The Dissertation Committee for Oswaldo Luis Olivo
certifies that this is the approved version of the following dissertation:

Automatic Static Analysis of Software Performance

Committee:

Calvin Lin, Supervisor

Işil Dillig, Co-Supervisor

Thomas Dillig

Shuvendu Lahiri

Vitaly Shmatikov

Automatic Static Analysis of Software Performance

by

Oswaldo Luis Olivo, B.S., M.S.C.S.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2016

Dedicated to my parents.

Acknowledgments

I would like to thank my advisors Calvin Lin and Işil Dillig for their technical contributions, professional advice, and overall support during graduate school.

Thomas Dillig, Shuvendu Lahiri and Vitaly Shmatikov for taking the time to serve on my committee and providing useful comments about my work.

My undergraduate advisor from *Universidad Simón Bolívar*, Prof. Ascánder Suárez, for introducing me to the world of academic research.

My technical friends for helpful and enjoyable discussions that contributed to making my research projects a reality. It was a pleasure to collaborate with Roopsha Samanta, Sarfraz Khurshid, Shiyu Dong, and Lingming Zhang.

My many friends outside of grad school for helping me avoid burnout.

My parents, Luis and Margarita, and my sister María Elisa, for their understanding of the demands of grad school and their unconditional support.

Automatic Static Analysis of Software Performance

Oswaldo Luis Olivo, Ph.D.
The University of Texas at Austin, 2016

Supervisors: Calvin Lin
Işil Dillig

Performance is a critical component of software quality. Software performance can have drastic repercussions on an application, frustrating its users , breaking the functionality of its components, or even rendering it defenseless against hackers. Unfortunately, unlike in the program verification domain, robust analysis techniques for software performance are almost non-existent.

In this thesis we formalize important classes of performance-related bugs and security vulnerabilities, and implement novel static analysis techniques for automatically detecting them in widely used open-source applications. Our tools are able to uncover 92 performance bugs and 47 security vulnerabilities, while analyzing hundreds of thousands of lines of code and reporting a modest amount of false positives.

Our work opens a new avenue for research: the development of rigorous automatic analyses for effective software performance understanding, inspired by traditional research in functional verification.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Society is becoming increasingly dependent on software. We rely on computer programs for entertainment purposes such as browsing the Web, playing games or watching videos, as well as more critical tasks such as medical surgery, industrial automation, and banking. Emerging technologies such as mobile devices and the Internet-of-Things are a strong indicator that software will continue to form an integral part of our daily lives. Thus, ensuring the quality of software is a critical goal in modern times.

Functionality bugs are clearly important, so techniques for testing and verification of software functionality have been studied and developed for decades in both academia and industry. However, software quality is comprised of other aspects beyond functionality such as performance, security, maintainability, usability, and these components can have an influence on each other. In particular, problematic performance can have subtle effects on different dimensions of software quality. Inefficient performance can hang the application and break its functionality, or at least frustrate its users and affect its usability. Performance that is controllable by an attacker, even with the absence of implementation inefficiencies, can constitute a denial-of-service vulnerability. A performance difference while servicing a request depending on private information, say a short execution path and a long execution path, can leak private information to a hacker. These are just a few examples of the drastic consequences of problematic performance, and its different manifestations beyond slow program execution.

There is a need for automated techniques to achieve effective performance understanding, recognizing the different manifestations of problematic performance and their effect on the other components of software quality. This thesis provides the necessary groundwork to make progress towards that goal, by defining classes of important performance-related bugs and vulnerabilities and describing practical detection techniques.

1.1 Challenges

A key challenge in static performance analysis is defining the performance properties to verify. While functional static analyses typically rely on checking for undesirable values in a program's state (null pointer dereference, array access out of bounds, among others), it is harder to specify undesirable performance in a similar manner. In general it's not apparent whether a program's performance can be improved, since it depends on user inputs, the execution environment, and what constitutes unnaceptable performance for one feature of the application might be optimal for another one. Another complication is that problematic performance has very different manifestations: a denial-of-service vulnerability might be caused due to performance controlled by an attacker, while a privacy leak might result from a performance difference in terms of private information. In summary, problematic software performance goes beyond the traditional notion of *slow execution*.

Even with the fundamental definitions of performance bugs and vulnerabilities in place, there remains the technical challenge of implementing precise and scalable static analyses for their detection. Given that static analyses lack information that is only available at runtime, they tend to overapproximate the behavior of the program, which can lead to false positives (reported warnings that do not represent real problems). It is imperative that static analyses minimize the number of false positives, preserve automation as much as possible (by not relying excessively on user annotations) and find important issues in realistic applications. Existing techniques for performance analysis are mostly dynamic. They rely on program testing and profiling, and require subsequent human judgment to detect and eliminate performance problems. But finding the right inputs that trigger performance issues is extremely laborious, and these approaches rely excessively on human judgment to detect problematic performance. Static approaches involving worst-case execution time have been shown to suffer from scalability or precision issues for realistic programs, while still relying on human judgment in similar spirit to dynamic approaches. Additionally, there's been substantial effort spanning the fields of algorithms, computer architecture, and compilers to improve performance in terms of program running time. But they cover a limited aspect of performance —the need to make programs run faster— and are unable to assess the more subtle dimensions of performance and its influence on other factors of software quality. For example, performance bugs involve the notion of features of the program being replaced by a more efficient implementation; denial-of-service vulnerabilities on whether an attacker can abuse the performance of an application and potentially fully control the performance of the application; side channel attacks leverage the existence of a fast path and a slow path, depending on private data, which can leak sensitive information to an attacker. These kind of problems go beyond the symptoms of *slow performance*, have effects on other aspects of software quality including functionality and security, and cannot be fixed by simply making local, low-level changes. Instead they might involve code refactoring across different classes and procedures, imposing limitations of what a feature can process, or ultimately warrant the removal of features from the application. These scenarios lie outside of the scope of state-of-the compiler optimizations and software evolution techniques.

The main idea underlying our techniques is to perform an approximate asymptotic complexity analysis on the program, leveraging existing static analysis techniques such as weakest precondition computation and taint analysis. We use these techniques to detect potential redundant computations, opportunities for an attacker to control the application's performance and launch a denial-of-service attack, or a performance imbalance involving secret information that can result in privacy leaks. In particular, we rely on taint analysis to keep track of user input data and secret application data throughout the program and infer the effects of tainted data in the program's performance. Our analyses are able to scale due to the effective use of existing lightweight static analysis techniques in performing a limited form of asymptotic complexity analysis.

1.2 Contributions

In this thesis we present novel techniques for systematically analyzing software performance. We formalize important classes of performance-related bugs and security vulnerabilities in a manner that is amenable to detection. We have used our tools to automatically uncover 92 performance bugs and 47 security vulnerabilities in open source Java packages from Google and the Apache Foundations, among others, along with e-commerce, bidding and medical web applications in PHP. The performance bugs have the potential to severely affect the functionality and usability of the applications due to decreased response times of their components, while 37 of the security vulnerabilities can be exploited to cause denial-of-service attacks against web servers, and 10 vulnerabilities represented the leak of sensitive data such as purchase histories, bidding histories and medical information about patients. Apart from the detection algorithms, we have implemented exploit generation tools to facilitate a more detailed assessment of the applications.

The rest of the thesis is organized as follows:

- Chapter 2 describes asymptotic redundant collection traversals, a class of performance bugs.

- Chapter 3 presents a static analysis for detecting the performance bugs from chapter 2, and experimental results.
- Chapter 4 cover second-order denial-of-service vulnerabilities, a class of security vulnerabilities that can be leveraged against application availability.
- Chapter 5 presents a static analysis for detecting the vulnerabilities from chapter 4, and experimental results.
- Chapter 6 describes asymptotic resource-usage side-channel vulnerabilities, a class of security vulnerabilities that can be leveraged to against application privacy.
- Chapter 7 presents a static analysis for detecting the vulnerabilities from chapter 6, and experimental results.
- Chapter 8 discusses related work.
- Chapter 9 concludes.

Chapter 2

Defining Performance Bugs[1]

2.1 Performance Bugs

A *functionality bug* occurs when a piece of software crashes or produces an incorrect result. Fortunately, research in program analysis has produced significant advances in the automated detection of such bugs [14, 93, 6, 27, 28]. By contrast, a *performance bug* arises when a program produces the correct result but a simple functionality-preserving change can provide a substantial performance improvement [104]. Performance bugs are significant because they can render a program unusable; they can also be exploited by malicious users to create denial-of-service attacks. Unfortunately, performance bugs are more difficult to detect than functionality bugs for several reasons:

- First, it is difficult to know whether a program's performance can be expected to improve, since it depends on user inputs, on the many details of the program's execution environment, and on some notion of how a "good" solution should perform.

[1]O. Olivo, I. Dillig, C. Lin. *Static Detection of Asymptotic Performance Bugs in Collection Traversals.* 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15).

- Second, while functionality bugs can be tested using assertions or various automated testing tools [73, 157, 28], the detection of performance bugs typically requires a human to monitor the program and make a judgment call on its performance.

- Third, performance bugs often manifest themselves only with large inputs, so the *small input hypothesis* [127], which forms the basis of most software testing methodologies, does not hold.

For these reasons, performance bugs remain a nebulous and evasive problem, and most existing tools for detecting performance problems either rely on rule-based pattern matching of syntactic program constructs or on some degree of runtime analysis and human intervention.

This thesis presents a new static analysis—and its implementation in a tool called Clarity—for automatically detecting an important class of asymptotic performance bugs. We say that a code snippet has an *asymptotic performance bug* if its computational complexity is $O(f(n))$ but the same functionality can be implemented by code with complexity $O(g(n))$ such that $g(n)$ is $O(f(n))$ but $f(n)$ is not $O(g(n))$. Although the detection of arbitrary asymptotic performance bugs is beyond the scope of program analysis[2], we have identified a restricted but prevalent class of asymptotic performance bugs that we call *redundant traversal bugs*. A redundant traversal bug arises

[2]Observe that identifying arbitrary asymptotic performance bugs requires knowing a "best" algorithm for implementing a given functionality.

```
public boolean render(Graphics2D g2, Rectangle2D
                      dataArea, int index, ...) {
  ...
  XYDataset dataset = getDataset(index);
  XYItemRenderer renderer = getRenderer(index);
  ...
  int sCount = dataset.getSeriesCount();
  int series;
  for (series=sCount-1; series >= 0; series--) {
   int first = 0;
   int last = dataset.getItemCount(series) - 1;
   ...
   for (item=first; item <= last; item++) {
    renderer.drawItem(dataset, series, item,...);
   }
   ...
  }
  ...
}
public void drawItem(XYDataSet dataset,
                  int series, int item, ...) {
  ...
  OHLCDataset highLowData = (OHLCDataset)dataset;
  itemCount = highLowData.getItemCount(series);
  double xxWidth = dataArea.getWidth();
  for(int i=0; i< itemCount; i++) {
   ...
   if(last != -1) {
    xxWidth=Math.min(xxWidth,Math.abs(pos-last));
    }
  }
}
```

Figure 2.1: A previously unknown performance bug in the JFreeChart application that was identified by Clarity.

if a program fragment repeatedly iterates over a data structure, such as an array or list, that has not been modified between successive traversals of the data structure. Since such computation can be memoized and re-used across loop iterations, redundant traversal bugs typically result in at least an $O(n)$ performance degradation, where n is the size of the data structure. Furthermore, such performance bugs are typically easy to fix and often only require the addition of a parameter to a method, the addition of a field to an object, or the use of a slightly different data structure.

Motivating Example. As an example of a redundant traversal bug, consider the program snippet shown in Figure 2.1. This code is taken from version 1.0.17 of the JFreeChart software and exhibits a previously unknown performance bug uncovered by Clarity. In particular, the `render` method (lines 1–19) plots a series of data items in the form (x, y) and invokes the `drawItem` method on line 14 to draw a single point within a given series. Here, the method invocation on line 14 is a virtual call with many possible targets, one of which is the `drawItem` method of `CandlestickRenderer` (lines 20–32).

The performance problem in this example arises because the `drawItem` method iterates over *all points* within the series in order to draw a *single* data point. In particular, the code traverses all data points to compute a value called `xxWidth`, which corresponds to the minimum gap between adjacent x-coordinates in the series. However, since the data set is not modified between successive calls to `drawItem`, the recomputation of `xxWidth` in each call to `drawItem` is redundant and needlessly traverses a potentially large list of data items many times. Hence, this code fragment exhibits a serious performance bug that can be fixed either by passing `xxWidth` as an argument to `drawItem` or by storing it as a field. Not only does such a fix result in a theoretical asymptotic performance improvement of $O(n)$, but it produces an order of magnitude performance improvement in practice.[3]

While it may seem surprising that such a blatant performance bug exists in a mature software project like JFreeChart, there are several reasons why this bug could be missed during development and testing. First, the impact of this performance bug is proportional to the size of the data series and requires data points to be drawn in the shape of candlesticks. Hence, test cases that either use small data series or render objects in a different shape, such as a square, will not reveal the performance bug. Second, the heavy-use of object oriented abstractions obscures the performance bug, making it difficult to spot the problem during manual code inspection. In particular, observe that the `drawItem()` method is virtual, and the performance bug only occurs in the `CandlestickRenderer` implementation. Similarly, the collection that is traversed is hidden behind an interface, so to identify the exact data structure, another virtual method call must be resolved. Finally, there is a function call depth of three between the loop that traverses the data structure and the access of the actual item in the data structure.

[3]For example, using the existing test harnesses from SourceForge, we observe speedups of 8× to 11× when we fix this performance bug and modify the test harness to render each data item in the shape of a candlestick.

```
boolean containsAny1(HashSet<Foo> mySet,
                     ArrayList<Foo> myList) {
  for(Foo f: mySet)
    if(myList.contains(f))
      return true;
  return false;
}
```

Figure 2.2: Check if `myList` contains an element from `mySet`

2.2 Defining Redundant Traversal Bugs

Definition 1. (Traversal) *We say that a code snippet S* traverses *a data structure δ if it performs a computation whose average-case complexity is $\Omega(n)$, where n denotes the number of elements in δ.*

For instance, consider the `contains` methods provided by various data structures in the Java Collections Framework. According to Definition 1, the `contains` method of `ArrayList` performs a traversal of the data structure, as its average-case complexity is $O(n)$. On the other hand, while `HashSet`'s `contains` method has worst-case complexity $O(n)$, it is not considered a traversal because its average-case complexity is $O(1)$.

Definition 2. (Traversal Footprint) *The* traversal footprint *of a code snippet S, written* TraversalFP(S)*, is the set of data structures traversed by S.*

Definition 3. (Write Footprint) *The* write footprint *of code S, written* WriteFP(S)*, is the set of data structures that S modifies.*

```
boolean containsAny2(HashSet<Foo> mySet,
                     ArrayList<Foo> myList) {
  for(int i=0; i < myList.size(); i++) {
    Foo elem = myList.get(i);
    if(mySet.contains(elem))
      return true;
  }
  return false;
}
```

Figure 2.3: Different implementation of code in Figure 2.2.

Definition 4. (Redundant Traversal Bug) *A loop L exhibits a* redundant traversal bug *if there exists a data structure δ such that:*

1. $\delta \in \mathrm{TraversalFP}(L)$ *and* $\delta \notin \mathrm{WriteFP}(L)$
2. δ *is traversed* $\Omega(m)$ *times in L, where m is the number of times that L executes*

In other words, a redundant traversal bug arises if a loop-invariant data structure is traversed a linear number of times within the loop. We believe that this definition captures the intuitive notion of redundancy, as the computation that is performed by traversing the data structure can be done once and re-used across all loop iterations.

Example 1. *Figures 2.2 and 2.3 show the implementation of two methods called* `containsAny1` *and* `containsAny2` *that determine if the intersection of* `myList` *and* `mySet` *is non-empty. While* `containsAny1` *and* `containsAny2` *are functionally equivalent,* `containsAny1` *has a performance bug according to Definition 4 while* `containsAny2` *does not. In particular, since the* `contains` *method of* `ArrayList` *traverses the data structure,* `myList` *is part of the traversal footprint of the loop. By contrast, the* `contains` *method of* `HashSet` *does not perform a traversal; so, the traversal footprint of the loop from Figure 2.3 is empty. Thus,* `containsAny1` *contains an asymptotic performance bug because its average-case complexity is* $O(n \cdot m)$*, whereas the average-case complexity of* `containsAny2` *is* $O(n)$ *for a list of size* n *and set of size* m*.*

The need for the second condition of Definition 4 is illustrated by the following example.

Example 2. *Consider the following code snippet, where the* `computeAvg` *method traverses its input:*

```
int calculate(ArrayList<ArrayList<int>> a) {
 int avgSum = 0;
 for(int i=0; i < a.size(); i++)
   avgSum += computeAvg(a.get(i));
 return avgSum;
}
```

Here, condition (1) of Definition 4 is satisfied because each element of `a` *is part of the traversal footprint but not the write footprint of the loop. However, this code does not have a performance bug because a* different *element of* `a` *is traversed in each loop iteration. Hence, condition (2) is violated.*

Chapter 3

Detecting Performance Bugs[1]

3.1 Core Ideas for Detecting Redundant Traversals

This section explains the key challenges underlying the static detection of redundant traversal bugs and outlines the core ideas behind our static analysis.

First, based on condition (1) of Definition 4, we see that a sound static analysis for detecting redundant data structure traversals must be able to perform the following task:

$$\boxed{\begin{array}{l}\text{Given code snippet } S\text{, overapproximate the emptiness} \\ \text{of the set } \theta \equiv \text{TraversalFP}(S) - \text{WriteFP}(S)\end{array}}$$

Since θ is defined to be the difference of TraversalFP(S) and WriteFP(S), a sound static analysis for detecting redundant traversal bugs must *overapproximate* the traversal footprint but *underapproximate* the write footprint. Furthermore, since our analysis will track data structures in terms of program expressions, we need over- and under-approximating preconditions of program expressions with respect to a given program fragment. For this purpose, we

[1]O. Olivo, I. Dillig, C. Lin. *Static Detection of Asymptotic Performance Bugs in Collection Traversals.* 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15).

define the following notions of *necessary* and *sufficient preconditions*:

Definition 5. (Necessary precondition) *A set of expressions* $\{e_1, \ldots, e_n\}$ *is called a* necessary precondition *of an expression e with respect to a code snippet S, written* $\text{pre}^+(e, S)$*, if, for any constant c, the following Hoare triple is valid:*

$$\{e_1 \neq c \wedge \ldots \wedge e_n \neq c\}\ S\ \{e \neq c\}$$

In other words, for e to have value c after S, it is necessary that some element in $\text{pre}^+(e, S)$ has value c before S; hence, we refer to the set $\{e_1, \ldots, e_n\}$ as a necessary precondition of e with respect to S. Now, we also define a dual notion of *sufficient preconditions*:

Definition 6. (Sufficient precondition) *A set of expressions E is called a* sufficient precondition *of expression e with respect to code S, written* $\text{pre}^-(e, S)$*, if, for all constants c and all* $e' \in E$*, the following Hoare triple is valid:*

$$\{e' = c\}\ S\ \{e = c\}$$

In other words, for e to have value c after S, it is sufficient that elements in $\text{pre}^-(e, S)$ have value c before S. Thus, sufficient conditions underapproximate the weakest precondition of an expression e with respect to code S.

