have $\bar{\gamma}(\bar{\alpha}(\sigma^\flat)) \subseteq \gamma(\alpha(A))$. Because $\bar{\gamma}$ maps each concrete byte value $v \in \mathbb{B}$ to the corresponding non-secret byte value v, and red to any secret byte value, we have $\sigma^\flat \in \bar{\gamma}(\bar{\alpha}(\sigma^\flat))$ and thus $\sigma^\flat \in \gamma(\alpha(A))$.

□

Lemma 12. *The functions $\alpha : \mathcal{P}(\Sigma^\flat) \to \mathcal{P}(\Sigma^\sharp)/\!\sim$ and $\gamma : \mathcal{P}(\Sigma^\sharp)/\!\sim \to \mathcal{P}(\Sigma^\flat)$ form a Galois connection. That is, for any $A \in \mathcal{P}(\Sigma^\flat)$ and $B \in \mathcal{P}(\Sigma^\sharp)/\!\sim$, we have :*

$$\alpha(A) \sqsubseteq B \iff A \subseteq \gamma(B)$$

Proof. We show both directions of the equivalence.

($\Rightarrow$). Suppose $\alpha(A) \sqsubseteq B$. By definition of $\sqsubseteq$, we have $\gamma(\alpha(A)) \subseteq \gamma(B)$. By Lemma 11, we have $A \subseteq \gamma(\alpha(A))$. Thus, by transitivity of $\subseteq$, we have $A \subseteq \gamma(B)$.

($\Leftarrow$). Suppose $A \subseteq \gamma(B)$. By Lemma 11, we have $A \subseteq \gamma(\alpha(A))$. By transitivity of $\subseteq$, we have $\gamma(\alpha(A)) \subseteq \gamma(B)$. By definition of $\sqsubseteq$, we have $\alpha(A) \sqsubseteq B$.

□

Lemma 13. *The functions $\mathcal{A} : \mathcal{M}^\flat \to \mathcal{M}^\sharp$ and $\Gamma : \mathcal{M}^\sharp \to \mathcal{M}^\flat$ (defined in Section 6) form a Galois connection. That is, for any $\mu^\flat \in \mathcal{M}^\flat$ and $\mu^\sharp \in \mathcal{M}^\sharp$, we have :*

$$\mathcal{A}(\mu^\flat) \sqsubseteq \mu^\sharp \iff \mu^\flat \subseteq \Gamma(\mu^\sharp)$$

Where the orders on $\mathcal{M}^\flat$ and $\mathcal{M}^\sharp$ are defined as the pointwise extension of the orders on $\mathcal{P}(\Sigma^\flat)$ and $\mathcal{P}(\Sigma^\sharp)/\!\sim$, respectively.

Proof. We show both directions of the equivalence.

($\Rightarrow$). Suppose $\mathcal{A}(\mu^\flat) \sqsubseteq \mu^\sharp$. By definition of the pointwise extension of $\sqsubseteq$, for all program points $\ell \in \mathsf{points}$, we have $\alpha(\mu^\flat(\ell)) \sqsubseteq \mu^\sharp(\ell)$. By Lemma 12, for all program points $\ell \in \mathsf{points}$, we have $\mu^\flat(\ell) \subseteq \gamma(\mu^\sharp(\ell))$. Thus, by definition of the pointwise extension of $\subseteq$, we have $\mu^\flat \subseteq \Gamma(\mu^\sharp)$.

($\Leftarrow$). Suppose $\mu^\flat \subseteq \Gamma(\mu^\sharp)$. By definition of the pointwise extension of $\subseteq$, for all program points $\ell \in \mathsf{points}$, we have $\mu^\flat(\ell) \subseteq \gamma(\mu^\sharp(\ell))$. By Lemma 12, for all program points $\ell \in \mathsf{points}$, we have $\alpha(\mu^\flat(\ell)) \sqsubseteq \mu^\sharp(\ell)$. Thus, by definition of the pointwise extension of $\sqsubseteq$, we have $\mathcal{A}(\mu^\flat) \sqsubseteq \mu^\sharp$. □

