Symbolic Path Cost Analysis for Side-Channel Detection*

Tegan Brennan
University of California Santa Barbara
Santa Barbara, CA, USA
tegan@cs.ucsb.edu

Seemanta Saha
University of California Santa Barbara
Santa Barbara, CA, USA
seemantasaha@cs.ucsb.edu

Tevfik Bultan
University of California Santa Barbara
Santa Barbara, CA, USA
bultan@cs.ucsb.edu

Corina S. Păsăreanu
CMU West and NASA Ames Research Center
Moffet Field, CA, USA
corina.s.pasareanu@nasa.gov

ABSTRACT

Side-channels in software are an increasingly significant threat to the confidentiality of private user information, and the static detection of such vulnerabilities is a key challenge in secure software development. In this paper, we introduce a new technique for scalable detection of side- channels in software. Given a program and a cost model for a side-channel (such as time or memory usage), we decompose the control flow graph of the program into nested branch and loop components, and compositionally assign a symbolic cost expression to each component. Symbolic cost expressions provide an over-approximation of all possible observable cost values that components can generate. Queries to a satisfiability solver on the difference between possible cost values of a component allow us to detect the presence of imbalanced paths (with respect to observable cost) through the control flow graph. When combined with taint analysis that identifies conditional statements that depend on secret information, our technique answers the following question: Does there exist a pair of paths in the program's control flow graph, differing only on branch conditions influenced by the secret, that differ in observable side-channel value by more than some given threshold? Additional optimization queries allow us to identify the minimal number of loop iterations necessary for the above to hold or the maximal cost difference between paths in the graph. We perform symbolic execution based feasibility analyses to eliminate control flow paths that are infeasible. We implemented our techniques in a prototype, and we demonstrate its favourable performance against state-of-the-art tools as well as its effectiveness and scalability on a set of sizable, realistic Java server-client and peer-to-peer applications.

CCS CONCEPTS

• **Security and privacy** → *Software and application security*; • **Software and its engineering** → *Automated static analysis*;

KEYWORDS

Side-channel analysis, static analysis, symbolic execution

ACM Reference Format:
Tegan Brennan, Seemanta Saha, Tevfik Bultan, and Corina S. Păsăreanu. 2018. Symbolic Path Cost Analysis for Side-Channel Detection. In *Proceedings of 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'18).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3213846.3213867

1 INTRODUCTION

Modern software systems frequently store and manipulate personal data, such as location, credit card information or medical history. Confidentiality of such private data is critical for users of these systems. Many software development practices, such as the encryption of packages sent over a network, aim to protect the confidentiality of private data by ensuring that an observer is unable to parse any meaningful output from the system. Under these protections, the software system's main communication channels, such as the content of the network packets it sends, or the output it writes to a public file, should not leak information about the private data. However, protecting the main communication channels is not sufficient, and serious security vulnerabilities can still exist even if information is not leaked from a main channel.

A class of information leaks, referred to as side-channels, use non-functional properties of program execution to obtain information about private data. Potential side-channels include those in execution time, memory usage, size and timings of network packets that are sent, and power consumption.

Though side-channel vulnerabilities have been known for many years [18], they are still often neglected by software developers. They are commonly thought of as impractical despite a growing number of demonstrations of realistic side-channel attacks that result in critical security vulnerabilities [11, 23, 27]. This makes scalable analysis techniques and tools devoted to the detection of side-channel vulnerabilities a necessity [14].

*This material is based on research sponsored by NSF under grant CCF-1548848 and by DARPA under the agreement number FA8750-15-2-0087. 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 DARPA or the U.S. Government.

ISSTA'18, July 16–21, 2018, Amsterdam, Netherlands

ACM ISBN 978-1-4503-5699-2/18/07…$15.00
https://doi.org/10.1145/3213846.3213867

In this paper, we present a static, scalable analysis technique for detecting side-channels in software systems. Given a set of secret variables, a type of side-channel, and a program, our goal is to detect the set of branch conditions responsible for potential side-channels of the given type, and generate a pair of witness paths in the control flow graph for the detected side-channel. Our technique achieves this by analyzing the control flow graph of the program with respect to a cost model (such as time or memory usage), and identifies if a change in the secret value can cause a detectable change in the observed cost of the program behavior. It also generates a pair of witness paths in the control flow graph, differing only on the branch conditions influenced by the secret, whose observable output under the given side-channel differs by more than some user defined threshold. We present an extension to our technique to determine the minimum number of loop iterations necessary for a side-channel to be detectable as well as the maximal cost difference between any two paths through the control flow graph. We implemented our approach in a prototype tool, CoCo-Channel (COmpositional COnstraint-based side-CHANNEL analyzer), for analyzing Java programs. Our approach consists of following four phases:

(1) Annotated Control Flow Graph Extraction and Decomposition: We use taint analysis to identify the branch conditions in the program that are influenced by the secret information. Then, we extract a reducible control flow graph where each basic block is annotated with a cost based on the cost model used. We decompose the control flow graph to a set of nested loop and branch components to facilitate compositional symbolic cost expression construction.

(2) Symbolic Cost Expression Construction: We compositionally construct symbolic cost expressions for components based on their nesting relationships. We use one integer variable per branch and loop component, that encode which branch is taken and how many times a loop is executed, respectively.

(3) Constraint Solver Query Generation: We generate queries for constraint solvers that are satisfiable if only if there exist two paths in the control flow graph, that differ only on the secret-dependent branch conditions, whose observable output under the given side-channel differs by more than some user defined threshold. As an extension, we also generate optimization queries that identify two such paths while minimizing the number of loop executions or maximizing the cost difference.

(4) Feasibility Analysis: Not all paths through a control flow graph are executable. To reduce potential false positives, we perform two types of feasibility checks. The first determines whether the witness paths themselves are executable and the second generates constraints that must be satisfied in order for a pair of individually feasible witness paths to be executable on the same public input.

After a motivating example, we will explain the phases of our analysis in order, followed with experimental evaluations, related work and conclusions.

2 A MOTIVATING EXAMPLE

Consider the application code given in Figure 1(a) and its corresponding control flow graph in Figure 1(b). This code was extracted from a peer-to-peer messaging application provided as part of the DARPA STAC (Space-Time Analysis for Cybersecurity) Program [1, 3] and contains a timing side-channel leaking whether two users have been previously connected. We demonstrate how an analysis of the program paths and their costs could detect the side-channel.

Annotation. We first identify the branch conditionals impacted by the secret value. There are three such conditionals denoted by b_1, b_2 and b_3 in the control flow graph. We also ascribe to each node a cost. To simplify the presentation, we use a simple cost model where a node can take one of three costs. 1 denotes an inexpensive node, 2 a moderately expensive node, and 3 an expensive node. In Figure 1, these costs are shown in the center of each node.

Decomposition. We decompose the annotated graph into a set of nested or disjoint components. In this example, all components are determined by the branch conditionals. The first of the four components present in the given control flow graph is determined by the branch node b_2 and its merge node m_2. This component is nested within the component determined by the branch node b_1 and its merge node m_1. The components determined by the pairs (b_3, m_3) and (b_4, m_4) constitute the remaining graph components.

Symbolic Cost Expression Construction. Our analysis compositionally derives a symbolic cost expression and corresponding set of constraints for each component. Evaluating this cost expression on satisfying instantiations of its symbolic variables generates a set of costs that span the set of all possible costs of paths through the component. In our example, the cost expression for the component associated with b_2 is given by $0 \cdot v_{b2} + 2 \cdot (1 - v_{b2}) + 1 + 1$, where v_{b2} is a symbolic variable constrained to 0 or 1 and denotes which branch was taken at b_2. The cost expression for the component associated with b_1, $0 \cdot v_{b1} + (1 - v_{b1}) \cdot (0 \cdot v_{b2} + 2 \cdot (1 - v_{b2}) + 1 + 1) + 1 + 1$, reuses the cost expression of its nested component. The cost expressions for the other components are formed analogously to that of b_2.