Example 3. *Consider the following code snippet S:*

```
if(*) x := y else x := z;
```

Here, we have $\text{pre}^+(x, S) = \{y, z\}$ *and* $\text{pre}^-(x, S) = \emptyset$*. In particular, for* `x` *to be equal to a certain value* c *after* S*, it is necessary that either* `y` *or* `z` *have value* c *before* S*. However, the sufficient precondition for* `x` *is* $\emptyset$ *because neither* $y = c$ *nor* $z = c$ *before* S *guarantees that* $x = c$ *after* S*.*

Now, given a statement S and a sub-statement π nested inside S, we will use the notation $S^-[\pi]$ to denote the code that comes before π in S. For instance, if S is the code:

```
x:=y; if(x>10) x++; y--; else x := 0
```

and π is the statement `y--`, then $S^-[\pi]$ is:

```
x:=y; assume(x>10); x++;
```

The following theorem explains why necessary and sufficient preconditions are useful for checking condition (1) of Definition 4.

Theorem 1. *Let* S *be a code snippet containing two sets of statements* Π_1 *and* Π_2 *such that:*

1. *Each statement* $\pi_i \in \Pi_1$ *traverses a data structure referred to by program expression* e_i
2. *Each* $\pi'_j \in \Pi_2$ *modifies a data structure referred to by* e'_j

Then, $\text{TraversalFP}(S) - \text{WriteFP}(S) = \emptyset$ *if:*

$$(\bigcup_{\pi_i \in \Pi_1} \text{pre}^+(e_i, S^-[\pi_i]) - \bigcup_{\pi'_j \in \Pi_2} \text{pre}^-(e'_j, S^-[\pi'_j]) \ ) = \emptyset \qquad (*)$$

Proof. Suppose $\text{TraversalFP}(S) - \text{WriteFP}(S) \neq \emptyset$ but $(*)$ holds. Then there must be some statement π that traverses a data structure δ that is referred to by expression e and δ is not modified in S. Let $\text{pre}^+(e, S^-[\pi]) = \{e_1, \ldots, e_n\}$. By definition of necessary precondition, this implies that $e_1 = \delta \vee \ldots \vee e_k = \delta$ before S. Now, since condition $(*)$ holds, every e_i is in the set $\text{pre}^-(e'_j, S^-[\pi'_j])$ for some statement π'_j modifying data structure referred to by expression e'_j. By definition of sufficient precondition, this means that $\{e_i = \delta\}\ S^-[\pi'_j]\ \{e'_j = \delta\}$. But this implies that δ must also be modified, i.e., a contradiction.

□

Theorem 1 is useful because it provides a method for statically checking condition (1) of Definition 4. In particular, to determine whether $\text{TraversalFP}(S) - \text{WriteFP}(S)$ *may* be empty, we compute necessary preconditions E of all program expressions that are traversed and sufficient preconditions E' of all expressions that are modified. If $E - E'$ is empty, then Theorem 4 implies that all expressions that are traversed are also modified; hence, we can rule out a potential redundant traversal bug. This is a key insight underlying our static analysis, and we will compute necessary preconditions of data structures that are traversed and sufficient preconditions for expressions that are modified in Section 3.2.

We now turn to the problem of statically checking condition (2) from Definition 4. That is, given a loop-invariant data structure δ that is traversed within the loop, is δ traversed at least a linear number of times? In our analysis, we will check this linearity requirement by over-approximating the following slightly stronger condition:

Given a loop L and a data structure δ, is δ traversed in all iterations of L?

The above criterion is stronger than checking whether δ is traversed $\Omega(m)$ times in the loop. However, since our static analysis is path-insensitive, soundly answering the above question overapproximates condition (2) of Definition 4 for all practical purposes.

Example 4. *Consider the following code snippet, where* `n` *is a positive integer and* `traverse` *performs list traversal:*

```
for(i=0; i<n; i++) {if(i%2 == 0) traverse(myList);}
```

Here, `myList` *is not traversed in all iterations, but it is traversed* $\Omega(n)$ *times. However, a sound static analysis that treats the test* `i%2 == 0` *as a non-deterministic choice will conclude that* `myList` *may be traversed in all iterations.*

The following theorem is useful in determining whether a data structure δ may be traversed in all loop iterations:

Theorem 2. *Let e be a program expression, and let E be a necessary precondition of e with respect to code snippet S. Then, the following Hoare triple is valid for any constant c:*

$$\{e = c \ \wedge \bigwedge_{e_i \in E} e \neq e_i\} \ S \ \{e \neq c\}$$

Proof. First, note that the following implication is valid:

$$e = c \wedge \ (\bigwedge_{e_i \in E} e \neq e_i) \Rightarrow (\bigwedge_{e_i \in E} e_i \neq c)$$

Now, by definition of necessary precondition, we have:

$$\{(\bigwedge_{e_i \in E} e_i \neq c)\} \ S \ \{e \neq c\}$$

Hence, the theorem holds by precondition strengthening. □

Simply put, this theorem states that the value of an expression e has different values before and after executing S *provided that* e is distinct from every $e_i \in \text{pre}^+(e, S)$. To see the relevance of this theorem, suppose that e is a program expression that may be traversed in the loop. Now, if the value of e changes between any two consecutive loop iterations, then two different data structures δ and δ' are traversed; hence, δ is not traversed in all loop iterations. Thus, the key question to answer is whether the value of e can change between different loop iterations. Fortunately, we can answer this question using Theorem 2. Specifically, let E be the necessary precondition of e with respect to the loop body S. Based on Theorem 2, if we can prove that

$$\begin{array}{lcl}
\text{Program } P & := & \tau_1\ v_1; \ldots\ \tau_n\ v_n;\ S \\
\text{Type } \tau & := & \text{Int} \mid \text{Collection}\langle \tau_1, \tau_2 \rangle \\
\text{Statement } S & := & \text{skip} \mid v := e \mid v.\text{traverse}() \\
 & & \mid v_1 := v_2.\text{get}(v_3) \mid v_1.\text{put}^{\rho}(v_2, v_3) \\
 & & \mid S_1; S_2 \mid \text{if}(\star)\ \text{then}\ S_1\ \text{else}\ S_2 \\
 & & \mid \text{while}(\star)\ \text{do}^{\rho}\ S \\
\text{Expression } e & := & \text{int} \mid v \mid e_1 \oplus e_2 \quad (\oplus \in \{+, -, \times\})
\end{array}$$

Figure 3.1: Language used for formal development.

e is distinct from every $e_i \in E$, then we know that the same data structure is not traversed in all loop iterations.

Example 5. *Consider again the code from Example 2, where* `computeAvg` *traverses the input array. Here, the loop traverses program expression* `a[i]`*, but the necessary precondition of* `a[i]` *with respect to the loop body is* {`a[i+1]`}*. Thus, assuming* `a[i]` *and* `a[i+1]` *do not alias, we can determine that condition (2) of Definition 4 is violated.*

3.2 Static Analysis

We now use the ideas introduced in Section 3.1 to describe our static analysis for detecting performance bugs.

3.2.1 Language

To formally describe our analysis, we use the small imperative language shown in Figure 7.1. This language contains two types of variables, namely, scalars of type Int and references of type Collection. We model collections as key-value stores that support insertion and retrieval of values associated with a given key. Hence, a variable of type Collection$\langle \tau_1, \tau_2 \rangle$ models a key-value store where keys are of type τ_1 and values are of types τ_2. Observe that both keys and values may be of type Collection; hence, it is possible to nest an arbitrary number of data structures within another one.

In the language shown in Figure 7.1, statements include skip, assignments of the form $v := e$, and the following three collection-manipulating operations:

- A statement v.traverse() traverses collection v, where traversal encompasses any operation that is consistent with Definition 1.
- A statement $v_1 := v_2$.get(v_3) retrieves the value v_1 of key v_3 in the data structure pointed to by variable v_2.
- A statement v_1.put$^{\rho}$(v_2, v_3), where ρ denotes a program point, associates value v_3 with key v_2 in the data structure referenced by variable v_1.

In addition, statements also include sequences $S_1; S_2$, if statements, and while loops. Since our analysis does not interpret conditionals (i.e., is path-insensitive), we model conditionals using non-deterministic choices indicated as $\star$ in Figure 7.1. Furthermore, we assume that while loops are annotated with a program point ρ which denotes the program location right before the first instruction in the loop body.

$$
\begin{array}{lcl}
\text{Symbolic exp } \pi & := & c \mid v \mid \pi_1\langle\pi_2\rangle \\
\text{Read footprint } \Phi & := & 2^\pi \\
\text{Write footprint } \Psi & := & 2^\pi \\
\text{Alias environment } \mathcal{E} & := & \rho \times \pi \rightarrow (2^\pi, 2^\pi)
\end{array}
$$

Figure 3.2: Summary of notation used in formalization

To simplify the formalization of our analysis, we omit function calls from this language and assume that the only way to traverse a data structure is by calling v.traverse(). In Section 3.3, we explain the inference of methods that traverse data structures as well as our interprocedural analysis.

3.2.2 Computing Traversal and Write Footprints

We now describe our static analysis for over- and under-approximating each statement's traversal and write footprints. Our analysis is a backwards dataflow analysis and is presented in Figure 3.3 using judgments of the form:

$$\mathcal{E}, \Phi, \Psi \vdash S : \Phi', \Psi'$$

This judgment means that under the aliasing relations given by environment $\mathcal{E}$, if Φ and Ψ denote the traversal and write footprints after statement S, then Φ' and Ψ' over- and under-approximate the traversal and write footprints before S respectively. That is, assuming the correctness of $\mathcal{E}, \Phi$ and Ψ, the set Φ' over-approximates all collections that are traversed in or after S in terms of program expressions *before* S. Similarly, the set Φ' under-approximates all collections that must be modified in terms of program expressions before S.

As summarized in Figure 3.2, our analysis tracks traversal and write footprints using sets of symbolic expressions π. Symbolic expressions π can be constants c, variables v, or expressions of the form $\pi_1\langle\pi_2\rangle$, which represents the value associated with key π_2 in a data structure represented by expression π_1. For example, if the traversal footprint Φ of some statement S includes an expression $v\langle 3\rangle$, then the data structure stored at index 3 of the collection referenced by variable v may be traversed by statement S.

Since variables of type Collection are references in our language, our analysis must take possible aliasing relations into account if it is to soundly compute traversal and write footprints. Thus, our analysis rules utilize an *aliasing environment* $\mathcal{E}$ which maps each expression to its set of aliases. However, since our goal is to under-approximate write footprints, we need *must alias* facts as well as *may alias* information, so the aliasing environment $\mathcal{E}$ has signature $\rho \times \pi \rightarrow (2^\pi, 2^\pi)$, which maps each expression π and program point ρ to π's may- and must-aliases at program point ρ. In what follows, we will assume that such an aliasing environment $\mathcal{E}$ has been computed by performing may- and must-alias analyses prior to our footprint computation.

Let us now consider the analysis rules shown in Figure 3.3. In particular, rule (2) describes the analysis of assignments of the form $v := e$. Here, we replace any variable v used in Φ and Ψ by expression e because e is both a necessary and sufficient precondition for v with respect to statement S. For instance, for a statement $v_1 := v_2$, traversal footprint $\Phi = \{v_1, y\}$ and write footprint $\Psi = \{x\langle v_1\rangle\}$, our analysis computes $\Phi' = \{v_2, y\}$ and $\Psi' = \{x\langle v_2\rangle\}$.

$$(1)\quad \frac{}{\mathcal{E}, \Phi, \Psi \vdash \text{skip} : \Phi, \Psi}$$

$$(2)\quad \frac{\Phi' = \Phi[e/v] \quad \Psi' = \Psi[e/v]}{\mathcal{E}, \Phi, \Psi \vdash v := e : \Phi', \Psi'}$$

$$(3)\quad \frac{\Phi' = \Phi \cup \{v\}}{\mathcal{E}, \Phi, \Psi \vdash v.\text{traverse}() : \Phi', \Psi}$$

$$(4)\quad \frac{\begin{array}{c}\Phi' = \Phi[v_2\langle v_3\rangle/v_1] \\ \Psi' = \Psi[v_2\langle v_3\rangle/v_1]\end{array}}{\mathcal{E}, \Phi, \Psi \vdash v_1 := v_2.\text{get}(v_3) : \Phi', \Psi'}$$

$$(5)\quad \frac{\begin{array}{c}\mathcal{E}(\rho, v_1) = (\mathcal{A}_1^+, \mathcal{A}_1^-) \quad \mathcal{E}(\rho, v_2) = (\mathcal{A}_2^+, \mathcal{A}_2^-) \\ \Phi' = \Phi[v_3/\mathcal{A}_1^+\langle\mathcal{A}_2^+\rangle] \cup (\Phi \ominus \mathcal{A}_1^-\langle\mathcal{A}_2^-\rangle) \\ \Psi' = \Psi[v_3/\mathcal{A}_1^-\langle\mathcal{A}_2^-\rangle] \cup (\Phi \ominus \mathcal{A}_1^+\langle\mathcal{A}_2^+\rangle)\end{array}}{\mathcal{E}, \Phi, \Psi \vdash v_1.\text{put}^\rho(v_2, v_3) : \Phi', \Psi' \cup \{v_1\}}$$

$$(6)\quad \frac{\begin{array}{c}\mathcal{E}, \Phi, \Psi \vdash S_2 : \Phi', \Psi' \\ \mathcal{E}, \Phi', \Psi' \vdash S_1 : \Phi'', \Psi''\end{array}}{\mathcal{E}, \Phi, \Psi \vdash S_1; S_2 : \Phi'', \Psi''}$$

$$(7)\quad \frac{\begin{array}{c}\mathcal{E}, \Phi, \Psi \vdash S_1 : \Phi_1, \Psi_1 \\ \mathcal{E}, \Phi, \Psi \vdash S_2 : \Phi_2, \Psi_2\end{array}}{\mathcal{E}, \Phi, \Psi \vdash \text{if}(\star) \text{ then } S_1 \text{ else } S_2 : \Phi_1 \cup \Phi_2, \Psi_1 \cap \Psi_2}$$

$$(8)\quad \frac{\begin{array}{c}\Phi' \supseteq \Phi \quad \Psi \supseteq \Psi' \\ \mathcal{E}, \Phi', \Psi' \vdash \widetilde{S} : \Phi', \Psi'\end{array}}{\mathcal{E}, \Phi, \Psi \vdash \text{while}(\star) \text{ do } S : \Phi', \Psi'}$$

Figure 3.3: Analysis rules for computing traversal and write footprints. The notations $\mathcal{A}_1\langle\mathcal{A}_2\rangle$ and $\Phi[v/\mathcal{A}]$ are defined in Equations 3.1 and 3.2, and operator $\ominus$ is defined in Equation 3.3.

Rule (3) describes the analysis of traversals of the form v.traverse. In this case, we simply add variable v to the traversal footprint Φ; the write footprint Ψ remains unchanged.

Rule (4) describes the analysis of retrieval (i.e., load) operations of the form $v_1 := v_2.\text{get}(v_3)$. Similar to the assignment rule, we replace variable v_1 in the traversal and write footprints with the expression $v_2\langle v_3\rangle$, which denotes the value associated with key v_3 in the collection referenced by v_2. Observe that $v_2\langle v_3\rangle$ is both a necessary and sufficient precondition for v_1 with respect to the statement $v_1 := v_2.\text{get}(v_3)$: In particular, the value of v_1 is equal to constant c after this statement if and only if $v_2\langle v_3\rangle = c$ before executing $v_1 := v_2.\text{get}(v_3)$.

The most involved part of the analysis is Rule (5) for analyzing insertion (i.e., store) operations. To build intuition, let us first consider the statement $S = v_1.\text{put}(v_2, v_3)$ and an expression $x\langle y\rangle \in \Phi$. There are two cases to consider:

- If x must alias v_1 and y must alias[2] v_2, then the necessary precondition for $x\langle y\rangle$ is just $\{v_3\}$ since, for any value c, condition $v_3 \neq c$ before S guarantees $x\langle y\rangle \neq c$ after S.
- On the other hand, if x may alias v_1 and y may alias v_2 (but either may-alias relation is not also a must-alias relation), then the necessary precondition for $x\langle y\rangle$ is the set $\{x\langle y\rangle, v_3\}$. Observe that neither condition

[2]Here, since y and v_2 may be scalars, we overload the term "alias" to also mean equality for scalars.

$x\langle y\rangle \neq c$ nor $v_3 \neq c$ before S on its own guarantees $x\langle y\rangle \neq c$ after S, as the value of $x\langle y\rangle$ may—but does not have to– be affected by S. However, if we know $x\langle y\rangle \neq c \wedge v_3 \neq c$ before S, we can guarantee that $x\langle y\rangle \neq c$ after S.

Now, let us also consider the analogous case where $x\langle y\rangle \in \Psi$. Again, there are two cases to consider:

- If x must alias v_1 and y must alias v_2, then the sufficient precondition for $x\langle y\rangle$ is just $\{v_3\}$ since, for any value c, condition $v_3 = c$ before S guarantees $x\langle y\rangle = c$ after S.
- Otherwise, if x may alias v_1 and y may alias v_2, then the sufficient precondition for $x\langle y\rangle$ is the empty set, as there is no program expression whose value before S is guaranteed to be the same as the value of $x\langle y\rangle$ after S.

As this example illustrates, the computation of traversal and write footprints for store operations requires aliasing information for pointers (and equality information for scalars). With this intuition in mind, we now explain Rule (5) from Figure 3.3. As expected, we first need to look up the set of may- and must-aliases $(\mathcal{A}_1^+, \mathcal{A}_1^-)$ of v_1 as well as those of v_2 $(\mathcal{A}_2^+, \mathcal{A}_2^-)$. Now, any expression of the form $x\langle y\rangle$ may be affected by the statement $v_1.\text{put}(v_2, v_3)$ if x is an alias of v_1 and y is an alias of v_2. Given set of symbolic expressions $\mathcal{A}$ and $\mathcal{A}'$, we use the notation $\mathcal{A}\langle\mathcal{A}'\rangle$ to represent:

$$[\![\mathcal{A}\langle\mathcal{A}'\rangle]\!] = \bigcup_{\pi\in\mathcal{A}}\bigcup_{\pi'\in\mathcal{A}'} \pi\langle\pi'\rangle \tag{3.1}$$

Hence, in Rule (5), $\mathcal{A}_1^+\langle\mathcal{A}_2^+\rangle$ yields the set of expressions that *may* be affected by the update, while $\mathcal{A}_1^-\langle\mathcal{A}_2^-\rangle$ represents expressions that *must* be overwritten.