Lemma 14. *The abstract semantics of compute instructions is sound with respect to the concrete semantics. That is, for any $A \in \mathcal{P}(\Sigma^\sharp)$ and any compute instruction $\langle \mathsf{compute}, f, I, o, \rangle$, we have :*

$$\alpha \circ [\![\langle \mathsf{compute}, f, I, o \rangle]\!]^\flat \circ \gamma(A) \sqsubseteq [\![\langle \mathsf{compute}, f, I, o \rangle]\!]^\sharp(A)$$

Proof. Consider an arbitrary $\sigma^\flat \in \gamma(A)$. Its existence implies that (i) there is $\sigma^\sharp \in A$ such that $\sigma^\flat \in \bar{\gamma}(\sigma^\sharp)$ and (ii) there is $\sigma^{\flat\prime} \in [\![\langle \mathsf{compute}, f, I, o \rangle]\!]^\flat \circ \gamma(A)$ such that $\sigma^{\flat\prime} = [\![\langle \mathsf{compute}, f, I, o \rangle]\!]^\flat(\sigma^\flat)$. Moreover, $\sigma^\sharp \in A$ implies that there is $\sigma^{\sharp\prime} \in [\![\langle \mathsf{compute}, f, I, o \rangle]\!]^\sharp(A)$ such that $\sigma^{\sharp\prime} = [\![\langle \mathsf{compute}, f, I, o \rangle]\!]^\sharp(\sigma^\sharp)$.

We need to show that $\sigma^{\flat\prime} \in \bar{\gamma}(\sigma^{\sharp\prime})$, that is, for any architectural byte $b \in \mathcal{B}$:

$$\begin{cases} \sigma^{\flat\prime}[b].\mathsf{taint} = \mathsf{Sec} & \text{if } \sigma^{\sharp\prime}[b] = \mathsf{red} \\ \text{no constraint} & \text{if } \sigma^{\sharp\prime}[b] = \mathsf{orange} \\ \sigma^{\flat\prime}[b].\mathsf{taint} = \mathsf{Pub} & \text{if } \sigma^{\sharp\prime}[b] = \mathsf{green} \\ \sigma^{\flat\prime}[b].\mathsf{taint} = \mathsf{Pub}, \sigma^{\flat\prime}[b].\mathsf{val} = v & \text{if } \sigma^{\sharp\prime}[b] = v \in \mathbb{B} \end{cases} \tag{1}$$

Condition 1 is clearly satisfied for $b \neq o$, because we have $\sigma^\flat \in \bar{\gamma}(\sigma^\sharp)$ and both the concrete and abstract semantics of compute instructions do not modify bytes other than the output byte o.

We now discuss the case $b = o$.

Suppose that there is $b' \in I$ such that $\sigma^\sharp[b'] = \mathsf{red}$. This implies (by definition of $\bar{\gamma}$) that $\sigma^\flat[b'].\mathsf{taint} = \mathsf{Sec}$ and thus (by definition of the concrete semantics) $\sigma^{\flat\prime}[o].\mathsf{taint} = \mathsf{Sec}$. Now, $\sigma^\sharp[b'] = \mathsf{red}$ implies (by definition of the abstract semantics) that $\sigma^{\sharp\prime}[o] = \mathsf{red}$. Thus, Condition 1 is satisfied.

Suppose now that there exists no $b' \in I$ such that $\sigma^\sharp[b'] = \mathsf{red}$, but there is $b' \in I$ such that $\sigma^\sharp[b'] = \mathsf{orange}$. This implies (by definition of the abstract semantics) that $\sigma^{\sharp\prime}[o] = \mathsf{orange}$. Thus, Condition 1 is satisfied.

Suppose now that there exists no $b' \in I$ such that $\sigma^\sharp[b'] \in \{\mathsf{red}, \mathsf{orange}\}$, but there is $b' \in I$ such that $\sigma^\sharp[b'] = \mathsf{green}$. This implies (by definition of $\bar{\gamma}$) that, for all $b'' \in I$, $\sigma^\flat[b''].\mathsf{taint} = \mathsf{Pub}$. Thus (by definition of the concrete semantics) $\sigma^{\flat\prime}[o].\mathsf{taint} = \mathsf{Pub}$. This also implies (by definition of the abstract semantics) that $\sigma^{\sharp\prime}[o] = \mathsf{green}$. Thus, Condition 1 is satisfied.