Constraint Solver Query Generation. Queries on the cost expressions and the associated constraints on their variables can be formulated to determine the existence of two paths through any component differing only on secret-dependent branches whose costs differ by more than some threshold. Since both b_1 and b_2 are secret-dependent, if there are two pairs of satisfying assignments to v_{b2} and v_{b1} such that the difference in the evaluation of the cost expression, b_1, $0 \cdot v_{b1} + (1 - v_{b1}) \cdot (0 \cdot v_{b2} + 2 \cdot (1 - v_{b2}) + 1 + 1) + 1 + 1$ is greater than the threshold, then there are two paths through the component satisfying this criteria. However, 1 and 2 are not expensive costs, meaning such a satisfying assignment is not possible. The next component to analyze is that of b_3 whose associated cost expression $1 \cdot v_{b3} + (1 + 3) \cdot (1 - v_{b3}) + 1 + 1$ *does* have two satisfying assignments ($v_{b3} = 0, v_{b3} = 1$) to its secret-dependent symbolic variable v_{b3} such that the difference in its evaluation under those assignments is greater than our threshold. From this satisfying assignment, we learn that the choice made at branch conditional b_3 impacts the cost of two paths through the component in an observable manner. Analyzing the component associated with b_4, we observe that the two resultant branches also differ significantly in terms of cost. However, b_4 is a non secret-dependent conditional meaning that this cost imbalance does not introduce a side-channel to the application. This is modeled by a constraint that v_{b4} must match in any two assignments of variables that are compared for cost differences. In the general case, results across disjoint components would be combined to determine if multiple non-nested branch conditions introduce a side-channel.

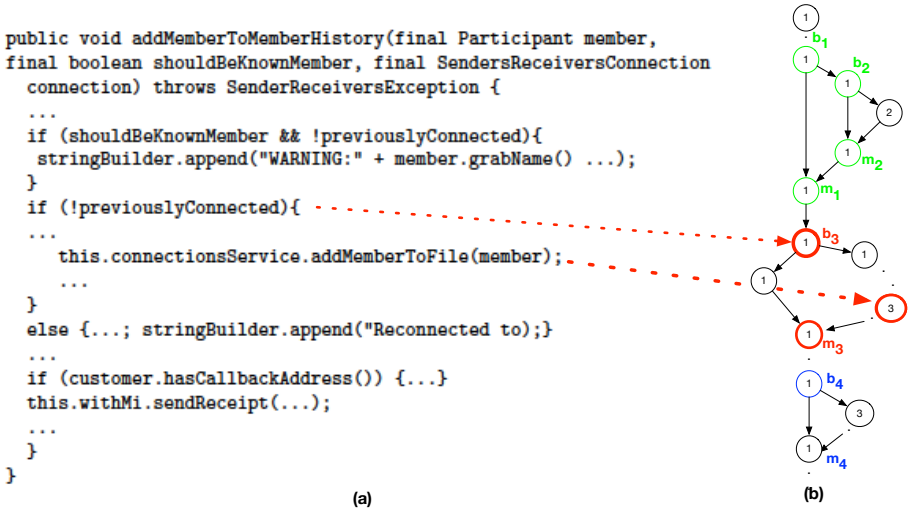

Figure 1: (a) Program vulnerable to a timing side-channel. (b) Control flow graph representation of the program. Arrows indicate the relationship between the code and resultant graph structures.

We have demonstrated how queries on the set of possible costs of paths through the control flow graph can aid in the detection of side-channels. Explicit enumeration of all costs is rarely possible on programs of a meaningful size. Thus, we introduce a technique to compositionally generate a set of symbolic expressions and constraints characterizing all possible observable costs, and formulate queries over these expressions answerable by constraint solvers.

3 CFG EXTRACTION AND DECOMPOSITION

A program's control flow graph provides a representation of all paths that might be traversed during its execution. We augment the control flow graph of a program by 1) a cost model and 2) secret-dependent tainting.

Cost Model. Each run of a program returns not only its specified output, but also an observable value denoting the side-channel measurements of that execution of the program. For a given type of side-channel, we use a **cost model** to obtain static approximations of the observables that can be determined without running the system under test. For example, a cost model for timing side-channels might return the number of bytecode instructions along the control flow path while a cost model for side-channels in space might return the number of bytes written to a file. For our analysis, we assume that each node of the control flow graph has been annotated with the value of its associated basic block under the desired cost model.

Secret-Dependent Tainting. In programs of interest, some input may be considered secret or private. These secret inputs may impact the results of evaluating certain conditional statements during execution. We refer to these conditional statements as secret-dependent and assume that all nodes of the control flow graph whose basic blocks contain such a conditional statement are marked appropriately.

Control Flow Graph Decomposition. The decomposition of the control flow graph into components provides both scalability and interpretability benefits. On one hand, it allows us to modularize both our construction of cost expressions and the solving of resultant queries. On the other, it provides insight into the minimal amount of difference necessary between two control flow paths to introduce a side-channel. We assume a control flow graph in which appropriate node splitting ensures that no node is both a branch and merge node and any entry to a loop has only one incoming edge. We define our components using dominator and post-dominator relationships [28, 32].

Loop Components. Each back edge of the control flow graph introduces a loop component. For a given back edge, (a, b), its associated loop component is the union of b and the set of nodes that can reach a without going through b.

Branch Components. A branch node b is any node that has more than one outgoing edge. The merge node m of a branch node b is the immediate post-dominator of b. The branch component associated with b is union of m and the set of all nodes that can be reached from b without going through m. If b is nested within a loop component that does not include its merge node, we define its loop-local branch component according to its immediate post-dominator of the loop component. These loop-local branch components are considered as branch components henceforth.

Hierarchical Structure Using previously computed cost expressions in the construction of new ones is possible if our components have a clearly defined nesting structure, meaning that for any two components, exactly one of the following three conditions hold: 1) The components are disjoint. 2) One component is a strict subset of the other i.e. they are nested. 3) The components are identical.

For two loop components, this assumption holds for *reducible* control flow graphs [20]. To gaurantee nesting for branch components, we assume a sufficient additional assumption that all non-branch, non-merge nodes of a branch component have only

$$A(G_n) = (c_n, \top) \quad (1)$$
$$A(G_s) = A(G_1) \oplus A(G_2) \oplus \ldots \quad (2)$$
$$A(G_b) = U_b(G'_{b1}, v) \oplus U_b(G'_{b2}, 1-v) \oplus A(b) \oplus A(m) \oplus (0, v = 0 \vee v = 1) \quad (3)$$
$$A(G_l) = U_l(G'_l, v) \oplus (0, v \geq 0) \quad (4)$$
$$U_b(G, v) = v \otimes A(G) \quad (5)$$
$$U_l(G_n, v) = (v \times c_n, \top) \quad (6)$$
$$U_l(G_s, v) = U_l(G_1, v) \oplus U_l(G_2, v) \oplus \ldots \quad (7)$$
$$U_l(G_b, v) = U_l(G'_{b1}, h) \oplus U_l(G'_{b2}, v-h) \oplus (0, h \leq v) \oplus U_l(b, v) \oplus U_l(m, v) \quad (8)$$
$$U_l(G_l, v) = \oplus_{i=1}^{v} v_i \otimes A(G'_l) \oplus (0, ((v-i) > 0 \wedge v_i = 1) \vee ((v-i) \leq 0 \wedge v_i = 0)) \quad (9)$$

Figure 2: Rules defining the *Annotate* (A), *UpdateBranch* (U_b), and *UpdateLoop* (U_l) functions.

immediate ancestors within that branch component. Semantically, this means that every entry into the body of an if statement must be through its conditional guard.

