Precise Detection of Side-Channel Vulnerabilities using Quantitative Cartesian Hoare Logic

Jia Chen
University of Texas at Austin
Austin, Texas
jchen@cs.utexas.edu

Yu Feng
University of Texas at Austin
Austin, Texas
yufeng@cs.utexas.edu

Isil Dillig
University of Texas at Austin
Austin, Texas
isil@cs.utexas.edu

ABSTRACT

This paper presents THEMIS, an end-to-end static analysis tool for finding resource-usage side-channel vulnerabilities in Java applications. We introduce the notion of *ϵ-bounded non-interference*, a variant and relaxation of Goguen and Meseguer's well-known non-interference principle. We then present *Quantitative Cartesian Hoare Logic (QCHL)*, a program logic for verifying ϵ-bounded non-interference. Our tool, THEMIS, combines automated reasoning in CHL with lightweight static taint analysis to improve scalability. We evaluate THEMIS on well known Java applications and demonstrate that THEMIS can find unknown side-channel vulnerabilities in widely-used programs. We also show that THEMIS can verify the absence of vulnerabilities in repaired versions of vulnerable programs and that THEMIS compares favorably against BLAZER, a state-of-the-art static analysis tool for finding timing side channels in Java applications.

CCS CONCEPTS

• **Security and privacy → Logic and verification**; **Software security engineering**; • **Theory of computation → Automated reasoning**;

KEYWORDS

vulnerability detection; side channels; static analysis; verification

1 INTRODUCTION

Side channel attacks allow an adversary to infer security-sensitive information of a system by observing its external behavior. For instance, in the case of *timing side channels*, an attacker can learn properties of a secret (e.g., user's password) by observing the time it takes to perform some operation (e.g., password validation). Similarly, *compression side channel* attacks allow adversaries to glean confidential information merely by observing the size of the compressed data (e.g., HTTP response). Numerous research papers and several real-world exploits have shown that such side channel attacks are both practical and harmful. For instance, side channels have been used to infer confidential data involving user accounts [26, 38], cryptographic keys [5, 19, 45], geographic locations [62], and medical data [22]. Recent work has shown that side channels can also lead to information leakage in cyber-physical systems [23].

Side channel attacks are made possible due to the presence of an underlying *vulnerability* in the system. For example, timing attacks are feasible because the application exhibits different timing characteristics based on some properties of the secret. In general, the most robust defense against side-channel attacks is to eradicate the underlying vulnerabilities by ensuring that the resource usage of the program (time, space, power etc.) does not vary with respect to the secret. Unfortunately, it can be challenging to write programs in a way that follows this discipline, and side-channel vulnerabilities continue to be uncovered on a regular basis in real-world security-critical systems [5, 19, 36, 68].

Our goal in this paper is to help programmers develop side-channel-free applications by automatically analyzing correlations between variations in resource usage and differences in security-sensitive data. In particular, given a program P and a "tolerable" resource deviation ϵ, we would like to *verify* that the resource usage of P does not vary by more than ϵ no matter what the value of the secret. Following the terminology of Goguen and Meseguer [34], we refer to this property as *ϵ-bounded non-interference*. Intuitively, a program that violates ϵ-bounded non-interference for even large values of ϵ exhibits significant secret-induced differences in resource usage.

The problem of verifying ϵ-bounded non-interference is challenging for at least two reasons: First, the property that we would like to verify is an instance of a so-called *2-safety property* [66] that requires reasoning about all possible interactions between *pairs* of program executions. Said differently, a *witness* to the violation of ϵ-bounded interference consists of a *pair* of program runs on two different secrets. Unlike standard safety properties that have been well-studied in verification literature and for which many automated tools exist, checking 2-safety is known to be a much harder problem. Furthermore, while checking 2-safety can in principle be reduced to standard safety via so-called *product construction* [12, 14] such a transformation either causes a blow-up in program size [12], thereby resulting in scalability problems, or yields a program that is practically very difficult to verify [14].

In this work, we solve these challenges by combining relatively lightweight static taint analysis with more precise *relational verification* techniques for reasoning about *k-safety* (i.e., properties that concern interactions between k program runs). Specifically, our approach first uses taint information to identify so-called *hot spots*, which are program fragments that have the potential to exhibit a secret-induced imbalance in resource usage. We then use much

CCS '17, October 30-November 3, 2017, Dallas, TX, USA

ACM ISBN 978-1-4503-4946-8/17/10…$15.00
https://doi.org/10.1145/3133956.3134058

more precise relational reasoning techniques to automatically verify that such hot spots do not violate ϵ-bounded non-interference.

At the core of our technique is a new program logic called *Quantitative Cartesian Hoare Logic (QCHL)* for verifying the ϵ-bounded non-interference property. QCHL leverages recent advances in relational verification by building on top of *Cartesian Hoare Logic (CHL)* [65] for verifying k-safety properties. Specifically, QCHL allows us to prove triples of the form $\langle \phi \rangle \ S \ \langle \psi \rangle$, where S is a program fragment and ϕ, ψ are first-order formulas that relate the program's resource usage (e.g., execution time) between an arbitrary pair of program runs. Starting with the precondition that two runs have the same public input but different values of the secret, QCHL proof rules allow us to prove that the difference in resource usage is bounded from above by some (user-provided) constant ϵ. Similar to CHL, our QCHL logic allows effective relational verification by symbolically executing two copies of the program in lockstep. However, QCHL differs from CHL in that it reasons about the program's resource usage behavior and exploits domain-specific assumptions to improve both analysis precision and scalability. Furthermore, since the QCHL proof rules are deterministic (modulo an oracle for finding loop invariants and proving standard Hoare triples), QCHL immediately lends itself to a fully automated verification algorithm.

We have implemented our proposed solution as a tool called Themis[1], a static analyzer for detecting resource-related side-channels in Java applications. To demonstrate its effectiveness, we evaluate Themis by performing a series of experiments. First, we compare Themis against Blazer [8], a state-of-the-art static analysis tool for finding timing side channels in Java applications, and we show that Themis compares favorably with Blazer, both in terms of accuracy and running time. Second, we use Themis to analyze known side-channel vulnerabilities in security-sensitive Java applications, such as Tomcat and Spring-Security. We show that Themis can identify the defects in the original vulnerable versions of these programs, and that Themis can *verify* the correctness of their repaired versions. Finally, we run Themis on several real-world Java applications and demonstrate that Themis uncovers *previously unknown* side-channel vulnerabilities in widely-used programs, such as the Eclipse Jetty HTTP web server.

Contributions. In summary, this paper makes the following key contributions:

- We propose the notion of *ϵ-bounded non-interference*, which can be used to reason about secret-induced variations in the application's resource usage behavior.
- We present *Quantitative Cartesian Hoare Logic (QCHL)*, a variant of CHL that can be used to verify ϵ-bounded non-interference.
- We show how to build a scalable, end-to-end side channel detection tool by combining static taint analysis and QCHL.
- We implement our approach in a tool called Themis and evaluate it on multiple security-critical Java applications. We also compare Themis against Blazer, a state-of-the-art timing side channel detector for Java. Our results demonstrate that Themis is precise, useful, and scalable.

```
1   BigInteger modPow(BigInteger base,
2       BigInteger exponent, BigInteger modulus) {
3       BigInteger s = BigInteger.valueOf(1);
4       // BigInteger r;
5       int width = exponent.bitLength();
6       for (int i = 0; i < width; i++) {
7           s = s.multiply(s).mod(modulus);
8           if(exponent.testBit(width - i - 1))
9               s = s.multiply(base).mod(modulus);
10          //else r = s.multiply(base).mod(modulus);
11      }
12      return s;
13  }
```

Figure 1: Gabfeed code snippet that contains a timing side channel (without the commented out lines). A possible fix can be obtained by commenting in lines 4 and 10.

```
{
  "epsilon":"0",  "costModel":"time",
  "secrets": [ "<com.cyberpointllc.stac.auth.
                KeyExchangeServer:java.math.BigInteger
                secretKey>" ]
}
```

Figure 2: Themis **configuration file for Gabfeed.**

- We use Themis to find previously unknown security vulnerabilities in widely-used Java applications. Five of the vulnerabilities uncovered by Themis were confirmed and fixed by the developers in less than 24 hours.

Organization. The rest of this paper is organized as follows. We start by giving an overview of Themis and explain our threat model (Section 2). After formalizing the notion of ϵ-bounded non-interference in Section 3, we then present our program logic, QCHL, for verifying this 2-safety property (Section 4). We then describe the design and implementation of Themis in Section 5 and present the results of our evaluation in Section 6. The limitations of the system as well as comparison against related work are discussed in Sections 7 and 8.

2 OVERVIEW

In this section, we give an overview of our technique with the aid of a motivating example and explain the threat model that we assume throughout the paper.

2.1 Motivating Example

Suppose that Bob, a security analyst at a government agency, receives a Java web application called Gabfeed, which implements a web forum that allows community members to post and search messages[2]. In this context, both the user names and passwords are considered confidential and are therefore encrypted before being stored in the database. Bob's task is to vet this application and verify that it does not contain timing side-channel vulnerabilities that may compromise user name or password information. However,

[1] Themis is a Greek goddess for justice and balance, hence the name.

[2] Gabfeed is one of the challenge problems from the DARPA STAC project. Please see http://www.darpa.mil/program/space-time-analysis-for-cybersecurity for more details about the STAC project.

Gabfeed contains around 30,000 lines of application code (not including *any* libraries); hence, manually searching for a vulnerability in the application is akin to finding a needle in the haystack.

A security analyst like Bob can greatly benefit from Themis by using it to automatically verify the absence of side-channel vulnerabilities in the target application. To use Themis, Bob first identifies application-specific confidential data (in this case, secretKey) and annotates them as such in a Themis-specific configuration file, as shown in Figure 2. In the same configuration file, Bob also tells Themis the type of side channel to look for (in this case, timing) by specifying the `costModel` field and provides a reasonable value of ϵ, using the `epsilon` field. Here, Bob wants to be conservative and initially sets the value of ϵ to zero.

Using the information provided by Bob in the configuration file, Themis first performs static taint analysis to identify methods that are dependent on confidential data. In this case, one of the methods that access confidential data is `modPow`, shown in Figure 1. Specifically, Themis determines that the second argument (`exponent`) of `modPow` is tainted and marks it as a "hot spot" that should be analyzed more precisely using relational verification techniques.

In the next phase, Themis uses its Quantitative Cartesian Hoare Logic (QCHL) verifier to analyze `modPow` in more detail. Specifically, the QCHL verifier considers two executions of `modPow` that have the same values of `base` and `modulus` but that differ in the value of `exponent`. In this case, the QCHL verifier fails to prove that the resource usage of any such two runs is identical and therefore issues a warning about a possible timing side channel in the `modPow` procedure.

Next, Bob wonders whether the imbalance in resource usage is large enough to be actually exploitable in practice. For this reason, he plays around with different values of the bound ϵ, gradually increasing it to larger and larger constants. In the case of timing side channels, ϵ represents the difference in the executed number of Java bytecode instructions. However, no matter what value of ϵ Bob picks, Themis complains about a possible timing side channel. This observation indeed makes sense because the difference in resource usage is proportional to the secret and can therefore not be bounded by a constant.

Bob now inspects the source code of `modPow` and realizes that a possible vulnerability arises due to the resource imbalance in the secret-dependent branch from line 8. To fix the vulnerability, Bob adds the code from lines 4 and 10, with the goal of ensuring that the timing behavior of the program is not dependent on `exponent`. To confirm that his fix is valid, Bob now runs Themis one more time and verifies that his repair eliminates the original vulnerability.

2.2 Threat Model

In this paper, we assume that an adversary can observe a program's total resource usage, such as timing, memory, and response size. When measuring resource usage, we further assume that any variations are caused at the *application software* level. Hence, side channels caused by the microarchitecture such as cache contention [70] and branch prediction [2] are out of the scope of this work. Physical side channels (including power and electromagnetic radiation [31]) can, in principle, be handled by our our system as long as a precise model of the corresponding resource usage is given. We assume that the attacker is not able to observe anything else about the program other than its resource usage.

One possible real-world setting in which the aforementioned assumptions hold could be that the attacker and the victim are connected through a network, and the victim runs a server or P2P software that interacts with other machines through encrypted communications. In this scenario, the attacker and the victim are physically separated; hence, the attacker cannot exploit physical side channels, such as power usage. Furthermore, the attacker does not have a co-resident process or VM running on the victim's machine, thus it is hard to passively observe or actively manipulate OS and hardware-level side channels. What the attacker can do is to either interact with the server and measure the time it takes for the server to respond, or observe the network traffic and measure request and response sizes. In our setting, we assume that data encryption has been properly implemented and the attacker cannot directly read the contents of any packet.