Finally, suppose that for all $b' \in I$, $\sigma^\sharp[b'] \in \mathbb{B}$. This implies (by definition of $\bar{\gamma}$) that, for all $b' \in I$, $\sigma^\flat[b'].\mathsf{taint} = \mathsf{Pub}$ and $\sigma^\flat[b'].\mathsf{val} = \sigma^\sharp[b']$. Thus (by definition of the concrete semantics) $\sigma^{\flat\prime}[o].\mathsf{taint} = \mathsf{Pub}$ and $\sigma^{\flat\prime}[o].\mathsf{val} = f(\sigma^\flat[I].\mathsf{val})$. This also implies (by definition of the abstract semantics) that $\sigma^{\sharp\prime}[o] = f(\sigma^\flat[I].\mathsf{val})$. Thus, Condition 1 is satisfied.

□

Lemma 15. *The abstract semantics of load instructions is sound with respect to the concrete semantics. That is, for any $A \in \mathcal{P}(\Sigma^\sharp)$ and any load instruction $\langle \mathsf{load}, f, S, d \rangle$, we have :*

$$\alpha \circ [\![\langle \mathsf{load}, f, S, d \rangle]\!]^\flat \circ \gamma(A) \sqsubseteq [\![\langle \mathsf{load}, f, S, d \rangle]\!]^\sharp(A)$$

Proof. Consider an arbitrary $\sigma^\flat \in \gamma(A)$. Its existence implies that (i) there is $\sigma^\sharp \in A$ such that $\sigma^\flat \in \bar{\gamma}(\sigma^\sharp)$ and (ii) there is $\sigma^{\flat\prime} \in [\![\langle \mathsf{load}, f, S, d \rangle]\!]^\flat \circ \gamma(A)$ such that

$\sigma^{\flat\prime} = [\![\langle \mathsf{load}, f, S, d\rangle]\!]^\flat(\sigma^\flat)$. Moreover, $\sigma^\sharp \in A$ implies that there is $\sigma^{\sharp\prime} \in [\![\langle \mathsf{load}, f, S, d\rangle]\!]^\sharp(A)$ such that $\sigma^{\sharp\prime} = [\![\langle \mathsf{load}, f, S, d\rangle]\!]^\sharp(\sigma^\sharp)$.

We need to show that $\sigma^{\flat\prime} \in \bar{\gamma}(\sigma^{\sharp\prime})$, that is, for any architectural byte $b \in \mathcal{B}$, condition 1 holds.

Condition 1 is clearly satisfied for $b \neq d$, because we have $\sigma^\flat \in \bar{\gamma}(\sigma^\sharp)$ and both the concrete and abstract semantics of load instructions do not modify bytes other than the destination byte d.

We now discuss the case $b = d$.

Suppose that there is $b' \in S$ such that $\sigma^\sharp[b'] = \mathsf{red}$. This implies (by definition of $\bar{\gamma}$) that $\sigma^\flat[b'].\mathsf{taint} = \mathsf{Sec}$ and thus (by definition of the concrete semantics) $\sigma^{\flat\prime}[d].\mathsf{taint} = \mathsf{Sec}$. Now, $\sigma^\sharp[b'] = \mathsf{red}$ implies (by definition of the abstract semantics) that $\sigma^{\sharp\prime}[d] = \mathsf{red}$. Thus, Condition 1 is satisfied.

Suppose now that there exists no $b' \in S$ such that $\sigma^\sharp[b'] = \mathsf{red}$, but there is $b' \in S$ such that $\sigma^\sharp[b'] \in \{\mathsf{orange}, \mathsf{green}\}$. This implies (by definition of the abstract semantics) that $\sigma^{\sharp\prime}[d] = \mathsf{orange}$. Thus, Condition 1 is satisfied.

