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 $\epsilon$-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 $\epsilon$-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 $\epsilon$, we would like to verify that the resource usage of $P$ does not vary by more than $\epsilon$ no matter what the value of the secret. Following the terminology of Goguen and Meseguer [34], we refer to this property as $\epsilon$-bounded non-interference. Intuitively, a program that violates $\epsilon$-bounded non-interference for even large values of $\epsilon$ exhibits significant secret-induced differences in resource usage.

The problem of verifying $\epsilon$-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 $\epsilon$-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
more precise relational reasoning techniques to automatically verify
that such hot spots do not violate $\epsilon$-bounded non-interference.

At the core of our technique is a new program logic called \textit{Quantitative Cartesian Hoare Logic (QCHL)} for verifying the $\epsilon$-bounded
non-interference property. QCHL leverages recent advances in re-
lational verification by building on top of \textit{Cartesian Hoare Logic (CHL)} [65] for verifying $k$-safety properties. Specifically, QCHL al-
 lows us to prove triples of the form $\langle \psi \rangle \models S \equiv (\phi)$, where $S$ is a program fragment and $\phi, \psi$ 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 $\epsilon$. Similar
to CHL, our QCHL logic allows effective relational verification by
symbolically executing two copies of the program in lockstep. How-
ever, 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
\textit{Themis}, a static analyzer for detecting resource-related side-channels
in Java applications. To demonstrate its effectiveness, we evaluate
\textit{Themis} by performing a series of experiments. First, we compare
\textit{Themis} against \textit{Blazer} [8], a state-of-the-art static analysis tool
for finding timing side-channels in Java applications, and we show
that \textit{Themis} compares favorably with \textit{Blazer}, both in terms of ac-
curacy and running time. Second, we use \textit{Themis} to analyze known
side-channel vulnerabilities in security-sensitive Java applications,
such as Tomcat and Spring-Security. We show that \textit{Themis} can
identify the defects in the original vulnerable versions of these pro-
grams, and that \textit{Themis} can verify the correctness of their repaired
versions. Finally, we run \textit{Themis} on several real-world Java appli-
cations and demonstrate that \textit{Themis} uncovers previously unknown
side-channel vulnerabilities in widely-used programs, such as the
Eclipse Jetty HTTP web server.

\textbf{Contributions}. In summary, this paper makes the following key
contributions:

- We propose the notion of $\epsilon$-bounded non-interference, which
can be used to reason about secret-induced variations in the
application’s resource usage behavior.
- We present \textit{Quantitative Cartesian Hoare Logic (QCHL)}, a
variant of CHL that can be used to verify $\epsilon$-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 \textit{Themis} and
evaluate it on multiple security-critical Java applications. We
also compare \textit{Themis} against \textit{Blazer}, a state-of-the-art timing
side channel detector for Java. Our results demonstrate that
\textit{Themis} is precise, useful, and scalable.

---

\footnote{\textit{Themis} is a Greek goddess for justice and balance, hence the name.}

\begin{figure}[h]
\centering
\begin{lstlisting}
public static BigInteger modPow(BigInteger base, 
    BigInteger exponent, BigInteger modulus) {
    BigInteger s = BigInteger.valueOf(1);
    // Big Integer r;
    int width = exponent.bitLength();
    for (int i = 0; i < width; i++) {
        s = s.multiply(s.mod(modulus));
        if (exponent.testBit(width - i - 1))
            s = s.multiply(base).mod(modulus);
    }
    //else r = s.multiply(base).mod(modulus);
    return s;
}
\end{lstlisting}
\caption{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.}
\end{figure}

\begin{figure}[h]
\centering
\begin{lstlisting}
{
    "epsilon":"0", "costModel":"time",
    "secrets": ["<com.cyberpointllc.stac.auth.KeyExchangeServer:java.math.BigInteger secretKey>"
}
\end{lstlisting}
\caption{\textit{Themis} configuration file for Gabfeed.}
\end{figure}

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

\textbf{Organization}. The rest of this paper is organized as follows. We
start by giving an overview of \textit{Themis} and explain our threat
model (Section 2). After formalizing the notion of $\epsilon$-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 \textit{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.

\section{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.