3 SIDE-CHANNELS AND BOUNDED NON-INTERFERENCE

In this section, we introduce the property of *ϵ-bounded non-interference*, which is the security policy that will be subsequently verified using the Themis system.

Let P be a program that takes a list of input values $\vec{a}$, and let $R_P(\vec{a})$ denote the resource usage of P on input $\vec{a}$. Following prior work in the literature [30, 35, 60], we assume that each input is marked as either *high* or *low*, where high inputs denote security-sensitive data and low inputs denote public data. Let $\vec{a}^h$ (resp. $\vec{a}^l$) be the sublist of the inputs that are marked as *high* (resp. *low*). Prior work in the literature [6, 25, 66] considers a program to be side-channel-free if the following condition is satisfied:

Definition 1. *A program P is free of resource-related side-channel vulnerabilities if*

$$\forall \vec{a}_1, \vec{a}_2.\ (\vec{a}_1{}^l = \vec{a}_2{}^l \wedge \vec{a}_1{}^h \neq \vec{a}_2{}^h) \Rightarrow R_P(\vec{a}_1) = R_P(\vec{a}_2)$$

The above definition, which is a direct adaptation of the classical notion of *non-interference* [34], states that a program is free of side channels if the resource usage of the program is deterministic with respect to the public inputs. In other words, the program's resource usage does not correlate with any of its secret inputs.

We believe that Definition 1 is too strong in practice: There are many realistic programs that are considered side-channel-free but that would be deemed vulnerable according to Definition 1. For example, consider a setting in which the attacker is co-located with the victim on a slow network and the resource usage of the program varies by only a few CPU cycles depending on the value of the high input. Since the resource usage of the program is not identical for different high inputs, this program is vulnerable according to Definition 1, but it is practically impossible for an attacker to exploit this vulnerability given the noise in the program's execution environment.

In this paper, we therefore use a relaxed version of the above definition, with the goal of giving security analysts greater flexibility and helping them understand the severity of the resource usage imbalance. Specifically, we propose the following variant of non-interference that we call *ϵ-bounded non-interference*:

$\langle expr \rangle ::= \langle const \rangle \mid \langle var \rangle \mid \langle expr \rangle \circ \langle expr \rangle$
$\quad (\circ \in \{+, -, \times, \vee, \wedge, \ldots\})$

$\langle stmt \rangle ::= \textbf{skip} \mid \textbf{consume}(\langle expr \rangle) \mid \langle var \rangle := \langle expr \rangle$

$\langle stmts \rangle ::= \langle stmt \rangle \mid \langle stmt \rangle; \langle stmts \rangle$
$\mid\ \textbf{if}\ \langle expr \rangle\ \textbf{then}\ \langle stmts \rangle\ \textbf{else}\ \langle stmts \rangle$
$\mid\ \textbf{while}\ \langle expr \rangle\ \textbf{do}\ \langle stmts \rangle$

$\langle params \rangle ::= \langle param \rangle \mid \langle param \rangle, \langle params \rangle$

$\langle param \rangle ::= \langle annot \rangle\ \langle var \rangle$

$\langle annot \rangle ::= \textbf{low} \mid \textbf{high}$

$\langle prog \rangle ::= \lambda \langle params \rangle.\ \langle stmts \rangle$

Figure 3: Language used in our formalization

Definition 2. *A program P obeys the ϵ-bounded non-interference property if*

$$\forall \vec{a_1}, \vec{a_2}.\quad (\vec{a_1}^{\,l} = \vec{a_2}^{\,l} \wedge \vec{a_1}^{\,h} \neq \vec{a_2}^{\,h}) \Rightarrow |R_P(\vec{a_1}) - R_P(\vec{a_2})| \leq \epsilon$$

In this definition, any variation in resource usage below ϵ is deemed to be a minor imbalance that is unlikely to be exploitable under real-world scenarios. Hence, compared to Definition 1, the notion of ϵ-bounded interference considers a program to be secure as long as the difference in resource usage is "minor" according to the constant ϵ. In practice, the value of ϵ should be chosen by security analysts in light of the security requirements of the application and the underlying threat model. If Definition 2 is violated even for large values of ϵ, this means the application potentially exhibits large secret-induced variations in resource usage, and hence, the underlying vulnerability is potentially more serious.

4 VERIFYING BOUNDED NON-INTERFERENCE USING QCHL

One of the key technical contributions of this paper is a new method for verifying ϵ-bounded non-interference using QCHL, a variant of Cartesian Hoare Logic introduced in recent work for verifying k-safety [65]. As mentioned in Section 1, QCHL proves triples of the form $\langle \phi \rangle\ S\ \langle \psi \rangle$, where S is a program fragment and ϕ, ψ are first-order formulas that relate the program's resource usage between an arbitrary pair of program runs. Starting with the precondition that the program's low inputs are the same for a pair of program runs, QCHL tries to derive a post-condition that logically implies ϵ-bounded non-interference.

4.1 Language

We will describe our program logic, QCHL, using the simplified imperative language shown in Figure 3. In this language, program inputs are annotated as *high* or *low*, indicating private and public data respectively. Atomic statements include skip (i.e., a no-op), assignments of the form $x := e$, and *consume* statements, where "**consume**(e)"indicates the consumption of e units of resource. Our language also supports standard control-flow constructs, including sequential composition, if statements, and loops.

$$\frac{P = \lambda\vec{p}.S \qquad \forall p_i \in \vec{p}.\ \Gamma(p_i) = a_i \qquad \Gamma \vdash S : \Gamma', r}{R_P(\vec{a}) = r}$$

$$\frac{S = \textbf{skip}}{\Gamma \vdash S : \Gamma, 0}$$

$$\frac{S = (x := e) \qquad \Gamma \vdash e : v \qquad \Gamma' = \Gamma[x \leftarrow v]}{\Gamma' \vdash S : \Gamma', 0}$$

$$\frac{S = \textbf{consume}\ (e) \qquad \Gamma \vdash e : v}{\Gamma \vdash S : \Gamma, v}$$

$$\frac{S = S_1; S_2 \qquad \begin{matrix} \Gamma \vdash S_1 : \Gamma_1, r_1 \\ \Gamma_1 \vdash S_2 : \Gamma_2, r_2 \end{matrix}}{\Gamma \vdash S : \Gamma_2, r_1 + r_2}$$

$$\frac{S = \textbf{if}\ e\ \textbf{then}\ S_1\ \textbf{else}\ S_2 \qquad \begin{matrix} \Gamma \vdash e : \textbf{true} \\ \Gamma \vdash S_1 : \Gamma', r' \end{matrix}}{\Gamma \vdash S : \Gamma', r'}$$

$$\frac{S = \textbf{if}\ e\ \textbf{then}\ S_1\ \textbf{else}\ S_2 \qquad \begin{matrix} \Gamma \vdash e : \textbf{false} \\ \Gamma \vdash S_2 : \Gamma', r' \end{matrix}}{\Gamma \vdash S : \Gamma', r'}$$

$$\frac{S = \textbf{while}\ e\ \textbf{do}\ S' \qquad \Gamma \vdash e : \textbf{false}}{\Gamma \vdash S : \Gamma, r}$$

$$\frac{\begin{matrix} S = \textbf{while}\ e\ \textbf{do}\ S' \qquad \Gamma \vdash S' : \Gamma_1, r_1 \\ \Gamma \vdash e : \textbf{true} \qquad \Gamma_1 \vdash S : \Gamma_2, r_2 \end{matrix}}{\Gamma \vdash S : \Gamma_2, r_1 + r_2}$$

Figure 4: Rules for computing resource usage

Figure 4 defines the cost-instrumented operational semantics of this language using judgments of the form $\Gamma \vdash S : \Gamma', r$. The meaning of this judgment is that, assuming we execute S under environment Γ (mapping variables to values), then S consumes r units of resource and the new environment is Γ'. As shown in Figure 4, we use the notation $R_P(\vec{a})$ to denote the resource usage of program P on input vector $\vec{a}$. In cases where the resource usage is irrelevant, we simply omit the cost and write $\Gamma \vdash S : \Gamma'$.

4.2 QCHL Proof Rules

We now turn our attention to the proof rules of Quantitative Cartesian Hoare Logic (QCHL), which forms the basis of our verification methodology. Similar to CHL [65], QCHL is a relational program logic that allows proving relationships between multiple runs of the program. However, unlike CHL, QCHL is concerned with proving properties about the difference in *resource usage* across multiple runs. Towards this goal, QCHL performs cost instrumentation and explicitly tracks the program's resource usage. Furthermore, since our goal is to prove the specific property of ϵ-bounded non-interference, QCHL exploits domain-specific assumptions by incorporating taint information into the proof rules. Finally, since the QCHL proof rules we describe here are deterministic, our program logic can be immediately translated into a verification algorithm (modulo an oracle for providing loop invariants and proving standard Hoare triples).

Figure 5 presents the proof rules of QCHL. Here, all proof rules, with the exception of Rule (0), derive judgments of the form $\Sigma \vdash \langle\Phi\rangle\ S_1 \circledast S_2\ \langle\Psi\rangle$, where S_1 and S_2 contain a disjoint set of variables and Σ is a *taint environment* mapping variables to a taint value drawn from the set $\{low, high\}$. The notation $S_1 \circledast S_2$ describes a program that is semantically equivalent to $S_1; S_2$ but that is somehow easier to verify (because it tries to execute loops from different executions in lock step). Hence, we have $\Sigma \vdash \langle\Phi\rangle\ S_1 \circledast S_2\ \langle\Psi\rangle$ if $\{\Phi\}S_1; S_2\{\Psi\}$ is a valid Hoare triple. As we will see shortly, the taint environment Σ is used as a way of increasing the precision and scalability of the analysis. In the remainder of this section, we assume that Σ is sound, i.e., if $\Sigma(x)$ is *low*, then the value of x does not depend (either explicitly or implicitly) on any high inputs. We now explain each of the rules from Figure 5 in more detail.

The first rule labeled (0) corresponds to the top-level verification procedure. If we can derive $\Sigma \vdash SideChannelFree(P, \epsilon)$, then P obeys the ϵ-bounded non-interference property. In this rule, we use the notation S^τ to denote the *cost-instrumented* version of S, defined as follows:

Definition 3. *Given a program $P = \lambda\vec{p}.S$, its cost-instrumented version is another program P^τ obtained by instrumenting P with a counter variable τ that tracks its resource usage. More formally, $P^\tau = \gamma(P)$ where the instrumentation procedure γ is defined as:*

- $\gamma(\lambda\vec{p}.S) = \lambda\vec{p}.(\tau := 0; \gamma(S))$
- $\gamma(\texttt{skip}) = \texttt{skip}$
- $\gamma(x := e) = (x := e)$
- $\gamma(\texttt{consume }(e)) = (\tau := \tau + e)$
- $\gamma(S_1; S_2) = \gamma(S_1); \gamma(S_2)$
- $\gamma(\texttt{if } e \texttt{ then } S_1 \texttt{ else } S_2) = \texttt{if } e \texttt{ then } \gamma(S_1) \texttt{ else } \gamma(S_2)$
- $\gamma(\texttt{while } e \texttt{ do } S) = \texttt{while } e \texttt{ do } \gamma(S)$

Essentially, the program P^τ is the same as P except that it contains an additional variable τ that tracks the program's resource usage. As stated by the following lemma, our instrumentation is correct with respect to the operational semantics from Figure 4.

Lemma 4.1. *Let program $P = \lambda\vec{p}.S$ and let $P^\tau = \lambda\vec{p}.S^\tau$. We have*

- *S^τ does not contain any* `consume` *statement.*
- *If $\Gamma(\vec{p}) = \vec{a}$ and $\Gamma \vdash S^\tau : \Gamma'$, then $R_P(\vec{a}) = \Gamma'(\tau)$.*

Hence, rule (0) from Figure 5 instruments the original program $\lambda\vec{p}.\ S$ to obtain a new program $\lambda\vec{p}.\ S^\tau$ that uses a fresh variable τ to track the program's resource usage. Since bounded non-interference is a 2-safety property, it then creates two α-renamed copies S_1^τ and S_2^τ of S^τ that have no shared variables and uses the remaining QCHL proof rules to derive a triple

$$\langle \vec{p_1}^{\,l} = \vec{p_2}^{\,l} \wedge \vec{p_1}^{\,h} \neq \vec{p_2}^{\,h} \rangle\ S_1^\tau \circledast S_2^\tau\ \langle\Psi\rangle$$

If the post-condition Ψ logically implies $|\tau_1 - \tau_2| \leq \epsilon$, we have a proof that the program obeys bounded non-interference. Intuitively, this proof rule considers an arbitrary pair of executions of S where the low inputs are the same and tries to prove that the resource usage of the two runs differs by at most ϵ.