Finally, suppose that for all $b' \in S$, $\sigma^\sharp[b'] \in \mathbb{B}$. This implies (by definition of $\bar{\gamma}$) that, for all $b' \in S$, $\sigma^\flat[b'].\mathsf{taint} = \mathsf{Pub}$ and $\sigma^\flat[b'].\mathsf{val} = \sigma^\sharp[b']$. Thus (by definition of the concrete semantics) $\sigma^{\flat\prime}[d] = \sigma^\flat[s]$, where $s = f(\sigma^\flat[S].\mathsf{val})$ is the source architectural byte of the load operation. This also implies (by definition of the abstract semantics) that $\sigma^{\sharp\prime}[d] = \sigma^\sharp[s]$. As $\sigma^\flat \in \bar{\gamma}(\sigma^\sharp)$, condition 1 is satisfied.

□

Lemma 16. *The abstract semantics of store instructions is sound with respect to the concrete semantics. That is, for any $A \in \mathcal{P}(\Sigma^\sharp)$ and any store instruction $\langle \mathsf{store}, f, D, s\rangle$, we have :*

$$\alpha \circ [\![\langle \mathsf{store}, f, D, s\rangle]\!]^\flat \circ \gamma(A) \sqsubseteq [\![\langle \mathsf{store}, f, D, s\rangle]\!]^\sharp(A)$$

Proof. Consider an arbitrary $\sigma^\flat \in \gamma(A)$. Its existence implies that (i) there is $\sigma^\sharp \in A$ such that $\sigma^\flat \in \bar{\gamma}(\sigma^\sharp)$ and (ii) there is $\sigma^{\flat\prime} \in [\![\langle \mathsf{store}, f, D, s\rangle]\!]^\flat \circ \gamma(A)$ such that $\sigma^{\flat\prime} = [\![\langle \mathsf{store}, f, D, s\rangle]\!]^\flat(\sigma^\flat)$. Moreover, $\sigma^\sharp \in A$ implies that there is $\sigma^{\sharp\prime} \in [\![\langle \mathsf{store}, f, D, s\rangle]\!]^\sharp(A)$ such that $\sigma^{\sharp\prime} = [\![\langle \mathsf{store}, f, D, s\rangle]\!]^\sharp(\sigma^\sharp)$. We need to show that $\sigma^{\flat\prime} \in \bar{\gamma}(\sigma^{\sharp\prime})$, that is, for any architectural byte $b \in \mathcal{B}$, condition 1 holds.

Suppose that there exists $b \in D$ such that $\sigma^\sharp[b] \in \{\mathsf{red}, \mathsf{orange}, \mathsf{green}\}$. This implies (by definition of the abstract semantics) that $\sigma^{\sharp\prime} = \bot^\sharp$. Thus, Condition 1 is satisfied.

Suppose now that for all $b \in D$, $\sigma^\sharp[b] \in \mathbb{B}$. This implies (by definition of $\bar{\gamma}$) that, for all $b \in D$, $\sigma^\flat[b].\mathsf{taint} = \mathsf{Pub}$ and $\sigma^\flat[b].\mathsf{val} = \sigma^\sharp[b]$. Thus (by definition of the concrete semantics) $\sigma^{\flat\prime} = \sigma^\flat[d \mapsto \sigma^\flat[s]]$, where $d = f(\sigma^\flat[D].\mathsf{val})$ is the destination architectural byte of the store operation. This also implies (by definition of the abstract semantics) that $\sigma^{\sharp\prime} = \sigma^\sharp[d \mapsto \sigma^\sharp[s]]$. As $\sigma^\flat \in \bar{\gamma}(\sigma^\sharp)$, condition 1 is satisfied. □

Lemma 17. *The abstract semantics of branch instructions is sound with respect to the concrete semantics. That is, for any $A \in \mathcal{P}(\Sigma^\sharp)$ and any branch instruction $\langle \mathsf{branch}, g, I, T, F\rangle$, we have :*

$$\alpha \circ [\![\langle \mathsf{branch}, g, I, T, F\rangle]\!]^\flat \circ \gamma(A) \sqsubseteq [\![\langle \mathsf{branch}, g, I, T, F\rangle]\!]^\sharp(A)$$