\subsection{Motivating Example}

Suppose that Bob, a security analyst at a government agency, re-
cieves a Java web application called Gabfeed, which implements
a web forum that allows community members to post and search
messages\footnote{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.}. 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 ver-
ify that it does not contain timing side-channel vulnerabilities that
may compromise user name or password information. However,
When measuring resource usage, we further assume that any variations 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 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 $\epsilon$-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

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

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 $\epsilon$-bounded non-interference:
(expr) ::= (const) | (var) | (expr) o (expr)
(o ∈ {+, −, ×, V, ∧, ...})
(stmt) ::= skip | consume((expr)) | (var) ::= (expr)
(stmts) ::= (stmt) | (stmt); (stmts)
| if (expr) then (stmts) else (stmts)
| while (expr) do (stmts)

〈params〉 ::= 〈param〉 | 〈param〉, 〈params〉
〈param〉 ::= 〈annot〉 〈var〉
〈annot〉 ::= low | high
〈prog〉 ::= λ〈params〉. (stmts)

Figure 3: Language used in our formalization

Definition 2. A program $P$ obeys the $\epsilon$-bounded non-interference property if

$$\forall a_1, a_2. \quad (\vec{a}^l_1 = \vec{a}^l_2 \land \vec{a}^h_1 \neq \vec{a}^h_2) \Rightarrow |R_P(\vec{a}_1) - R_P(\vec{a}_2)| \leq \epsilon$$

In this definition, any variation in resource usage below $\epsilon$ 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 $\epsilon$-bounded interference considers a program to be secure as long as the difference in resource usage is "minor" according to the constant $\epsilon$. In practice, the value of $\epsilon$ 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 $\epsilon$, 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 $\epsilon$-bounded non-interference using QCHL, a variant of Cartesian Hoare Logic introduced in recent work for verifying $\kappa$-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 $\phi, \psi$ 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 $\epsilon$-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.

$$\begin{align*}
P = \lambda \vec{a}. S & \quad \forall p_i \in \vec{a}. \Gamma(p_i) = a_i \quad \Gamma \vdash S : \Gamma', r \\
r & \quad R_P(\vec{a}) = r \\
S = \text{skip} & \quad \Gamma \vdash S : \Gamma, 0 \\
S = (x := e) & \quad \Gamma \vdash e : v \\
\Gamma \vdash S : \Gamma', 0 \\
S = \text{consume} (e) & \quad \Gamma \vdash e : v \\
S = S_1 ; S_2 & \quad \Gamma \vdash S_1 ; S_2 : \Gamma', r_1 \\
\Gamma \vdash S_1 : \Gamma_1, r_1 \\
\Gamma \vdash S_2 : \Gamma_2, r_1 + r_2 \\
\Gamma \vdash S : \Gamma', r'
\end{align*}$$

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 $\Gamma$ (mapping variables to values), then $S$ consumes $r$ units of resource and the new environment is $\Gamma'$. 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 $\epsilon$-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...
with the exception of Rule (0), derive judgments of the form
\[ \lambda \rho, S_t = \alpha(\lambda \rho, S_t) \]
\[ \Sigma \vdash \langle \Phi \rangle S_t \supset S'_t (\Psi) \]
\[ \Sigma \vdash \text{SideChannelFree}(\lambda \rho, S, \epsilon) \]
\[ \Phi = (\rho_1^l = \rho_2^l \land \rho_1^h \neq \rho_2^h) \]
\[ |r_1 - r_2| \leq \epsilon \]
\[ \Sigma \vdash \langle \Phi \rangle S_2 \vdash S_2' (\Psi) \]
\[ \Sigma \vdash \langle \Phi \rangle S_1 \vdash S_1' (\Psi) \]
\[ \Sigma \vdash \langle \Phi \rangle S \vdash \text{SideChannelFree}(\epsilon) \]
\[ \Sigma \vdash \langle \Phi \rangle S \vdash \text{CanSynchronize}(e_1, e_2, S_1, S_2, I) \]
\[ |\Phi \rightarrow I| = |\Phi' \rightarrow I'| \]
\[ \Sigma \vdash \langle \Phi \rangle \text{while} e_1 \text{ do } S_1 \text{ and while } e_2 \text{ do } S_2; S'_t (\Psi) \]
\[ \Sigma \vdash \langle \Phi \rangle \text{ while } e_2 \text{ do } S_2; S'_t (\Psi) \]
\[ \Sigma \vdash \langle \Phi \rangle \text{ while } e_2 \text{ do } S_2; S'_t (\Psi) \]
\[ \Sigma \vdash \langle \Phi \rangle \text{ while } e_1 \text{ do } S_1; \text{ and while } e_2 \text{ do } S_2; S'_t (\Psi) \]

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

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.