The remaining rules from Figure 5 derive QCHL triples of the form $\langle\Phi\rangle\ S_1 \circledast S_2\ \langle\Psi\rangle$. Our verification algorithm applies these rules in the reverse order shown in Figure 5. That is, we only use rule labeled i if no rule with label $j > i$ is applicable. Hence, unlike standard CHL, our verification method does not perform backtracking search over the proof rules.

$$\frac{\begin{array}{cc} & \Phi = (\vec{p_1}^{\,l} = \vec{p_2}^{\,l} \wedge \vec{p_1}^{\,h} \neq \vec{p_2}^{\,h}) \\ \lambda\vec{p}_1.S_1^\tau = \alpha(\lambda\vec{p}.S^\tau) & \Sigma \vdash \langle\Phi\rangle\ S_1^\tau \circledast S_2^\tau\ \langle\Psi\rangle \\ \lambda\vec{p}_2.S_2^\tau = \alpha(\lambda\vec{p}.S^\tau) & \models \Psi \rightarrow |\tau_1 - \tau_2| \leq \epsilon \end{array}}{\Sigma \vdash SideChannelFree(\lambda\vec{p}.S,\ \epsilon)}\ (0)$$

$$\frac{\Sigma \vdash \langle\Phi\rangle\ S_2 \circledast S_1\ \langle\Psi\rangle}{\Sigma \vdash \langle\Phi\rangle\ S_1 \circledast S_2\ \langle\Psi\rangle}\ (1)$$

$$\frac{\begin{array}{c} S \neq (S_1; S_2) \\ \Sigma \vdash \langle\Phi\rangle\ S; \textbf{skip} \circledast S'\ \langle\Psi\rangle \end{array}}{\Sigma \vdash \langle\Phi\rangle\ S \circledast S'\ \langle\Psi\rangle}\ (2)$$

$$\frac{\begin{array}{c} \vdash \{\Phi\}\ S_1\ \{\Phi'\} \\ \Sigma \vdash \langle\Phi'\rangle\ S_2 \circledast S_3\ \langle\Psi\rangle \\ S_1 = \textbf{skip} \vee S_1 = (v := e) \end{array}}{\Sigma \vdash \langle\Phi\rangle\ S_1; S_2 \circledast S_3\langle\Psi\rangle}\ (3)$$

$$\frac{\vdash \{\Phi\}\ S\ \{\Psi\}}{\Sigma \vdash \langle\Phi\rangle\ S \circledast \textbf{skip}\ \langle\Psi\rangle}\ (4)$$

$$\frac{\begin{array}{c} \Sigma \vdash \langle\Phi \wedge e\rangle\ S_1; S \circledast S_3\langle\Psi_1\rangle \\ \Sigma \vdash \langle\Phi \wedge \neg e\rangle\ S_2; S \circledast S_3\langle\Psi_2\rangle \end{array}}{\Sigma \vdash \langle\Phi\rangle\ \textbf{if } e \textbf{ then } S_1 \textbf{ else } S_2; S \circledast S_3\langle\Psi_1 \vee \Psi_2\rangle}\ (5)$$

$$\frac{\begin{array}{c} \vdash \{\Phi\}\textbf{while } e_1 \textbf{ do } S_1\{\Phi'\} \\ \vdash \{\Phi'\}\textbf{while } e_2 \textbf{ do } S_2\{\Psi'\} \\ \Sigma \vdash \langle\Psi'\rangle\ S \circledast S'\ \langle\Psi\rangle \end{array}}{\Sigma \vdash \langle\Phi\rangle\ \textbf{while } e_1 \textbf{ do } S_1; S \circledast \textbf{while } e_2 \textbf{ do } S_2; S'\langle\Psi\rangle}\ (6)$$

$$\frac{\begin{array}{c} \Sigma \vdash \text{CanSynchronize}(e_1, e_2, S_1, S_2, I) \\ \Sigma \vdash \langle I \wedge e_1 \wedge e_2\rangle S_1 \circledast S_2\langle I'\rangle \\ \Sigma \vdash \langle I \wedge \neg e_1 \wedge \neg e_2\rangle S \circledast S'\langle\Psi\rangle \\ \models \Phi \rightarrow I \qquad \models I' \rightarrow I \end{array}}{\Sigma \vdash \langle\Phi\rangle\ \textbf{while } e_1 \textbf{ do } S_1; S \circledast \textbf{while } e_2 \textbf{ do } S_2; S'\langle\Psi\rangle}\ (7)$$

Figure 5: QCHL proof rules. The notation $\alpha(S)$ denotes an α-renamed version of statement S.

Let us now consider the remaining rules in more detail: Rule (1) is the same as commutativity rule in CHL and states that the $\circledast$ operator is symmetric. Intuitively, since S_1 and S_2 do not share variables, any interleaving of S_1 and S_2 will yield the same result, and we can therefore commute the two operands when deriving QCHL triples. As will become clear shortly, the commutativity rule ensures that our verification algorithm makes progress when none of the other rules are applicable.

The next rule states that we are free to append a skip statement to any non-sequential statement without affecting its meaning. While this rule may not seem very useful on its own, it allows us to avoid redundancies in the proof system by bringing each $S_1 \circledast S_2$ to a canonical form where S_1 is always of the form $S; S'$ or S_2 is skip.

Rule (3) specifies the verification logic for $S_1 \circledast S_2$ when S_1 is of the form $A; S$ where A is an atomic statement. In this case, we simply "consume" A by deriving the Hoare triple $\{\Phi\}A\{\Phi'\}$ and then use Φ' as a precondition for $S \circledast S_2$.

Rule (4) serves as the base case for our logic. When we want to prove $\langle\Phi\rangle\ S \circledast \textbf{skip}\ \langle\Psi\rangle$, we immediately reduce this judgement to the standard Hoare triple $\{\Phi\}\ S\ \{\Psi\}$ because skip is just a no-op.

Example. Suppose we want to prove (0-bounded) non-interference for the following program:

```
λ(low x). consume(x); skip;
```

First we apply transformation γ and get the resource instrumented program:

```
λ(low x). τ=0; τ = τ + x; skip;
```

Ignore the taint environment for now, as we will not use it in this example. According to rule (0), we only need to prove

$$\langle x_1 = x_2\rangle\ \tau_1 = 0; \tau_1 = \tau_1 + x_1; \textbf{skip};\ \circledast$$
$$\tau_2 = 0; \tau_2 = \tau_2 + x_2; \textbf{skip};\ \langle \tau_1 = \tau_2\rangle$$

Applying rule (3) twice, we can reduce the above judgement to the following one:

$$\langle x_1 = x_2 \land \tau_1 = x_1\rangle\ \textbf{skip};\ \circledast\ \tau_2 = 0; \tau_2 = \tau_2 + x_2; \textbf{skip};\ \langle \tau_1 = \tau_2\rangle$$

Swapping the two operands of $\circledast$ with rule (1), we get

$$\langle x_1 = x_2 \land \tau_1 = x_1\rangle\ \tau_2 = 0; \tau_2 = \tau_2 + x_2; \textbf{skip};\ \circledast\ \textbf{skip}; \langle \tau_1 = \tau_2\rangle$$

After applying rule (4), we get

$$\{x_1 = x_2 \land \tau_1 = x_1\}\ \tau_2 = 0; \tau_2 = \tau_2 + x_2; \textbf{skip};\ \{\tau_1 = \tau_2\}$$

Applying Hoare-style strongest postcondition computation, the above Hoare triple can be reduced to

$$\{x_1 = x_2 \land \tau_1 = x_1 \land \tau_2 = x_2\}\ \textbf{skip};\ \{\tau_1 = \tau_2\}$$

Since this Hoare triple is clearly valid, we have proven non-interference using the QCHL proof rules. □

Rule (5) specifies the general verification logic for branch statements. This rule is an analog of the conditional rule in standard Hoare logic: we can verify an if statement by embedding the branch condition e into the true branch and its negation $\neg e$ into the false branch and carry out the proof for both branches accordingly.

Rule (6) specifies the general verification logic for loops. Without loss of generality, this rule requires both sides of the $\circledast$ operator to be loops: If one side is a not a loop, we can always apply one of the other rules, using rule (1) to swap the loop to the other side if necessary. The idea here is to apply self-composition [14]: we run the loop on the left-hand side first, followed by the loop on the right-hand side, and try to derive the proof as if the two loops are sequentially composed.

While rule (6) is sound, it is typically difficult to prove 2-safety using rule (6) alone. In particular, rule (6) does not allow us to synchronize executions between the two loops, so the resulting Hoare triples are often hard to verify. The following example illustrates this issue:

$$\frac{e_1 = \alpha(e) \qquad e_2 = \alpha(e)}{e_1 \equiv_\alpha e_2}$$

$$\frac{S_1 = \alpha(S) \qquad S_2 = \alpha(S)}{S_1 \equiv_\alpha S_2}$$

$$\frac{\begin{array}{cc} e_1 \equiv_\alpha e_2 & \Sigma \vdash e_1 : \textbf{low} \\ S_1 \equiv_\alpha S_2 & \Sigma \vdash e_2 : \textbf{low} \end{array}}{\Sigma \vdash \text{CanSynchronize}(e_1, e_2, S_1, S_2, I)}$$

$$\frac{\models I \rightarrow (e_1 \leftrightarrow e_2)}{\Sigma \vdash \text{CanSynchronize}(e_1, e_2, S_1, S_2, I)}$$

Figure 6: Helper rules for figure 5

Example. Consider the following code snippet:

```
λ(low n, low k).
  i = 0;
  while (i < n) {
    consume(i); i = i + k;
  }
```

To prove that this program obeys ϵ-bounded non-interference, we need to show that the difference in resource consumption after executing the two copies of the loop is at most ϵ. However, to prove this property using rule (6), we would need to infer a precise post-condition about resource consumption. Unfortunately, this requires inferring a complex *non-linear* loop invariant involving i, n, k. Since such loop invariants are difficult to infer, we cannot prove non-interference using rule (6). □

Rule (7) is one of the most important rules underlying QCHL, as it allows us to execute loops from different executions in lockstep. This loop can be applied only when the *CanSynchronize* predicate is true, meaning that the two loops are guaranteed to execute the same number of times. The definition of the *CanSynchronize* predicate is shown in Figure 6: Given two loops $L_1 \equiv \textit{while}(e_1)\ \textit{do}\ S_1$ and $L_2 \equiv \textit{while}(e_2)\ \textit{do}\ S_2$, and a loop invariant I for the "fused" loop $\textit{while}(e_1 \land e_2)\ \textit{do}\ S_1; S_2$, *CanSynchronize* determines if L_1 and L_2 must execute the same number of times. In the easy case, this information can be determined using only taint information: Specifically, suppose that L_1, L_2 are identical modulo variable renaming and e_1, e_2 contains only untainted (low) variables. Since we prove bounded non-interference under the assumption that low variables from the two runs have the same value, this assumption implies L_1 and L_2 must execute the same number of times. If we cannot prove the *CanSynchronize* predicate using taint information alone, we may still be able to prove it using the invariant I for the fused loop. Specifically, if I logically implies $e_1 \leftrightarrow e_2$, we know that after each iteration e_1, e_2 have the same truth value; hence, the loops must again execute the same number of times.

Now, suppose we can prove that *CanSynchronize* evaluates to true. In this case, rule (7) conceptually executes the two loops in lockstep. Specifically, the premise $\Sigma \vdash \langle I \land e_1 \land e_2\rangle S_1 \circledast S_2 \langle I'\rangle$, together with $\models I' \rightarrow I$, ensures that I is an inductive invariant of the fused loop $\textit{while}(e_1 \land e_2)\ \textit{do}\ S_1; S_2$. Thus, I must hold when the both loops terminate. Thus, we can safely use the predicate $I \land \neg e_1 \land \neg e_2$ as a precondition when reasoning about the "continuations" S and S'.