Proof. Consider an arbitrary $\sigma^\flat \in \gamma(A)$. Its existence implies that there is $\sigma^\sharp \in A$ such that $\sigma^\flat \in \bar{\gamma}(\sigma^\sharp)$. We need to show that

$$\begin{aligned} &\sigma^\flat \in [\![\langle \mathsf{branch}, g, I, T, F\rangle]\!]^\flat \circ \gamma(A) \implies \\ &\sigma^\sharp \in [\![\langle \mathsf{branch}, g, I, T, F\rangle]\!]^\sharp(A) \end{aligned} \tag{2}$$

Suppose that there exists $b' \in I$ such that $\sigma^\sharp[b'] \in \{\mathsf{red}, \mathsf{orange}, \mathsf{green}\}$. By definition of the abstract semantics of branch instructions, we have $\sigma^\sharp \in [\![\langle \mathsf{branch}, g, I, T, F\rangle]\!]^\sharp(A)$. Thus, Condition 2 is satisfied.

Suppose now that for all $b \in I$, $\sigma^\sharp[b] \in \mathbb{B}$. By definition of $\bar{\gamma}$, this implies that, for all $b \in I$, $\sigma^\flat[b].\mathsf{val} = \sigma^\sharp[b]$. By definition of the concrete semantics of branch instructions, we have $\sigma^\flat \in [\![\langle \mathsf{branch}, g, I, T, F\rangle]\!]^\flat \circ \gamma(A)$ if and only if $g(\sigma^\flat[I].\mathsf{val}) = \mathsf{true}$. Also, by definition of the abstract semantics of branch instructions, we have $\sigma^\sharp \in [\![\langle \mathsf{branch}, g, I, T, F\rangle]\!]^\sharp(A)$ if and only if $g(\sigma^\sharp[I].\mathsf{val}) = \mathsf{true}$. Thus, Condition 2 is satisfied. □

Lemma 18. *For any pair of program points (ℓ, ℓ') and any $A \in \mathcal{P}(\Sigma^\sharp)$, we have :*

$$\alpha \circ [\![(\ell, \ell')]\!]^\flat \circ \gamma(A) \sqsubseteq [\![(\ell, \ell')]\!]^\sharp(A)$$

Proof. This follows directly from Lemmas 14, 15, 16, and 17, by case analysis on the type of instruction at program point ℓ. □

Lemma 19. *For any $\mu^\sharp \in \mathcal{M}^\sharp$, we have :*

$$\mathcal{A} \circ \mathcal{F}^\flat \circ \Gamma(\mu^\sharp) \sqsubseteq \mathcal{F}^\sharp(\mu^\sharp)$$

Proof. Consider an arbitrary program point ℓ. By definition of the pointwise extension of $\sqsubseteq$, we need to show that

$$(\mathcal{A} \circ \mathcal{F}^\flat \circ \Gamma(\mu^\sharp))[\ell] \sqsubseteq \mathcal{F}^\sharp(\mu^\sharp)[\ell]$$

Using the definitions of $\mathcal{A}$, Γ, $\mathcal{F}^\flat$, $\mathcal{F}^\sharp$, and Lemma 18, we have

$$\begin{aligned} &(\mathcal{A} \circ \mathcal{F}^\flat \circ \Gamma(\mu^\sharp))[\ell] \\ =&\alpha \circ \gamma(\mu^\sharp[\ell]) \cup \bigcup_{\ell' \in \mathsf{points}} \alpha \circ [\![(\ell', \ell)]\!]^\flat \circ \gamma(\mu^\sharp[\ell']) \\ \sqsubseteq&\mu^\sharp[\ell] \cup \bigcup_{\ell' \in \mathsf{points}} [\![(\ell', \ell)]\!]^\sharp(\mu^\sharp[\ell']) \\ =&\mathcal{F}^\sharp(\mu^\sharp)[\ell] \end{aligned}$$

□

Lemma 20. *The least fixed point of the abstract transformer overapproximates the least fixed point of the concrete transformer. That is :*

$$\mathcal{A}(\mathit{lfp}(\mathcal{F}^\flat)) \sqsubseteq \mathit{lfp}(\mathcal{F}^\sharp)$$