4 SYMBOLIC COST EXPRESSIONS

We construct a symbolic cost expression and associated constraints for each component G of the control flow graph such that: 1) for each cost c obtained along some path through G there exists a corresponding satisfying assignment of symbolic variables such that the cost expression evaluates to c, 2) for each value c generated by a satisfying assignment of the symbolic variables, there exists a corresponding path through G with cost c.

The function *Annotate*, denoted as $A(G) \rightarrow (E, F)$, takes a graph G as input and returns a symbolic cost expression E along with a constraint F on the symbolic variables in E. The addition of two tuples (E, F) and (E', F') and the multiplication of a tuple (E, F) by a symbolic expression over integer variables v are performed by the $\oplus$ and $\otimes$ operators respectively, and defined:

$$(E, F) \oplus (E', F') = (E + E', F \wedge F')$$

$$v \otimes (E, F) = (v \times E, F)$$

$A(G)$ is defined in terms of two update functions, *UpdateBranch* $U_b(G, v) \rightarrow (E, F)$ and *UpdateLoop* $U_l(G, v) \rightarrow (E, F)$, where v is a symbolic expression over integer variables.

Annotate, *UpdateBranch* and *UpdateLoop* are defined by the rules given in Figure 2. There are four possible types of graphs these functions can take as arguments. If G is a single node, we denote it with G_n and denote its cost as c_n. If G is the sequential composition of other components or single nodes, we call it G_s and write $G_s = G_1, G_2, \ldots$ where each G_i is either a component or a single node. If G is a branch component, we use G_b. For any G_b, G'_{b1} and G'_{b2} denote the two disjoint subgraphs formed when the branch node b and merge node m are removed from G_b. If G is a loop component, we use G_l, and G'_l is the subgraph formed when the back edge of G is removed. G refers to a graph that may be any of the above.

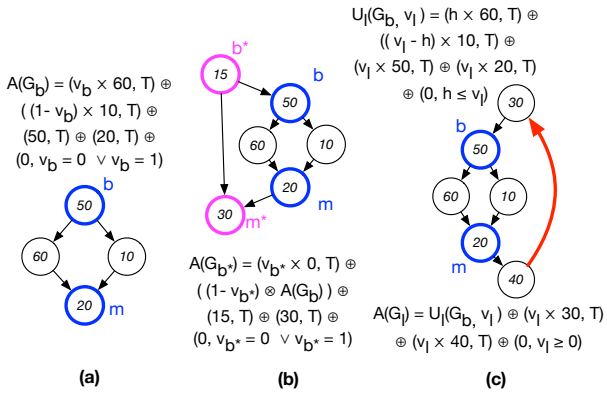

Figure 3: Symbolic cost expressions for (a) a simple branch component, and a branch within (b) a branch and (c) a loop.

Intuitively, E and F are constructed by introducing a symbolic integer variable and its appropriate constraints for each branch and loop, which encodes the branch taken or number of loop iterations, respectively. This can be observed in the $A(G_b)$ and $A(G_l)$ rules (rules (3) and (4)) where a fresh symbolic integer variable v is created and used in the appropriate *Update* function.

Symbolic cost expressions and their associated constraints are constructed hierarchically. Applying A to some component G assumes the results of applying A to all nested components of G. Below, we give examples demonstrating symbolic expressions constructed according to the rules shown in Figure 2.

Figure 3(a) shows the symbolic cost expression E constructed for a simple branch component. The integer variable v_b corresponds to the integer variable created for the branch node in rule (3) in Figure 2. v_b is constrained to 1 or 0 and intuitively denotes whether the if or the else branch is taken at node b, respectively.

Figure 3(b) shows the construction of the symbolic cost expression for a branch component with a nested branch component. The value $A(G_b)$ for the nested component G_b was computed in Figure 3(a) and is reused in the construction of $A(G_{b*})$ by another application of rule (3) in Figure 2.

Figure 3(c) shows the symbolic cost expression construction for a loop component with a nested branch component. $A(G_l)$ is constructed by first applying rule (7) in Figure 2) and then by applications of rules (6) and (8).

Figure 4(a) shows the symbolic cost expression E constructed for a simple loop component. The symbolic integer variable v_l, associated with the loop component (rule (4) in Figure 2), is constrained to be greater than or equal to 0 and denotes the number of times that the back edge of the loop is taken.

Figure 4(b) shows the symbolic cost expression construction for a loop component with a nested loop component. Here $A(G_l)$ is derived from applications of rules (7), (6) and (9).

Claim. *Given any component G with $A(G) = (E, F)$, there is a satisfying assignment to the variables in F such that E evaluates to cost c if and only if there is a path through G with cost c.*

Proof Sketch. We first show that for the cost c of any path through G, there is a satisfying assignment to the variables of F

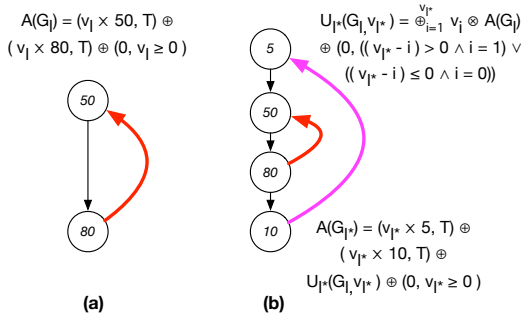

Figure 4: Symbolic cost expressions for (a) a simple loop component, and (b) a loop component nested in a loop.

such that E evaluates to c. We do this through the construction of such an assignment. The cost of any path p through G is uniquely determined by the sequence of edges chosen when more than one outgoing edge could have been taken (i.e. at branch nodes). Following p through G, each time a branch condition is encountered, the value of the appropriate symbolic variable is set as follows.

If the branch condition is the exit of a loop, then only if the exit is taken is the symbolic variable denoting the appropriate exit of the loop (if multiple exits are present) set to 1 and the value of the symbolic variable associated with the back edge of the loop set to the number of traversals of that edge. If the branch condition is nested within a loop, then a running tally is kept of how many times each branch is taken and the variable is set appropriately when the loop is exited. If the branch condition is not nested within a loop, then the update to its symbolic variable is simply the choice made in p. When loops are nested within other loops, the above annotation is performed for each iteration of the outer loop on the symbolic variables corresponding to that iteration. Symbolic variables assigned through this process will both satisfy F and evaluate in E to c.

Next, we show that any cost value obtained by evaluating E on a satisfying assignment of F is the cost of some path G. Let c be the cost generated by the E. By definition, there is some instantiation of the symbolic variables in F resulting in cost c. Let p be the path through component G dictated by that instantiation. In other words, each time a branch condition b is encountered along p, it is chosen according to the instantiation of the symbolic variable corresponding to b. If b is an exit for a loop, then it is taken if and only if the back edge of the loop has been traversed the exact number of times specified by its associated symbolic variable and if b is the exit dictated by the instantiation of the symbolic variables corresponding to the possible loop exits. The cost of the path through G constructed in this manner will be c. □

5 QUERY GENERATION

The construction of symbolic cost expressions allows for pertinent queries to be formulated concerning the set of costs they generate. To formulate these queries, we divide the set of symbolic variables in symbolic cost expressions to two types: secret-dependent and secret-independent. Recall that our control flow graph is annotated with information specifying which branch nodes are secret-dependent. We use this information to mark any symbolic variable associated with a branch component whose branch node is secret-dependent as secret-dependent, and any symbolic variable associated with the back edge of a loop with a secret-dependent exit node as secret-dependent. All other variables are secret-independent. Given this, we generate two types of queries for side-channel analysis.