Let us now consider the remaining rules in more detail: Rule (1) is the same as commutativity rule in CHL and states that the \( \oplus \) operator is symmetric. Intuitively, since \( S_1 \) and \( S_2 \) do not share variables, any interleaving of \( S_1 \) and \( S_2 \) 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.
Applying Hoare-style strongest postcondition computation, the above judgement can be reduced to the standard Hoare triple \( \{ \Phi \} S \{ \Psi \} \) because skip is just a no-op.

**Example.** Suppose we want to prove \((0\text{-bounded})\) non-interference for the following program:

\[
\begin{align*}
\lambda (\text{low } x) . & \quad \text{consume}(x); \text{skip}; \\
\end{align*}
\]

First we apply transformation \( \gamma \) and get the resource instrumented program:

\[
\lambda (\text{low } x) . \quad \tau = \tau + x; \text{skip};
\]

Ignoring the taint environment for now, as we will not use it in this example. According to rule (6), we only need to prove

\[
\begin{align*}
\langle x_1 = x_2 \rangle \quad & (\tau_1 = 0; \tau_1 + x_1; \text{skip}; \; \tau_2 = 0; \tau_2 + x_2; \text{skip}; \; \tau_1 = \tau_2)
\end{align*}
\]

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

\[
\begin{align*}
\langle x_1 = x_2 \& \tau_1 = x_1 \rangle \quad & \text{skip}; \; \tau_2 = 0; \tau_2 + x_2; \text{skip}; \; \tau_1 = \tau_2
\end{align*}
\]

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

\[
\begin{align*}
\langle x_1 = x_2 \& \tau_1 = x_1 \rangle \quad & \tau_2 = 0; \tau_2 + x_2; \text{skip}; \; \tau_1 = \tau_2
\end{align*}
\]

After applying rule (4), we get

\[
\begin{align*}
\langle x_1 = x_2 \& \tau_1 = x_1 \rangle \quad & \text{skip}; \; \tau_2 = 0; \tau_2 + x_2; \text{skip}; \; \tau_1 = \tau_2
\end{align*}
\]

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

\[
\begin{align*}
\langle x_1 = x_2 \& \tau_1 = x_1 \& \tau_2 = x_2 \rangle \quad & \text{skip}; \; \tau_1 = \tau_2
\end{align*}
\]

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

**Example.** Consider the following code snippet:

\[
\begin{align*}
e_1 = \alpha(e) & \quad e_2 = \alpha(e) \\
S_1 = \alpha(S) & \quad S_2 = \alpha(S) \\
S_1 \equiv S_2 & \quad S \equiv S \\text{low} \\
I \equiv e_1 \leftrightarrow e_2 & \quad S \equiv S \\text{low} \\
\Sigma \vdash \text{CanSynchronize}(e_1, e_2, S_1, S_2, I) & \quad \Sigma \vdash I \rightarrow (e_1 \leftrightarrow e_2)
\end{align*}
\]

To prove that this program obeys \( \epsilon \)-bounded non-interference, we need to show that the difference in resource consumption after executing the two copies of the loop is at most \( \epsilon \). 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 \( L_1 \equiv \text{while}(e_1) \) do \( S_1 \) and \( L_2 \equiv \text{while}(e_2) \) do \( S_2 \), and a loop invariant \( I \) for the "fused" loop while\( (e_1 \& e_2) \) 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 \( \Sigma \vdash (I \& e_1 \& e_2) S_1 \oplus S_2 (I') \), together with \( \models I' \rightarrow I \), ensures that \( I \) is an inductive invariant of the fused loop while\( (e_1 \& e_2) \) do \( S_1; S_2 \). Thus, \( I \) must hold when both loops terminate. Thus, we can safely use the predicate \( I \& \neg e_1 \& \neg e_2 \) 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 $\not\vdash e \rightarrow g$ then
7:            Guesses $\leftarrow$ Guesses\{g\}
8:    inductive $\leftarrow$ false
9:    while $\neg$ inductive do
10:       $I \leftarrow \bigwedge_{g \in$Guesses} $g$
11:       assume $\Sigma \vdash (I \land e_1 \land e_2)S_1 \otimes S_2(I')$
12:       inductive $\leftarrow$ true
13:    for $g \in$ Guesses do
14:        if $\not\vdash I' \rightarrow g$ then
15:            Guesses $\leftarrow$ Guesses\{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. Specifically, consider the universe of predicates $\{v_1 = v_2 \mid v \in V\}$ which is a suitable inductive relational loop invariant, because:

- $i_1, i_2, \tau_1, \tau_2$ 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 $k_1 = k_2$.
- $\tau_1$ and $\tau_2$ 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 $\Sigma$, if $\Sigma \vdash SideChannelFree(\lambda b. S, e)$, then the program $\lambda b. S$ does not have an $e$-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.