Now, let us focus on the computation of the traversal footprint Φ' in the second line of Rule (5). Here, for a set $\mathcal{A} = \{\pi_1, \ldots, \pi_n\}$, we use the notation $\Phi[v/\mathcal{A}]$ as shorthand for:

$$\Phi[v/\{\pi_1, \ldots, \pi_n\}] = \Phi[v/\pi_1, \ldots, v/\pi_n] \tag{3.2}$$

Hence, the set $\Phi[v_3/\mathcal{A}_1^+\langle\mathcal{A}_2^+\rangle]$ is the same as Φ except that every (sub-) expression that may correspond to $v_1\langle v_2\rangle$ has been replaced with v_3. However, since we want to over-approximate the footprint, Φ' must also contain any expression π such that (i) $\pi \in \Phi$ and (ii) no prefix of π is in the set $\mathcal{A}_1^-\langle\mathcal{A}_2^-\rangle$, because such an expression π is not guaranteed to be killed by the store operation. To capture all expressions in Φ that are preserved by the statement $v_1.\text{put}(v_2, v_3)$, we define an $\ominus$ operation on expression sets as follows:

$$\pi \in (\mathcal{A}_1 \ominus \mathcal{A}_2) \Leftrightarrow \pi \in \mathcal{A}_1 \wedge \forall\pi' \in \mathcal{A}_2.(\pi' \neq \text{prefix}(\pi)) \tag{3.3}$$

In other words, $\mathcal{A}_1 \ominus \mathcal{A}_2$ preserves exactly those expressions π in $\mathcal{A}_1$ where π is not an extension of some expression in $\mathcal{A}_2$. Hence, the overall effect is two-fold:

- Φ' contains v_3 if Φ contains some expression $x\langle y\rangle$ where x and y may alias v_1 and v_2 respectively
- Φ' contains any expression $\pi \in \Phi$ such that π is not guaranteed to be modified by the statement $v_1.\text{put}(v_2, v_3)$

We now explain the computation of the write footprint Ψ', which is described in the third line of Rule (5). First, observe that Ψ' contains v_3 iff there exists some $x\langle y\rangle \in \Psi$ such that x and y must alias v_1 and v_2. Furthermore, we kill all expressions in Ψ of the form $x'\langle y'\rangle$ where x' and y' may alias v_1 and v_2, respectively. Finally, since the statement $v_1.\text{put}(v_2, v_3)$ modifies collection v_1, the write footprint before this statement includes variable v_1.

Example 6. *Consider the following code snippet where x and y are variables of type* Collection$\langle$Int, Collection$\langle$Int$\rangle\rangle$ *and z, w are variables of type* Collection$\langle$Int, Int$\rangle$*:*

1.	$y.\text{put}(0, w);$	$\Phi_1 = \{x\langle 0\rangle, w\}$	$\Psi_1 = \{y, w\}$
2.	$z := x.\text{get}(0);$	$\Phi_2 = \{x\langle 0\rangle\}$	$\Psi_2 = \{w\}$
3.	$w.\text{put}(2, 5);$	$\Phi_3 = \{z\}$	$\Psi_3 = \{w\}$
4.	$z.\text{traverse}();$	$\Phi_4 = \{z\}$	$\Psi_4 = \emptyset$

The annotations Φ_i and Ψ_i show the traversal and write footprints right before statement S_i under the assumption that x and y may alias (but are not guaranteed to).

We now consider the last two rules in Figure 3.3. When analyzing if statements in Rule (7), we take the union of the traversal footprints Φ_1 and Φ_2 obtained from the two branches. On the other hand, since we need to underapproximate the write footprint, we take the intersection of Ψ_1 and Ψ_2. rather than their union.

The final rule (8) describes the analysis of while loops[3]. In this rule, we use the notation $\widetilde{S}$ to denote the resulting statement when all traversal statements in S are replaced by skip. In particular, to avoid reporting the same warning for both inner and outer loops, our analysis ignores traversals in nested loops when computing the traversal footprint associated with an outer loop.

Now, continuing with rule (8), the traversal and write footprints Φ' and Ψ' must satisfy the following properties:

- Φ' must be a superset of Φ since any expression that is traversed after the loop may also be traversed before the loop (observe that the loop may execute zero times).
- Ψ' must be a subset of Ψ since only those expressions that are modified after the loop are guaranteed to be modified before the loop.
- Φ' and Ψ' must be inductive with respect to loop body $\widetilde{S}$.

[3]This rule is only needed when detecting performance bugs in loops that contain at least one nested loop.

Example 7. *Consider the following code snippet:*

1. while($\star$) do $i = i + 1$; if($\star$) then $a = b$ else $b = a$;
2. a.traverse(); b.put$(2, 5)$;

Right before line 2, we have the traversal and write footprints $\Phi_2 = \{a\}, \Psi_2 = \{b\}$. *On the other hand, the traversal and write footprints before line 1 are* $\Phi_1 = \{a, b\}$ *and* $\Psi_1 = \emptyset$.

3.2.3 Detecting Redundant Traversal Bugs

We now explain how the computed traversal and write footprints are used to detect redundant traversal bugs. Figure 3.4 summarizes our performance bug detection algorithm using the judgments from Figure 3.3. As expected, our analysis only reports warnings when analyzing loops. Specifically, as shown on the first line of the Err rule, we first compute the traversal and write footprints Φ, Ψ associated with the loop body. Now recall from Section 3.1 that the loop contains a redundant traversal bug if there exists a $\pi \in \Phi$ such that (i) π is traversed in all loop iterations, and (ii) π is loop invariant (i.e., is not modified within the loop).

To determine if condition (i) holds, we use Theorem 2 from Section 3.1 to check whether π is distinct from every expression $\pi' \in \text{pre}^+(\pi, S)$. More specifically, in the All rule, $\mathcal{NC}$ corresponds to $\text{pre}^+(\pi, S)$, and the check $\mathcal{A}^+ \cap \mathcal{NC} = \emptyset$ stipulates that π does not alias any expression in $\text{pre}^+(\pi, S)$. Hence, by Theorem 2, if the predicate traversal_all(S^ρ, π) evaluates to true, then π may be traversed in all loop iterations.

$$\frac{\begin{array}{c}\mathcal{E}, \{\pi\}, \_ \vdash \widetilde{S} : \mathcal{NC}, \_ \\ \mathcal{E} \vdash (\rho, \pi) : (\mathcal{A}^+, \mathcal{A}^-) \\ \mathcal{A}^+ \cap \mathcal{NC} \neq \emptyset\end{array}}{\mathcal{E} \vdash \text{traverse\_all}(S^\rho, \pi)} \quad (\textsc{All})$$

$$\frac{\begin{array}{c}\mathcal{E} \vdash (\rho, \pi) : (\mathcal{A}^+, \mathcal{A}^-) \\ \mathcal{A}^- \cap \Psi = \emptyset\end{array}}{\mathcal{E}, \Psi \vdash \text{loop\_inv}(S^\rho, \pi)} \quad (\textsc{Inv})$$

$$\frac{\begin{array}{c}\mathcal{E}, \emptyset, \emptyset \vdash S : \Phi, \Psi \\ \pi \in \Phi \\ \mathcal{E} \vdash \text{traverse\_all}(S^\rho, \pi) \\ \mathcal{E}, \Psi \vdash \text{loop\_inv}(S^\rho, \pi)\end{array}}{\mathcal{E} \vdash \text{while}(\star)\ \text{do}^\rho\ S \rightsquigarrow \textsc{Error}} \quad (\textsc{Err})$$

Figure 3.4: Summary of performance bug detection. As before, $\widetilde{S}$ denotes traverse statements in S replaced by skip.

Now, we still need to check that π is not modified within the loop. For this purpose, we use the Inv rule, which checks if some must-alias of π is in the write footprint Ψ. If not (i.e., $\mathcal{A}^+ \cap \Psi = \emptyset$), this implies that π may be loop invariant, so our analysis reports a potential performance bug.

Example 8. *Consider the following code snippet:*

$$\text{while}(\star)\ \text{do}\ \ t := a.\text{get}(i);\ t.\text{traverse}();\ i := i + 1;$$

Here, the traversal footprint Φ for the loop body is $\{a\langle i\rangle\}$, and the write footprint Ψ is $\emptyset$. Since the necessary precondition of $a\langle i\rangle$ is $\Phi' = \{a\langle i+1\rangle\}$, the All *rule fails (assuming $a\langle i\rangle$ and $a\langle i+1\rangle$ do not alias), so our analysis does not report a performance bug.*

Example 9. *Consider the following code snippet:*

1. if($\star$) then $b := a$; else skip;
2. while($\star$) do a.traverse(); b.put$(2, 4)$;

Here, we have $\Phi = \{a\}$ and $\Psi = \{b\}$. The necessary precondition of a with respect to the loop body is $\{a\}$, so using the All *rule, we determine that a may be traversed in all loop iterations. Furthermore, since a and b are not guaranteed to alias, the* Inv *rule succeeds, so our analysis reports a redundant traversal bug. However, if we changed line* 1 *to $b := a$, then our analysis would not report a performance bug, since a and b are now guaranteed to alias.*

3.3 Implementation

We have implemented our proposed analysis in a tool called Clarity, which is written in Java and consists of approximately 8,000 lines of code. Clarity is implemented in the Soot compiler framework [171] and uses the Jimple intermediate representation. Clarity relies on Soot for CFG and callgraph construction and issues a warning if any possible target of a virtual method call contains a performance bug. We have also implemented our own summary-based pointer analysis for computing may and must aliases by adapting the ideas described by Dillig et al. [63].

In this section, we focus on two important aspects of the implementation: (1) the identification of data structure traversals and (2) interprocedural analysis.

Identifying Data Structure Traversals. Our implementation uses a combination of automatic inference and a small amount of manual annotations to identify data structure traversals. In particular, we manually annotate 28 methods that (a) are implemented by the Java collections library and (b) have average-case complexity that is at least linear in the size of the data structure.[4] For instance, as we saw in Section 3.1, the `contains` method of `ArrayList` is annotated as a traversal, while `HashSet`'s `contains` method is not.

In addition to such manual annotations, Clarity performs automated inference to identify code snippets that traverse data structures through iterators or loops. However, our current implementation does not detect data structure traversals in recursive functions. Hence, if a Java application implements its own tree traversal algorithm, Clarity will not be able to detect performance bugs that arise from redundant traversals of this custom tree data structure.

To detect data structures that are traversed in loops, our static analysis also maintains a so-called *read footprint*. In particular, a data structure δ is added to the read footprint for code snippet S if S may access δ. For instance, using the notation from Section 3.2, if a code snippet accesses an element of array `a`, we simply add `a` to the read footprint $\mathcal{R}$ and compute $\mathcal{R}$'s necessary precondition during our backwards analysis. When we encounter a loop, we then check whether an element $a \in \mathcal{R}$ is accessed multiple times

[4]The annotated methods represent a tiny fraction of the analyzed methods.

using Theorem 2. This analysis is similar to the check that tests whether a data structure is traversed multiple times (see the ALL rule in Figure 3.4). If a given data structure may be accessed in multiple loop iterations, we then add it to the traversal footprint Φ. The following example illustrates this analysis:

Example 10. *Consider the following code snippet:*

$$\text{while}(\star)\ \text{do}\ t := l.\text{get}(i);\ \ \text{sum} := \text{sum} + t;\ \ i := i + 1;$$

While analyzing the loop body, we add variable l to $\mathcal{R}$. Then, when analyzing the while loop, we check whether l may be accessed in all loop iterations by testing whether the necessary precondition for l includes itself. Since it does in this case, l is added to the traversal footprint Φ of the while loop.

Interprocedural Analysis. Since the key idea underlying our technique is to compute traversal and write footprints, our analysis is amenable to modular reasoning. In particular, our interprocedural analysis computes a *procedure summary* for each method that tracks its read, traversal, and write footprints as well as transfer functions that give the necessary and sufficient preconditions for each data structure used in that method.

When we encounter a call to method m, we simply retrieve m's summary and *instantiate* its read, write, and traversal footprints by applying the appropriate formal-to-actual mapping. The instantiated callee footprints are then added to the corresponding footprints of the caller. In addition, the summary for each procedure also contains transfer functions that describe side effects of the callee. These transfer functions correspond to necessary and sufficient preconditions of program expressions with respect to the callee's body. Hence, for each expression e in the caller's read, write, and traversal footprints, we apply the appropriate transfer function to obtain the new footprints before the method call.

3.4 Experimental Evaluation

Table 3.1: Experimental results: Performance Bugs Detected.

Name	LoC	LoC w/ libraries	Number of Methods	Analysis Time (sec)	Reported Bugs	Previously Unknown Bugs	False Positives
Charts4j	21,396	28,667	715	122	0	0	0
Janino	39,832	305,660	7,149	556	3	3	0
Apache Collections	58,186	58,186	3,759	180	19	10	4
Ode4J	83,207	83,207	4,048	128	0	0	0
Java3D	115,859	146,376	1,875	335	0	0	0
Guava (Google Core)	129,745	129,745	12,338	338	55	44	1
LWJGL (Game Library)	202,902	267,248	10,740	1,584	10	10	0
Apache Ant	205,371	265,828	10,029	1,325	2	2	0
JFreeChart	226,623	362,584	9,162	602	3	3	0
Total	1,083,121	1,647,501	59,815	5,470	92	72	5

We evaluate Clarity by applying it to nine mature and widely-used open source code bases (see Table 5.4). We conduct our experiments on an x86_64 Ubuntu 12.04 desktop machine with 8 GB of RAM and a dual-core 3 GHz processor. We evaluate our approach using the following metrics: (1) the number of performance bugs detected by Clarity, (2) the number of false positives reported, and (3) the impact of fixing these bugs.

Table 5.4 summarizes the results of our experimental evaluation, with the benchmarks listed in order of increasing code size. Our benchmarks range from 21,396 to 226,623 lines of Java source code and contain between 715 and 12,338 methods (including external library calls). Since CLARITY also analyzes the external libraries called by each application, the actual number of lines of code analyzed by the tool can be much larger (see third column of Table 5.4). We see that even though CLARITY performs a non-trivial interprocedural static analysis, the running time of the tool is quite reasonable, with LWJGL taking the longest at 26.4 minutes. Note that the reported times include pointer analysis as well as the time required to analyze library code.

Most significantly, we see that CLARITY reports a total of 92 performance bugs, with only five of these being false positives. Furthermore, among the 92 true positives, 72 are previously unreported according to the SourceForge development logs. The numbers in this table include only unique performance bugs; that is, performance bugs that are inherited by a sub-class are not counted multiple times.

Finally, to evaluate CLARITY's performance impact, we repair each of the identified performance bugs, for example, by adding additional fields to classes, passing extra parameters to methods, or transforming data structures (e.g., lists into sets). We then compare the execution time of the original and repaired programs on input sizes ranging from a few thousand to a few hundred thousand elements. In this evaluation, we observe speedups in the range 2.5-548.2$\times$ on inputs sizes of 50,000 and speedups between 6.6-3,350$\times$ on input sizes of 100,000. This empirical comparison between the original and modified programs confirms that the redundant traversals identified by Clarity indeed correspond to asymptotic performance problems.

3.4.1 Discussion

We now explain the most commonly reported performance bugs and share some observations about the nature of the performance bugs detected by Clarity.

RetainAll Performance Bug. The *RetainAll* bug occurs in code that removes all elements in a collection A that are not also present in another collection B (often passed as a parameter). The inefficiency occurs when B has a slow containment checking method that is invoked a linear number of times. This bug can typically be fixed by converting the parameter collection B to a set, either within the algorithm or at its call site, or by more complicated means such as keeping an iterator on the parameter collection and advancing it accordingly to avoid redundant checks. In addition to appearing in doubly-nested loops, this bug can also appear in other ways, such as the pruning of multi-maps, removal of data points from a plot, and intersection of build resources while compiling a Java application. In fact, we observe some variant of the *RetainAll* bug in four code bases, namely, Apache Ant, Guava, JFreeChart, and Apache Collections.

ContainsAll Performance Bug. The *ContainsAll* bug is similar to *RetainAll* and occurs in code that checks whether a collection contains all the elements in some other collection, which is often a method parameter. As in the *RetainAll* bug, we see that the flexibility of generic types for collection parameters can lead to severe performance degradation. We found several instances of this performance bug in Guava and Apache Collections.

FilterPredicate Bug. This bug occurs when a containment predicate P is attached to a collection C, and elements can only be added to C if they satisfy P. Typically, the root cause of the performance problem is an inefficient data structure used in the implementation of the predicate. We see several instances of this bug in the Guava libraries.

TransformPredicate Bug. This type of performance bug is similar to the *FilterPredicate* bug. It appears when an inefficient predicate is used to identify elements that should be removed from a collection. Again, we find several occurrences of this type of performance bug in the Guava libraries.

ExtremeVal Bug. The *ExtremeVal* performance bug occurs when the maximum or minimum value of a list of elements is computed multiple times, even though the list does not change. An instance of this type of bug is the JFreeChart example from Figure 2.1.

PatternMatching Bug. This bug occurs when checking a set of elements against a set of patterns. The redundant traversals could be avoided by combining the set of patterns into one pattern, simplifying it, and then checking the elements against this pattern. An instance of this bug arises in Apache Ant when testing if files in a directory satisfy a regular expression describing an include-path.

False Positives. Four of the false positives in our experiment arise from imprecise virtual method call resolution. For example, in some cases, CLARITY believes that the target of a virtual method call may be `LinkedList::contains` even though it can only be `HashMap::contains`. The fifth false positive arises when CLARITY believes that a data structure is traversed multiple times in a loop that can be traversed only once. In this case, the code has a nontrivial loop invariant that CLARITY cannot reason about.

Nature of Performance Bugs. We now discuss some observations about the nature of performance bugs uncovered by CLARITY.

First, while the majority of the bugs detected by CLARITY are *conceptually* quite simple, they are not easily identifiable through either manual code review or simple static analysis. Due to the heavy use of abstraction in Java, the collection that is traversed is often hidden behind an interface, so identifying data structures that are accessed requires virtual method call resolution. Furthermore, the loop where the performance bug arises is typically defined in a different class or method from the code that actually traverses the data structure. Hence, the detection of such bugs requires fairly sophisticated interprocedural static analysis.

Second, even though there are conceptual similarities between the performance bugs identified by CLARITY, different code snippets exhibiting conceptually similar bugs can be syntactically *very different*. For example, a performance bug that is classified in the *RetainAll* category appears in a method called `createConsolidatedPieDataset`, which tries to create a new dataset with keys above a certain threshold. Meanwhile, another instance of the *RetainAll* bug appears in the Apache collections in a method called `retainAll` and looks very different from the JFreeChart instance of the *RetainAll* bug. Thus, despite conceptual similarities among various performance bugs detected by CLARITY, these bugs cannot be detected by performing syntactic pattern matching on program constructs.

Finally, our empirical evaluation sheds light on the difficulty of detecting these performance bugs during testing. First, several performance bugs identified by CLARITY arise in rarely executed program paths. For instance, recall the performance bug from Figure 2.1, which is triggered when the user renders items in the shape of a candlestick. Since this shape is unlikely to be a popular choice, it is unlikely to be triggered during testing. Second, as observed in our empirical performance comparison between the original and repaired programs, the true cost of a performance bug may not be apparent unless tested with large inputs. Since most test suites are designed with the *small input hypothesis* in mind, they are unlikely to reveal these performance problems.

Feedback from developers. The Apache Software Foundation developers have acknowledged the performance bugs in Apache Collections and Apache Ant, and have either fixed the bugs or changed the documentation to warn users of the performance implications for some input values. The Google developers have decided not to fix the bugs due to software maintainability considerations.

Chapter 4

Defining Denial-of-Service Vulnerabilities[1]

4.1 Denial-of-Service Vulnerabilities

Web applications form the backbone of the modern Internet and provide a plethora of useful services, including banking, commerce, education, blogging, and social networking. Since web applications do not require the user to install any software beyond a standard web browser, they are enormously popular. Unfortunately, this growing popularity has also made web applications a desirable target for many kinds of security exploits, including denial-of-service (DoS) attacks.