Algorithm 1 Relational Invariant Generation

```
Input:  $\Sigma$, the taint environment.
Input:  $\Phi$, the pre-condition of the loop.
Input:  $e, S$, loop condition and loop body.
Input:  $V$, the set of all variables appeared in the loop.
Output:  An inductive relational loop invariant
1:  function RelationalInvGen($\Sigma, \Phi, e, S, V$)
2:      $(e_1, S_1) \leftarrow \alpha(e, S)$
3:      $(e_2, S_2) \leftarrow \alpha(e, S)$
4:      $Guesses \leftarrow \{v_1 = v_2 \mid v \in V\}$
5:      for $g \in Guesses$ do
6:          if $\not\models \Phi \rightarrow g$ then
7:              $Guesses \leftarrow Guesses \backslash \{g\}$
8:      $inductive \leftarrow$ false
9:      while $\neg$ $inductive$ do
10:         $I \leftarrow \bigwedge_{g \in Guesses} g$
11:         assume $\Sigma \vdash \langle I \wedge e_1 \wedge e_2 \rangle S_1 \circledast S_2 \langle I' \rangle$
12:         $inductive \leftarrow$ true
13:         for $g \in Guesses$ do
14:             if $\not\models I' \rightarrow g$ then
15:                 $Guesses \leftarrow Guesses \backslash \{g\}$
16:                 $inductive \leftarrow$ false
17:     return $\bigwedge_{g \in Guesses} g$
```

Example. In the previous example, we illustrated that it is difficult to prove non-interference using rule (6) even for a relatively simple example. Let us now see why rule (7) makes verifying 2-safety easier. Since i and n are both low according to the taint environment Σ, we can show that the the *CanSynchronize* predicate evaluates to true. To prove that the program obeys non-interference, we use the relational loop invariant $I = (i_1 = i_2 \wedge \tau_1 = \tau_2 \wedge k_1 = k_2)$. It is easy to see that I is a suitable inductive relational loop invariant, because:

- i_1, i_2, τ_1, τ_2 are all set to 0 before the loop starts.
- We know $k_1 = k_2$ from the precondition (since they are low inputs)
- i_1 and i_2 are increased by the same amount in each iteration of the loop since $k_1 = k_2$.
- τ_1 and τ_2 are also increased by the same amount in each iteration of the loop since $i_1 = i_2$.
- I implies the post condition $|\tau_1 - \tau_2| \leq 0$.

Observe that the use of rule (7) allows us to prove the desired property without reasoning about the total resource consumption of the loop. Hence, we do not need complicated non-linear loop invariants, and the verification task becomes much easier to automate. □

Theorem 4.2 (Soundness). *Assuming soundness of taint environment Σ, if $\Sigma \vdash SideChannelFree(\lambda\vec{p}.S,\ \epsilon)$, then the program $\lambda\vec{p}.S$ does not have an ϵ-bounded resource side-channel.*

Proof of this theorem can be found in appendix A.

4.3 Loop Invariant Generation

In the previous subsection, we assumed the existence of an oracle for finding suitable relational loop invariants (recall rule 7). Here, by "relational loop invariant", we mean a simulation relation over variables in programs S_1, S_2. Specifically, we use such relational loop invariants in two ways: First, we use them to check whether two loops execute the same number of times. Second, we use the relational loop invariant to compute the precondition for the continuations of the two programs. Hence, to apply rule 7, we need an algorithm for computing such relational loop invariants.

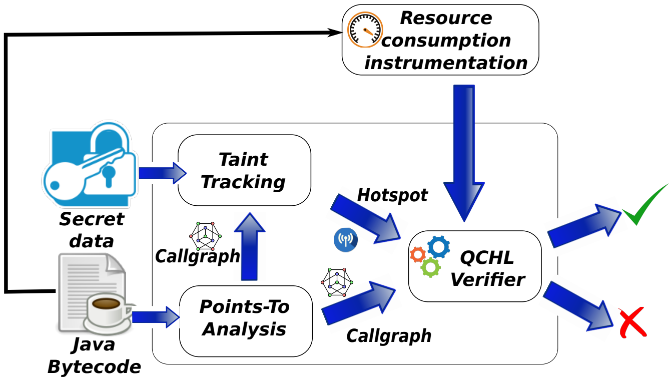

Figure 7: Workflow of the Themis tool

Algorithm 1 shows our inference engine for computing relational loop invariants. This algorithm can be viewed as an instance of monomial predicate abstraction (i.e., *guess-and-check*) [27, 47, 61]. Specifically, we consider the universe *Guesses* of predicates $v_1 = v_2$ relating variables from the two loops. Because synchronizable loops execute the same number of times, they typically contain one or more "anchor" variables that are pairwise equal. Thus, we can often find useful relational invariants over this universe of predicates.

Considering Algorithm 1 in more detail, we first filter our those predicates that are not implied by the precondition Φ (lines 5-7). In lines 9-16, we then further filter out those predicates that are not preserved in the loop body. In particular, on line 10, we construct the candidate invariant by conjoining all remaining predicates in our guess set, and, on line 11, we compute the post condition I' of the loop using the proof rules shown in figure 5. Since we have $\Sigma \vdash \langle I \wedge e_1 \wedge e_2 \rangle S_1 \circledast S_2 \langle I' \rangle$ and $\not\models I' \rightarrow g$, this means predicate g is not preserved by the loop body and is therefore removed from our set of predicates. When the loop in lines 9-16 terminates, we have $\Sigma \vdash \langle I \wedge e_1 \wedge e_2 \rangle S_1 \circledast S_2 \langle I' \rangle$ and $\models I' \rightarrow I$; thus, I is an inductive relational loop invariant.

5 SYSTEM DESIGN AND IMPLEMENTATION

In this section, we discuss the design and implementation of Themis, our end-to-end static analysis tool for verifying bounded non-interference. While the QCHL verifier discussed in the previous section is one of the key components of Themis, it is not necessary (and also not scalable) to employ such precise relational reasoning throughout the entire program. Hence, as mentioned earlier, our approach employs taint analysis to identify program parts that require more precise analysis.

5.1 Design Overview

Figure 7 gives a high-level schematic overview of Themis's architecture. In addition to the QCHL verifier discussed in detail in Section 4, Themis also incorporates pointer and taint analyses and instruments the program to explicitly track resource usage. We now give a brief overview of each of these components.

Pointer analysis. Given the bytecode of a Java application, Themis performs (field- and object-sensitive) pointer analysis to build a precise call graph and identify all variables that may alias each other. The resulting call graph and alias information are used by the subsequent taint analysis as well as the QCHL verifier.

Taint analysis. The use of taint analysis in Themis serves two goals: First, the QCHL verifier uses the results of the taint analysis to determine whether two loops can be synchronized. Second, we use taint analysis to identify hotspots that need to be analyzed more precisely using the QCHL verifier.

The taint analyzer uses the annotations in Themis's configuration file to determine taint sources (i.e., high inputs) and propagates taint using a field- and object-sensitive analysis. Our taint analyzer tracks both explicit and implicit flows. That is, a variable v is considered tainted if (a) there is an assignment $v := e$ such that e is tainted (explicit flow), or (b) a write to v occurs inside a branch whose predicate is tainted (implicit flow).

We use the results of the taint analysis to identify methods that should be analyzed by the QCHL verifier. A method m is referred to as *hot spot* if it reads from a tainted variable. We say that a hot spot m *dominates* another hot spot m' if m' is a transitive callee of m but not the other way around. Any hot spot that does not have dominators is given as an entry point to the QCHL verifier. In principle, this strategy of running the QCHL verifier on only hot spots can cause our analysis to report false positives. For instance, consider the following example:

```
main(...) { foo(); bar(); }

foo() {
  int x = readSecret();
  if(x > 0) consume(1); else consume(100);
}

bar() {
  int y = readSecret();
  if(y <= 0) consume(1); else consume(100);
}
```

While this program does not have any secret-dependent imbalance in resource usage, foo and bar individually do not obey non-interference, causing our analysis to report false positives. However, in practice, we have not observed any such false positives, and this strategy greatly increases the scalability of the tool.

Resource usage instrumentation. The language we considered for our formalization in Section 4 is equipped with a *consume*(x) statement that models consumption of x units of resource. Unfortunately, since Java programs do not come with such statements, our implementation uses a *cost model* to instrument the program with such *consume* statements. In principle, our framework can detect different classes of side channels, provided that the tool is given a suitable cost model for the corresponding resource type.

Our current implementation provides cost models for two kinds of resource usage, namely, timing and response size. For timing, we use a coarse cost model where every byte code instruction is assumed to have unit cost. For response size, each string s that is appended to the response consumes $s.length()$ units of resource.

Counterexample generation. If Themis fails to verify the bounded non-interference property for a given ϵ, it can also generate counterexamples by using the models provided by the underlying SMT solver. In particular, when the verification condition (VC) generated by Themis is invalid, the tool asks the SMT solver for a falsifying assignment and pretty-prints the model returned from Z3 by replacing Z3 symbols with their corresponding variable names. Since the VCs depend on automatically inferred loop invariants, the counterexamples generated by Themis may be spurious if the inference engine does not infer sufficiently strong loop invariants.

5.2 System Implementation

The Themis system is implemented in a combination of Java and OCaml and leverages multiple existing tools, such as Soot [67], Z3 [24], and Apron [44]. Specifically, our pointer analysis builds on top of Soot [67], and we extend the taint analysis provided by FlowDroid [9], which is a state-of-the-art context-, field-, flow-, and object-sensitive taint analyzer, to also track implicit flows. Our QCHL verifier is implemented in OCaml and uses the Z3 SMT solver [24] to discharge the generated verification conditions. To prove the Hoare triples that arise as premises in the QCHL proof rules, we perform standard weakest precondition computation, leveraging the Apron numerical abstract domain library [44] to infer standard loop invariants. Recall that we infer relational loop invariants using the monomial predicate abstraction technique described in Section 4.3.

Our formal description of QCHL in section 4 uses a simplified programming language that does not have many of the complexities of Java. Themis handles these complexities by first leveraging the Soot framework to parse the Java bytecode to Soot IR, and then using an in-house "front-end" that further lowers Soot IR into a form closer to what is presented in section 4. In particular, the transformation from Soot to our IR recovers program structures (loops, conditionals etc.) and encodes heap accesses in terms of arrays. The verifier performs strongest postcondition calculation over our internal IR and encodes verification conditions with SMT formulae. In the remainder of this section, we explain how we handle various challenges that we encountered while building the Themis frontend.

Object encoding. Since objects are pervasive in Java applications, their encoding has a significant impact on the precision and scalability of the approach. In Themis, we adopt a heap encoding that is similar to ESC-Java [28]. Specifically, instance fields of objects are represented as maps from object references (modeled as integer variables) to the value of the corresponding field. Reads and writes to the map are modeled using select and update functions defined by the theory of arrays in SMT solvers. If two object references are known not to be the same (according to the results of the pointer analysis), we then add a disequality constraint between the corresponding variables.

Method invocation. Since the simplified language from Section 4 did not allow function calls, we only described an intraprocedural version of the QCHL verifier. We currently perform interprocedural analysis by function inlining, which is performed as a preprocessing step at the internal IR level before the analysis takes place. Since the QCHL verifier only needs to analyze hot spots (which typically constitute a small fraction of the program), we do not find inlining to be a major scalability bottleneck. However, since recursive procedures cannot be handled using function inlining, our current implementation requires models for recursive procedures that correspond to hot spots.

Virtual calls and instanceof encoding. The result of certain operations in the Java language, such as virtual calls and the `instanceof` operator, depends on the runtime values of their operands. To faithfully model those operations , we encode the type of each allocation site as one of its field, and we transform virtual calls and `instanceof` to a series of if statements that branch on this field. For example, if variable `a` may point to either allocation A_1 of type `T1` or allocation A_2 of type `T2`, then the polymorphic call site `a.foo()` will be modeled as:

```
if (a.type == T1)
  ((T1)a).foo();
else if (a.type == T2)
  ((T2)a).foo();
```

We handle the `instanceof` operator in a similar way.

6 EVALUATION

In this section, we describe our evaluation of Themis on a set of security-critical Java applications. Our evaluation is designed to answer the following research questions:

Q1. How does Themis compare with state-of-the-art tools for side channel detection in terms of accuracy and scalability?
Q2. Is Themis able to detect known vulnerabilities in real-world Java applications, and can Themis verify their repaired versions?
Q3. Is Themis useful for detecting zero-day vulnerabilities from the real world?

In what follows, we describe a series of experiments that are designed to answer the above questions. All experiments are conducted on an Intel Xeon(R) computer with an E5-1620 v3 CPU and 64G of memory running on Ubuntu 16.04.