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 of predicates $\{v_1 = v_2 \mid v \in V\}$ 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 out those predicates that are not implied by the precondition $\Phi$ (lines 3-7). 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 $\Sigma \vdash (I \land e_1 \land e_2)S_1 \otimes S_2(I')$ and $\not\vdash I' \rightarrow g$, 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 $\Sigma \vdash (I \land e_1 \land e_2)S_1 \otimes S_2(I'')$ and $\vdash I' \rightarrow I$; 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 a `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:

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

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

bar() {
    int y = readSecret();
    if(y \( \leq 0 \)) 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 \( \epsilon \), 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 \( T_1 \) or allocation \( A_2 \) of type \( T_2 \), then the polymorphic call site \( a.foo() \) will be modeled as:

```java
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 64GB 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 \( \epsilon \)-bounded variant), we set the value of \( \epsilon \) to be 0 when running Themis.

We summarize the results of our comparison against Blazer in Table 8. 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].

---

\[3\] 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.
While we compare the lines of code (in Soot IR) in the original application which we do not have suitable drivers, we manually construct a 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 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 analysis, whereas the number of reachable methods after taint analysis ranges from 6 to 35, with an average of 13, after taint analysis. Thus, pruning using taint information makes the job of the QCHL verifier significantly easier.

**Benefit of $\epsilon$.** To justify the need for our relaxed notion of non-interference, Figure 9 also shows the results of the same experiment using an $\epsilon$ 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 $\epsilon$ 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 $\epsilon$-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 invariants 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,
public boolean check(Object credentials) {
    if (credentials instanceof char[]) {
        credentials = new String((char[]) credentials);
    }
    if (!((credentials instanceof String) && !(credentials instanceof Password))){
        LOG.warn("Can't check " + credentials.getClass() + " against CRYPT");
    }
    String passwd = credentials.toString();
    // FIX: return stringEquals(_cooked, UnixCrypt.crypt(passwd,_cooked));
    return _cooked.equals(UnixCrypt.crypt(passwd,_cooked));
}

/**
 * Utility method that replaces String.equals() to avoid timing attacks.
 */
static boolean stringEquals(String s1, String s2) {
    boolean result = true;
    int l1 = s1.length();
    int l2 = s2.length();
    if(l1 != l2) result = false;
    for (int i = 0; i < n; i++)
        result &= s1.charAt(i) == s2.charAt(i);
    return result;
}

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 \( \epsilon \) 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 Taintflex 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 surveyed 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
The general idea underlying self-composition is as follows: Given a program $P$ and a safety property $\phi$, we create a new program $P'$ which sequentially composes two $\alpha$-renamed copies of $P$ and then asserts that $\phi$ 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 [63], 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 $\epsilon$-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 $\epsilon$-bounded non-interference for any value of $\epsilon$.