DoS attacks, which can render websites inaccessible and can cause significant financial damage to website owners, come in two flavors. *Network-based DoS attacks* require an attacker to flood a target server with many requests, thereby saturating server resources and rendering the target web service unavailable. In contrast, *application-level DoS attacks* take advantage of an underlying vulnerability in the web application and either cause the server to crash or exhaust its computational resources. These application-level DoS

[1]O. Olivo, I. Dillig, C. Lin. *Detecting and Exploiting Second Order Denial-of-Service Vulnerabilities in Web Applications.* 22nd ACM Conference on Computer and Communications Security (CCS '15).

attacks are typically more dangerous and cannot be prevented using standard network-level defense mechanisms [130, 4].

In this thesis, we address the issue of application-level DoS attacks that cause excessive CPU usage. In particular, we identify a new type of DoS attack, which we refer to as Second Order DoS attacks, because like recent work on second-order XSS and SQLI vulnerabilities [54], these attacks consist of two phases: In the first step, the attacker uses a bot to pollute the database with junk entries. In the second step, the attacker performs some expensive computation over the junk entries in the database, rendering the application unavailable for a considerable amount of time.

These second-order DoS attacks are made possible by the presence of lurking security vulnerabilities in web applications. Specifically, the first phase of the attack is feasible because the application does not employ an appropriate defense mechanism, such as a *CAPTCHA*, that prevents users from inserting database entries through a bot. Similarly, the second part of the attack is possible because the application does not sanitize the retrieved database entries (e.g., by bounding their size). Furthermore, such DoS attacks cannot be prevented using standard network-based defense mechanisms: Since the inserted database entries are typically small in size and temporally separated, second-order DoS attacks use low-bandwidth and can evade detection by techniques that monitor anomalous network traffic.

4.2 Defining Second Order Denial-of-Service Vulnerabilities

In this section, we first give a precise definition of *second-order DoS vulnerabilities* and then discuss some common mechanisms that programmers employ for avoiding such security holes.

Definition 7. (Second-Order DoS Vulnerability)
Let P be a program using a database table R, and let V_R be a view of R. We say that P contains a second-order DoS vulnerability if:

1. *The quantity $|V_R|$ can be controlled by a bot*
2. *The worst-case resource usage of P is $\Omega(|V_R|)$*

As explained in this definition, there are two necessary conditions that enable a second-order DoS attack. First, the application must allow a bot to control the result size of some query on database table R. Second, the resource usage of the application must be at least linear in the size of this attacker-controlled view of R. If the application satisfies both of these conditions, then an attacker can insert junk entries into R in a way that drives $|V_R|$ to ∞ in the limit. Furthermore, since the worst-case resource usage of the application is proportional to $|V_R|$, the attacker can then craft inputs that trigger this excessive resource usage behavior.

In practice, there are several ways to avoid second-order DoS vulnerabilities in web applications. The most common way to protect against second-order DoS attacks is to perform some sort of Turing test before inserting any user input into the database. Hence, in this context, *CAPTCHA*s and other similar mechanisms (e.g., text message verification) provide a form of sanitization that prevents bots from polluting a database.

However, performing a Turing test is not the only way to defend web applications against second-order DoS attacks. For example, another form of sanitization is to require administrator credentials or to bound the size of a view V_R before performing computation whose resource usage behavior depends on $|V_R|$. For example, consider a web application that iterates over a collection obtained using the following database query:

```
$res = SELECT * FROM Papers WHERE Author=$author
```

Here, if the application disallows the insertion of more than 10 papers by the same author into the database, then the worst-case resource usage of the application will be bound by a constant and will therefore not be susceptible to a second-order DoS attack.

Chapter 5

Detecting Denial-of-Service Vulnerabilities[1]

5.1 Static Analysis

In this section, we describe our algorithm for statically detecting DoS vulnerabilities. As shown schematically in Figure 5.1, our approach consists of two consecutive static taint analyses:

- ***Phase I:*** The goal of the first phase is to identify *tainted database attributes*. We say that an attribute $\mathcal{K}$ of some database table $\mathcal{R}$ is tainted if $|\sigma_{\varphi_{\mathcal{K}}}(\mathcal{R})|$ can be controlled by a bot, where $\varphi_{\mathcal{K}}$ is a condition involving attribute $\mathcal{K}$. Hence, our first static analysis determines which user inputs can reach which attributes of a database table without being sanitized. In this context, a sanitizer is a piece of code that prevents a bot from arbitrarily increasing the size of $\sigma_{\varphi_{\mathcal{K}}}(\mathcal{R})$.
- ***Phase II:*** In the second phase of our analysis, we start with tainted database attributes inferred in Phase I and then perform taint propagation to identify query results whose sizes may be arbitrarily large. Our

[1]O. Olivo, I. Dillig, C. Lin. *Detecting and Exploiting Second Order Denial-of-Service Vulnerabilities in Web Applications.* 22nd ACM Conference on Computer and Communications Security (CCS '15).

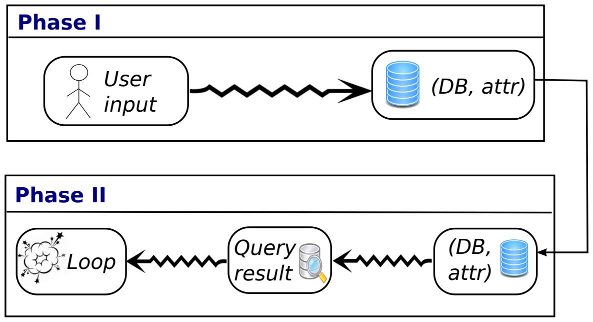

Figure 5.1: Schematic illustration of our approach. Here, squiggly arrows indicate flows between different resources.

$$
\begin{array}{lcl}
\text{Binop } \oplus & \in & \{+, -, ==, !=, <, >, ...\} \\
\text{Expr } e & := & c \mid v \mid e_1 \ \oplus \ e_2 \mid e_1[e_2] \mid \mathit{count}(e) \\
\text{Cond } \Phi & := & \mathcal{K} = v \mid \Phi_1 \text{ AND } \Phi_2 \mid \Phi_1 \text{ OR } \Phi_2 \\
\text{Stmt } S & := & v := e \mid S_1; S_2 \mid f(e_1, \ldots, e_k) \\
 & & \mid \text{if } (e) \text{ then } S_1 \text{ else } S_2 \\
 & & \mid \text{foreach } (v_1 \text{ as } v_2)\ S \\
 & & \mid \text{INSERT } (v_1, ..., v_n) \text{ INTO } \mathcal{R} \\
 & & \mid v := \text{SELECT } (\mathcal{K}_1, ..., \mathcal{K}_n) \\
 & & \qquad \text{FROM } \mathcal{R} \text{ WHERE } \Phi
\end{array}
$$

Figure 5.2: Language used for describing our analysis

analysis then issues a warning if the number of executions of a loop is controlled by such a *tainted query result*.

In the rest of this section, we will use the simplified language of Figure 7.1 to formally describe our static analyses. In particular, we consider a simple PHP-like language that has built-in support for database operations, such as INSERT and SELECT. Expressions in our simplified language include constants (1, "xyz", $\{1, 2, 3\}, \ldots$), variables, and binary operations $e_1 \oplus e_2$

where $\oplus \in \{+, -, *, =, \leq, ...\}$. In addition, we allow array reads $e_1[e_2]$ to model reading from PHP superglobals, such as `$_GET` and `$_POST`. Finally, the expression $count(e)$ yields the size of collection e.

Continuing with the grammar of Figure 7.1, statements include assignments $v := e$, composition $S_1; S_2$, if statements, and loops of the form foreach(v_1 as v_2) S where v_1 is a collection (i.e., array or map), and v_2 is bound to each element v_1. In what follows, we will use function calls $f(e_1, \ldots, e_k)$ to model various kinds of sanitizers. To simplify our presentation, we only consider the two most important database operations, namely INSERT and SELECT[2]. Database insertion operations have the syntax

$$\text{INSERT } (v_1, \ldots, v_k) \text{ INTO } \mathcal{R}$$

where $\mathcal{R}$ is the name of a database table and $(v_1, \ldots, v_k)$ is a tuple to be inserted into $\mathcal{R}$. Selection operations have the syntax:

$$\text{SELECT } (\mathcal{K}_1, \ldots, \mathcal{K}_n) \text{ FROM } \mathcal{R} \text{ WHERE } \Phi$$

Here, each $\mathcal{K}_i$ is the name of an attribute of table $\mathcal{R}$ and Φ is a condition used for filtering relevant tuples in $\mathcal{R}$. Note that condition Φ is a boolean combination of atomic predicates of the form $\mathcal{K} = v$ where $\mathcal{K}$ is the name of an attribute and v is a variable.

[2]Our actual implementation handles most MYSQL commands, not just `INSERT` and `SELECT`.

5.1.1 Phase I Analysis

As mentioned earlier, the first phase of our algorithm performs static taint analysis to identify tainted database attributes. Here, taint sources correspond to user inputs, and sinks are database insertions. Our analysis distinguishes between two classes of sanitizers:

- ***Full sanitizers:*** Such sanitizers protect the application against bots. Examples of full sanitizers include Turing tests (e.g., *CAPTCHAs* or SMS verification) as well as administrative credential checks. We refer to these checks as full sanitizers because they sanitize *every* input received by the application henceforth.

- ***Conditional sanitizers:*** These checks sanitize a particular variable v for some attribute $\mathcal{K}$ of database table $\mathcal{R}$. Specifically, if v is *conditionally sanitized* for $(\mathcal{R}, \mathcal{K})$, this means that the value stored in variable v cannot occur an unbounded number of times in attribute $\mathcal{K}$ of table $\mathcal{R}$. The following PHP function exemplifies such a conditional sanitizer:

```
cond_sanitize(v) {
   $rows = SELECT * FROM Contacts where name = $v;
   if(count($rows) == 0) return true;
   return false;
}
```

  This function returns true iff attribute `name` of table `Contacts` does not already contain value v. Since this function is used to prevent insertion of duplicate names in the `Contacts` table, it sanitizes v for context

($\mathit{Contacts}, \mathit{name}$). These kinds of conditional sanitizers are ubiquitous in real code.

Since our analysis needs to differentiate between full and conditional sanitizers, our taint analysis is somewhat non-standard. We describe our Phase I static analysis in Figure 5.3 using judgments of the form:

$$b, \Gamma \vdash S : b', \Gamma', \Lambda$$

Here, b is a boolean value, referred to as a *bot checker*, indicating whether we have encountered a full sanitizer in the code analyzed so far. Environment Γ, called the *conditional taint environment*, maps each program variable to a propositional formula ϕ and is used to reason about conditional sanitization. Specifically, formula ϕ is used to represent all contexts under which a variable is tainted and is formed according to the following grammar:

$$\phi \;:=\; \mathit{true} \mid \mathcal{R}_{\mathcal{K}} \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2$$

Here, $\mathcal{R}_{\mathcal{K}}$ is a boolean variable representing context $(\mathcal{R}, \mathcal{K})$. Hence, if $\Gamma(v) = \mathcal{R}_{\mathcal{K}}$, this means that v is tainted under context $(\mathcal{R}, \mathcal{K})$. On the other hand, if $\Gamma(v) = \neg\mathcal{R}_{\mathcal{K}} \wedge \neg\mathcal{R}'_{\mathcal{K}'}$, then v is tainted in all contexts except $(\mathcal{R}, \mathcal{K})$ and $(\mathcal{R}', \mathcal{K}')$.

Going back to the judgment $b, \Gamma \vdash S : b', \Gamma', \Lambda$, we use a set Λ to keep track of tainted database attributes $(\mathcal{R}, \mathcal{K})$. Hence, the judgment $b, \Gamma \vdash S : b', \Gamma', \Lambda$ means the following: Assuming we analyze statement S in an environment where b indicates the presence/absence of a full sanitizer and Γ

indicates conditional taint information, the analysis of statement S produces a new bot checker b', a new conditional taint environment Γ' and a set Λ of tainted database attributes. Thus, for a program P, if our analysis produces the judgment

$$\mathit{false}, [\,] \vdash P : b, \Gamma, \Lambda$$

then the set Λ gives us all the tainted database attributes inferred by phase I.

Let us now consider the rules from Figure 5.3 in more detail. According to the first rule labeled *Input*, a call to function *user_input* is a taint source; hence variable v is unconditionally tainted after this statement. The next two rules describe taint propagation for assignments $v := e$ and cause variable v be to become tainted under the same contexts as e.

The next two rules, labeled *Stz 1* and *Stz 2*, describe the analysis of full and conditional sanitizers, respectively. According to *Stz 1*, the analysis of full sanitizers simply sets the bot checker to *true*.[3] The second rule labeled *Stz 2* analyzes conditional sanitizers that bound the number of occurrences of v that can appear in attribute $\mathcal{K}$ of table $\mathcal{R}$. Since variable v is now sanitized for context $(\mathcal{R}, \mathcal{K})$, we only propagate taint for v when $\neg\mathcal{R}_{\mathcal{K}}$ is true. Even though we model conditional sanitizers using the function call $\mathit{cond\_sanitizer}(v, \mathcal{R}, \mathcal{K})$, our analysis automatically infers PHP/SQL expressions that perform such conditional sanitization. By contrast, our analysis requires manual annotations

[3]The careful reader may wonder why we do not analyze full sanitizers by simply setting Γ to be the empty map. While this strategy can be made sound, it would be imprecise because the program may apply full sanitization before asking the user for input.

$$(\mathit{Input}) \quad \frac{\Gamma' = \Gamma[v \mapsto \mathit{true}]}{b, \Gamma \vdash v := \mathit{user\_input}() : b, \Gamma', \emptyset}$$

$$(\mathit{Asn.}\ 1) \quad \frac{e \notin \mathrm{dom}(\Gamma)}{b, \Gamma \vdash v := e : b, \Gamma, \emptyset}$$

$$(\mathit{Asn.}\ 2) \quad \frac{e \in \mathrm{dom}(\Gamma)}{b, \Gamma \vdash v := e : b, \Gamma[v \mapsto \Gamma(e)], \emptyset}$$

$$(\mathit{Stz.}\ 1) \quad \frac{}{b, \Gamma \vdash \mathit{full\_sanitizer}() : \mathit{true}, \Gamma, \emptyset}$$

$$(\mathit{Stz.}\ 2) \quad \frac{\Gamma' = \Gamma[v \mapsto (\Gamma(v) \wedge \neg \mathcal{R}_{\mathcal{K}})]}{b, \Gamma \vdash \mathit{cond\_sanitizer}(v, \mathcal{R}, \mathcal{K}) : b, \Gamma', \emptyset}$$

$$(\mathit{Insert}) \quad \frac{\begin{array}{c} \mathit{Attributes}(\mathcal{R}) = (\mathcal{K}_1, \ldots, \mathcal{K}_n) \\ \Lambda = \{(\mathcal{R}, \mathcal{K}_i) \mid b = \mathit{false} \wedge \Gamma(v_i) \not\Rightarrow \neg \mathcal{R}_{\mathcal{K}_i}\} \end{array}}{b, \Gamma \vdash \text{INSERT } (v_1, ..., v_n) \text{ INTO } \mathcal{R} : b, \Gamma', \Lambda}$$

$$(\mathit{Seq}) \quad \frac{\begin{array}{c} b, \Gamma \vdash S_1 : b_1, \Gamma_1, \Lambda_1 \\ b_1, \Gamma_1 \vdash S_2 : b_2, \Gamma_2, \Lambda_2 \end{array}}{b, \Gamma \vdash S_1; S_2 : b_2, \Gamma_2, \Lambda_1 \cup \Lambda_2}$$

$$(\mathit{If}) \quad \frac{\begin{array}{c} b, \Gamma \vdash S_1 : b_1, \Gamma_1, \Lambda_1 \\ b, \Gamma \vdash S_2 : b_2, \Gamma_2, \Lambda_2 \\ \Gamma' = \Gamma_1 \sqcup \Gamma_2 \\ \Lambda' = \Lambda_1 \cup \Lambda_2 \end{array}}{b, \Gamma \vdash \text{if}(e) \text{ then } S_1 \text{ else } S_2 : \ b_1 \wedge b_2, \Gamma', \Lambda'}$$

Figure 5.3: Phase I static analysis

for full sanitizers (please see Section 5.3).

The rule labeled *Insert* describes the analysis of database insertions, which correspond to taint sinks. To understand this rule, suppose that we are performing an insertion into database table $\mathcal{R}$ which has attributes $\mathcal{K}_1, \ldots, \mathcal{K}_n$. Now, if we have previously performed full sanitization, we know that this insertion is not being performed by a bot. Hence, if b is *true*, then $|\sigma_{\varphi_{\mathcal{K}}}(\mathcal{R})|$ is not controlled by a bot, and $(\mathcal{R}, \mathcal{K})$ should not become tainted. Similarly, if we have performed conditional sanitization to restrict the number of occurrences of value v_i in the $\mathcal{K}_i$'th attribute of $\mathcal{R}$, then $|\sigma_{\varphi_{\mathcal{K}_i}}(\mathcal{R})|$ is bounded; hence, $(\mathcal{R}, \mathcal{K}_i)$ is not tainted. Thus, our analysis considers an insertion to be a sink for $(\mathcal{R}, \mathcal{K}_i)$ only when b is false and $\Gamma(v_i) \not\Rightarrow \neg\mathcal{R}_{\mathcal{K}_i}$ (i.e., it is possible that v_i is tainted under context $(\mathcal{R}, \mathcal{K}_i)$).

The last two rules of Figure 5.3 summarize taint propagation for sequences and if statements. In the *Seq* rule, observe that we take the union of Λ_1 and Λ_2 because a database attribute $(\mathcal{R}, \mathcal{K})$ is tainted if it is either tainted in S_1 *or* S_2.

Finally, let us consider taint propagation for a conditional statement of the form if(e) then S_1 else S_2. Here, we can only be sure that a full sanitization has been performed if it has been performed in both branches; thus, our analysis takes the *conjunction* of b_1 and b_2. On the other hand, since we want to overapproximate tainted database attributes, we propagate the union of Λ_1 and Λ_2. Similarly, since we also want to be conservative about taint information for variables, we compute the join $(\sqcup)$ of Γ_1 and Γ_2, defined as follows:

$$(\Gamma_1 \sqcup \Gamma_2)(v) = \begin{cases} \Gamma_1(v) & \text{if } v \notin \mathrm{dom}(\Gamma_2) \\ \Gamma_2(v) & \text{if } v \notin \mathrm{dom}(\Gamma_1) \\ \Gamma_1(v) \vee \Gamma_2(v) & \text{otherwise} \end{cases}$$

In other words, this join operation ensures that variable v is tainted if it is tainted in either branch.

Observe that the rules from Figure 5.3 omit the analysis of loops and selection operations because (1) our analysis unrolls loops a fixed number of times, and (2) selection operations are effectively no-ops for the first analysis.

5.1.2 Phase II Analysis

The second phase of our algorithm performs a different kind of static taint analysis to infer $\Omega(n)$ computation over *tainted query results (views)*. For the Phase II analysis, taint sources are elements of set Λ (i.e., tainted database attributes) inferred by the Phase I analysis. On the other hand, taint sinks are loops that iterate over collections. Unlike the Phase I analysis where we need to differentiate between two classes of sanitizers, the notion of sanitization for the Phase II analysis is more straightforward: Specifically, we say that a collection v is sanitized if its size is bounded.

The taint propagation rules for Phase II are described in Figure 5.4 using judgments of the form:

$$\Lambda, \Upsilon \vdash S : \Upsilon', \mathcal{E}$$

Here, Λ is the output of the first static analysis (i.e., tainted database attributes), and $\mathcal{E}$ is a boolean variable indicating whether a vulnerability has been encountered. The set Υ is a set of variables that correspond to tainted query results. Essentially, our phase II analysis propagates taint arising from selection operations and issues a warning when the number of loop iterations depends on a tainted variable $v \in \Upsilon$.