5.1 Existential Query

The first type of query we generate is aimed at answering the question: Do there exist two paths through the component differing only on conditionals impacted by the secret such that the difference between the two paths is greater than some threshold?

Given a component G, let (E, F) be the symbolic cost expression and the set of constraints generated by $A(G)$. The set of variables in F can be partitioned into two categories, the set of secret-dependent variables $\vec{v_s}$ and the set of secret-independent variables $\vec{v}$. We create two instances of (E, F): (E_1, F_1) and (E_2, F_2). The set of secret-dependent variables in F_1 is $\vec{v}_{s1}$ and that in F_2 is $\vec{v_{s2}}$. The set of secret independent variables $\vec{v}$ is shared across F_1 and F_2. We then formulate and send the following query to an SMT-Solver:

$$\exists \vec{v_{s1}}, \vec{v_{s2}}, \vec{v} \text{ such that } |E_1 - E_2| > \delta \wedge F_1 \wedge F_2 \tag{10}$$

The satisfiability of this query illustrates the existence of two paths through G differing only on secret-dependent conditionals whose costs differ by more than the threshold δ. The satisfying assignments to variables $\vec{v}$, $\vec{v_{s1}}$ and $\vec{v_{s2}}$, denoted as $\vec{a}$, $\vec{a_{s1}}$ and $\vec{a_{s2}}$, provide a specification of such paths through G. This specification can be examined for its feasibility in the program and the process can be iterated to find new assignments if it is suspected to be infeasible.

Since the symbolic cost expressions constructed for multiple nested loops contain a sum over a symbolic variable (see rule (9) in Figure 2), a maximum value for this symbolic variable must be provided to ensure that our expression is within the theories supported by SMT Solvers. This value bounds the maximum number of loop iterations. As such, the unsatisfiability of our query only demonstrates the non-existence of two paths up to the given loop bound. Nevertheless, the analysis can be repeated across multiple bounds to increase certainty in the results.

Note that symbolic cost expressions may contain non-linear arithmetic expressions in the presence of nested loops, making the generated query potentially unsolvable by an SMT-Solver. Additionally, the number of generated variables grows exponentially with the nesting depth, leading to scalability concerns for components with highly nested loops. To address both of these challenges, we use a compositional approach to query generation that simplifies the cost expressions for nested components.

Compositional Query Solving. The symbolic cost expression generation technique we discussed earlier is compositional, and we exploit this to implement a compositional query solving technique that expedites the query solving process. For each component G nested within a secret-dependent component, we calculate the maximum and minimum cost paths through G and, when applicable, two paths through G that differ only on secret-dependent conditionals and for which the cost difference is maximized. We call the assignment yielding the maximum cost value $\vec{a}_{max}$, that yielding the minimum $\vec{a}_{min}$, and that yielding the maximum difference $\vec{a}_{diff}$.

The following claim allows us to determine the satisfiability of the existential query more cheaply using these values.

Claim. *For any component, G, $A(G) = (E, F)$ for which $\exists \vec{v_{s1}}, \vec{v_{s2}}, \vec{v}$ such that $|E_1 - E_2| > \delta \wedge F_1 \wedge F_2$, there exist assignments from $\vec{a}_{\max}$, $\vec{a}_{\min}$, or $\vec{a}_{\text{diff}}$ to the variables corresponding to the nested components of G that also satisfy $|E_1 - E_2| > \delta \wedge F_1 \wedge F_2$.*

Proof. Let $\vec{a}_1$ and $\vec{a}_2$ be the two assignments such that $|E_1 - E_2| > \delta$ and assume, without loss of generality, that $\vec{a}_1$ generates the larger cost value. There are two possibilities: $\vec{a}_1$ and $\vec{a}_2$ either differ in the symbolic variable associated with G, or they do not. First, let us assume that they do. Let $\vec{v_1}$ be the set of variables excluding the variable associated with G, whose assignment contributes to the value of E_1, meaning that a change in their assignment could result in a difference in the value of E_1. Replace the assignment of every variable in $\vec{v_1}$ with its assignment in $\vec{a}_{max}$ and replace the assignment of every variable in the complement of $\vec{v_1}$ with its assignment in $\vec{a}_{min}$. The two resulting assignments differ in cost by at least as much as the original $\vec{a}_1$ and $\vec{a}_2$.

Now, let us assume that $\vec{a}_1$ and $\vec{a}_2$ do not differ in the variable associated with G. Replacing the assignment of every variable associated with a nested component of G by that of $\vec{a}_{diff}$ (ensuring that the more expensive of the two assignments in each component always contributes to the value of E_1) again results in a pair of assignments that differ by at least as much as the original. □

We can similarly show that finding $\vec{a}_{max}$, $\vec{a}_{min}$, and $\vec{a}_{diff}$ for some component G can also be done using the $\vec{a}_{max}$, $\vec{a}_{min}$, and $\vec{a}_{diff}$ assignments of its nested components. The paths built in this manner; however, tend to be the most extreme and thus are potentially more likely to be infeasible in the program.

5.2 Optimization Query

The existential query formulated in Section 5.1 is aimed at addressing the boolean question of the existence of a side-channel observable at some threshold for some number of loop iterations. As choosing a good value for either parameter is challenging a priori, we reformulate the existential query as an optimization problem.

Our first reformulation is the following: What is the minimum number of loop iterations needed for the difference in cost between two paths in the component differing only on secret-dependent conditionals to be greater than some threshold? Let G, E_1, E_2, F_1, F_2, $\vec{v}$, $\vec{v_{s1}}$ and $\vec{v_{s2}}$ be defined as in Eq. 10. We define $\vec{v_k}$ to be the set of all variables associated with a back edge in the union of $\vec{v}$, $\vec{v_{s1}}$ and $\vec{v_{s2}}$. We generate the following minimization query:

$$\textbf{minimize } \vec{v_k} \text{ such that } |E_1 - E_2| > \delta \wedge F_1 \wedge F_2 \qquad (11)$$

This expression can easily be adapted to weigh the minimization of certain loops differently. While a maximum number of loop iterations must still be specified to ensure the appropriate translation to the SMT-Solver in the case of iteratively nested loops, this optimization query relaxes the importance of the loop bound. Additionally, the result of the minimization query can provide some insight into the realizability of the side-channel. Though not always the case, it is likely that a side-channel achievable with relatively few loop iterations is more likely to be realizable than one that requires a large number of loop iterations.

The second reformulation is the following: What is the maximum difference between the costs of two paths through the component differing only on secret-dependent conditionals? Again, let G, E_1, E_2, F_1, F_2, $\vec{v}$, $\vec{v_{s1}}$ and $\vec{v_{s2}}$ be defined as in Eq. 10 and 11. We generate the following maximization query:

$$\textbf{maximize } |E_1 - E_2| \text{ such that } F_1 \wedge F_2 \qquad (12)$$

The result of this query provides a greater understanding of the observability of the side-channel in cases where an analyst is less sure of an appropriate value for the threshold δ.

6 FEASIBILITY ANALYSIS

Assignments returned by SMT solvers to queries generated from symbolic cost expressions may be un-exploitable in the program. To mitigate the impact of false positives, we augment CoCo-Channel with two types of symbolic-execution based feasibility analyses.

6.1 Single Path Feasibility Analysis

We perform a guided symbolic execution along each witness path to check its individual feasibility. Here we constrain our search to the largest branch component containing the vulnerable component. There is exactly one path to the vulnerable component in this subgraph, so this expansion does not increase the size of the analysis. Since the symbolic execution is guided along a single path, the exponential path explosion commonly debilitating to symbolic execution is avoided. However, because we are assuming that the expanded component can be reached by any input, we may incorrectly determine that a path that is feasible in the expanded component but infeasible in the larger program is executable. Nevertheless, this focused feasibility test can still eliminate false positives that are due to infeasible paths.