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 $\epsilon$-bounded non-interference for arbitrary values of $\epsilon$. 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 Potter 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 whileThemis 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 \( \epsilon \)-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 \( \epsilon \), but we would also like to automatically infer the smallest \( \epsilon \) 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

APPENDIX A: PROOF OF SOUNDNESS

LEMMA 1. Let program \( P = \lambda \vec{p}.S \). If the following premises hold:

- \( \vec{p}_1 = \alpha(\vec{p}), \vec{p}_2 = \alpha(\vec{p}) \)
- \( \models I \rightarrow \vec{p}_1 = \vec{p}_2 \land \vec{p}_1 \neq \vec{p}_2 \)
- \( \Sigma \) is sound
- \( \Sigma \vdash \text{CanSynchronize}(e_1, e_2, S_1, S_2, I) \)

Then \( \models I \rightarrow (e_1 \leftrightarrow e_2) \).

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:

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

If the first condition is true, then the conclusion trivially holds. Otherwise, since \( \Sigma \) is sound, we know that \( e_1 \) and \( e_2 \) depend solely on \( \vec{p}_1 \) and \( \vec{p}_2 \), respectively. According to the first two premises, \( I \rightarrow \vec{p}_1 = \vec{p}_2 \). It follows that \( I \rightarrow e_1 = e_2 \) and therefore \( \models I \rightarrow (e_1 \leftrightarrow e_2) \).

LEMMA 2. Let \( \text{vars}(S) \) be the set of all free variables in \( S \). If \( \text{vars}(S_1) \cap \text{vars}(S_2) = 0 \), then \( S_1, S_2 \) is semantically equivalent to \( S_2, S_1 \).

Proof. Suppose \( \Gamma \vdash S_1, S_2 : \Gamma', r \). Since \( \text{vars}(S_1) \) and \( \text{vars}(S_2) \) are mutually disjoint, we could break \( \Gamma \) into three partitions \( \Gamma = \Gamma_1 \cup \Gamma_2 \cup \Gamma_3 \) where \( \text{dom}(\Gamma_1) = \text{vars}(S_1), \text{dom}(\Gamma_2) = \text{vars}(S_2) \) and \( \text{dom}(\Gamma_3) = \text{dom}(\Gamma) \setminus \text{vars}(S_1) \setminus \text{vars}(S_2) \). Since \( S_1 \) does not touch \( \Gamma_j \) where \( i \neq j \), we have

\[
\Gamma_1 \vdash S_1 : \Gamma'_1 \cup \Gamma_3, r_1 \quad \Gamma_2 \vdash S_2 : \Gamma'_2 \cup \Gamma_3, r_2
\]

It follows that

\[
\Gamma_1 \vdash S_1 : \Gamma'_1 \cup \Gamma_2 \cup \Gamma_3, r_1 + r_2 \quad \Gamma_2 \vdash S_2 : \Gamma'_2 \cup \Gamma_1 \cup \Gamma_3, r_1 + r_2
\]

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

\[
\Gamma \vdash S_1, S_2 : \Gamma'_1 \cup \Gamma'_2 \cup \Gamma_3, r_1 + r_2
\]

As \( S_1, S_2 \) both have the same effect on \( \Gamma \) 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:

- \( \vec{p}_1 = \alpha(\vec{p}), \vec{p}_2 = \alpha(\vec{p}) \)
- \( \models \Phi \rightarrow \vec{p}_1 = \vec{p}_2 \land \vec{p}_1 \neq \vec{p}_2 \)
- \( \Sigma \) is sound

If \( \Sigma \vdash \Phi \vdash S_1 \otimes S_2(\Psi) \), then \( \vdash \Phi \vdash S_1 \otimes S_2(\Psi) \).

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

- Rule (1).
  - By inductive hypothesis, \( \vdash \Phi \vdash S_1 \otimes S_2(\Psi) \). Since \( S_1 \) and \( S_2 \) belong to two different alpha-renamed copies of the program, we have \( \text{vars}(S_1) \cap \text{vars}(S_2) = 0 \). Using lemma 2, we get
    \( \vdash \Phi \vdash S_1 \otimes S_2(\Psi) \)
• Rule (2).
By inductive hypothesis, ⊢ {Φ} S₁; skip S₂ {Ψ}. As S₁; skip is semantically equivalent to S₁, we have ⊢ {Φ} S₁; S₂ {Ψ}.
• Rule (3).