Let us now consider the rules from Figure 5.4 in more detail. The first rule for assignments $v := e$ simply propagates taint to v if e is tainted. The second rule analyzes sanitizers that perform some bounds checking on the size of a collection v. In this case, since the size of collection of v is bounded, we simply remove v from the set of tainted variables Υ. While Figure 5.4 models sanitizers using a function call *sanitize(v)*, our implementation automatically infers sanitization expressions that perform bounds checking on the size of collections.

The most interesting aspect of Phase II is the analysis of selection operations. Here, given a set Λ of tainted database attributes, we need to infer whether the result of a selection query is also tainted. Unsurprisingly, this depends on the expression Φ used in the WHERE clause of the query. For this purpose, our analysis utilizes the helper rules shown in Figure 7.2. Given tainted attributes Λ and a database table $\mathcal{R}$, these rules decide whether to propagate taint or not. In particular, given a query whose WHERE clause is Φ, the judgment $\Lambda, \mathcal{R} \vdash \Phi : \mathit{true}$ indicates that taint should be propagated to the result of the query.

The first two rules in Figure 7.2 deal with the base case where Φ is an atomic predicate of the form $\mathcal{K} = v$. In this case, if $(\mathcal{R}, \mathcal{K})$ is not tainted (i.e., $(\mathcal{R}, \mathcal{K}) \notin \Lambda$), then the result of the query cannot be unbounded; hence, $\Lambda, \mathcal{R} \vdash \mathcal{K} = v : \mathit{true}$ if and only if $(\mathcal{R}, \mathcal{K}) \in \Lambda$. If the WHERE clause involves AND, then the result is tainted iff both Φ_1 and Φ_2 are tainted. In contrast, Φ_1 OR Φ_2 yields true iff $\Lambda, \mathcal{R} \vdash \Phi_i : \mathit{true}$ for some $i \in \{1, 2\}$.

Based on this discussion, let us go back to the *Select* rule from Figure 5.4. As expected, the query result v becomes tainted if and only if $\Lambda, \mathcal{R} \vdash \Phi : \mathit{true}$; hence, we only add v to set Υ under this condition. Since the rules *Seq* and *If* are similar to taint propagation from the first phase, we do not describe them in detail.

In Figure 5.4, the last rule for loops describes the analysis of *sinks*. In particular, since the loop iterates over collection v_1, the CPU usage of the program is $\Omega(\mathit{count}(v_1))$.[4] Furthermore, if $v_1 \in \Upsilon$, we know that the size of v_1 may be unbounded, so the analysis issues a warning if $v_1 \in \Upsilon$. Note that this rule also propagates taint for the loop body.

5.2 Attack Vector Generation

So far, we have described a static analysis for identifying second-order DoS vulnerabilities. However, our static analysis has two main limitations: First, it can have false positives. Second, even when the tool reports a true

[4]Unless there is a *break* or *return* statement, in which case the attacker should insert values in the first phase that don't trigger these statements. In our experience, we don't find this to be an obstacle for the attacker.

positive, it can be hard to understand the conditions under which the detected vulnerability can be exploited. Hence, to help programmers understand and confirm the warnings generated by the tool, we have also developed an *attack vector generation engine*. In the context of second-order DoS vulnerabilities, an *attack vector* consists of the following:

- **Component 1:** A set S of tuples inserted into the database, together with other user inputs that are needed to trigger these insertion operations
- **Component 2:** A particular database query that causes the application to perform some expensive computation over S (again, along with other user inputs that are necessary for triggering this behavior)

As shown schematically in Figure 5.6, our approach to automated attack generation is based on backwards symbolic execution and constraint solving. Specifically, given a pair of program locations π_0, π_1 associated with sources and sinks, we perform backwards symbolic execution to collect a set of constraints ϕ describing the conditions under which execution will reach from π_0 to π_1. We then use an SMT solver to find a satisfying assignment σ to ϕ and use σ to generate a candidate attack vector.

5.2.1 Backwards Symbolic Execution

Given source and sink locations π_0, π_1, the goal of our symbolic execution is to generate a constraint ϕ that describes the conditions under which control will reach from π_0 to π_1. This analysis is described in Figure 5.7 using judgments of the following shape:

$$\phi, b, \pi_0, \pi_1 \vdash S : \phi', b'$$

Here, π_0 and π_1 are the source and sink locations respectively and b is a boolean value indicating whether we have reached the source. Given condition ϕ, formula ϕ' represents the constraint under which control will reach π_1 starting from the current statement S. Hence, if our analysis produces the judgment:

$$\mathit{false}, \mathit{false}, \pi_0, \pi_1 \vdash P : \phi, \_$$

then ϕ yields the condition under which π_1 is reachable from π_0 in program P.

Let us now consider the rules from Figure 5.6 in more detail. According to rule 1, if we encounter a sink statement at program point π_1, we know that π_1 is unconditionally reachable from this statement; hence, we propagate *true*. On the other hand, if we encounter a source at program point π_0, we have reached the desired source; hence, we set the boolean variable b to *true* to indicate that we should stop computing weakest preconditions.

Continuing with rule (3) for assignment $v := e$, we use the standard rule for weakest precondition computation by substituting e for v in constraint ϕ. However, if we have already reached the source (i.e, b is true), we do not need to compute weakest preconditions; hence, the generated constraint is $(b \rightarrow \phi) \land (\neg b \rightarrow \phi[e/v])$. Rule (4) for composition $(S_1; S_2)$ also computes weakest preconditions in the standard way.

Rule (5) for conditionals is a bit more involved. First, we compute the weakest precondition of ϕ with respect to the then and else branches. Now, if we have encountered the source π_0 in either branch, we need to stop propagating the weakest precondition; hence the new value of boolean b' is $b_1 \vee b_2$. Assuming we have not yet encountered the desired source, the new precondition is computed as $(e \wedge \phi_1) \vee (\neg e \wedge \phi_2)$. To see why this is correct, consider two cases: If there is sink π_1 is in the then branch, then condition e needs to be true for control to reach π_1; hence, we conjoin e with ϕ_1. On the other hand, if the desired sink is in the else branch, $\neg e$ needs to hold for control to reach π_1. Note that at least one of ϕ_1 and ϕ_2 is guaranteed to be false because the sink location cannot be in both branches simultaneously.

The last rule describes the analysis of `SELECT` statements. This rule is similar to the assignment rule, except that we now substitute v with a more complex term t that involves set comprehensions. In particular, term t is a set that contains all tuples in $\mathcal{R}$ satisfying condition Φ.

5.2.2 Using Constraints to Generate Inputs

We now describe how to use the constraints from Section 5.2.1 to generate attack vectors. As in Section 5.1, the attack vector generation consists of two phases: In the first phase, the input to the analysis is a (user input, database insertion) pair discovered to be a feasible source-sink flow in the Phase I static analysis of Section 5.1.1. Similarly, for the second phase of attack vector generation, the input is a (database selection, loop) pair that is deemed to be a feasible source-sink flow according to the Phase 2 static analysis (Section 5.1.2).

The second phase of attack vector generation is simpler than the first phase because we only need to compute a single input. For this purpose, we get a satisfying assignment σ to the constraint generated using backwards symbolic execution. The input values given by σ, together with the database query for the sink, are now used to generate the second component of the attack vector.

The first phase of attack vector generation is a bit more involved since we need to generate a *sequence* of tuples, rather than a single input. A key challenge here is that the constraint ϕ generated using symbolic execution can refer to database $\mathcal{R}$, which evolves as we insert new tuples into the database. To address this challenge, we employ an iterative algorithm that repeatedly instantiates constraint ϕ with respect to the actual tuples inserted into the database. In particular, given a set variable $\mathcal{R}$ used in constraint ϕ, we first initialize it to the empty set and get a satisfying assignment σ_0 for $\phi[\emptyset/\mathcal{R}]$. This assignment σ_0 is now used to generate a concrete tuple t to insert into the database. In the next iteration, we instantiate $\mathcal{R}$ with the set $\{t\}$ and ask for a new satisfying assignment σ_1 to $\phi[\{t\}/\mathcal{R}]$. This process continues until we have inserted a sufficiently large number of tuples into the database.

Example 11. *We illustrate attack vector generation using a realistic code pattern commonly found in PHP programs:*

```
L1: x := $_POST["a"]; y := $_POST["b"]; z := $_POST["c"];
    if (z == "1") {
       v := SELECT (k1, k2) from R where k2 = y;
       if(count(v)==0)
L2:       INSERT (x, y) INTO R;
    }
```

Our goal is to generate a sequence of tuples that can reach the database insertion at program point `L2` *starting from program location* `L1`. *Using backwards symbolic execution, we first generate the following constraint ϕ:*

$$\begin{gathered}\$\_\texttt{POST}[\text{“c”}] = 1 \wedge \\ count(\{(e.\mathcal{K}_1, e.\mathcal{K}_2) \mid e \in \mathcal{R} \wedge e.\mathcal{K}_2 = \$\_\texttt{POST}[\text{“b”}]\}) = 0\end{gathered}$$

To generate the first tuple, we instantiate $\mathcal{R}$ with $\emptyset$, which yields `$_POST["c"] = "1"`. *Hence, the values of* `$_POST["a"]` *and* `$_POST["b"]` *are unconstrained, but* `$_POST["c"]` *must be* `"1"`*; so suppose we generate the input* `"aa", "bb", "1"`. *In the next iteration, we instantiate $\mathcal{R}$ with* { `("aa", "bb")` } *in ϕ. This new constraint now tells us that, in addition to* `$_POST["c"] = "1"`*, we need* `$_POST["b"]` $\neq$ `"bb"`. *Hence, we now pick the next input to be* `"aa", "bc", "1"`. *In the next iteration, we instantiate $\mathcal{R}$ with the set* { `("aa", "bb"), ("aa", "bc")` }*, which tells us that* `$_POST["c"] = "1"` $\wedge$ `$_POST["b"]` $\neq$ `"bb"` $\wedge$ `$_POST["b"]` $\neq$ `"bc"`. *This process continues until we have inserted sufficiently many tuples into the database.*

5.2.3 Constraint Solving

As described in the previous subsections, the formulas we need to solve for attack vector generation belong to the following constraint language :

$$\begin{array}{lll} \text{Term } t & := & c \mid x \mid f(t) \mid \{x \mid x \in S \wedge \phi\} \\ \text{Formula } \phi & := & \text{true} \mid \text{false} \mid \ t_1 \text{ op } t_2 \\ & & \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \neg\phi \mid \phi_1 \rightarrow \phi_2 \end{array}$$

Here, c is a constant, x is a variable, f is an uninterpreted function, and S is a set. If we exclude *set comprehension terms* of the form $\{x \mid x \in S \wedge \phi\}$ from this constraint language, then our formulas would belong to the standard first order theory of equality with uninterpreted functions (and linear arithmetic) and could be directly solved using off-the-shelf SMT solvers. Unfortunately, to allow the computation of weakest preconditions in the presence of database queries, our analysis also needs to introduce set comprehension terms which are not supported by standard SMT solvers. For this purpose, we employ a customized decision procedure that overapproximates satisfiability by performing a transformation and using off-the-shelf SMT solvers. In particular, our algorithm consists of the following steps:

1. Replace each set comprehension term $t_i \in \phi$ with a fresh variable x_i and generate the formula $\phi' = \phi[x_i/t_i]$ and the mapping $\Gamma : \{x_i \mapsto t_i\}$

2. Let $\Gamma(x_i)$ be $\{x \mid x \in S_i \wedge \phi_i\}$. Now, generate the constraint:

$$\phi'' = (\bigwedge_{x_i \in dom(\Gamma)} (count(x_i) = 0 \rightarrow \bigwedge_{e_{ij} \in S_i} \neg\phi_i[e_{ij}/x]))$$

3. Use an off-the-shelf SMT solver to get a satisfying assignment to $\phi' \wedge \phi''$

It is easy to see that our procedure overapproximates satisfiability: In particular, we construct ϕ' by replacing set comprehension terms with fresh variables. Second, observe that $count(x_i) = 0$ implies that x_i must be the empty set; hence if x_i represents a set comprehension term $\{x \mid x \in S_i \wedge \phi_i\}$, we know that ϕ_i must evaluate to false for *all* elements in set S_i, which is expressed using ϕ''.

Since our procedure overapproximates satisfiability, it is possible that the inputs generated using our technique do not trigger the desired source-sink flow in reality. However, we have not observed this overapproximation to be a problem in practice. Specifically, the overwhelming majority of the set comprehension terms t_i appear in the form $count(t) = 0$; and, under this restriction, it can be shown that $\phi' \wedge \phi''$ is *equisatisfiable* to the original constraint ϕ.

5.3 Implementation

Torpedo consists of approximately 4,000 lines of PHP code. We use Nikic's PHP parser [142] to obtain an initial AST and then construct our own CFG representation for each function. Since an SMT solver (recall Section 5.2) is needed to automatically generate attack vectors, Torpedo uses the Z3 SMT solver [193] and its string solving plug-in [194] for finding satisfying assignments to constraints. For interprocedural analysis, Torpedo conceptually performs function inlining by "gluing together" the CFGs of callees at method invocation sites.

Internally, Torpedo consists of four different modules, namely, (1) static taint analysis, (2) symbolic execution engine, (3) sanitization inference, and (4) an engine for inferring database schemas. Since we have already described the taint analyzer and symbolic execution engine in detail, we now briefly outline the design of modules (3) and (4).

Sanitizer Inference Engine. To infer sanitizers, Torpedo uses a combination of manual annotations and automated static analysis. For the Phase I static analysis from Section 5.1.1, Torpedo requires manual annotations of full sanitizers, since automated inference of Turing tests is beyond the scope of static analysis. Hence, we have manually annotated *CAPTCHA* routines, as well as methods that check for website administrator credentials. However, Torpedo does perform static analysis to infer conditional sanitizers that impose a bound on the size of data structures. Specifically, Torpedo first statically analyzes the source code to generate constraints on collection sizes (for Phase II) and on the number of occurrences of keys in database tables (for Phase I). Torpedo then uses an SMT solver to check if the generated constraints imply an upper bound on the size of some data structure of interest. Hence, a statement S is considered to be a sanitizer if the size of a data structure is unbounded before S but becomes bounded after S.

Database Schema Inference. Recall from Section 5.1 that our static analysis needs to know the attributes for a given database table, so Torpedo performs a pre-analysis that infers the schema for each database table. Specifically, after parsing all files in the application, TORPEDO looks for `CREATE TABLE` instructions in the source code, and it records the ordered set of attributes associated with each table name. Since database queries in PHP can be generated dynamically through the use of string variables, TORPEDO can only over-approximate the set of string values provided to the queries. As we discuss later, this source of imprecision is one of the main causes of false positives.

5.4 Evaluation

We evaluate TORPEDO by applying it to six server-side web applications written in PHP. Specifically, our benchmarks include HotCRP (a widely-used conference management software), WordPress (a popular blogging application), Gallery (a photo management application), SCARF (a research discussion forum), osCommerce (an online store management software), and OpenConf (another conference management system). In total, we use TORPEDO to analyze 483,675 lines of PHP code. We perform our experiments on a MacBook Air laptop with Mac OS X 10.9.3, a 2 GHz Intel Core i7 processor, and 8 GB of RAM.

Table 5.4 summarizes our experimental results, showing that TORPEDO finds a total of 37 vulnerabilities across six applications and only reports 18 false positives. On average, we see that TORPEDO has a 33% false detection rate.

Application	Files	LOC	# TP	# FP
SCARF	19	1686	11	0
OpenConf	130	22,889	7	0
HotCRP	103	48,144	8	11
Gallery	505	62,699	1	0
osCommerce	702	86,693	5	3
Wordpress	479	261,564	5	4
Total	1938	483,675	37	18

Table 5.1: DoS vulnerability detection results. "TP" indicates true positives (i.e., real vulnerabilities), and "FP" denotes false positives.

Application	n=25,000	n=50,000	n=75,000	n=100,000
SCARF	6m32s	22m53s	*TO*	*TO*
OpenConf	2m26s	17m49s	*TO*	*TO*
HotCRP	7m46s	26m33s	*TO*	*TO*
Gallery	1m57s	6m12s	14m8s	29m47s
osCommerce	7m16s	15m35s	33m17s	*TO*
Wordpress	46s	2m11s	8m37s	21m53s

Table 5.2: Impact of DoS attacks for different numbers (n) of entries inserted into the database during the first phase. *TO* means that the application becomes unresponsive for more than one hour. This data only includes a subset of the uncovered vulnerabilities.

Discussion of true positives. For the true vulnerabilities detected by our tool, we use TORPEDO's attack vector generation capability to confirm that the uncovered vulnerability can be exploited to launch a DoS attack. Specifically, we devise a bot to insert a large number of junk entries into a given database and then issue a query that triggers an $\Omega(n)$ computation over these entries. As expected, the severity of the DoS attack is proportional to the number of entries inserted during the first phase. Table 5.2 shows the number of database entries inserted during the first phase against the amount of time the server is unresponsive during the second phase. For example, for a vulnerability in HotCRP, when we insert 75,000 entries into the database in the first phase, we can bring down the server for more than an hour by issuing a *single* query in the second phase.

Upon manual inspection of the true vulnerabilities, we find that the second phase of the attack does not necessarily need to be carried out by a bot. In fact, all of the running times reported in Table 5.2 are caused by a single database query, so automation of the second phase is not a prerequisite for the DoS attack. By contrast, since the uncovered DoS attacks typically involve the insertion of thousands of entries into the database, automation is crucial to the first phase of the attack.

Another interesting aspect of the vulnerabilities is that a few tainted database attributes typically lead to several security vulnerabilities in the same application. In other words, many of the source-sink flows identified by Torpedo's Phase II taint analysis share the same source. For example, the 8 different vulnerabilities found in HotCRP involve two distinct tainted database attributes. This observation suggests that several vulnerabilities within the same application can be prevented by a single fix that blocks the first phase of the attack (e.g., by employing some kind of Turing test).

Finally, we observe that the severity of the uncovered vulnerability is proportional not only to the number of database entries inserted in the first phase of the attack but also to the *kind* of sink encountered in the second phase. In particular, $\Omega(n)$ computations that involve network operations, file I/O or database manipulation typically lead to more serious vulnerabilities. For example, one of the vulnerabilities in osCommerce involves sending emails to all registered users, which can lead to the collapse of the server's network for hours.

Discussion of false positives. We now discuss the root causes of the false positives reported by TORPEDO. Manual inspection of all false alarms reveals that there are only two root causes: (1) incomplete sanitizer inference, and (2) incomplete database query resolution. In particular, all 11 false positives in HotCRP are due to missed detection of sanitizers, and the remaining 7 false alarms in osCommerce and WordPress are caused by imprecise string analysis used for database schema detection. TORPEDO fails to identify some of the sanitizers in HotCRP because the constraints generated for sanitizer inference are overapproximate: Since TORPEDO heuristically drops some interprocedural path conditions for scalability reasons, the SMT solver may decide that the overapproximate constraint is satisfiable even though the exact constraint is in fact unsatisfiable. We note that all false positives in HotCRP can be eliminated using a few simple annotations that explicitly identify sanitizers missed by TORPEDO. Incomplete database query resolution occurs when TORPEDOis unable to identify the string values of the database name and/or attributes in a dynamic database query, and this over-approximation results in false positives. We believe the remaining 8 false positives in osCommerce and Wordpress can be eliminated by employing a more precise string analysis for inferring the tables and attributes of database queries.