6.1 Comparison Against Blazer

To evaluate how competitive Themis is with existing tools, we compare Themis against Blazer [8], a state-of-the-art tool for detecting timing side channels in Java bytecode. Blazer is a static analyzer that uses a novel decomposition technique for proving non-interference properties. Since the Blazer tool is not publicly available, we compare Themis against Blazer on the same 22 benchmarks that are used to evaluate Blazer in their PLDI'17 paper [8]. These benchmarks include a combination of challenge problems from the DARPA STAC program, classic examples from previous literature[33, 46, 55], and some microbenchmarks constructed by the developers of Blazer. Since Blazer verifies standard non-interference (rather than our proposed ϵ-bounded variant), we set the value of ϵ to be 0 when running Themis.

Benchmark	Version	Size	Time (s)	
			Blazer	Themis
MicroBench				
array	Safe	16	1.60	0.28
array	Unsafe	14	0.16	0.23
loopAndbranch	Safe	15	0.23	0.33
loopAndbranch	Unsafe	15	0.65	0.16
nosecret	Safe	7	0.35	0.20
notaint	Unsafe	9	0.28	0.12
sanity	Safe	10	0.63	0.41
sanity	Unsafe	9	0.30	0.17
straightline	Safe	7	0.21	0.49
straightline	Unsafe	7	22.20	5.30
STAC				
modPow1	Safe	18	1.47	0.61
modPow1	Unsafe	58	218.54	14.16
modPow2	Safe	20	1.62	0.75
modPow2	Unsafe	106	7813.68	141.36
passwordEq	Safe	16	2.70	1.10
passwordEq	Unsafe	15	1.30	0.39
Literature				
k96	Safe	17	0.70	0.61
k96	Unsafe	15	1.29	0.54
gpt14	Safe	15	1.43	0.46
gpt14	Unsafe	26	219.30	1.25
login	Safe	16	1.77	0.54
login	Unsafe	11	1.79	0.70

Figure 8: Comparison between Themis and Blazer.

We summarize the results of our comparison against Blazer in Table 8. [3] One of the key points here is that Themis is able to automatically verify all 25 programs from the Blazer data set. Moreover, we see that Themis is consistently faster than Blazer except for a few benchmarks that take a very short time to analyze. On average, Themis takes a median of 7.73 seconds to verify a benchmark, whereas the median running time of Blazer is 376.92 seconds.

6.2 Detection of Known Vulnerabilities

To demonstrate that Themis can be used to detect non-trivial vulnerabilities in real-world Java programs, we further evaluate Themis on security-sensitive Java frameworks. The benchmarks we collect come from the following sources:

(1) Response-size side-channel benchmarks from existing publication [73][4].

[3] The Blazer paper reports two sets of numbers for running time, namely time for safety verification alone, and time including attack specification search. Since Themis does not perform the latter, we only compare time for safety verification. For the "Size" column in the table, we use the original metric from Blazer, which indicates the number of basic blocks.

[4] We are only able to obtain the source codes for 2 of 3 benchmarks mentioned in the paper.

Benchmark	Version	LOC	LOC'	$\epsilon = 64$	$\epsilon = 0$	Time (s)
Spring-Security	Safe	1630	41	✓	✓	1.70
Spring-Security	Unsafe	1602	32	✓	✓	1.09
JDK7-MsgDigest	Safe	633	30	✓	✓	1.27
JDK6-MsgDigest	Unsafe	619	27	✓	✓	1.33
Picketbox	Safe	208	73	✓	✗	1.79
Picketbox	Unsafe	180	65	✓	✓	1.55
Tomcat	Safe	12221	100	✓	✗	9.93
Tomcat	Unsafe	12173	96	✓	✓	8.64
Jetty	Safe	2667	77	✓	✓	2.50
Jetty	Unsafe	2619	76	✓	✓	2.07
orientdb	Safe	19564	134	✓	✗	37.99
orientdb	Unsafe	19413	131	✓	✓	38.09
pac4j	Safe	1978	104	✓	✗	3.97
pac4j	Unsafe	1900	105	✓	✓	1.85
boot-auth	Safe	7106	74	✓	✗	9.12
boot-auth	Unsafe	6977	69	✓	✓	8.31
tourPlanner	Safe	7735	46	✓	✓	22.22
tourPlanner	Unsafe	7660	34	✓	✓	22.01
Dyna_table	Unsafe	175	40	✓	✓	1.165
Advanced_table	Unsafe	232	55	✓	✓	2.01

Figure 9: Evaluation on existing vulnerabilities. A checkmark (✓) indicates that Themis gives the correct result, while ✗ indicates a false positive.

(2) One benchmark that contains a response-size side channel from the DARPA STAC project.
(3) A well-known timing side channel in the MessageDigest class from JDK6.
(4) Seven other benchmarks with known vulnerabilities collected from Github.

Benchmarks that fall in the first two categories contain response-size side-channel vulnerabilities, and all other benchmarks contain timing side-channels. All benchmarks except for those in category (1) also come with a repaired version that does not exhibit the original vulnerability.

Before running Themis, we need to specify the entry points of each application. Since most applications come with test cases, we use these test harnesses as entry points. For those applications for which we do not have suitable drivers, we manually construct a harness and specify it as the entry point.

Main results. The table in Figure 9 shows the accuracy and running time of Themis on these benchmarks. Using a value of $\epsilon = 64$, Themis successfully finds vulnerabilities in the original vulnerable versions of these frameworks and is able to verify that the original vulnerability is no longer present in the repaired versions. The running time of Themis is also quite reasonable, taking an average 8.81 seconds to analyze each benchmark.

Benefit of taint analysis. Recall from Sections 1 and 5 that Themis performs taint analysis to identify hot spots, which overapproximate program fragments that may contain a side-channel vulnerability. The QCHL verifier only analyzes such hot spots rather than the entire program. To demonstrate the usefulness of taint analysis, we compare the lines of code (in Soot IR) in the original application (reported in the LOC column) with the lines of code (also in Soot IR) with those analyzed by the QCHL verifier (reported in the LOC' column). As we can see from Figure 9, taint analysis significantly prunes security-irrelevant parts of the application in terms of lines of codes. This pruning effect can also be observed using other statistics. For example, the number of reachable methods ranges from 15 to 1487, with an average of 479, before taint analysius, whereas the number of reachable methods after taint analysis ranges from 6 to 35, with an average of 15, after taint analysis. Thus, pruning using taint information makes the job of the QCHL verifier significantly easier.

Benchmark	LOC	Category	#Reports	Time (s)
Jetty	2619	Server	4	10.17
Tomcat	12173	Server	1	5.86
OpenMRS	10721	Healthcare	1	9.71
OACC	78	Authentication	1	1.83
Apache Shiro	4043	Authentication	0	6.54
Apache Crypto	4505	Crypto	0	4.33
bc-java	5759	Crypto	0	6.89

Figure 10: Evaluation Themis on identifying zero-day vulnerabilities from popular Java applications

Benefit of ϵ. To justify the need for our relaxed notion of non-interference, Figure 9 also shows the results of the same experiment using an ϵ value of 0. Hence, the $\epsilon = 0$ column from Figure 9 corresponds to the standard notion of non-interference. As we can see from the table, Themis reports several false positives using an ϵ value of 0. In particular, the repaired versions of some programs still exhibit a minor resource usage imbalance but this difference is practically infeasible to exploit, so the developers consider these versions to be side-channel-free. However, these programs are deemed unsafe using standard non-interference. We believe this comparison shows that our relaxed policy of ϵ-bounded non-interference is useful in practice and allows security analysts to understand the severity of the side channel.

Benefit of relational analysis. To investigate the benefit of relational invariants, we analyze the safe versions of the 20 benchmarks from Figures 8 and 9 with relational invariant generation disabled. In this case, Themis can only verify the safety of 10 of the benchmarks.

Although this number can potentially be increased by using a more sophisticated non-relational loop invariant generation algorithm, Themis circumvents this need, instead using simple *relational* in- variants that are conjunctions of simple equality constraints. This experiment corroborates the hypothesis that QCHL makes verification easier by requiring simpler loop invariants compared to other techniques like self-composition.

6.3 Discovery of Zero-Day Vulnerabilities

To evaluate whether Themis can discover *unknown* vulnerabilities in real world Java applications, we conduct an experiment on seven popular Java frameworks. Our data set covers a wide range of Java applications from different domains such as HTTP servers,

```
1  public boolean check(Object credentials)
2  {
3    if (credentials instanceof char[])
4      credentials = new String((char[])credentials);
5    if (!(credentials instanceof String) && !(credentials
         instanceof Password))
6      LOG.warn("Can't check " + credentials.getClass() + "
          against CRYPT");
7
8    String passwd = credentials.toString();
9    // FIX: return stringEquals(_cooked, UnixCrypt.crypt(
         passwd,_cooked));
10   return _cooked.equals(UnixCrypt.crypt(passwd,_cooked));
11
12 }
13
14 /**
15  * <p>Utility method that replaces String.equals() to
        avoid timing attacks.</p>
16  */
17 static boolean stringEquals(String s1, String s2)
18 {
19   boolean result = true;
20   int l1 = s1.length();
21   int l2 = s2.length();
22   if(l1 != l2) result = false;
23   int n = (l1 < l2) ? l1 : l2;
24   for (int i = 0; i < n; i++)
25     result &= s1.charAt(i) == s2.charAt(i);
26   return result;
27 }
```

Figure 11: Eclipse Jetty code snippet that contains a timing side channel. Line 10 is the original buggy code. This vulnerability can be fixed by implementing `stringEquals` (lines 14 – 26) and calling it instead of the built-in `String.equals` method.

health care platforms, authentication frameworks, etc. For example, Eclipse Jetty is a well-known web server that is embedded in products such as Apache Spark, Google App Engine, and Twitter's Streaming API. OpenMRS is the world's leading open source enterprise electronic medical record system platform; OACC is a well-known Java application security framework, and bc-java is an implementation of the Bounty Castle crypto API in Java.

As in our previous experiment, we first manually annotate each application to indicate the sources of confidential information. We then use Themis to find timing side channels in these applications using an ϵ value of 10. The results of this experiment are summarized in Figure 10. As we can see from this figure, Themis reports a total of seven vulnerabilities in four of the analyzed applications. We manually inspected each report and confirmed that the detected vulnerabilities are indeed true positives. We also reported the vulnerabilities detected by Themis to the developers, and the majority of these vulnerabilities were confirmed and fixed by the developers in less than 24 hours. However, the vulnerability that we reported for OpenMRS was rejected by the developers. The reason for this false positive is that the leaked password is actually hashed and salted in the database, but, because the logic for hashing and salting is not part of the Java implementation, Themis was not able to reason about this aspect.

To give the reader some intuition about the kinds of side channels detected by Themis, Figure 11 shows a security vulnerability from the Eclipse Jetty web server. The `check` procedure from Figure 11 checks whether the password provided by the user matches the expected password (`_cooked`). The original code performs this check by calling the built-in equality method provided by the `java.lang.String` library. Since the built-in equality method returns false *as soon as* it finds a mismatch between two characters, line 10 in the `check` method introduces a timing side-channel vulnerability.

The developers have fixed the vulnerability [1] in this code snippet by replacing line 10 with the (commented out) code shown in line 9. In particular, the fix involves calling the safe version of equals, called `stringEquals`, which checks for equality between *all* characters in the strings. This repaired version of the `check` method no longer contains a vulnerability for any $\epsilon > 1$, and Themis can verify that the `check` procedure is now safe.

7 LIMITATIONS

Like any other program analysis tool, Themis has a number of limitations. First, due to the fundamental undecidability of the underlying static analysis problem, Themis is incomplete and may report false positives (e.g., due to imprecision in pointer analysis or loop invariant generation). For example, our method for inferring relational invariants is based on monomial predicate abstraction using a fixed set of pre-defined templates, and we restrict our templates to equalities between variables. In addition, our non-relational invariant generator is based on traditional abstract interpretation, which does not distinguish array elements precisely.

Second, dynamic features of the Java language, such as reflective calls, dynamic class loading, and exceptional handling, pose challenges for Themis. Our current implementation can handle some cases of reflection (e.g., reflective calls with string constants), but reflection can, in general, cause Themis to have false negatives. We plan to mitigate this issue by integrating the Tamiflex tool [16] for reasoning about reflection into the Themis tool chain.

Finally, Themis unconditionally trusts all human inputs into the system, which may result in false negatives if the user inputs are not accurate. Said user inputs include application entry points, taint sources, cost instrumentations, and models of library methods.