By inductive hypothesis, ⊢ {Φ'} S₁; S₂ {Ψ}. Also, we know {Φ} S₁ {Φ'}. Using the sequence rule in standard Hoare logic, we derive ⊢ {Φ} S₁; S₂ {Ψ}.
• Rule (4).
By inductive hypothesis, ⊢ {Φ} S {Ψ}. As S is semantically equivalent to S₁; skip, we get {Φ} S; skip {Ψ}.
• Rule (5).
By inductive hypothesis, ⊢ {Φ ∧ e} S₁; S₂; S₁ {Ψ₁} and ⊢ {Φ ∧ ¬e} S₂; S₂ {Ψ₂}. Since ⊢ Ψ₁ → Ψ₁ ∨ Ψ₂ and ⊢ Ψ₂ → Ψ₁ ∨ Ψ₂, according to the consequence rule in standard Hoare logic we have {Φ ∧ e} S₁; S₂; S₁ {Ψ₁ ∨ Ψ₂} and {Φ ∧ ¬e} S₂; S₂ {Ψ₁ ∨ Ψ₂}. With the sequence rule in standard Hoare logic, assume
(1) ⊢ {Φ ∧ e} S₁ {Ψ₂}
(2) ⊢ {Φ} S₁; S₂ {Ψ₁ ∨ Ψ₂}
(3) ⊢ {Φ ∧ ¬e} S₂ {Ψ₂}
(4) ⊢ {Φ} S₁; S₂ {Ψ₁ ∨ Ψ₂}.
Let Φ' = wp(Ψ₁ ∨ Ψ₂). It follows immediately from (2) and (4) that Φ₁ → Φ' and Φ₂ → Φ'. We could apply the consequence rule again to (1) and (3) and derive ⊢ {Φ ∧ e} S₁ {Φ'} and ⊢ {Φ ∧ ¬e} S₂ {Φ'}. Using the condition rule in standard Hoare logic, we have {Φ} if e then S₁ else S₂ {Φ'}. Combining (2), (4), sequence rule and the definition of wp, we could finally derive ⊢ {Φ} if e then S₁ else S₂; S₁ {Ψ₁ ∨ Ψ₂}.
• Rule (6).
By inductive hypothesis, ⊢ {Ψ'} S₁; S₁' {Ψ}. We also know that ⊢ {Φ} while e₁ do S₁ {Ψ'} and ⊢ {Φ'} while e₂ do S₂ {Ψ'}. Applying the sequence rule in standard Hoare logic twice, we get ⊢ {Φ} while e₁ do S₁; while e₂ do S₂ {Ψ'}. Additionally, S and while e₂ do S₂ comes from two different alpha-renamed copies so \( \text{var}(S) \cap \text{var}(\text{while e₂ do S₂}) = \emptyset \). We could apply lemma 2 and get ⊢ {Φ} while e₁ do S₁; S₂; while e₂ do S₂; S₁' {Ψ}.
• Rule (7).
By inductive hypothesis, ⊢ {I ∧ e₁ ∧ e₂} S₁; S₂ {I'} and ⊢ {I ∧ ¬e₁ ∧ e₂} S₁; S₁' {Ψ}. As \( \models I' \rightarrow I \), we have ⊢ {I ∧ e₁ ∧ e₂} S₁; S₂ {I} due to consequence rule. Now we may apply the while rule in standard Hoare logic to obtain ⊢ {I} while e₁ ∧ e₂ do (S₁; S₂)(I ∧ ¬(e₁ ∧ e₂)).
Now, as following two statements are semantically equivalent:
- while e₁ ∧ e₂ do (S₁; S₂)
- while e₁ ∧ e₂ do (S₁; S₂); while e₁ do S₁; while e₂ do S₂
we could replace the former with the latter:

\[ \text{while e₁ ∧ e₂ do (S₁; S₂); while e₁ do S₁; while e₂ do S₂} \]

According to lemma 1, \( \models I \rightarrow (e₁ ↔ e₂) \). But we also know that the precondition \( I ∧ ¬(e₁ ∧ e₂) \) holds before the second loop while e₁ do S₁. This implies \( I \rightarrow ¬(e₁ ∧ ¬e₂) \) and therefore both of the two loops while e₁ do S₁ and while e₂ do S₂ would not execute, which means ⊢ {I ∧ ¬(e₁ ∧

**Theorem 4 (Soundness).** Assuming soundness of taint environment \( \Sigma \), if \( \Sigma \vdash \text{SideChannelFree}(λ\mathcal{P}.S, e) \), then the program \( λ\mathcal{P}.S \) does not have an \( ϵ \)-bounded resource side-channel.

**Proof.** We know that \( Σ \) is sound and \( \models \Phi \rightarrow p₁^I = p₂^I \land \overline{p₁}^h \neq p₂^h \). Therefore, lemma 3 applies, and we get ⊢ {Φ} S₁' ; S₂' {Ψ}. Additionally, \( \models \Psi \rightarrow |τ₁ - τ₂| \leq ϵ \). Using the consequence rule in standard Hoare logic, we obtain ⊢ {Φ} S₁' ; S₂' {Ψ}. By the soundness of Hoare logic, it follows that \( \models [Φ] S₁' ; S₂' \{ |r₁ - r₂| \leq ϵ \} \). By the soundness of self-composition, this means that

\[ \forall \overline{a₁}, \overline{a₂}, \overline{b₁}, \overline{b₂}, (\overline{a₁} \land \overline{a₂} \land \overline{b₁} \neq \overline{b₂}) \implies |R(\overline{a₁}) - R(\overline{a₂})| \leq ϵ \]

By lemma 4.1, \( r₁ = R(\overline{a₁}) \) and \( r₂ = R(\overline{a₂}) \). Substitute \( r \) with \( R \) we arrive at our conclusion

\[ \forall \overline{a₁}, \overline{a₂}, (\overline{a₁} = \overline{a₂} \land \overline{b₁} \neq \overline{b₂}) \implies |R(\overline{a₁}) - R(\overline{a₂})| \leq ϵ \]

□