Lasting Damage to Applications. While the results in Table 5.2 focus on the damage of a specific second-phase attack, the impact of the first phase can often be much more significant: Once the database has been polluted, the application is often primed for multiple possible second-phase attacks, becoming virtually unusable. For example, imagine a conference submission site whose database has been populated with an enormous number n of spurious papers. This database is unlikely to be useful for the review process because the conference chair would have to be careful to not trigger the high-complexity behavior in the application. Alternatively, the conference chair could try to cleanse the database. In OpenConf, the program chair might try to flag for withdrawal all submissions with no uploaded files. This natural reaction to the attack unfortunately triggers the second phase of the attack. In general, a careful attacker can perform a dictionary attack that makes it difficult to distinguish malicious from benign entries, complicating the task of automatically cleansing the tables without removing legitimate entries.

Feedback from developers.

Since our work, the Wordpress developers have added CAPTCHA mechanisms that prevent the DoS attacks against their application. The developers of osCommerce and HotCRP have decided not to fix the vulnerabilities.

In the remainder of this section, we describe two representative vulnerabilities uncovered by TORPEDO and outline how an attacker can exploit these vulnerabilities to launch a second-order DoS attack.

5.4.1 Exploit in HotCRP

One of the vulnerabilities uncovered by TORPEDO is in the HotCRP conference management application. This vulnerability arises due to an interaction between three HotCRP features, namely account creation, paper registration, and merging of accounts.

To understand the vulnerability and how it can be exploited, we first observe that HotCRP allows a registered user to add an *unrestricted* number of papers to an underlying database called `Paper`. Furthermore, it is possible, although not trivial, to automatically pollute this `Paper` database by ensuring that certain conditions are met (for example, `$_REQUEST`[“p”] is set to “new”, `$_POST`[“submitfinal”] and `$_POST`[“submitpaper”] are both set to “1”, the hidden *formid* value, which is actually leaked from the cookie, is valid, and so on).

Second, let us consider the HotCRP functionality that allows users to merge different accounts, shown in Figure 5.9. This merge operation first retrieves the set S of all papers associated with one account, and then, for each paper in S, it performs an update operation on the `Paper` database to modify the corresponding author information. Thus, when merging an account A with another account B, the amount of work that is performed is directly proportional to the number of papers associated with A's account.

Thus, to launch a DoS attack on HotCRP, an attacker can implement a bot that performs the following steps:

1. It registers a large number m of bogus user accounts.[5]

2. For each user account, it then registers a large number n of papers by providing inputs that pass the paper registration filters.

3. Finally, it merges account i with account $i+1$ for each $i \in [1, m-1]$, again by generating inputs that pass HotCRP's checks that (unsuccessfully) attempt to prevent illicit account merging.

Observe that this attack causes the server to perform work whose complexity is $\Theta(m \times n)$, where m is the number of accounts and n is the number of papers per account. Furthermore, since m and n can both be made arbitrarily large, this attack can feasibly cause HotCRP to become unavailable for significant periods of time. For example, when $m = 5$ and $n = 25,000$, the bot we created was able to bring down HotCRP for more than an hour on our local server.[6]

5.4.2 Exploit in osCommerce

We now describe a vulnerability uncovered by Torpedo in the osCommerce application. Since osCommerce provides infrastructure for online stores,

[5]Even though HotCRP disallows the registration of multiple accounts associated with the same email address, the attacker can still generate arbitrarily many HotCRP accounts because some email services, including Yahoo, do not prevent bots from creating accounts.

[6]For the HotCRP results in Table 5.2, we use $m = 1$.

DoS attacks involving osCommerce directly cost money to businesses that use this application. An interesting aspect of the vulnerability found in osCommerce is that the second phase of the attack is only *indirectly* triggered by the attacker. Thus, the point of this example is to illustrate that second-order DoS attacks can be serious even if the second phase is not under the direct control of the attacker.

The vulnerability in osCommerce arises from an interaction between two features, namely account creation and email subscription. Let us first consider account creation. The key point here is that osCommerce does not employ a mechanism that prevents bots from registering spurious user accounts. As shown in the code snippet of Figure 5.9, osCommerce uses a database relation called `TABLE_CUSTOMERS` that stores information about all registered users. When a user fills out an HTML form to create a new account, the code performs checks to ensure that certain conditions are met, for instance, that the customer's first name and email address are a certain minimum length, that there are no other users with the same email address, etc. However, these checks are not sufficient to rule out the possibility that the web form is being filled out by a bot. In fact, the application does not even check that the user has entered a valid (i.e., existing) email address. As a result, it is fairly straightforward to create a bot that registers many spurious users and pollutes the `TABLE_CUSTOMERS` database.

The next feature relevant to the attack is an email subscription feature that notifies users when certain events occur. For example, a user can subscribe to a *product category*, which notifies the user when a new product is added to that category (e.g., electronics or groceries). Similarly, users can subscribe to individual products so that osComerce can notify them of changes to the inventory (e.g., when a certain product is re-stocked). Since the code implementing this subscription feature does not protect itself against bots, it is possible for an attacker to subscribe a huge number of spurious accounts to all possible categories and products. Hence, each time there is a minor change in the inventory, the application will send an enormous number of email messages to all of the spurious users registered by the attacker.

At first sight, this exploit may seem insignificant because the amount of work performed upon each inventory change is only *linear* in the number of subscribed users. However, when we perform experiments to evaluate the impact of this kind of attack, we find that we can bring down our own server for over 10 minutes by registering only 40,000 users and subscribing them to a product. If the website becomes unavailable for over 10 minutes every time there is a minor change to the inventory, customers are unlikely to enjoy their online shopping experience. Thus, even this innocuous-looking attack makes websites based on osCommerce quite unusable for all practical purposes.

Moreover, we find that after polluting the database, the application's administrator becomes essentially helpless, as many of the application's administrative features become unusable due to their long setup or execution times. The panel for displaying user profiles becomes too slow to use. Sending a general newsletter or new product announcements can take hours or even collapse the network, preventing the legitimate users from receiving the message. Personalized emails to specific users can become almost impossible to send due to the need to navigate a vast amount of junk entries. Of course, the administrator likely has no idea which operations have been affected by the attack, further adding to his despair. Unfortunately, as mentioned earlier, it can be extremely difficult to automatically cleanse the database in the presence of a sophisticated first-phase attack.

$(Assign)$
$$\frac{\Upsilon' = \begin{cases} \Upsilon \cup \{v\} & \text{if } e \in \Upsilon \\ \Upsilon & \text{if } e \notin \Upsilon \end{cases}}{\Lambda, \Upsilon \vdash v := e : \Upsilon', \mathit{false}}$$

$(Sanitize)$
$$\frac{}{\Lambda, \Upsilon \vdash \mathit{sanitize(v)} : \Upsilon \backslash \{v\}, \mathit{false}}$$

$(Select)$
$$\frac{\begin{array}{c}\Lambda, \mathcal{R} \vdash \Phi : c \\ \Upsilon' = \begin{cases} \Upsilon \cup \{v\} & \text{if } c = \mathit{true} \\ \Upsilon & \text{if } c = \mathit{false} \end{cases}\end{array}}{\Lambda, \Upsilon \vdash v := \text{SELECT } (\mathcal{K}_1, ..., \mathcal{K}_n) \text{ FROM } \mathcal{R} \text{ WHERE } \Phi : \Upsilon', \mathit{false}}$$

(Seq)
$$\frac{\begin{array}{c}\Lambda, \Upsilon \vdash S_1 : \Upsilon_1, \mathcal{E}_1 \\ \Lambda, \Upsilon_1 \vdash S_2 : \Upsilon_2, \mathcal{E}_2\end{array}}{\Lambda, \Upsilon \vdash S_1; S_2 : \Upsilon_2, \mathcal{E}_1 \vee \mathcal{E}_2}$$

(If)
$$\frac{\begin{array}{c}\Lambda, \Upsilon \vdash S_1 : \Upsilon_1, \mathcal{E}_1 \\ \Lambda, \Upsilon \vdash S_2 : \Upsilon_2, \mathcal{E}_2\end{array}}{\Lambda, \Upsilon \vdash \text{if}(e) \text{ then } S_1 \text{ else } S_2 : \ \Upsilon_1 \cup \Upsilon_2, \mathcal{E}_1 \vee \mathcal{E}_2}$$

$(Loop)$
$$\frac{\begin{array}{c}\Upsilon' \supseteq \Upsilon \\ \Lambda, \Upsilon' \vdash S : \Upsilon', \mathcal{E}\end{array}}{\Lambda, \Upsilon \vdash \text{foreach}(v_1 \text{ as } v_2)\ S : \ \Upsilon', (v_1 \in \Upsilon) \vee \mathcal{E}}$$

Figure 5.4: Inference rules describing the second phase of static analysis

$$(Base\ 1) \qquad \frac{(\mathcal{R}, \mathcal{K}) \in \Lambda}{\Lambda, \mathcal{R} \vdash \mathcal{K} = v : \mathit{true}}$$

$$(Base\ 2) \qquad \frac{(\mathcal{R}, \mathcal{K}) \notin \Lambda}{\Lambda, \mathcal{R} \vdash \mathcal{K} = v : \mathit{false}}$$

$$(AND) \qquad \frac{\begin{array}{c}\Lambda, \mathcal{R} \vdash \Phi_1 : c_1 \\ \Lambda, \mathcal{R} \vdash \Phi_2 : c_2\end{array}}{\Lambda, \mathcal{R} \vdash \Phi_1 \text{ AND } \Phi_2 : c_1 \wedge c_2}$$

$$(OR) \qquad \frac{\begin{array}{c}\Lambda, \mathcal{R} \vdash \Phi_1 : c_1 \\ \Lambda, \mathcal{R} \vdash \Phi_2 : c_2\end{array}}{\Lambda, \mathcal{R} \vdash \Phi_1 \text{ OR } \Phi_2 : c_1 \vee c_2}$$

Figure 5.5: Helper rules for SELECT

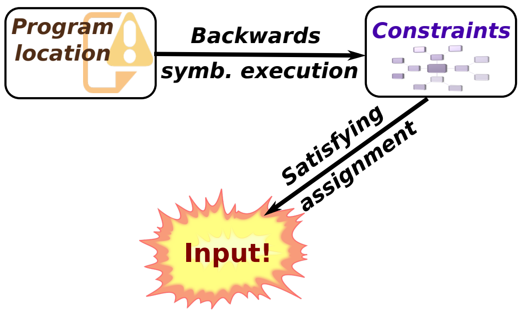

Figure 5.6: Illustration of attack vector generation

$$(1) \qquad \frac{\pi = \pi_1}{\phi, b, \pi_0, \pi_1 \vdash \mathit{sink}_{@\pi} : \mathit{true}, b}$$

$$(2) \qquad \frac{\pi = \pi_0}{\phi, b, \pi_0, \pi_1 \vdash \mathit{source}_{@\pi} : \phi, \mathit{true}}$$

$$(3) \qquad \frac{\phi' = (b \to \phi) \land (\neg b \to \phi[e/v])}{\phi, b, \pi_0, \pi_1 \vdash v := e : \phi', b}$$

$$(4) \qquad \frac{\begin{array}{c}\phi', b', \pi_0, \pi_1 \vdash S_1 : \phi'', b'' \\ \phi, b, \pi_0, \pi_1 \vdash S_2 : \phi', b'\end{array}}{\phi, b, \pi_0, \pi_1 \vdash S_1; S_2 : \phi'', b''}$$

$$(5) \qquad \frac{\begin{array}{c}\phi, b, \pi_0, \pi_1 \vdash S_1 : \phi_1, b_1 \\ \phi, b, \pi_0, \pi_1 \vdash S_2 : \phi_2, b_2 \\ b' = b_1 \lor b_2 \\ \phi' = (e \land \phi_1) \lor (\neg e \land \phi_2)\end{array}}{\phi, b, \pi_0, \pi_1 \vdash \text{if}(e) \text{ then } S_1 \text{ else } S_2 : (b' \to \phi) \land (\neg b' \to \phi'), b'}$$

$$(6) \qquad \frac{\begin{array}{c}t = \{(x.\mathcal{K}_1, \ldots, x.\mathcal{K}_n) \mid x \in \mathcal{R} \land \Phi[x.\mathcal{K}_i/\mathcal{K}_i]\} \\ \phi' = (b \to \phi) \land (\neg b \to \phi[t/x])\end{array}}{\phi, b, \pi_0, \pi_1 \vdash v := \text{SELECT } (\mathcal{K}_1, ..., \mathcal{K}_n) \text{ FROM } \mathcal{R} \text{ WHERE } \Phi : \phi', b}$$

Figure 5.7: Backward symbolic execution rules for generating attack vector constraints. Statements whose preconditions are not met (or are not shown in the figure) are assumed to be no-ops.

```
...
if ($Me->is_empty())
    $Me->escape();
...
if (isset($_REQUEST["merge"]) && check_post()) {
    if (!$_REQUEST["email"])
        ...
    else if (!$_REQUEST["password"])
        ...
    else {
     $MiniMe = Contact::find_by_email($_REQUEST["email"]);
       if (!$MiniMe)
          ...
     else if
       (!$MiniMe->check_password($_REQUEST["password"]))
          ...
        else if ($MiniMe->contactId == $Me->contactId) {
          ...
        } else if (!$MiniMe->contactId || !$Me->contactId)
          ...
        else {
            ...
            $result = $Conf->qe("select paperId,
            authorInformation
            from Paper where authorInformation like '%\t" .
            sqlq_for_like($MiniMe->email) . "\t%'");
            $qs = array();
            while (($row = edb_row($result))) {
             $row[1] = str_ireplace("\t"
                         . $MiniMe->email .
                       "\t", "\t" .
                       $Me->email . "\t", $row[1]);
             $qs[] ="update Paper set authorInformation='"
               .sqlq($row[1]) . "' where paperId=$row[0]";
            }
          ...
```

Figure 5.8: Code snippet showing merging of user accounts functionality in HotCRP.

```
if (isset($HTTP_POST_VARS['action'])
    && ($HTTP_POST_VARS['action'] == 'process')
    && isset($HTTP_POST_VARS['formid'])
 && ($HTTP_POST_VARS['formid'] == $sessiontoken)) {

 ...

  if (strlen($firstname)
    < ENTRY_FIRST_NAME_MIN_LENGTH) {
    $error = true;

    $messageStack->add('create_account',
         ENTRY_FIRST_NAME_ERROR);
  }
  ...
  } else {
    $check_email_query = tep_db_query(
        "select count(*) as total from "
           . TABLE_CUSTOMERS .
          " where customers_email_address = '" .
          tep_db_input($email_address) . "'");
    $check_email = tep_db_fetch_array
         ($check_email_query);
    if ($check_email['total'] > 0) {
      $error = true;
      $messageStack->add('create_account',
           ENTRY_EMAIL_ADDRESS_ERROR_EXISTS);
    }
  }
  ...
  if ($error == false) {
    ...
    tep_db_perform(TABLE_CUSTOMERS, $sql_data_array);

  ...
```

Figure 5.9: Code snippet showing user registration functionality in osCommerce

Chapter 6

Defining Side-Channel Vulnerabilities

6.1 Side-Channel Vulnerabilities

Web applications have become enormously popular due to the ubiquity of the Internet and the existence of rich development frameworks. Hence, in today's Internet-rich world, most people perform their daily activities, including banking and e-commerce, using web applications. Unfortunately, this growing popularity of privacy-sensitive applications has also led to a surge of illegal activities by hackers trying to steal confidential data.

To secure private data, web applications currently rely on a combination of network-level security mechanisms (e.g., encryption and firewalls) and application-level protection techniques, such as credential checks and session handling. While these mechanisms provide some degree of privacy assurance, they do not prevent the application from leaking confidential data through unintended communication channels, known as *side channels*.

Most side-channel leaks in web applications are related to resource usage (e.g., time or space). As an example, consider a health-related web application whose response time depends on whether the user is taking a certain medication. In this case, the server response time can reveal confidential information about ailments of specific users. For instance, recent work [71] has shown that timing and response-size side-channel vulnerabilities can be exploited using *cross-site search attacks*.

This thesis presents an effective static analysis for automatically detecting resource-related side-channel vulnerabilities in web applications. Our approach is pragmatic and aims to detect as many exploitable side channels as possible with a low false positive rate. Towards this goal, we focus on a subset of side-channel vulnerabilities that arise due to *asymptotic differences* in resource usage. To gain some intuition about such *asymptotic side-channel vulnerabilities*, consider an application whose response size is either constant or linear (in the application's input size), *depending on* the value of some secret data. In this case, an attacker can easily observe a substantial difference in response size by supplying a sufficiently large input to the application. In addition to being easy to exploit, asymptotic side-channel vulnerabilities also have the advantage of being amenable to detection using a fairly lightweight static analysis. In particular, our detection algorithm uses a combination of dataflow and taint analyses to automatically uncover serious side-channel vulnerabilities in widely-used PHP applications.

While the techniques we describe in this thesis can be used to detect any resource-related asymptotic side-channel vulnerability, our implementation focuses on two kinds of resources, namely, *time* and *response size*. Specifically, we implement our static analysis in a tool called SCANNER[1], which can be used to find timing and response-size side-channel vulnerabilities in PHP applications. In addition to pinpointing vulnerable components, SCANNER further aids security analysts by identifying confidential database fields that may be leaked due to the detected vulnerability. Furthermore, SCANNER helps users assess the severity of the uncovered vulnerability by semi-automatically generating a Javascript exploit that performs a cross-site search attack.

We evaluate SCANNER on 5 open-source PHP applications and show that it uncovers 10 side-channel vulnerabilities and reports no false positives. Furthermore, the vulnerabilities uncovered by SCANNER are fairly serious and can be used to obtain confidential user information, such as purchase histories, medical records, and bids placed by the user.

6.2 Non-Interference

Since our definition of resource side-channel vulnerabilities is inspired by *non-interference*, we briefly review this security policy and introduce some standard terminology that we use throughout the next two chapters. Non-interference [74] is a security policy that informally states that confidential data may not "interfere with" (i.e., affect) non-confidential data. In particular, non-interference allows a program to manipulate confidential data as long as the observable outputs of the program do not reveal anything about the secret.

[1]SCANNER stands for *Side Channel ANalyzER*

The literature on secure information flow classifies program inputs as being either *"high"* or *"low"*. Specifically, high inputs represent confidential security-sensitive data, whereas low inputs denote non-confidential public data. Using this terminology, non-interference is typically formally defined as follows:

Definition 8. (Non-Interference) *Let H, L denote* ***high*** *and* ***low*** *inputs of a program P, and let $O_P(I)$ denote the observable output of program P on input I. We say that P obeys the non-interference security policy if the following condition is satisfied:*

$$\forall H_1, H_2, L_1, L_2.\;\; L_1 = L_2 \Rightarrow O_P(H_1, L_1) = O_P(H_2, L_2)$$

Intuitively, non-interference states that the program always yields the same observable output on the same low input, regardless of the values of the high inputs.

6.3 Defining Side-Channel Vulnerabilities

In this section, we formally state the class of side-channel vulnerabilities studied in this thesis and justify our decision to focus on this subclass.