In the case where CoCo-Channel specifies not just one, but a class of witness paths, we check the feasibility of one example path. If it is infeasible, another path from the class can be selected. While this process itself may be exponential, it can be terminated at any time and an additional constraint may be added to CoCo-Channel's queries to disallow that class of paths from consideration. If CoCo-Channel can find another pair of assignments demonstrating the side-channel, it is possible that proving the feasibility of the side-channel over these witness paths may be simpler.

6.2 Relational Feasibility Analysis

Two paths might be individually feasible but not executable for the same public input. In the function **sanity_safe** (Figure 5), both taking the if branch and executing the while loop 1 time and taking the else branch and executing the while loop 5 times are feasible paths. However, they are not both feasible for the same public input (the value of b). The reason for this lies in the correlation over public input in the number of iterations of the two loops.

Let v_1 and v_2 be the symbolic variables associated with two distinct conditionals and V_1 and V_2 be the set of feasible values for v_1 and v_2 respectively. For two value assignments $a_1 \in V_1$, $a_2 \in V_2$, we use $\mathcal{P}(a_1, a_2)$ to denote there is no public input on which both a_1 and a_2 are executable.

The two conditionals are said to be correlated over public input if $\exists a_1 \in V_1, \exists a_2 \in V_2$ such that $\mathcal{P}(a_1, a_2)$. Given a set of k-paths,

```
public static boolean sanity_safe(int a, int b){
        int i = b; int j = b;
        if (b < 0) {return false;}
        if (a > 0) {while (i>0) {i--;} }
        else {while (j>0) {j--;}}
        return false;
}
```

Figure 5: sanity_safe example from Blazer dataset.

we call the problem of determining the correlated set of conditional statements over public input *k-path relational analysis*. Because CoCo-Channel returns a witness pair of paths, we are interested in particular in the 2-path relational analysis problem. Additionally, we constrain the broad problem of determining any correlation between conditionals to determining only their equivalence or non-equivalence. In other words to determine if $\forall a_1 \in V_1, \forall a_2 \in V_2, a_1 \neq a_2, \mathcal{P}(a_1, a_2)$ or if $\forall a_1 \in V_1, \forall a_2 \in V_2, a_1 = a_2, \mathcal{P}(v_1, v_2)$.

Assume two assignments $\vec{a}_1$ and $\vec{a}_2$. If $\vec{v}^*$ are the secret-dependent conditionals on which the two paths differ, then the only public-dependent conditionals we need consider lie in the components of $v_i \in \vec{v}^*$. Since we are concerned with the cost differential between the paths, we additionally constrain our analysis to only those public-dependent conditionals whose value in $\vec{a}_1$ or $\vec{a}_2$ introduce a meaningful cost differential between the paths.

Let v_1 and v_2 be symbolic variables corresponding to such public-dependent conditionals whose values in $\vec{a}_1$ and $\vec{a}_2$ respectively introduce a cost differential in the component of $v_i \in \vec{v}^*$. Without loss of generality, we assume that $v_i = 1$ in $\vec{a}_1$ and $v_i = 0$ in $\vec{a}_2$. Note that v_1 and v_2 may be either branch or loop variables. We use symbolic execution to determine if there is a correlation between the conditionals associated with v_1 and v_2. First we instrument the bytecode of a program by introducing integer variables for v_i, v_1 and v_2 that increment each time the corresponding conditional is taken, resulting in either a loop counter or a branch indicator. We then duplicate the program under test and execute both versions of the program symbolically using the same symbolic public input and fresh symbolic private input for the second. Letting the suffix denote the version of the program, we then confirm whether any of the following properties can be violated:

$$v_{i1} = 1 \wedge v_{i2} = 0 \wedge v_1 = v_2$$
$$v_{i1} = 1 \wedge v_{i2} = 0 \wedge v_1 \neq v_2$$

If the first one is never violated, then v_1 and v_2 are correlated and equivalent. If the second one is never violated, then v_1 and v_2 are correlated and non-equivalent. In either case, we can add an additional constraint to our query reflecting this information.

7 IMPLEMENTATION AND EXPERIMENTS

We implemented our analysis in our tool CoCo-Channel [2], a Python prototype that performs the decomposition and annotation described in Sections 3 and 4 and interfaces with tools for control flow graph extraction and satisfiability and optimization querying.

Control Flow Graph Extraction. The extraction of the control flow graph was done using Janalyzer [8], an in-development research tool for the static analysis of Java Byte-code developed by Balasubramanian et. al at Vanderbilt.

Taint Analysis. Janalyzer was also used to perform the taint analysis needed by our approach. Static taint analysis is an area of ongoing research [34], and obtaining sound results without over approximation is challenging. We observed that for a sizable class of side-channel vulnerabilities, taint analysis based only on data dependency would suffice to correctly mark secret-dependent branches and would achieve a lower false positive rate than taint analysis based on both data and control dependency. While this introduces a source of unsoundness, we constrain our taint analysis to track only data-dependencies for our experiments.

Satisfiability and Optimization Queries. We integrated CoCo-Channel with the SMT-Solver Z3 [16] through Z3Py, the Z3 API in Python [4]. All queries of the type formulated in Section 5 are sent to Z3 in order to determine either their unsatisfiability or to obtain a satisfying or optimizing assignment.

Feasibility Analysis. We guided the search of the symbolic execution engine Symbolic Path-Finder [6] (SPF) along the pair of witness paths returned by CoCo-Channel to determine their individual feasibility. To determine their relational feasibility, we used javassist [15] to instrument the bytecode with the additional integer variables discussed in Section 6.2. We perform symbolic execution of the instrumented bytecode and find any violations of the properties given in Section 6.2 using SPF.

7.1 Experiments on the STAC Benchmark

We evaluate CoCo-Channel on a benchmark suite of fourteen examples from the DARPA Space/Time Analysis for Cybersecurity (STAC) program [1, 3]. All fourteen questions require determining the vulnerability of large Java client-server or peer-to-peer applications to timing side-channels. Each specifies a secret value, and we evaluate CoCo-Channel's ability to determine if there is a timing side-channel that leaks information about that secret.

Information about these applications is provided in Table 1. The STAC applications are non-trivial applications including web-forums, image-sharing applications, peer-to-peer chat programs, and peer-to-peer applications for hosting auctions. Multiple versions of the same application, often differing in their vulnerability to side-channels, are provided in the STAC benchmark. The number following the application name refers to the version.

For each question, we manually marked the input or program variable corresponding to the specified secret and an entry point into the application. We then used Janalyzer to perform taint analysis from our the variable and extract the control flow graph from the entry point. Function calls were handled through inlining. Presently, CoCo-Channel does not handle recursion, but instead unrolls the recursive calls through iterative inlining until some specified depth is reached. For our experiments, we used a depth of 1. Our cost model was the number of bytecode instructions in a basic block. Since not all bytecode instructions take the same amount of time, we augmented our cost model through a manually compiled list of bytecode operations known to be more expensive. Our list consisted of write-to-file operations, the sleep instruction, and the expensive arithmetic operations (multiply, mod, div, exp, pow, sqrt) and their BigInteger counterparts. The cost of these operations was increased according to the values provided in [22] except for sleep which was increased by its argument.

Table 1: Application Statistics

Application	#Classes	#Methods	#Bytes_of_instruction
gabfeed_1	111	332	12180
gabfeed_2	69	283	8862
gabfeed_3	112	332	12143
snapbuddy_2	338	2224	81636
snapbuddy_3	1083	3823	120351
powerbroker_1	272	3158	61288
powerbroker_4	261	3146	59948
withmi_4	218	2527	46011
withmi_5	213	2526	45996
collab	26	121	7236
lawdb	43	173	9987

CoCo-Channel was run on the resultant annotated control flow graph for a threshold of 1000 and a maximum loop depth of 20. All experiments were run on a single computer with an Intel Core i5-6600K CPU @ 3.50GHz and 32GB of RAM.