8 RELATED WORK

In this section, we survey related work from the security and program analysis communities and explain how Themis differs from prior techniques.

Side channel attacks. Side-channel attacks related to resource usage have been known for decades. Specifically, side channels have been used to leak private cryptographic keys [3, 19, 46], infer user accounts [17], steal cellphone and credit card numbers [32], obtain web browsing history [26], and recover the plaintext of encrypted TLS traffic [5]. Chen et al. presents a comprehensive study of side-channel leaks in web applications [22].

Verification for non-interference. As mentioned in Section 3, we can prove that a program is free of side channel leaks by proving that it obeys a certain kind of non-interference property. There has been a significant body of work on proving non-interference. The simplest and most well-known technique for proving non-interference (and, in general, any 2-safety property) is *self-composition* [14]. The general idea underlying self-composition is as follows: Given a program P and 2-safety property ϕ, we create a new program P' which sequentially composes two α-renamed copies of P and then asserts that ϕ holds. Effectively, self-composition reduces verification of 2-safety to standard safety. While this self-composition technique is sound and relatively complete, successfully verifying the new program often requires the safety checker to come up with intricate invariants that are difficult to infer automatically [66]. Dufay et al. try to solve this problem by providing those invariants through JML annotations [25]; however, the resulting approach requires significant manual effort on the part of the developer or security analyst.

Another popular approach for proving k-safety is to construct so-called *product programs* [12, 13, 71]. Similar to self-composition, the product program method also reduces k-safety to standard safety by constructing a new program containing an assertion. While there are several different methods for constructing the product program, the central idea –shared in this work– is to execute the different copies of the program in lock step whenever possible. One disadvantage of this approach is that it can cause a blow-up in program size. As shown in the work of Sousa and Dillig [65], the product program approach can therefore suffer from scalability problems.

The approach advocated in this paper is most closely related to relational program logic, such as *Cartesian Hoare Logic* [65] and *Relational Hoare Logic* [15]. Specifically, the QCHL program logic introduced in Section 4 builds on top of CHL by instantiating it in the ϵ-bounded non-interference setting and augmenting it with additional rules for tracking resource usage and utilizing taint information. One advantage of this approach over explicit product construction is that we decompose the proof into smaller lemmas by constructing small product programs on-the-fly rather than constructing a monolithic program that is subsequently checked by an off-the-shelf verifier.

The approach described in this paper also shares similarities with the work of Terauchi and Aiken, in which they extend self-composition with *type-directed translation* [52, 66]. In particular, this technique uses a type system for secure information flow to guide product construction. Specifically, similar to our use of taint information to determine when two loops can be synchronized, Terauchi and Aiken use type information to construct a better product program than standard self-composition. Our verification technique differs from this approach in two major ways: First, our algorithm is not guided purely by taint information and uses other forms of semantic information (e.g., relational loop invariants) to determine when two loops can be executed in lock step. Second, similar to other approaches for product construction, the type-directed translation method generates a new program that is subsequently verified by an off-the-shelf verifier. In contrast, our method decomposes the proof into smaller lemmas by constructing mini-products on-the-fly, as needed.

Almeida et al. implement a tool named ct-verif based on aforementioned techniques (involving both product programs and self-composition) [6]. In particular, ct-verif is designed for verifying the *constant-time policy*, which roughly corresponds to our notion of 0-bounded non-interference instantiated with a timing cost model. In addition to using different techniques based on QCHL and taint analysis, Themis provides support for verifying a more general property, namely ϵ-bounded non-interference for any value of ϵ.

An alternative approach for verifying k-safety is the decomposition method used in Blazer [8]: This method decomposes execution traces into different partitions using taint information and then verifies k-safety of the whole program by proving a standard safety property of each partition. One possible disadvantage of this approach is that, unlike our method and product construction techniques, Blazer does not directly reason about the relationship between a pair of program executions. As illustrated through some of the examples in Section 4, such relational reasoning can greatly simplify the verification task in many cases. Furthermore, as we demonstrate in Section 6, Themis can verify benchmarks that cannot be proven by Blazer within a 10-minute time limit.

In their recent work, Ngo et al. propose a language-based system for verifying and synthesizing synthesizes programs with *constant-resource usage*, meaning that every execution path of the program consumes the same amount of resource [53]. This technique uses a novel type system to reason both locally and globally about the resource usage bounds of a given program. Similar to work for verifying *constant-time policy*, this technique also does not allow proving ϵ-bounded non-interference for arbitrary values of ϵ. Furthermore, as a type-based solution for a functional language, this technique puts heavier annotation burden on the developer and is not immediately applicable to standard imperative languages like Java or C.

Secure information flow. There has been a significant body of work on language-based solutions for enforcing information flow properties [51, 56, 69, 72]. For instance, Zhang et al. [72] propose a language-based approach that tracks side-channel leakage, and Pottier et al. [56] design a type-based information flow analysis inside an ML-style language. Themis differs from these language-based solutions in that it requires minimal annotation effort and works on existing Java programs.

One of the most popular tools for tracking information flow in existing Java applications is FlowDroid [9], and Themis builds on top of FlowDroid to identify secret-tainted variables. FlowTracker [59] is another information flow analysis for C/C++ featuring efficient representation of implicit flow. We believe these techniques are complimentary to our approach, and a tool like Themis can directly benefit from advances in such static taint tracking tools.

There have also been attempts at verifying the constant-time policy directly using information-flow checking [11]. However, this approach is flow-insensitive (and therefore imprecise) and imposes a number of restrictions on the input program.

Automatic resource bound computation. There has been a flurry of research on statically computing upper bounds for the resource usage of imperative programs. Existing techniques for this purpose leverage abstract interpretation [37], size-change abstraction [74], lossy vector addition systems [63], linear programming [20], difference constraints [64], recurrence relations [4, 7, 29], and term rewriting [18]. Another line of research, called AARA [39–43], performs bound analysis on functional languages.

Our approach differs from these approaches in that we perform *relational* reasoning about resource usage. That is, rather than computing an upper bound on the resource usage of the program, we use QCHL to prove an upper bound on the *difference* between the resource usage of two program runs. Similar to our QCHL, recent work by Çiçek et al. performs *relational cost analysis* to reason about the difference in resource usage of a pair of programs [21]. Their work shares with us the insight that relational analysis may be simplified by exploiting the structural similarity between the inputs as well as the program codes. However, their non-relational reasoning relies on range analysis while Themis relies on Hoare-style weakest precondition computation; as a result Themis is more precise. Also, Themis analyzes real-world Java programs, while [21] is built on top of a hypothetical higher-order functional language.

Other defenses against side channels. In this paper, we consider a purely static approach for detecting resource side channels. However, there are other possible ways of detecting vulnerabilities and preventing against side channel attacks. For instance, Bang et al. use symbolic execution and model counting to quantify leakage for a particular type of side channel [10]. Pasareanu et al. have recently implemented a symbolic execution based algorithm for generating inputs that maximize side channel measurements (namely timing and memory usage) [54]. Sidebuster [73] uses a hybrid static/dynamic analysis to detect side-channels based on irregularities in the One key advantage of our approach compared to these other techniques is that it can be used to verify the absence of side-channel vulnerabilities in programs.

There has also been a line of research that focuses for defending against side channels using runtime systems [49], compilers [50, 57, 58], or secure hardware [48]. Unlike these techniques, our approach does not result in runtime overhead.

9 CONCLUSIONS

We have proposed a new security policy called *ϵ-bounded non-interference* that can be used to verify the absence of resource side channels in programs. We have also proposed an automated verification algorithm, implemented in a tool called Themis, for proving this property. Our approach verifies the absence of side channels by combining lightweight static taint analysis with precise relational verification using a new program logic called QCHL.

We have evaluated our tool, Themis, in a number of ways and have shown that (a) it can find previously unknown vulnerabilities in widely used Java programs, (b) it can verify that the repaired versions of vulnerable programs do not exhibit the original vulnerability, and (c) it compares favorably against Blazer, a state-of-the-art tool for finding timing side channels in Java programs.

There are a number of directions for future work. First, our current implementation only provides cost models for timing and response size; however, we would like to broaden the applicability of Themis by providing cost models for other kinds of resources. Second, our current implementation can verify bounded non-interference for a given ϵ, but we would also like to automatically infer the smallest ϵ for which the program is safe. While this extension can be easily done by solving an optimization rather than a satisfiability problem, our current implementation does not provide this capability due to limitations in Z3's OCaml API.

ACKNOWLEDGEMENT

We would like to thank the anonymous reviewers for their helpful feedback. We also thank Calvin Lin for his insightful comments, Marcelo Sousa for setting up the DESCARTES [65] tool, and the Jetty developer team for their responsiveness as well as the assistance they provide.

This material is based on research sponsored by DARPA award FA8750-15-2-0096 as well as NSF Award CCF-1712067. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U.S. Government.

REFERENCES

[1] 2017. A timing channel in Jetty. https://github.com/eclipse/jetty.project/commit/2baa1abe4b1c380a30deacca1ed367466a1a62ea. (2017).

[2] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. 2007. On the Power of Simple Branch Prediction Analysis. In *Proceedings of the 2Nd ACM Symposium on Information, Computer and Communications Security (ASIACCS '07)*. ACM, 312–320.

[3] Onur Aciiçmez and Werner Schindler. 2008. A Vulnerability in RSA Implementations Due to Instruction Cache Analysis and Its Demonstration on OpenSSL. In *Proceedings of the 2008 The Cryptopgraphers' Track at the RSA Conference on Topics in Cryptology (CT-RSA'08)*. Springer-Verlag, 256–273.

[4] Elvira Albert, Jesús Correas Fernández, and Guillermo Román-Díez. 2015. Non-cumulative Resource Analysis. In *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 9035*. Springer-Verlag New York, Inc., 85–100.

[5] Nadhem J. AlFardan and Kenneth G. Paterson. 2013. Lucky Thirteen: Breaking the TLS and DTLS Record Protocols. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*. 526–540.

[6] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. 2016. Verifying Constant-Time Implementations. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.* 53–70.

[7] Diego Esteban Alonso-Blas and Samir Genaim. 2012. On the Limits of the Classical Approach to Cost Analysis. In *Proceedings of the 19th International Conference on Static Analysis (SAS'12)*. Springer-Verlag, 405–421.

[8] Timos Antonopoulos, Paul Gazzillo, Michael Hicks, Eric Koskinen, Tachio Terauchi, and Shiyi Wei. 2017. Decomposition Instead of Self-Composition for k-Safety. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*.

[9] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick D. McDaniel. 2014. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*. 259–269.

[10] Lucas Bang, Abdulbaki Aydin, Quoc-Sang Phan, Corina S. Păsăreanu, and Tevfik Bultan. 2016. String Analysis for Side Channels with Segmented Oracles. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. ACM, 193–204.

[11] Gilles Barthe, Gustavo Betarte, Juan Campo, Carlos Luna, and David Pichardie. 2014. System-level Non-interference for Constant-time Cryptography. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*. ACM, 1267–1279.

[12] Gilles Barthe, Juan Manuel Crespo, and César Kunz. 2011. Relational Verification Using Product Programs. In *FM 2011: Formal Methods - 17th International Symposium on Formal Methods, Limerick, Ireland, June 20-24, 2011. Proceedings.* 200–214.

[13] Gilles Barthe, Juan Manuel Crespo, and César Kunz. 2013. Beyond 2-Safety: Asymmetric Product Programs for Relational Program Verification. In *Logical Foundations of Computer Science, International Symposium, LFCS 2013, San Diego, CA, USA, January 6-8, 2013. Proceedings*, Sergei N. Artëmov and Anil Nerode (Eds.), Vol. 7734. Springer, 29–43.

[14] Gilles Barthe, Pedro R D'Argenio, and Tamara Rezk. 2004. Secure information flow by self-composition. In *Computer Security Foundations Workshop, 2004. Proceedings. 17th IEEE.* IEEE, 100–114.

[15] Nick Benton. 2004. Simple relational correctness proofs for static analyses and program transformations. In *ACM SIGPLAN Notices*, Vol. 39. ACM, 14–25.

[16] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. 2011. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*. 241–250.

[17] Andrew Bortz and Dan Boneh. 2007. Exposing Private Information by Timing Web Applications. In *World Wide Web*. ACM, 621–628.

[18] Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl. 2016. Analyzing Runtime and Size Complexity of Integer Programs. *ACM Trans. Program. Lang. Syst.* 38, 4, Article 13 (Aug. 2016), 50 pages.