We first start by defining *resource side-channel vulnerabilities*, which effectively arise when a program violates non-interference with respect to resource usage:

Definition 9. (Resource Side-Channel Vulnerabilities) *Let H, L denote* ***high*** *and* ***low*** *inputs of a program, respectively, and let $R_P(I)$ denote the re-*

source usage of program P on input I. We say that program P has a ***resource side-channel vulnerability*** *if:*

$$\exists H_1, H_2, L.\ \ R_P(H_1, L) \neq R_P(H_2, L)$$

As in traditional non-interference terminology, *high* variables represent secret values, while low variables denote values that are not security-sensitive. Hence, according to the above definition, a resource side-channel vulnerability arises if it is possible to observe different resource usages when program P is run on the same low input but different high inputs. Hence, an adversary can glean information about the secret simply by observing the program's resource usage.

Since the above definition does not specify the specific kind of resource, it is quite general and can be instantiated in a variety of ways to yield different classes of side-channel vulnerabilities previously discussed in the literature. For instance, if the resource of interest is CPU cycles, then this vulnerability corresponds to a *timing side channel*. On the other hand, if we instantiate R_P with power consumption, then the vulnerability could be exploited to cause a *power monitoring attack*.

The above definition also gives some intuition about a possible way to detect resource side-channel vulnerabilities. In particular, since Definition 9 is a variant of the standard notion of non-interference, we could simply instrument the program with additional *"ghost variables"* to track resource usage and then utilize an existing static analysis to track implicit information flow [18, 17, 134].

However, since non-interference is a very strong condition that is violated by almost any program, we believe that such an approach is not practical. For instance, consider a program that has a *very minor* resource imbalance (e.g., one CPU cycle) across two different execution paths. While such a program has a resource side-channel vulnerability according to Definition 9, it is very unlikely to be exploitable because an attacker cannot reliably observe this minor imbalance in resource usage.

Since our goal is to develop a practical static analysis with a low false positive rate, we instead focus on a subclass of resource side-channel vulnerabilities that can be easily and reliably exploited by attackers. We refer to this subclass as *asymptotic* resource side channels:

Definition 10. (Asymptotic Resource Side-Channel Vulnerabilities) *Let H denote the high inputs of a program, and let $R_P(I)$ denote the resource usage of program P on input I. We say that program P has an* ***asymptotic resource side-channel vulnerability*** *if:*

$$\exists H_1, H_2. \;\; R_P(H_1) \neq \Theta(R_P(H_2))$$

In this definition, observe that the high inputs H_1 and H_2 are still fixed, but the low inputs are unconstrained, so $R_P(H_1)$ and $R_P(H_2)$ are both functions of the low inputs. Hence, the above definition states that it is possible to find a pair of secrets H_1 and H_2 for which the resource usage of P will be asymptotically different with respect to the low-inputs. Since the attacker can control the program's low inputs, he can easily tell whether the secret is H_1 or H_2 by running the program on arbitrarily large values of the low input.

Observe that every asymptotic resource side-channel vulnerability also satisfies Definition 9, but not vice versa. We illustrate the differences between Definitions 9 and 10 using the following two examples.

Example 12. *Consider the following code snippet:*

```
foo(int n) {
   if(secret) {
       for(int i = 0; i < n; i++) {
          consume(1);
       }
   }
   else consume(1);
}
```

Let consume(x) be a statement that consumes x units of resource. If the value of `secret` *is* true, *then the resource usage of* `foo` *is n. On the other hand, if* `secret` *is* false, *then the resource usage is* 1. *Since, $n \neq \Theta(1)$, this program contains an asymptotic side-channel vulnerability according to Definition 10.*

Example 13. *Consider the following code snippet:*

```
bar(int n) {
   if(secret) consume(2);
   else  consume(1);
}
```

Here, function `bar` *contains a vulnerability according to Definition 9 because the resource usage of the program differs depending on whether* `secret` *is true or false. However, this function does not exhibit an* asymptotic *vulnerability because $R_P(H_1)$ and $R_P(H_2)$ only differ by a constant.*

This second example illustrates why we choose to focus on the subclass of vulnerabilities given by Definition 10 as opposed to Definition 9: Since the difference in resource usage is minimal in Example 13, it will be hard to exploit this imbalance in practice due to various kinds of noise in the program's execution environment (e.g., network traffic). Hence, our approach deliberately targets vulnerabilities that can be amplified by the attacker through carefully crafted inputs.

6.4 Assumptions

A key assumption in our work is that the attacker does not have direct access to the database records of other users, and cannot steal another user's credentials. Thus, an attacker needs to resort to a side channel attack to obtain sensitive information.

In order for the side channel vulnerability to be remotely exploitable, a *cross-site request forgery (CSRF)* vulnerability must also be present. In our experience, requests involving search queries do not typically change the state of the application, thus developers overlook the need for deploying *anti-CSRF* mechanisms.

We consider asymptotic resource usage whenever the resource usage depends on string values that can be arbitrarily controlled by an attacker. That is, the string value must not be bounded or converted to an integer before the resource usage region of the code is reached.

In the underlying vulnerability, the attacker must have control over one of the parameters related to a private field in a database query. Additionally, the conditional node that depends on the secret must leak whether there is a record or not with specific values provided to the query. These last two conditions ensure that due to the vulnerability the attacker can use the request as an oracle to try different values for a private field and rely on side channel measurements to determine if the query was a hit or a miss.

Chapter 7

Detecting Side-Channel Vulnerabilities

7.1 Detecting Side-Channel Vulnerabilities

Now that we have defined asymptotic side-channel vulnerabilities, we turn our attention to a static analysis algorithm for detecting these vulnerabilities. As mentioned earlier, our detection philosophy is pragmatic and aims to minimize false positives, rather than targeting theoretical soundness or completeness.

7.1.1 Key Ideas

To understand the key idea underlying our detection algorithm, observe that Definition 10 requires reasoning about the time or space complexity of programs. Unfortunately, it is difficult to statically reason about worst-case resource usage, and existing techniques for reasoning about complexity typically do not scale to large programs [84, 83, 94, 32]. Hence, to analyze realistic web applications with a low false positive rate, we further restrict our attention to the following subclass of asymptotic side-channel vulnerabilities that can be detected using lightweight static analysis:

Definition 11. (Detectable Resource Side-Channel Vulnerabilities) *Let H, L denote the high and low inputs of a program, and let $R_P(I)$ represent resource usage of program P on input I. We say that program P has a* ***detectable resource side-channel vulnerability*** *if:*

$$\exists H_1, H_2.\ \ R_P(H_1) = O(1) \ \wedge \ R_P(H_2) \neq O(1)$$

In other words, we are interested in detecting a subclass of asymptotic side-channel vulnerabilities where the resource usage is constant for some values of the secret but a function of L for other values of the secret. Our detection algorithm targets this specific subclass of vulnerabilities because we can find instances of Definition 11 using standard taint analysis rather than heavy-weight resource usage analysis.

To see how we can reduce side-channel vulnerability detection to taint analysis, consider a program that contains a conditional statement with predicate C and two branches S_1 and S_2. In this case, the following three criteria constitute a sufficient condition for exhibiting the behavior from Definition 11:

1. Predicate C depends on the secret
2. Resource usage in S_1 is a function of low input L
3. Resource usage in S_2 is *not* dependent on L

To see why these three conditions imply Definition 11, observe that condition (1) entails a possible secret-dependent resource usage imbalance between the two branches. Furthermore, since resource usage in S_2 does not

$$\begin{array}{lcl}
\text{Expr } E & := & c \mid v \mid E_1 \ \star \ E_2 \ \ (\star \in \{+, -, \times\}) \\
\text{Cond } C & := & E_1 \circ E_2 \ \ (\circ \in \{<, >, =\}) \\
 & & \mid C_1 \ \wedge \ C_2 \mid C_1 \ \vee \ C_2 \\
\text{Stmt } S & := & \mathit{consume}(E) \mid \mathit{source}(v, \mathit{label}) \\
 & & \mid v := E \mid S_1; S_2 \mid \ C \ ? \ \ S_1 : \ S_2 \\
 & & \mid \text{while } (C) \ \mathit{do} \ S
\end{array}$$

Figure 7.1: Language used for describing our analysis

depend on L according to condition (3), the resource usage must be $O(1)$ in executions where predicate C is true. In contrast, since resource usage in S_1 depends on L according to condition (2), it is not $O(1)$ and can be controlled by the attacker by varying L.

The three conditions stated above also provide some intuition about how vulnerability detection can be reduced to taint analysis. In particular, consider two sources of taint, namely H and L denoting high and low inputs, respectively. The first condition above can simply be restated as predicate C being tainted by H. Now, to understand how conditions (2) and (3) can be reduced to taint analysis, let us consider what it means for resource usage to be a function of L in terms of the program syntax: (i) Either the resource consumption is directly controlled by L (e.g., `malloc(x)` where x is tainted by L), or (ii) the resource consumption occurs in a loop or recursive procedure whose bound is controlled by L. We can check both of these scenarios by tracking expressions that are tainted by L. Hence, the key idea underlying our detection algorithm is to combine two kinds of taint analyses, one that tracks H-tainted variables and another one that tracks L-tainted expressions.

7.1.2 Formalization

With this intuition in mind, we are now ready to explain our static analysis for detecting asymptotic resource side channels. To formally describe our algorithm, we make use of the grammar presented in Figure 7.1, which describes a simplified imperative language with assignments, conditionals, and loops. This language has two non-standard constructs, namely , *consume* and *source* statements. Here, *consume*(E) models the consumption of E units of resource, where E is an integer expression. On the other hand, *source*$(v, \textit{label})$ denotes a taint source of kind *label* where *label* is either $\mathcal{H}$ (for high) or $\mathcal{L}$ (for low). For instance, *source*$(v, \mathcal{H})$ indicates that variable v is assigned to a secret value. In practice, taint sources with label H model database queries that retrieve security-sensitive data (e.g., password). On the other hand, taint sources with label $\mathcal{L}$ represent operations that accept some input from the user.

Our analysis is described using rules of the form

$$\Gamma \vdash S : \Gamma', \Delta, \chi$$

where χ is a boolean value indicating whether or not a vulnerability was encountered during the analysis of S. Environment Γ used in this judgment is a *taint environment* that maps each program variable to a dataflow value η drawn from the lattice shown below:

$$\begin{tikzcd}
 & \top \arrow[dl, dash] \arrow[dr, dash] & \\
\mathcal{H} \arrow[dr, dash] & & \mathcal{L} \arrow[dl, dash] \\
 & \bot &
\end{tikzcd}$$

Here, $\mathcal{H}$ indicates a secret value and $\mathcal{L}$ indicates a (low) input that can be controlled by the attacker. Variables with abstract value $\bot$ are not tainted, and $\top$ indicates unknown taint.

Continuing with the judgment $\Gamma \vdash S : \Gamma', \Delta, \chi$, we use Δ to denote an *abstraction* of the resource usage of statement S. In particular, 0 indicates no resource usage and 1 indicates *constant* (but not necessarily unit) resource usage. In contrast, ∞ indicates that the resource usage cannot be statically bounded (e.g., because it is controlled by the user). We define a $\oplus$ operation on elements of set $\mathcal{R} = \{0, 1, \infty\}$ as follows:

$$\begin{array}{ll} \forall x \in \mathcal{R}. & x \oplus \infty = \infty \\ \forall x \in \mathcal{R}. & x \neq \infty \Rightarrow x \oplus 1 = 1 \\ \forall x \in \mathcal{R}. & x = 0 \Rightarrow x \oplus 0 = 0 \end{array}$$

We also define a total order $\succ$ on set $\mathcal{R}$ as $\infty \succ 1 \succ 0$. Finally, we write $\Delta_1 \gg \Delta_2$ if $\Delta_1 = \infty$ and $\Delta_1 \succ \Delta_2$.

To summarize, the meaning of the judgment $\Gamma \vdash S : \Gamma', \Delta, \chi$ is the following: *"If we execute statement S in an environment that satisfies Γ, then the resource usage of S is given by Δ and the taint environment after S is Γ'".*

We describe our static analysis using the inference rules shown in Figures 7.2 and 7.3. First, the helper rules from Figure 7.2 allow us to determine the taint value η for each expression E and predicate C under taint environment Γ. According to these rules, constants are not tainted ($\bot$), and the taint value for each variable v is given by Γ. For composite expressions of the form $E_1 \otimes E_2$, the taint value is given by the join of the values of E_1 and E_2. For

$$\frac{}{\Gamma \vdash c : \bot}$$

$$\frac{}{\Gamma \vdash v : \Gamma(v)}$$

$$\frac{\begin{array}{c}\Gamma \vdash E_1 : \eta_1 \\ \Gamma \vdash E_2 : \eta_2 \\ \otimes = \star \text{ or } \circ\end{array}}{\Gamma \vdash E_1 \otimes E_2 : \eta_1 \sqcup \eta_2}$$

$$\frac{\begin{array}{c}\Gamma \vdash C_1 : \eta_1 \\ \Gamma \vdash C_2 : \eta_2 \\ \text{op} \in \{\wedge, \vee\}\end{array}}{\Gamma \vdash C_1 \text{ op } C_2 : \eta_1 \sqcup \eta_2}$$

Figure 7.2: Helper rules for determining taint values of expressions E and predicates C

instance, if E_1 has taint $\mathcal{H}$ and E_2 has taint $\bot$, then the taint value if $E_1 \otimes E_2$ is $\mathcal{H} \sqcup \bot = \mathcal{H}$.

Let us now consider the main analysis rules presented in Figure 7.3. Here, rule (1) describes the analysis of a taint source of the form *source*(v, *label*). In this case, the new taint environment Γ' is obtained from Γ by updating the taint value of v to *label*.

Rule (2) describes the analysis of *consume*(E) statements that model resource usage. Recall that resource usage is defined to be ∞ if expression E can be made arbitrarily large by an attacker. Hence, we first use the helper judgments from Figure 7.2 to determine the taint value η of E. If $\eta = \mathcal{L}$, then the resource usage of this statement is ∞ but constant otherwise.

Rule (3) describes taint propagation for assignments of the form $v := E$. As before, we use the helper rules from Figure 7.2 to determine the taint value η of E and update the taint environment by assigning v to η.

Rule (4) shows how we analyze sequence statements $S_1; S_2$. Observe that the resource usage of this statement is obtained by adding the resource usage Δ_1 of S_1 and Δ_2 of S_2 using the $\oplus$ operation defined earlier. Furthermore, $S_1; S_2$ contains a vulnerability if either S_1 or S_2 has a vulnerability; hence we take the disjunction of χ_1 and χ_2.

Rule (5) for conditionals is a bit more involved. Recall that $C\ ?\ S_1 : S_2$ exhibits a side-channel vulnerability if C depends on the secret and S_1 and S_2 have different resource usages. Hence, to determine if there is a vulnerability, we first check the taint value η of C using Figure 7.2. Clearly, if $\eta \not\sqsupseteq \mathcal{H}$, the statement does not exhibit a vulnerability; hence $\chi = \mathit{false}$ under this scenario. On the other hand, if C is secret-dependent (i.e., $\eta \sqsupseteq \mathcal{H}$), then an asymptotic side-channel vulnerability arises if the resource usage Δ_i is ∞ in one branch but constant or zero in the other branch. Hence, χ is *true* if $\Delta_1 \gg \Delta_2$ or $\Delta_2 \gg \Delta_1$, but it is false otherwise.

Continuing with rule (5), let us consider the taint values after analyzing $C\ ?\ S_1 : S_2$ under Γ. If the taint environment after S_i is given by Γ_i, then the value of each variable v after $C\ ?\ S_1 : S_2$ is given by $\Gamma_1(v) \sqcup \Gamma_2(v)$. Hence, the join operation $\Gamma_1 \sqcup \Gamma_2$ on taint environments takes the pairwise join for each variable.

$$(1) \quad \frac{\Gamma' = \Gamma[x \mapsto \mathit{label}]}{\Gamma \vdash \mathit{source}(v, \mathit{label}) : \Gamma', 0, \mathit{false}}$$

$$(2) \quad \frac{\Gamma \vdash E : \eta \qquad \Delta = \begin{cases} \infty & \text{if } \eta \sqsupseteq \mathcal{L} \\ 1 & \mathit{otherwise} \end{cases}}{\Gamma \vdash \mathit{consume}(E) : \Gamma, \Delta, \mathit{false}}$$

$$(3) \quad \frac{\Gamma \vdash E : \eta}{\Gamma \vdash v := E : \Gamma[v \mapsto \eta], 0, \mathit{false}}$$

$$(4) \quad \frac{\Gamma \vdash S_1 : \Gamma_1, \Delta_1, \chi_1 \qquad \Gamma_1 \vdash S_2 : \Gamma_2, \Delta_2, \chi_2}{\Gamma \vdash S_1; S_2 : \Gamma_2, \Delta_1 \oplus \Delta_2, \chi_1 \vee \chi_2}$$

$$(5) \quad \frac{\Gamma \vdash C : \eta \qquad \Gamma \vdash S_1 : \Gamma_1, \Delta_1, \chi_1 \qquad \Gamma \vdash S_2 : \Gamma_2, \Delta_2, \chi_2 \qquad \chi = \begin{cases} (\Delta_i \gg \Delta_j) & \text{if } \eta \sqsupseteq \mathcal{H} \\ \mathit{false} & \mathit{otherwise} \end{cases}}{\Gamma \vdash (C?\ S_1 :\ S_2) : \Gamma_1 \sqcup \Gamma_2, \Delta_1 \oplus \Delta_2, \chi_1 \vee \chi_2 \vee \chi}$$

$$(6) \quad \frac{\Gamma \vdash C : \eta \qquad \Gamma \vdash S : \Gamma, \Delta, \chi \qquad \Delta' = \begin{cases} \infty & \mathit{if}\ \Delta = \infty \vee (\Delta \succ 0 \wedge \eta \sqsupseteq \mathcal{L}) \\ 1 & \mathit{if}\ \Delta = 1\ \wedge \eta \not\sqsupseteq \mathcal{L} \\ 0 & \mathit{otherwise} \end{cases}}{\Gamma \vdash \mathit{while}(C)\ \mathit{do}\ S : \Gamma, \Delta', \chi \vee (\eta \sqsupseteq \mathcal{H} \wedge \Delta' = \infty)}$$

Figure 7.3: Analysis rules for asymptotic side-channel vulnerability detection.

Finally, let us consider the resource usage of the statement $C\ ?\ S_1 : S_2$, where the resource usage of each S_i is given by Δ_i. Since S_1 and S_2 cannot execute at the same time, the resource usage of $C\ ?\ S_1 : S_2$ is $\mathit{max}(\Delta_1, \Delta_2)$, which is in fact the same as $\Delta_1 \oplus \Delta_2$.

The final rule in Figure 7.3 describes the analysis of loops. First, the assumption $\Gamma \vdash S : \Gamma, \Delta, \chi$ at the second line of Rule (6) states that the taint environment Γ is a fixed-point, hence, the taint environment after the loop is also Γ. Now, let us consider the resource usage of the loop *while*(C) *do* S. Clearly, if the resource usage of the body S is ∞ (resp. 0), then the resource usage of the loop is also ∞ (resp. 0). However, if the resource usage of S is constant (i.e., $\Delta = 1$), then the resource usage of the loop depends on the taint value η of predicate C. In particular, if $\eta \sqsupseteq \mathcal{L}$, then the number of loop executions can be controlled by the attacker, causing the resource usage to be statically unbounded. Hence, if $\Delta = 1$, the resource usage Δ' of the loop is ∞ if $\eta \sqsupseteq \mathcal{L}$ but $\Delta' = 1$ otherwise.