Results. The results for the STAC benchmark are given in Table 2. The **Name** column identifies the example under test. **Size** is the number of basic blocks in our extracted control flow graph. **Vulnerable** is DARPA's official verdict and **Correct** denotes if our results agree with DARPA's. **UC-CoCo Time** is the time taken by CoCo-Channel to verify the application without compositional solving and **CoCo Time** is the time taken when compositional solving is used. It is extremely clear from the timing results on the larger STAC programs that compositional solving is key to scaling our technique to realistically sized applications. In five out of the fourteen cases, the naive solving approach times out after thirty minutes while the compositional solving approach is able to find a pair of witness paths in less than 3 seconds. CoCo-Channel's analysis agrees with DARPA's official verdict in all but two cases. Importantly, no vulnerable case is incorrectly classified as non-vulnerable, and CoCo-Channel specifies the correct component of the control flow graph in each correctly identified vulnerable case.

Discrepancies. DARPA considers **gb2_mdp** non-vulnerable, but CoCo-Channel's detection of a potential side-channel caused us to further examine the problem. The interesting method is a modular exponentiation function using the Montgomery Powering Ladder [24] to perform exponentiation. While a single iteration of the loop in this method is in fact non-vulnerable, a difference in a given bit of the secret exponent value does impact how certain variables are updated, which can lead to subsequent loop iterations taking a differing amount of time. A differential side-channel attack may be able to exploit this and the original presentation of the Montgomery Powering Ladder [24] acknowledges its potential vulnerability to such side-channel attacks. Thus, we are not convinced that there is no side-channel in **gb2_mdp**, but perhaps is not strong enough to recover the entire key or not exploitable within the application.

The other discrepancy concerns **withmi5_connect**. Manual inspection of the code reveals that the the branch conditional marked vulnerable by CoCo-Channel is both secret-dependent and that the differences in work between the two resultant branches is significant (including a file write). However, we determined that whether or not the more expensive path is taken is not visible based on the timing of network traffic because there are no network packets sent after the choice on conditionals is made, making the side-channel unexploitable. Such a distinction is out of the scope of our present analysis but motivates a potential extension of CoCo-Channel to include markers denoting where side-channel information is observable (such as when packets are sent). With this extension, our result on the problem would agree with DARPA's official verdict.

Table 2: Results on STAC Benchmark

Name	Size	Vulnerable	Correct	Time (s)	
				UC-CoCo	CoCo
collab	12	Safe	✓	0.026	0.052
gb1_mdp	90	Unsafe	✓	-	0.680
gb1_pwd	12	Safe	✓	0.042	0.090
gb2_mdp	86	Safe	x	-	0.857
gb3_search	286	Unsafe	✓	-	0.237
lawdb	736	Unsafe	✓	-	2.200
pb1_gen	83	Unsafe	✓	0.597	0.292
pb4_exp	94	Unsafe	✓	0.063	0.105
pb4_gen	18	Safe	✓	0.087	0.097
sb2_mdp	71	Unsafe	✓	-	0.730
sb3_pwd	13	Safe	✓	0.030	0.078
wm4_connect	21	Unsafe	✓	1.220	1.920
wm4_exp	901	Safe	✓	0.002	0.002
wm5_connect	916	Safe	x	35.200	2.100

Table 3: Optimization Results on STAC Benchmark

Name	#loop_iteration	Maximum Threshold	Time (s)
collab	-	19	0.031
gb1_pwd	37	506	0.061
pb1_gen	2	11853	1.280
pb4_exp	10	2052	0.087
pb4_gen	334	57	0.115
sb3_pwd	34	559	0.041
wm4_exp	-	0	0.002
wm4_connect	4	3688	1.240

Feasibility Checking. We present our results on the STAC dataset without the feasbility checking discussed in Section 6. Performing symbolic execution on the STAC applications would require symbolically manipulating complex non-primitives, handling randomness, and symbolically executing non-trivial networking code. SPF currently does not provide such functionality, and manually constructing drivers by stubbing out method calls and simplifying non-primitive objects is beyond the scope of this paper. Instead, we use this benchmark to test CoCo-Channel's performance without its refinement stage. Importantly, neither of the false positives is a result of infeasible witness paths.

Optimization Queries. We provide the results of our optimization queries in Table 3. For unsafe variants, we report the results of the optimization queries on the smallest vulnerable component. For safe variants, we report the results on the largest component containing a secret-dependent component. These queries scaled in eight of our examples. In the other cases, Z3 either ran out of memory or was terminated after an hour. In two benchmarks (**withmi4_exp** and **collab**), no loop was involved in a secret-dependent component, making the loop optimization query trivial. In the non-vulnerable cases to which the analysis scaled, it confirmed our suspicions of non-vulnerability by specifying the relatively large number of loop iterations necessary to introduce a side-channel or the relatively low maximum cost difference with our given loop bound.

7.2 Comparison to the State of the Art

We also evaluate CoCo-Channel against two recent tools for detecting side-channel vulnerabilites in Java applications, Blazer [7] and Themis [13]. Neither are publically available, so we were unable to test them on our STAC dataset. Instead, we compare CoCo-Channel on the benchmark introduced by the Blazer authors [7] and later used to evaluate Themis [13] as well as the additional benchmarks collected by the Themis authors [13].

7.2.1 Blazer Dataset. The Blazer dataset consists of 22 benchmarks drawn from a combination of challenge programs from the DARPA STAC program, classic examples from the literature [19, 25, 30], and microbenchmarks constructed by the Blazer authors. For our analysis, we used a loop bound of 10 and a threshold of 50. We summarize the results of our experiments in Table 4. CoCo-Channel was able to correctly verify every program in the benchmark. Each tool was run on different hardware, making the timing reported incomparable, but CoCo-Channel's running time without compositional solving appears to be similar to Themis and, with compositional solving, CoCo-Channel is the fastest.

Importance of Feasibility Checking. Without both kinds of feasibility analyses, CoCo-Channel would have incorrectly marked a safe variant as vulnerable. Our running time includes these analyses. The single path feasibility analysis is performed any time a pair of witness paths of paths are returned by CoCo-Channel while the relational feasibility analysis is only performed when the cost differential is due to choices on secret-independent branches inside a secret-dependent component across both witness paths.

Optimization Queries. Table 5 shows the results of our optimization queries, which scaled to all programs except the unsafe **ModPow** variants. Since our choices of threshold and loop bound may be somewhat arbitrary, these provide insight into the degrees of vulnerability in the programs beyond a binary classification. For example, the results on **array_safe** reveal that the program is very resilient to timing side-channels even for an adversary with refined observational abilities. Our threshold maximization results contradict the results reported in [13], where the authors claim that Themis is able to correctly determine the safety of the variants for a threshold of 0. We contacted the authors about this discrepancy, and they concede that it is likely due to a bug in their tool. While the value reported by the maximization query depends on the cost model, there is cost difference present in many of the safe variants.

Assumptions. In some of the safe variants (**k96**, **gpt14**, **modPow1**, **modPow2**), there is a conditional related to the length of the secret that results in information leakage, leading CoCo-Channel to initially determine that the variant was vulnerable. We consulted the authors of Themis about the discrepancy; they agreed and mentioned that since side-channels that only leak length are not considered strong enough, they manually annotated certain unsafe conditionals to tell Themis to ignore them. After doing the same, we re-ran the experiments and obtained the results reported.