[19] David Brumley and Dan Boneh. 2003. Remote Timing Attacks Are Practical. In *Proceedings of the 12th USENIX Security Symposium, Washington, D.C., USA, August 4-8, 2003.*

[20] Quentin Carbonneaux, Jan Hoffmann, and Zhong Shao. 2015. Compositional Certified Resource Bounds. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, 467–478.

[21] Ezgi Çiçek, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Jan Hoffmann. 2017. Relational Cost Analysis. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, 316–329.

[22] Shuo Chen, Rui Wang, XiaoFeng Wang, and Kehuan Zhang. 2010. Side-Channel Leaks in Web Applications: A Reality Today, a Challenge Tomorrow. In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*. 191–206.

[23] Sujit Rokka Chhetri and Mohammad Abdullah Al Faruque. 2017. Side-Channels of Cyber-Physical Systems: Case Study in Additive Manufacturing. *IEEE Design & Test* (2017).

[24] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[25] Guillaume Dufay, Amy Felty, and Stan Matwin. 2005. Privacy-sensitive Information Flow with JML. In *Proceedings of the 20th International Conference on Automated Deduction (CADE' 20)*. Springer-Verlag, 116–130.

[26] Edward W. Felten and Michael A. Schneider. 2000. Timing attacks on Web privacy. In *CCS 2000, Proceedings of the 7th ACM Conference on Computer and Communications Security, Athens, Greece, November 1-4, 2000*. 25–32.

[27] Cormac Flanagan and K. Rustan M. Leino. 2001. Houdini, an Annotation Assistant for ESC/Java. In *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity (FME '01)*. Springer-Verlag, 500–517.

[28] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. 2002. Extended Static Checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI '02)*. ACM, New York, NY, USA, 234–245.

[29] Antonio Flores-Montoya and Reiner Hähnle. 2014. *Resource Analysis of Complex Programs with Cost Equations*. Springer International Publishing, Cham, 275–295.

[30] Riccardo Focardi and Roberto Gorrieri. 1995. A Classification of Security Properties for Process Algebras1. *Journal of Computer security* 3, 1 (1995), 5–33.

[31] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. 2001. Electromagnetic Analysis: Concrete Results. In *Proceedings of the Third International Workshop on Cryptographic Hardware and Embedded Systems (CHES '01)*. Springer-Verlag, 251–261.

[32] Nethanel Gelernter and Amir Herzberg. 2015. Cross-Site Search Attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1394–1405.

[33] Daniel Genkin, Itamar Pipman, and Eran Tromer. 2014. Get Your Hands Off My Laptop: Physical Side-Channel Key-Extraction Attacks on PCs. In *Cryptographic Hardware and Embedded Systems - CHES 2014 - 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proceedings*. 242–260.

[34] Jan Goguen and Meseguer Jose. 1982. Security policies and security models. In *Symposium on Security and Privacy*. IEEE Computer Society Press, 11–20.

[35] James W Gray III. 1992. Toward a mathematical foundation for information flow security. *Journal of Computer Security* 1, 3-4 (1992), 255–294.

[36] David Gullasch, Endre Bangerter, and Stephan Krenn. 2011. Cache Games - Bringing Access-Based Cache Attacks on AES to Practice. In *32nd IEEE Symposium on Security and Privacy, S&P 2011, 22-25 May 2011, Berkeley, California, USA*. 490–505.

[37] Sumit Gulwani, Krishna K. Mehra, and Trishul Chilimbi. 2009. SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '09)*. ACM, 127–139.

[38] Andreas Haeberlen, Benjamin C. Pierce, and Arjun Narayan. 2011. Differential Privacy Under Fire. In *20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12, 2011, Proceedings.*

[39] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2012. Multivariate Amortized Resource Analysis. *ACM Trans. Program. Lang. Syst.* 34, 3, Article 14 (Nov. 2012), 62 pages.

[40] Jan Hoffmann, Ankush Das, and Shu-Chun Weng. 2017. Towards Automatic Resource Bound Analysis for OCaml. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, 359–373.

[41] Jan Hoffmann and Martin Hofmann. 2010. Amortized Resource Analysis with Polynomial Potential: A Static Inference of Polynomial Bounds for Functional Programs. In *Proceedings of the 19th European Conference on Programming Languages and Systems (ESOP'10)*. Springer-Verlag, 287–306.

[42] Jan Hoffmann and Zhong Shao. 2014. *Type-Based Amortized Resource Analysis with Integers and Arrays*. Springer International Publishing, 152–168.

[43] Martin Hofmann and Steffen Jost. 2003. Static Prediction of Heap Space Usage for First-order Functional Programs. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '03)*. ACM, 185–197.

[44] Bertrand Jeannet and Antoine Miné. 2009. Apron: A Library of Numerical Abstract Domains for Static Analysis. In *Proceedings of the 21st International Conference on Computer Aided Verification (CAV '09)*. Springer-Verlag, 661–667.

[45] Paul Kocher. 1996. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Advances in CryptologyâĂŤCRYPTOâĂŽ96*. Springer, 104–113.

[46] Paul C. Kocher. 1996. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*. 104–113.

[47] Shuvendu K. Lahiri and Shaz Qadeer. 2009. Complexity and Algorithms for Monomial and Clausal Predicate Abstraction. In *Proceedings of the 22Nd International Conference on Automated Deduction (CADE-22)*. Springer-Verlag, 214–229.

[48] Chang Liu, Austin Harris, Martin Maas, Michael Hicks, Mohit Tiwari, and Elaine Shi. 2015. GhostRider: A Hardware-Software System for Memory Trace Oblivious Computation. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. ACM, 87–101.

[49] Robert Martin, John Demme, and Simha Sethumadhavan. 2012. TimeWarp: Rethinking Timekeeping and Performance Monitoring Mechanisms to Mitigate Side-channel Attacks. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA '12)*. IEEE Computer Society, 118–129.

[50] David Molnar, Matt Piotrowski, David Schultz, and David Wagner. 2006. The Program Counter Security Model: Automatic Detection and Removal of Control-flow Side Channel Attacks. In *Proceedings of the 8th International Conference on Information Security and Cryptology (ICISC'05)*. Springer-Verlag, 156–168.

[51] Andrew C Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. 2001. Jif: Java information flow. *Software release. Located at http://www. cs. cornell. edu/jif* 2005 (2001).

[52] David A. Naumann. 2006. *From Coupling Relations to Mated Invariants for Checking Information Flow*. Springer Berlin Heidelberg, 279–296.

[53] Van Chan Ngo, Mario Dehesa-Azuara, Matthew Fredrikson, and Jan Hoffmann. 2017. Verifying and Synthesizing Constant-Resource Implementations with Types. In *38st IEEE Symposium on Security and Privacy, S&P*.

[54] Corina Pasareanu, Quoc-Sang Phan, and Pasquale Malacaria. 2016. Multi-Run Side-Channel Analysis Using Symbolic Execution and Max-SMT. In *Computer Security Foundations Symposium*. IEEE.

[55] Corina S. Pasareanu, Quoc-Sang Phan, and Pasquale Malacaria. 2016. Multi-run Side-Channel Analysis Using Symbolic Execution and Max-SMT. In *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*. 387–400.

[56] François Pottier and Vincent Simonet. 2003. Information Flow Inference for ML. *ACM Trans. Program. Lang. Syst.* 25, 1 (Jan. 2003), 117–158.

[57] Ashay Rane, Calvin Lin, and Mohit Tiwari. 2015. Raccoon: Closing Digital Side-Channels through Obfuscated Execution. In *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association, Washington, D.C., 431–446.

[58] Ashay Rane, Calvin Lin, and Mohit Tiwari. 2016. Secure, Precise, and Fast Floating-Point Operations on x86 Processors. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Austin, TX, 71–86.

[59] Bruno Rodrigues, Fernando Magno Quintão Pereira, and Diego F. Aranha. 2016. Sparse Representation of Implicit Flows with Applications to Side-channel Detection. In *Proceedings of the 25th International Conference on Compiler Construction (CC 2016)*. ACM, 110–120.

[60] Andrei Sabelfeld and Andrew C Myers. 2003. Language-based information-flow security. *IEEE Journal on selected areas in communications* 21, 1 (2003), 5–19.

[61] Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, Percy Liang, and Aditya V. Nori. 2013. A Data Driven Approach for Algebraic Loop Invariants. In *Proceedings of the 22Nd European Conference on Programming Languages and Systems (ESOP'13)*. Springer-Verlag, 574–592.

[62] Reza Shokri and Vitaly Shmatikov. 2015. Privacy-Preserving Deep Learning. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*. 1310–1321.

[63] Moritz Sinn, Florian Zuleger, and Helmut Veith. 2014. *A Simple and Scalable Static Analysis for Bound Analysis and Amortized Complexity Analysis*. Springer

International Publishing, 745–761.

[64] Moritz Sinn, Florian Zuleger, and Helmut Veith. 2017. Complexity and Resource Bound Analysis of Imperative Programs Using Difference Constraints. *Journal of Automated Reasoning* (2017), 1–43.

[65] Marcelo Sousa and Isil Dillig. 2016. Cartesian Hoare Logic for Verifying K-safety Properties. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 57–69.

[66] Tachio Terauchi and Alexander Aiken. 2005. Secure Information Flow as a Safety Problem. In *Static Analysis, 12th International Symposium, SAS 2005, London, UK, September 7-9, 2005, Proceedings*. 352–367.

[67] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java Bytecode Optimization Framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON '99)*. IBM Press, 13–.

[68] Serge Vaudenay. 2002. Security Flaws Induced by CBC Padding - Applications to SSL, IPSEC, WTLS .... In *Advances in Cryptology - EUROCRYPT 2002, International Conference on the Theory and Applications of Cryptographic Techniques, Amsterdam, The Netherlands, April 28 - May 2, 2002, Proceedings*. 534–546.

[69] Jean Yang, Kuat Yessenov, and Armando Solar-Lezama. 2012. A language for automatically enforcing privacy policies. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 85–96.

[70] Yuval Yarom, Daniel Genkin, and Nadia Heninger. 2016. *CacheBleed: A Timing Attack on OpenSSL Constant Time RSA.* Springer Berlin Heidelberg, 346–367.

[71] Anna Zaks and Amir Pnueli. 2008. CoVaC: Compiler Validation by Program Analysis of the Cross-Product. In *Proceedings of the 15th International Symposium on Formal Methods (FM '08)*. Springer-Verlag, Berlin, Heidelberg, 35–51.

[72] Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. 2012. Language-based Control and Mitigation of Timing Channels. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, 99–110.

[73] Kehuan Zhang, Zhou Li, Rui Wang, XiaoFeng Wang, and Shuo Chen. 2010. Sidebuster: Automated Detection and Quantification of Side-channel Leaks in Web Application Development. In *Computer and Communications Security*. ACM, 595–606.

[74] Florian Zuleger, Sumit Gulwani, Moritz Sinn, and Helmut Veith. 2011. Bound Analysis of Imperative Programs with the Size-change Abstraction. In *Proceedings of the 18th International Conference on Static Analysis (SAS'11)*. Springer-Verlag, 280–297.

APPENDIX A: PROOF OF SOUNDNESS

Lemma .1. *Let program $P = \lambda\vec{p}.S$. If the following premises hold:*

- $\vec{p_1} = \alpha(\vec{p}), \vec{p_2} = \alpha(\vec{p})$
- $\models I \longrightarrow \vec{p_1}^{\,l} = \vec{p_2}^{\,l} \land \vec{p_1}^{\,h} \neq \vec{p_2}^{\,h}$
- Σ *is sound*
- $\Sigma \vdash \mathit{CanSynchronize}(e_1, e_2, S_1, S_2, I)$

Then $\models I \longrightarrow (e_1 \leftrightarrow e_2)$.

Proof. According to figure 6, if $\Sigma \vdash \text{CanSynchronize}(e_1, e_2, S_1, S_2, I)$, then at least one of the two conditions must be true:

- $\models I \longrightarrow (e_1 \leftrightarrow e_2)$
- $e_1 \equiv_\alpha e_2 \land S_1 \equiv_\alpha S_2 \land \Sigma \vdash e_1 : \textbf{low} \land \Sigma \vdash e_2 : \textbf{low}$

If the first condition is true, then the conclusion trivially holds. Otherwise, since Σ is sound, we know that e_1 and e_2 depend solely on $\vec{p_1}^{\,l}$ and $\vec{p_2}^{\,l}$, respectively. According to the first two premises, $I \longrightarrow \vec{p_1}^{\,l} = \vec{p_2}^{\,l}$. It follows that $I \longrightarrow e_1 = e_2$ and therefore $\models I \longrightarrow (e_1 \leftrightarrow e_2)$. □

Lemma .2. *Let* $\texttt{vars}(S)$ *be the set of all free variables in* S*. If* $\texttt{vars}(S_1) \cap \texttt{vars}(S_2) = \emptyset$*, then* $S_1; S_2$ *is semantically equivalent to* $S_2; S_1$.