Proof. By Proposition 24 of [36] together with Lemma 13, we have :

$$\mathcal{A}(\mathrm{lfp}(\mathcal{F}^\flat)) \sqsubseteq \mathrm{lfp}(\mathcal{A} \circ \mathcal{F}^\flat \circ \Gamma)$$

By Proposition 25 of [36] together with Lemma 19, we have :

$$\mathrm{lfp}(\mathcal{A} \circ \mathcal{F}^\flat \circ \Gamma) \sqsubseteq \mathrm{lfp}(\mathcal{F}^\sharp)$$

By transitivity of $\sqsubseteq$, we obtain the desired result. □

Lemma 21. *For any $\sigma_1^\sharp, \sigma_2^\sharp \in \Sigma^\sharp$, we have :*

$$\sigma_1^\sharp \sqcup \sigma_2^\sharp \sqsubseteq \sigma_1^\sharp \wedge \sigma_2^\sharp$$

Where $\wedge$ is the join operator defined in Section 5.3.

Proof. We need to show that, for any architectural byte $b \in \mathcal{B}$, we have :

$$(\bar{\gamma}(\sigma_1^\sharp) \cup \bar{\gamma}(\sigma_2^\sharp))[b] \subseteq \bar{\gamma}(\sigma_1^\sharp \wedge \sigma_2^\sharp)[b]$$

Without loss of generality, we prove :

$$\bar{\gamma}(\sigma_1^\sharp)[b] \subseteq \bar{\gamma}(\sigma_1^\sharp \wedge \sigma_2^\sharp)[b] \tag{3}$$

We first consider the case $\sigma_1^\sharp[b] = \sigma_2^\sharp[b]$. Then, by definition of $\wedge$, we have $(\sigma_1^\sharp \wedge \sigma_2^\sharp)[b] = \sigma_1^\sharp[b]$. Thus, Equation 3 is satisfied.

Now consider the case $\sigma_1^\sharp[b] = \mathsf{orange}$ (for any value of $\sigma_2^\sharp[b]$). Then, by definition of $\wedge$, we have $(\sigma_1^\sharp \wedge \sigma_2^\sharp)[b] = \mathsf{orange}$. Thus, Equation 3 is satisfied.

Now, consider the case $\sigma_1^\sharp[b] = \mathsf{red}$ and $\sigma_2^\sharp[b] \in \{\mathsf{green}\} \cup \mathbb{B}$. Then, by definition of $\wedge$, we have $(\sigma_1^\sharp \wedge \sigma_2^\sharp)[b] = \mathsf{orange}$. Thus, by definition of $\bar{\gamma}$, Equation 3 is satisfied.

Now, consider the case $\sigma_1^\sharp[b] = \mathsf{green}$ and $\sigma_2^\sharp[b] \in \mathbb{B}$. Then, by definition of $\wedge$, we have $(\sigma_1^\sharp \wedge \sigma_2^\sharp)[b] = \mathsf{green}$. Thus, Equation 3 is satisfied.

Finally, consider the case $\sigma_1^\sharp[b] = v_1 \in \mathbb{B}$ and $\sigma_2^\sharp[b] = v_2 \in \mathbb{B}$ with $v_1 \neq v_2$. Then, by definition of $\wedge$, we have $(\sigma_1^\sharp \wedge \sigma_2^\sharp)[b] = \mathsf{green}$. Thus, by definition of $\bar{\gamma}$, Equation 3 is satisfied.

□

Lemma 22. *The operator ∇_τ defined in 5.3 is a widening operator, in the sense of Proposition 31 of [36].*

Proof. In Proposition 31 of [36], two conditions are given for an operator to be a widening operator.

The first condition requires that, for any ascending chain $(A_n)_{n\geq 0}$ in $\mathcal{M}^\sharp$, the sequence defined by $B_0 = A_0$ and $B_{n+1} = \nabla_\tau(B_n, A_{n+1})$ is not strictly increasing. As $\mathcal{M}^\sharp$ is a finite lattice, this condition is trivially satisfied.