7.2.2 Themis Dataset. The authors of Themis also evaluated their tool on a benchmark of real world Java programs with known vulnerabilities [13]. We compare our results on those containing potential timing side-channels. We set our threshold to 64 to match [13] and our loop bound to 10. Results are given in Tables 6 and 7.

Table 4: Results on Blazer Benchmark

Name	Version	Size	Time (s) Blazer	Themis	UC-CoCo	CoCo	Fe.	Rel.
MicroBench								
array	Safe	16	1.60	0.28	0.446	0.450	x	✓
	Unsafe	14	0.16	0.23	0.426	0.434	✓	x
loopAndbranch	Safe	15	0.23	0.33	0.968	0.982	✓	✓
	Unsafe	15	0.65	0.16	1.228	1.238	✓	✓
nosecret	Safe	7	0.35	0.20	0.001	0.001	x	x
	Unsafe	9	0.28	0.12	0.435	0.447	✓	x
sanity	Safe	10	0.63	0.41	0.624	0.623	x	✓
	Unsafe	9	0.30	0.17	0.439	0.450	✓	x
straightline	Safe	7	0.21	0.49	0.024	0.034	x	x
	Unsafe	7	22.20	5.30	0.413	0.423	✓	x
STAC								
modPow1	Safe	18	1.47	0.61	0.027	0.038	x	x
	Unsafe	58	218.54	14.16	18.016	0.597	✓	x
modPow2	Safe	20	1.62	0.75	0.028	0.038	x	x
	Unsafe	106	7813.68	141.36	181.412	0.675	✓	x
passwordEq	Safe	16	2.70	1.10	0.038	0.063	x	x
	Unsafe	15	1.30	0.39	0.494	0.519	✓	x
Literature								
k96	Safe	17	0.70	0.61	0.290	0.038	x	x
	Unsafe	15	1.29	0.54	0.411	0.424	✓	x
gpt14	Safe	15	1.43	0.46	0.028	0.030	x	x
	Unsafe	26	219.30	1.25	0.535	0.605	✓	x
login	Safe	16	1.77	0.54	0.07	0.089	x	x
	Unsafe	11	1.79	0.70	0.414	0.428	✓	x

Table 5: Optimization Results on Blazer Benchmark

Name	Version	#loop_iteration	Maximum Threshold	Time (s)
MicroBench				
array	safe	-	1	0.041
	unsafe	5	112	0.034
loopAndbranch	safe	21	28	0.116
	unsafe	5	98	0.054
nosecret	safe	-	0	0.001
	unsafe	6	88	0.020
sanity	safe	-	0	0.040
	unsafe	6	81	0.035
straightline	safe	-	5	0.030
	unsafe	-	1343	0.030
STAC				
modpow1	safe	51	9	0.035
modPow2	safe	51	9	0.035
passwordEq	safe	51	9	0.052
	unsafe	4	144	0.036
Literature				
k96	safe	17	27	0.037
	unsafe	-	74	0.020
gpt14	safe	26	18	0.037
	unsafe	4	175	0.069
login	safe	13	36	0.090
	unsafe	4	168	0.022

CFG extraction. Janalyzer requires a jar file containing all relevant classes for control flow graph extraction. In many cases, dependencies were missing from the manually modified source code provided by the Themis authors. In these cases, we wrote a driver for the provided source code and stubbed out any method calls we were unable to find the dependencies for. For one example, **pac4j**, stubbing out methods was a prohibitive process so we removed this example from our evaluation. In the other cases, we believe the integrity of the benchmark to remain intact but note that the subgraph of the control flow graph we analyze may differ from that in [13].

Table 6: Results on Themis Benchmark

Name	Version	Correct	Time (s)			Fe.	Rel.
			Themis	UC-CoCo	CoCo		
Spring-Security	Safe	✓	1.70	0.034	0.068	x	x
	Unsafe	✓	1.09	0.831	0.851	✓	x
JDK7-MsgDigest	Safe	✓	1.27	0.015	0.028	x	x
	Unsafe	✓	1.33	0.459	0.473	✓	x
Picketbox	Safe	✓	1.79	0.015	0.044	x	x
	Unsafe	✓	1.55	2.889	0.589	✓	x
Tomcat	Safe	x	9.93	1.280	1.010	x	✓
	Unsafe	✓	8.64	1.280	0.950	x	✓
Jetty	Safe	✓	2.50	0.038	0.201	x	x
	Unsafe	✓	2.07	0.480	1.531	✓	x
orientdb	Safe	✓	37.99	0.178	0.406	x	x
	Unsafe	✓	38.09	2.982	0.842	✓	x
boot-auth	Safe	✓	9.12	0.018	0.060	x	x
	Unsafe	✓	8.31	0.443	0.461	✓	x

Table 7: Optimization Results on Themis Benchmark

Name	Version	#loop_iteration	Maximum Threshold	Time (s)
Spring-Security	safe	-	0	0.041
	unsafe	3	179	0.061
JDK7-MsgDigest	safe	-	0	0.018
	unsafe	5	136	0.023
Picketbox	safe	-	1	0.018
	unsafe	4	208	2.980
Tomcat	safe	1	275	0.610
	unsafe	1	350	53.600
Jetty	safe	-	1	0.044
	unsafe	4	233	0.050
orientdb	safe	-	1	0.198
	unsafe	4	208	2.950
boot-auth	safe	-	36	0.020
	unsafe	5	144	0.026

Discrepancies. CoCo-Channel incorrectly determines that the safe variant of **tomcat** is vulnerable. In the Themis dataset, the authors manually patch the vulnerable **tomcat** variant by introducing an expensive method call into the shorter branch to mitigate the vulnerability. The effects of this patch can be seen in the results of our threshold maximization query across the two versions. Nevertheless, CoCo-Channel reports that it is not robust enough to fully hide the side-channel. This could be due to a difference in our cost model, our method stubs not providing the same time complexity as the original code, or a different assumption on the part of the Themis authors on what paths are feasible. We also did not perform our single path feasibility analysis on **tomcat** for the same reasons as for the STAC benchmark. Nevertheless, we confirmed that this false positive was not due to the infeasibility of the returned paths.

Assumptions. As in the Blazer examples, many of the safe variants actually do contain timing side-channels that leak the length of the secret or if the secret is null. Following the example of the Themis authors, we manually mark secret-dependent conditionals that depend only on the length of the secret, whether the secret is null, or if it is valid input as "safe". Exceptions are made for a length check in **spring-security** and for a null check in **tomcat** in which the intended vulnerabilities are side-channels of such kind.

8 RELATED WORK

Side-Channel Detection. Work on side-channel detection constitutes a large field of research. Our work extends that of Brennan et al [10] with feasibility analyses, compositional solving, and annotation rules. Our most immediate related work is that of Antopolous et al. [7] and of Chen et al. [13]. Their tools Blazer and Themis are both motivated by the same understanding that secret-dependent differences in control flow can introduce side-channel behavior. Our technical contributions; however, differ. Blazer uses a grammar-based approach to representing program trails while Themis introduces a simple imperative language with cost-instrumented operational semantics and verifies programs in this language using a relational program logic. Our compositional annotation and solving ability provide our approach with additional scalability. We use symbolic execution to check the feasibility of our results while both Blazer and Themis use abstract interpretation.

Another approach is taken by Pasareanu et al. [30], Bang et al. [9], and Phan et al. [31]. Their work uses a combination of symbolic execution and model counting to determine the probability of execution paths and tools from information theory to quantify the leakage based on those probabilities. Due to its reliance on full symbolic execution, the scalability of this approach suffers. Our approach can be considered complementary. Using CoCo-Channel, we can reduce a complicated program to a potentially much smaller set of suspicious components on which their analysis might scale.