Proof. Suppose $\Gamma \vdash S_1; S_2 : \Gamma', r$. Since $\texttt{vars}(S_1)$ and $\texttt{vars}(S_2)$ are mutually disjoint, we could break Γ into three partitions $\Gamma = \Gamma_1 \sqcup \Gamma_2 \sqcup \Gamma_3$, where $\mathit{dom}(\Gamma_1) = \texttt{vars}(S_1)$, $\mathit{dom}(\Gamma_2) = \texttt{vars}(S_2)$ and $\mathit{dom}(\Gamma_3) = \mathit{dom}(\Gamma) \setminus \texttt{vars}(S_1) \setminus \texttt{vars}(S_2)$. Since S_i does not touch Γ_j where $i \neq j$, we have

$$\Gamma_1 \vdash S_1 : \Gamma_1', r_1 \qquad \Gamma_2 \vdash S_2 : \Gamma_2', r_2$$

It follows that

$$\Gamma \vdash S_1 : \Gamma_1' \sqcup \Gamma_2 \sqcup \Gamma_3, r_1 \qquad \Gamma \vdash S_2 : \Gamma_1 \sqcup \Gamma_2' \sqcup \Gamma_3, r_2$$

$$\Gamma_1' \sqcup \Gamma_2 \sqcup \Gamma_3 \vdash S_2 : \Gamma_1' \sqcup \Gamma_2' \sqcup \Gamma_3, r_1 + r_2$$

$$\Gamma_1 \sqcup \Gamma_2' \sqcup \Gamma_3 \vdash S_1 : \Gamma_1' \sqcup \Gamma_2' \sqcup \Gamma_3, r_2 + r_1$$

Using the operational semantics rule for sequential composition shown in figure 4, this means

$$\Gamma \vdash S_1; S_2 : \Gamma_1' \sqcup \Gamma_2' \sqcup \Gamma_3, r_1 + r_2$$

$$\Gamma \vdash S_2; S_1 : \Gamma_1' \sqcup \Gamma_2' \sqcup \Gamma_3, r_2 + r_1$$

As $S_1; S_2$ and $S_2; S_1$ both have the same effect on Γ and consume the same amount of resource, they are semantically equivalent. □

Lemma .3. *Let program $P = \lambda\vec{p}.S$. Under the assumption that the following premises hold:*

- $\vec{p_1} = \alpha(\vec{p}), \vec{p_2} = \alpha(\vec{p})$
- $\models \Phi \longrightarrow \vec{p_1}^{\,l} = \vec{p_2}^{\,l} \land \vec{p_1}^{\,h} \neq \vec{p_2}^{\,h}$
- Σ *is sound*

If $\Sigma \vdash \langle\Phi\rangle\ S_1 \circledast S_2 \langle\Psi\rangle$*, then* $\vdash \{\Phi\}\ S_1; S_2\ \{\Psi\}$.

Proof. By structural induction on proof rules shown in figure 5.

- Rule (1).
  By inductive hypothesis, $\vdash \{\Phi\} S_2; S_1 \{\Psi\}$. Since S_1 and S_2 belongs to two different alpha-renamed copies of the program, we have $\texttt{vars}(S_1) \cap \texttt{vars}(S_2) = \emptyset$. Using lemma .2, we get $\vdash \{\Phi\} S_1; S_2 \{\Psi\}$

- Rule (2).
  By inductive hypothesis, $\vdash \{\Phi\}S_1; \mathbf{skip}; S_2\{\Psi\}$. As $S_1; \mathbf{skip}$ is semantically equivalent to S_1, we have $\vdash \{\Phi\}S_1; S_2\{\Psi\}$.
- Rule (3).
  By inductive hypothesis, $\vdash \{\Phi'\}S_2; S_3\{\Psi\}$. Also, we know $\{\Phi\}S_1\{\Phi'\}$. Using the sequence rule in standard Hoare logic, we derive $\vdash \{\Phi\}S_1; S_2; S_3\{\Psi\}$.
- Rule (4).
  By inductive hypothesis, $\vdash \{\Phi\}S\{\Psi\}$. As S is semantically equivalent to $S; \mathbf{skip}$, we get $\{\Phi\}S; \mathbf{skip}\{\Psi\}$.
- Rule (5).
  By inductive hypothesis, $\vdash \{\Phi \wedge e\}S_1; S; S_3\{\Psi_1\}$ and $\vdash \{\Phi \wedge \neg e\}S_2; S; S_3\{\Psi_2\}$. Since $\models \Psi_1 \rightarrow \Psi_1 \vee \Psi_2$ and $\models \Psi_2 \rightarrow \Psi_1 \vee \Psi_2$, according to the consequence rule in standard Hoare logic we have $\{\Phi \wedge e\}S_1; S; S_3\{\Psi_1 \vee \Psi_2\}$ and $\{\Phi \wedge \neg e\}S_2; S; S_3\{\Psi_1 \vee \Psi_2\}$. With the sequence rule in standard Hoare logic, assume

  (1) $\vdash \{\Phi \wedge e\}S_1\{\Phi_1\}$
  (2) $\vdash \{\Phi_1\}S; S_3\{\Psi_1 \vee \Psi_2\}$
  (3) $\vdash \{\Phi \wedge \neg e\}S_2\{\Phi_2\}$
  (4) $\vdash \{\Phi_2\}S; S_3\{\Psi_1 \vee \Psi_2\}$.

  Let $\Phi' = wp(\Psi_1 \vee \Psi_2)$. It follows immediately from (2) and (4) that $\Phi_1 \rightarrow \Phi'$ and $\Phi_2 \rightarrow \Phi'$. We could apply the consequence rule again to (1) and (3) and derive $\vdash \{\Phi \wedge e\}S_1\{\Phi'\}$ and $\vdash \{\Phi \wedge \neg e\}S_2\{\Phi'\}$. Using the condition rule in standard Hoare logic, we have $\{\Phi\}\mathbf{if}\ e\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\{\Phi'\}$. Combining (2), (4), sequence rule and the definition of wp, we could finally derive $\vdash \{\Phi\}\mathbf{if}\ e\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2; S; S_3\{\Psi_1 \vee \Psi_2\}$.
- Rule (6).
  By inductive hypothesis, $\vdash \{\Psi'\}S; S'\{\Psi\}$. We also know that $\vdash \{\Phi\}\mathbf{while}\ e_1\ \mathbf{do}\ S_1\{\Phi'\}$ and $\vdash \{\Phi'\}\mathbf{while}\ e_2\ \mathbf{do}\ S_2\{\Psi'\}$. Applying the sequence rule in standard Hoare logic twice, we get $\vdash \{\Phi\}\mathbf{while}\ e_1\ \mathbf{do}\ S_1; \mathbf{while}\ e_2\ \mathbf{do}\ S_2; S; S'\{\Psi\}$. Additionally, S and $\mathbf{while}\ e_2\ \mathbf{do}\ S_2$ comes from two different alpha-renamed copies so $\texttt{vars}(S) \cap \texttt{vars}(\mathbf{while}\ e_2\ \mathbf{do}\ S_2) = \emptyset$. We could apply lemma .2 and get
  $\vdash \{\Phi\}\mathbf{while}\ e_1\ \mathbf{do}\ S_1; S; \mathbf{while}\ e_2\ \mathbf{do}\ S_2; S'\{\Psi\}$
- Rule (7).
  By inductive hypothesis, $\vdash \{I \wedge e_1 \wedge e_2\}S_1; S_2\{I'\}$ and $\vdash \{I \wedge \neg e_1 \wedge e_2\}S; S'\{\Psi\}$. As $\models I' \rightarrow I$, we have $\vdash \{I \wedge e_1 \wedge e_2\}S_1; S_2\{I\}$ due to consequence rule. Now we may apply the while rule in standard Hoare logic to obtain $\vdash \{I\}\mathbf{while}\ e_1 \wedge e_2\ \mathbf{do}\ (S_1; S_2)\{I \wedge \neg(e_1 \wedge e_2)\}$.
  Now, as following two statements are semantically equivalent:
  - $\mathbf{while}\ e_1 \wedge e_2\ \mathbf{do}\ (S_1; S_2)$
  - $\mathbf{while}\ e_1 \wedge e_2\ \mathbf{do}\ (S_1; S_2); \mathbf{while}\ e_1\ \mathbf{do}\ S_1; \mathbf{while}\ e_2\ \mathbf{do}\ S_2$

  we could replace the former with the latter:

$$\vdash \{I\}\mathbf{while}\ e_1 \wedge e_2\ \mathbf{do}\ (S_1; S_2); \mathbf{while}\ e_1\ \mathbf{do}\ S_1; \\ \mathbf{while}\ e_2\ \mathbf{do}\ S_2\{I \wedge \neg(e_1 \wedge e_2)\}$$

  According to lemma .1, $\models I \rightarrow (e_1 \leftrightarrow e_2)$. But we also know that the precondition $I \wedge \neg(e_1 \wedge e_2)$ holds before the second loop $\mathbf{while}\ e_1\ \mathbf{do}\ S_1$. This implies $I \neg e_1 \wedge \neg e_2$ and therefore both of the two loops $\mathbf{while}\ e_1\ \mathbf{do}\ S_1$ and $\mathbf{while}\ e_2\ \mathbf{do}\ S_2$ would not execute, which means $\vdash \{I \wedge \neg(e_1 \wedge e_2)\}\mathbf{while}\ e_1\ \mathbf{do}\ S_1; \mathbf{while}\ e_2\ \mathbf{do}\ S_2\{I \wedge \neg(e_1 \wedge e_2)\}$. Applying the consequence rule here we end up with
  $\vdash \{I\}\mathbf{while}\ e_1\ \mathbf{do}\ S_1; \mathbf{while}\ e_2\ \mathbf{do}\ S_2\{I \wedge \neg(e_1 \wedge e_2)\}$. Combining this with $\models \Phi \rightarrow I$ and the second inductive hypothesis we finally get
  $\vdash \{\Phi\}\mathbf{while}\ e_1\ \mathbf{do}\ S_1; S; \mathbf{while}\ e_2\ \mathbf{do}\ S_2; S'\{\Psi\}$.

□

Theorem .4 (Soundness). *Assuming soundness of taint environment Σ, if $\Sigma \vdash SideChannelFree(\lambda\vec{p}.S,\ \epsilon)$, then the program $\lambda\vec{p}.S$ does not have an ϵ-bounded resource side-channel.*

Proof. We know that Σ is sound and $\models \Phi \rightarrow \vec{p_1}^{\,l} = \vec{p_2}^{\,l} \wedge \vec{p_1}^{\,h} \neq \vec{p_2}^{\,h}$. Therefore, lemma .3 applies, and we get $\vdash \{\Phi\}S_1^\tau; S_2^\tau\{\Psi\}$. Additionally, $\models \Psi \rightarrow |\tau_1 - \tau_2| \leq \epsilon|$. Using the consequence rule in standard Hoare logic, we obtain $\vdash \{\Phi\}S_1^\tau; S_2^\tau\{|\tau_1 - \tau_2| \leq \epsilon|\}$. By the soundness of Hoare logic, it follows that $\models \{\Phi\}S_1^\tau; S_2^\tau\{|\tau_1 - \tau_2| \leq \epsilon|\}$. By the soundness of self-composition, this means that

$$\forall \vec{a}_1, \vec{a}_2.\ \vec{a_1}^{\,l} = \vec{a_2}^{\,l} \wedge \vec{a_1}^{\,h} \neq \vec{a_2}^{\,h} \implies |\tau_1 - \tau_2| \leq \epsilon$$

By lemma 4.1, $\tau_1 = R_P(\vec{a}_1)$ and $\tau_2 = R_P(\vec{a}_2)$. Substitute τ with R_P we arrive at our conclusion

$$\forall \vec{a}_1, \vec{a}_2.\ (\vec{a_1}^{\,l} = \vec{a_2}^{\,l} \wedge \vec{a_1}^{\,h} \neq \vec{a_2}^{\,h}) \implies |R_P(\vec{a}_1) - R_P(\vec{a}_2)| \leq \epsilon$$

□