For the second condition, we need to show that for any $\mu_1^\sharp, \mu_2^\sharp \in \mathcal{M}^\sharp$, we have $\mu_1^\sharp \sqcup \mu_2^\sharp \sqsubseteq \mu_1^\sharp \nabla_\tau \mu_2^\sharp$, that is, for any program point ℓ, we have :

$$(\mu_1^\sharp \sqcup \mu_2^\sharp)[\ell] \sqsubseteq (\mu_1^\sharp \nabla_\tau \mu_2^\sharp)[\ell] \tag{4}$$

Suppose $|(\mu_1^\sharp \sqcup \mu_2^\sharp)[\ell]| < \tau$. Then, by definition of ∇_τ, we have $(\mu_1^\sharp \nabla_\tau \mu_2^\sharp)[\ell] = (\mu_1^\sharp \sqcup \mu_2^\sharp)[\ell]$. Thus, Equation 4 is satisfied.

Suppose now $|(\mu_1^\sharp \sqcup \mu_2^\sharp)[\ell]| \geq \tau$. Then, by definition of ∇_τ, we have $(\mu_1^\sharp \nabla_\tau \mu_2^\sharp)[\ell] = \bigwedge(\mu_1^\sharp[\ell] \sqcup \mu_2^\sharp[\ell])$. By Lemma 21, we have $(\mu_1^\sharp \sqcup \mu_2^\sharp)[\ell] \sqsubseteq \bigwedge(\mu_1^\sharp[\ell] \sqcup \mu_2^\sharp[\ell])$. Thus, Equation 4 is satisfied. □

Lemma 23. *The result $\mu_*^\sharp$ of the analysis, as defined in Section 5.3, overapproximates the least fixed point of the abstract transformer. That is :*

$$lfp(\mathcal{F}^\sharp) \sqsubseteq \mu_*^\sharp$$

Proof. This follows directly from Proposition 31 of [36] together with Lemma 22. □

Theorem. *The analysis is sound with respect to the concrete semantics. That is, the result $\mu_*^\sharp$ of the analysis overapproximates the least fixed point of the concrete transformer. Formally, we have :*

$$lfp(\mathcal{F}^\flat) \subseteq \Gamma(\mu_*^\sharp)$$

Proof. From Lemmas 19 and 23, by transitivity of $\sqsubseteq$, we have :

$$\mathcal{A}(\mathrm{lfp}(\mathcal{F}^\flat)) \sqsubseteq \mu_*^\sharp$$

By Lemma 13, we have the desired result. □

A.2. Corollaries

We state the proof of Corollary 2 stated in Section 6.

Proof. Suppose that there exists a concrete state $\sigma^\flat \in \mathrm{lfp}(\mathcal{F}^\flat)[\ell]$ and an architectural byte $b \in \mathcal{B}$ such that $\sigma^\flat[b].\mathsf{taint} = \mathsf{Sec}$.

Then, by Theorem 1, we have that $\sigma^\flat \in \Gamma(\mu_*^\sharp)[\ell]$.

By definition of the concretization function, this means that there exists an abstract state $\sigma^\sharp \in \mu_*^\sharp[\ell]$ such that $\sigma^\sharp[b]$ is either red or orange. □

We state the proof of Corollary 3 stated in Section 6.

Proof. Suppose that there exists an architectural byte $b \in \mathcal{B}$ such that for every abstract state $\sigma^\sharp \in \mu_*^\sharp[\ell]$, we have $\sigma^\sharp[b] = \mathsf{red}$.

By Theorem 1, we have that $\mathrm{lfp}(\mathcal{F}^\flat)[\ell] \subseteq \Gamma(\mu_*^\sharp)[\ell]$.

By definition of the concretization function, we have that for every concrete state $\sigma^\flat \in \Gamma(\mu_*^\sharp)[\ell]$, we have $\sigma^\flat[b].\mathsf{taint} = \mathsf{Sec}$, and thus the same holds for every concrete state $\sigma^\flat \in \mathrm{lfp}(\mathcal{F}^\flat)[\ell]$. □