Zhang et al. [36] detail a static approach to detecting side-channels in web applications which, like ours, involves determining the cost difference between paths resultant from secret-dependent conditionals. Our use of symbolic variables in cost expressions allow for a significantly more expressive and refined analysis than theirs which compares the remote procedure calls in each branch. Other static approaches to side-channel detection include type-based approaches [5, 21] or are particular to specific types of side-channels, such as CacheAudit [17]. Dynamic approaches to side-channel detection [12, 29] is also an area of active research.

Path-Based Worst-Case Analysis. Our work is similar to research to determine the worst-case execution time of a program [35]. Li and Malik [26] developed an approach which, like ours, implicitly considers all program paths. They determine a cost per basic block, introduce variables denoting the number of traversals of each basic block augmented by structural constraints, and then maximize the sum of the product of the costs of each basic block and its number of iterations. Puschner and Schedel [33] exploit the visual nature of the control flow graph to develop a set of constraints describing possible paths through it. Though both works share the goal of implicit path enumeration, our choice of formulation differs and is dictated by our need to detect not the maximum path through a program but rather the maximal difference between secret-sensitive paths.

9 CONCLUSIONS

CoCo-Channel provides a scalable static analysis for detecting side-channel vulnerabilities. We have demonstrated its ability to scale to large and highly-realistic applications and obtain meaningful results on those systems. CoCo-Channel provides compositional generation of symbolic path cost expressions, compositional query generation, optimization query formulation and evaluation, and symbolic execution based path feasibility analysis to eliminate false positives.

REFERENCES

[1] [n. d.]. https://github.com/Apogee-Research/STAC/tree/master/Engagement_Challenges. ([n. d.]).
[2] [n. d.]. https://github.com/vlab-cs-ucsb/coco-channel. ([n. d.]).
[3] [n. d.]. Space/Time Analysis for Cybersecurity (STAC). https://www.darpa.mil/program/space-time-analysis-for-cybersecurity. ([n. d.]).
[4] [n. d.]. Z3 API in Python. ([n. d.]). http://www.cs.tau.ac.il/~msagiv/courses/asv/z3py/guide-examples.htm
[5] Johan Agat. 2000. Transforming out timing leaks. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 40–53.
[6] Saswat Anand, Corina Păsăreanu, and Willem Visser. 2007. JPF–SE: A symbolic execution extension to java pathfinder. *Tools and Algorithms for the Construction and Analysis of Systems* (2007), 134–138.
[7] Timos Antopoulos, Paul Gazzillo, Michael Hicks, Eric Koskinen, Tachio Terauchi, and Shiyi Wei. 2017. Decomposition instead of self-composition for k-safety. (2017).
[8] Daniel Balasubramanian, Zhenkai Zhang, Dan McDermet, and Gabor Karsai. 2017. Janalyzer: A Static Analysis Tool for Java Bytecode. (08/2017 2017).
[9] 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*. ACM, 193–204.
[10] Tegan Brennan, Seemanta Saha, and Tevfik Bultan. 2018. Symbolic Path Cost Analysis for Side-Channel Detection. In *Software Engineering Companion (ICSE-C), 2018 IEEE/ACM 40th International Conference on*. IEEE.
[11] David Brumley and Dan Boneh. 2005. Remote timing attacks are practical. *Computer Networks* 48, 5 (2005), 701–716.
[12] Peter Chapman and David Evans. 2011. Automated black-box detection of side-channel vulnerabilities in web applications. In *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 263–274.
[13] Jia Chen, Yu Feng, and Isil Dillig. 2017. Precise Detection of Side-Channel Vulnerabilities using Quantitative Cartesian Hoare Logic. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 875–890.
[14] Shuo Chen, Rui Wang, XiaoFeng Wang, and Kehuan Zhang. 2010. Side-channel leaks in web applications: A reality today, a challenge tomorrow. In *Security and Privacy (SP), 2010 IEEE Symposium on*. IEEE, 191–206.
[15] Shigeru Chiba. 1998. JavassistâĂŤa reflection-based programming wizard for Java. In *Proceedings of OOPSLAâĂŹ98 Workshop on Reflective Programming in C++ and Java*, Vol. 174.
[16] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.
[17] Goran Doychev, Boris Köpf, Laurent Mauborgne, and Jan Reineke. 2015. Cacheaudit: A tool for the static analysis of cache side channels. *ACM Transactions on Information and System Security (TISSEC)* 18, 1 (2015), 4.
[18] Jeffrey Friedman. 1972. Tempest: A signal problem. *NSA Cryptologic Spectrum* 35 (1972), 76.
[19] Daniel Genkin, Itamar Pipman, and Eran Tromer. 2015. Get your hands off my laptop: Physical side-channel key-extraction attacks on PCs. *Journal of Cryptographic Engineering* 5, 2 (2015), 95–112.
[20] Matthew S Hecht and Jeffrey D Ullman. 1972. Flow graph reducibility. In *Proceedings of the fourth annual ACM symposium on Theory of computing*. ACM, 238–250.
[21] Daniel Hedin and David Sands. 2005. Timing aware information flow security for a javacard-like bytecode. *Electronic Notes in Theoretical Computer Science* 141, 1 (2005), 163–182.
[22] Vincent Hindriksen. [n. d.]. How expensive is an operation on a CPU. ([n. d.]). https://streamhpc.com/blog/2012-07-16/how-expensive-is-an-operation-on-a-cpu/
[23] Ralf Hund, Carsten Willems, and Thorsten Holz. 2013. Practical timing side channel attacks against kernel space ASLR. In *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 191–205.
[24] Marc Joye and Sung-Ming Yen. 2002. The Montgomery powering ladder. In *CHES*, Vol. 2. Springer, 291–302.
[25] Paul C Kocher. 1996. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Annual International Cryptology Conference*. Springer, 104–113.
[26] Yau-Tsun Steven Li and Sharad Malik. 1995. Performance analysis of embedded software using implicit path enumeration. In *ACM SIGPLAN Notices*, Vol. 30. ACM, 88–98.
[27] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. 2015. Last-level cache side-channel attacks are practical. In *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 605–622.
[28] Edward S Lowry and Cleburne W Medlock. 1969. Object code optimization. *Commun. ACM* 12, 1 (1969), 13–22.
[29] Luke Mather and Elisabeth Oswald. 2012. Quantifying Side-Channel Information Leakage from Web Applications. *IACR Cryptology ePrint Archive* 2012 (2012), 269.
[30] Corina S Pasareanu, Quoc-Sang Phan, and Pasquale Malacaria. 2016. Multi-run side-channel analysis using Symbolic Execution and Max-SMT. In *Computer Security Foundations Symposium (CSF), 2016 IEEE 29th*. IEEE, 387–400.
[31] Quoc-Sang Phan, Lucas Bang, Corina S Pasareanu, Pasquale Malacaria, and Tevfik Bultan. 2017. Synthesis of Adaptive Side-Channel Attacks. *IACR Cryptology ePrint Archive* 2017 (2017), 401.
[32] Reese T Prosser. 1959. Applications of boolean matrices to the analysis of flow diagrams. In *Papers presented at the December 1-3, 1959, eastern joint IRE-AIEE-ACM computer conference*. ACM, 133–138.
[33] Peter P Puschner and Anton V Schedl. 1997. Computing maximum task execution timesâĂŤa graph-based approach. *Real-Time Systems* 13, 1 (1997), 67–91.
[34] Omer Tripp, Marco Pistoia, Stephen J Fink, Manu Sridharan, and Omri Weisman. 2009. TAJ: effective taint analysis of web applications. In *ACM Sigplan Notices*, Vol. 44. ACM, 87–97.
[35] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, et al. 2008. The worst-case execution-time problemâĂŤoverview of methods and survey of tools. 7, 3 (2008), 36.
[36] 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 *Proceedings of the 17th ACM conference on Computer and communications security*. ACM, 595–606.