The last thing to consider is whether the loop contains a vulnerability. First, observe that the loop may have a vulnerability if the loop continuation condition C is secret-dependent. In particular, if C depends on a secret, the attacker might be able to learn the secret by observing the number of times that the loop executes, which in turn can potentially be inferred from the program's resource usage. To understand whether the loop introduces a vulnerability, observe that *while*(C) *do* S can be rewritten as $C?\ (S; \textit{while}(C)\ \textit{do}\ S) :$ skip. Clearly, the resource usage of the else branch is 0, and since $\Delta' \succeq \Delta$, the resource usage of the then branch is precisely $\Delta \oplus \Delta' = \Delta'$. Thus, the loop introduces a vulnerability if $\eta \sqsupseteq \mathcal{H}$ and $\Delta' = \infty$.

7.2 Implementation

The previous sections have explained the core technical insights underlying our side-channel vulnerability detection approach. However, to implement a tool, several practical issues need to be addressed. For instance, in the previous section, we assumed that taint sources were explicitly annotated using *source* statements, even though PHP applications typically retrieve secret data from a database. As another example, Section 7.1 models resource consumption using explicit *consume* statements, but a practical tool must directly reason about specific kinds of resource usage. In this section, we address these practical concerns by describing the design and implementation of the SCANNER tool in more detail.

7.2.1 SCANNER Basics

SCANNER analyzes PHP applications and detects two specific kinds of side-channel vulnerabilities, namely, *timing* and *response-size* side-channel vulnerabilities. Specifically, these side channels allow an attacker to infer confidential data by measuring server response times or response sizes, respectively. Both kinds of vulnerabilities can be easily exploited in web applications by launching a *cross-site search attack* [71].

Internally, the SCANNER tool consists of three different modules that perform complementary tasks:

- The *detection module* performs a static analysis that flags potential timing and response-size side-channel vulnerabilities. In essence, this module implements an instantiation of the algorithm described in Section 7.1 for two specific resources.

- The *error diagnostic module* performs additional static analysis to report descriptive warnings to the user. In particular, this module identifies the database fields that may be leaked by the detected vulnerabilities.
- The *exploit generation module* performs backwards symbolic execution to semi-automatically generate a Javascript exploit that can be used in a cross-site search attack.

The following three subsections describe each of these modules in greater depth.

7.2.2 Detection Module

Scanner's detection module implements the static analysis algorithm described in Section 7.1. Our implementation considers two kinds of *taint sources*, namely, *user inputs* and *database operations*. User inputs are taint sources with label $\mathcal{L}$ and correspond to reads from pre-defined PHP arrays, such as `GET`, `POST`, and `SESSION`. For instance, the PHP code fragment

```
$v = $_GET('amount')
```

corresponds to a source statement $\mathit{source}(v, \mathcal{L})$ in our formalization from Section 7.1.

In our implementation, taint sources with label $\mathcal{H}$ correspond to database queries that retrieve confidential attributes from a database. Since our tool does not know *a priori* the database columns that store confidential information, we require the user to annotate database fields that are considered to be secret. Armed with such annotations, SCANNER then infers whether or not a given `SELECT` statement is a taint source. As an example, consider the following database query:

```
$result = mysql_query("SELECT firstname,
    lastname, address, age  FROM friends
    WHERE firstname=$fs AND lastname=$ls");
```

Here, we consider variable `result` to be tainted if any attribute `firstname`, `lastname`, `address`, or `age` in table `friends` is annotated to be confidential. Furthermore, since previous work [71] has shown that the most effective way to exploit side-channel vulnerabilities is through *cross-site search* attacks in which the adversary can control the database query, we additionally require that the `WHERE` clause be tainted by the user. Hence, going back to our example, another necessary condition for considering `result` to be tainted is if either `$fs` or `$ls` depends on user input.

In our implementation, *consume* statements from our formalization are instantiated in different ways depending on the type of side-channel vulnerability. For timing vulnerabilities, we consider every instruction to consume one unit of resource; hence, resource usage only becomes ∞ if a loop bound is tainted by user input [1]. For response-size vulnerabilities, resource consumption corresponds to print statements. For instance, the resource usage of a statement `echo $foo` is ∞ if `foo` is tainted by user input, but has unit cost otherwise.

7.2.3 Error Diagnostic Module

The static analysis described so far allows SCANNER to detect the *existence* of a possible side-channel vulnerability. To help the user understand the severity and implications of a possible vulnerability, SCANNER performs an additional static analysis that identifies for the user the information that may be leaked.

Specifically, the error diagnostic module outputs, for each database table, the set of confidential attributes that may be leaked by the application. For instance, if the error diagnostic module outputs `{Age, Address}` for a database table called `Employees`, then the attacker can infer something about the age and address for each employee stored in this table.

To provide such diagnostic information, SCANNER performs a backwards static analysis that utilizes the information produced by the vulnerability detection module. Specifically, the input to the error diagnostic module is the *predicate of a conditional branch* along which there is a resource usage imbalance. Given such a predicate C, SCANNER then collects all secret-tainted

[1]Recall from Section 7.1 that ∞ indicates that the resource usage can be made arbitrarily large by the attacker.

variables used in C and performs backwards symbolic execution to trace each variable v to the database query that caused v to become tainted. Confidential database attributes that are mentioned in the `WHERE` clause of the query are then reported as being potentially leaked.

7.2.4 Exploit Generation Module

To further help programmers understand and assess the detected vulnerability, SCANNER also generates a Javascript program that can be used to launch a cross-site search attack to exploit the vulnerability. We first provide some relevant background on cross-site search attacks, and we then explain how SCANNER semi-automatically generates attack scripts.

Cross-site attacks. Recent work has shown that *cross-site search attacks* can effectively exploit side-channel vulnerabilities in web applications [71]. In this scenario, the attacker first tricks an unsuspecting user into executing a malicious script, for instance, by visiting the attacker's website or clicking on a link in a phishing email message. Now, the malicious script automatically submits a cross-site request with the user's legitimate credentials. Since web browsers allow a script to implement handlers for events triggered by cross-site requests, the malicious script can perform timing measurements between events and send this information back to the attacker[2]. Hence, if the underly-

[2]Observe that response-size side-channels can also be exploited using timing measurements since response parsing times are dependent on the response size.

ing website contains a side-channel vulnerability, then the attacker can glean confidential information about the victim by inflating certain parameters used in a database query.

The goal of SCANNER's exploit generation module is to synthesize Javascript programs that trigger the vulnerable component of the web application. This exploit generation module is only *semi-automatic*: SCANNER automatically infers the URL parameters needed to trigger the vulnerable functionality, but the user must still inflate various search strings to amplify resource usage. In our experience, the exploit generation module is extremely helpful in assessing the severity of the detected vulnerability.

Backwards symbolic execution. To generate a script that exploits the detected vulnerability, SCANNER performs backwards symbolic execution and automatically infers the URL parameters that are necessary for exercising the vulnerable component. To illustrate this process, consider the following code snippet:

```
$page = $_GET["page"];
$product_name = $_GET["product_name"];
if ($page != "1") {
     exit();
}
if (strlen($product_name) < 3 ) {
     exit();
8.    }
9     ...
10.   if(count($x)) {
11.     // do something very expensive!
12.   }
```

Here, assume that the value of variable `$x` is secret-dependent; hence, lines 10-12 suffer from a timing side-channel vulnerability. However, observe that the vulnerable component (i.e., lines 10-12) is only reachable under certain values of the URL parameters. In particular, the URL parameter *page* must be 1, and *product_name* must be at least three characters long.

To automatically infer these parameters, Scanner starts from the vulnerable component C and performs backwards symbolic execution to collect all path constraints that are necessary for the execution to reach C. The output of the symbolic execution engine is an SMT formula ϕ that encodes all possible ways that execution can reach C. Given such a constraint ϕ, Scanner uses an SMT solver (in our case, Z3 [193, 194]) to find a satisfying assignment σ of ϕ. This assignment σ corresponds to concrete values of the URL parameters that are sufficient to trigger the vulnerable functionality.

Going back to our code example, Scanner collects the following path constraint, which must hold for execution to reach the vulnerable code starting at line 10:

Application	Lines of Code	Files	Timing Side-Channel Vulnerabilities	Response Side-Channel Vulnerabilities	False Positives	Analysis Time (min:sec)
OpenClinic	30,849	180	0	2	0	1:09
WeBid	48,753	336	1	1	0	4:01
osCommerce	86,663	702	0	0	0	10:13
OpenCart	156,322	1,014	1	0	0	27:48
ZeusCart	166,400	612	5	0	0	31:06
Total	488,987	5,844	7	3	0	74:17

Table 7.1: Side-channel vulnerability detection results.

$$\begin{gathered}\neg length(\$\_GET[\text{"}product\_name\text{"}]) < 3 \\ \wedge \neg \$\_GET[\text{"}page\text{"}] \neq \text{"}1\text{"}\end{gathered}$$

For this constraint, an SMT solver that reasons about string operations [194] can yield a satisfying assignment, such as $product\_name = \text{"}aaaaa\text{"}$ and $page = \text{"}1\text{"}$. The synthesized Javascript program hard-codes these URL parameters, which can be further tweaked by the user to inflate resource usage.

7.3 Evaluation

To evaluate Scanner, we apply it to five widely used open-source PHP applications, namely, OpenClinic, WeBid, OpenCart, osCommerce and ZeusCart. We choose these applications because they contain security-sensitive information, such as medical records, account balances, and purchase histories. Furthermore, these are mature applications that have been developed with security in mind, so these applications constitute a good test-bed for evaluating our approach.

We run our experiments on a server laptop with Ubuntu 14.04, a dual-core 2 GHz processor, 8 GB of RAM, and the Apache 2.4.18 web server, connected to a campus wireless network. The client is a desktop machine running Ubuntu 14.04, with a dual-core 3 GHz processor, 8 GB of RAM, and the Firefox browser (version 44.0).

Table 7.1 gives statistics about the analyzed programs and summarizes the results of our evaluation. We see that the analyzed programs are quite large, ranging between 30K and 166K lines of code. We also see that Scanner's running time is quite reasonable, with the largest application taking 31 minutes to analyze. Most importantly, we see that Scanner uncovers a total of 10 vulnerabilities. Among these, seven are timing side-channel vulnerabilities, and the rest are response-size side channels.

Table 7.2 provides a more detailed overview of the vulnerabilities uncovered by Scanner. The second column summarizes the information that can be learned by the attacker by performing timing measurements.[3] We see that four of the 5 applications leak various kinds of confidential data, ranging from medical data to bidding histories to purchase data.

The columns labeled "Positive Query" and "Negative Query" in Table 7.2 report response times when the answer to the query is positive and negative, respectively. For instance, one of the vulnerabilities in OpenCart allows the attacker to infer whether a customer X has bought downloadable product Y because the average response time is 265 ms if X has bought Y

[3]Recall that response-size vulnerabilities can also be exploited by performing timing measurements.

Application	Information Inferred by Attacker	Positive Query		Negative Query	
		Avg Time	Std Dev	Avg Time	Std Dev
OpenClinic	Does patient X have a medical record?	1523.71	64.85	2020.32	137.60
OpenClinic	Has patient X been prescribed medication Y?	1655.69	29.84	2358.24	119.24
WeBid	Does user X have product Y on their watchlist?	96.27	10.23	513.35	22.67
WeBid	Is user X bidding on product Y?	2593.63	113.80	1225.34	18.47
OpenCart	Has user X bought downloadable product Y?	265.13	16.21	29.27	4.44
ZeusCart	Has user X bought downloadable product Y?	329.26	14.68	32.61	8.85
ZeusCart	Does user X have an order with amount Y?	4241.29	821.32	630.82	80.96
ZeusCart	Has user X bought something between dates X and Y?	4580.50	121.90	668.72	62.65
ZeusCart	Does user X's order Y have processing status Z ?	4616.64	130.10	676.79	99.84
ZeusCart	Is Y the account status for user X?	3740.37	468.63	798.55	64.92

Table 7.2: Summary of timing results for positive and negative queries. Times are in milliseconds.

but 29 ms otherwise. Since response times vary significantly for each pair of positive and negative queries, we see that the vulnerabilities uncovered by SCANNER can be exploited by attackers. The average and standard deviation metrics correspond to the collection of 100 samples, with URL request sizes of at most 90K characters.

In the next subsections, we describe two representative vulnerabilities that were automatically detected by SCANNER.

7.3.1 Response-size Side-Channel Vulnerability in OpenClinic

Our first example is a response-size side-channel vulnerability found in OpenClinic, which is an open-source medical records system. This vulnerability can be exploited to infer whether a patient has a medical record in the system.

The root cause of the vulnerability is that the response size of the application is asymptotically dependent on the search query provided by the attacker. When there is no record associated with the search string, the application returns a “Result not found” page, which also includes the entire search string associated with the query. On the other hand, if there is a record that matches the query, the response size of the application is constant and does not depend on the search string.

To explain this vulnerability in more detail, Figure 7.4 shows the relevant part of the vulnerable code, which searches for medical records matching a given query string. Specifically, lines 7-13 process the search string provided by the user, and lines 14-16 perform a database query involving this search string. The important point is that the user can set a URL parameter called `logical` (line 9), and if this parameter is set to “OR”, then the database query will return all rows for which the patient name matches *any* of the keywords in the search string. If the table does not contain patients matching the query (lines 17-23), then the application prints `No results found for X`, where X is the search string provided by the user. On the other hand, if there is a single record matching the query, then the application displays the patient’s record. (We omit the case where there are multiple matching entries.)

To see how an attacker can exploit this vulnerability, we show in Figure 7.5 part of the Javascript exploit. Here, the exploit inflates the query string by appending a long suffix (e.g., ”aaaaaaaa...”) after the patient of interest (in this case, Hoare). Note that the URL parameter `logical` is set to “OR”, so the database query will succeed if there is a patient with name Hoare. Since the query string has been intentionally inflated by the attacker, the attacker can tell that a negative answer to the query will be associated with a much longer response time. Thus, it is possible to infer the existence (or lack thereof) of a patient named Hoare by performing a cross-site search attack.

7.3.2 Timing Side-Channel Vulnerability in ZeusCart

Our second example is a timing side-channel vulnerability that SCANNER finds in ZeusCart, an open-source e-commerce system. In addition to the main store functionality, ZeusCart provides an administrator module that can be used by the store owner to query information about customers and their orders.

At a high level, ZeusCart contains a timing side-channel vulnerability in code that allows an administrator to determine the order status (processing, delivered, etc.) for a specific order and customer. Specifically, this component allows a request to search for a customer, order ID, and its order status. If there is no matching purchase, the administrator quickly gets an empty results table. Otherwise, the application performs a few more database queries as well as computations that depend on user input. Hence, the attacker can observe a significant difference in response times when a specific user and order have a specific order status. By leaving the order status field blank, a similar underlying vulnerability can be used to initially infer a valid order ID for a user.

Figure 7.6 shows the relevant ZeusCart code implementing the or-

```
$tab = "medical";
$nav = ``search'';
require_once(``../auth/login_check.php'');
loginCheck(OPEN_PROFILE_DOCTOR);

require_once(``../model/Query/Page/Patient.php'');
require_once(``../lib/Search.php'');

$currentPage = Check::postGetSessionInt('page', 1);
$searchType = Check::postGetSessionInt('search_type');
$logical = Check::postGetSessionString('logical');
$limit = Check::postGetSessionInt('limit');

$searchText = stripslashes(Check::postGetSessionString('search_text'));
$searchText = preg_replace(``/[[:space:]]+/i'', `` ``, $searchText);
$arraySearch = Search::explodeQuoted($searchText);

$patQ = new Query_Page_Patient();
$patQ->setItemsPerPage(OPEN_ITEMS_PER_PAGE);
$patQ->search($searchType, $arraySearch, $currentPage, $logical, $limit);

if ($patQ->getRowCount() == 0)
 {
   $patQ->close();
   FlashMsg::add(sprintf(_(``No results found for '%s'.''), $searchText));
   header("Location: ../medical/patient_search_form.php");
   exit();
 }

if ($patQ->getRowCount() == 1)
{
  $pat = $patQ->fetch();
  $patQ->freeResult();
  $patQ->close();
  header("Location: ../medical/patient_view.php?id_patient="
         . $pat->getIdPatient());
  exit();
}
...
```

Figure 7.4: Response-size vulnerability in OpenClinic.

```
<html>
...
<body> <script>
var video = document.createElement('video');
...
video.onloadstart = function() {
  console.log("Started load at: " + performance.now());
};
video.onerror = function() {
  console.log("Finished processing: " + performance.now());
};
var page = "1";
var search_type="1";
var logical = "OR";
var searchText = "Hoare aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
                  aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
                  aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa...";

video.src = "http://.../openclinic/medical/patient_search.php?"
            + "page=" + page + "&" + "search_type=" + search_type + "&"
            + "logical=" + logical + "&" + "search_text=" + searchText;
</script> </body>
</html>
```

Figure 7.5: Simplified Javascript exploit for OpenClinic's response size side-channel vulnerability

```
if($name!=’’)
{
$condition []= ``  b.user_display_name like ’%’’.$name."%’";
}
if($orderid!=’’)
{
$condition[]= " a.orders_id=’".$orderid."’";
}
...
if($orderstatus!=’’)
{
$condition []= "  a.orders_status=’".$orderstatus.’";
}
...
if(count($condition)>0)
  $sql.= ’ where ’. implode(’ and ’, $condition) .
         ’ order by a.date_purchased desc’ ;
...
else
{
$sql.=’ order by a.date_purchased desc’;
}
...
$obj=new Bin_Query();
if($obj->executeQuery($sql))
{
...
if (empty($condition))
  $sql1 =$sql." LIMIT ".$start.’’,’’.$end;
else
  $sql1 =$sql;

$query=new Bin_Query();
$obj1=new Bin_Query();
$obj1->executeQuery($sql1);
$sql3="select orders_status_id,orders_status_name from orders_status_table";
$obj3=new Bin_Query();
$obj3->executeQuery($sql3);
$query->executeQuery($sql);
}
```
Figure 7.6: Timing side-channel vulnerability in ZeusCart.

der status lookup functionality. The webpage allows the user to supply the `name`, `orderid` and `orderstartus` parameters, which correspond to the username, order ID, and the status of the order, respectively. The code creates a query over the supplied values and calls the `executeQuery` method to look for a match (line 20). If a result is found, the code performs three more database queries (calls to `executeQuery` at lines 28, 31, 32). The `executeQuery` method, omitted for brevity, is defined in the application and sanitizes the input string before calling the pre-defined `mysql_query` method to execute the query. Thus, the computation time of `executeQuery` is linear in the size of the query string.

Since the query string is controlled by the user, the attacker can arbitrarily inflate one of the query parameters to amplify the running time of `executeQuery`. For instance, Figure 7.7 shows an exploit in which the attacker inflates the user name parameter with spaces. Hence, in cases where there is a matching purchase, the attacker can observe a significant increase in running time of the application. Specifically, as shown in Table 7.2, positive queries take 3740ms on average, while negative queries take 799ms.

7.3.3 Feedback from Developers

The OpenClinic developers have acknowledged our reported vulnerabilities, and are working towards fixing them. We are collaborating with the developers of WeBid, OpenCart and ZeusCart to fix the remaining vulnerabilities.

```
<html>
...
<body> <script>
var video = document.createElement('video');
...
video.onloadstart = function() {
  console.log("Started load at: " + performance.now());
};
video.onerror = function() {
  console.log("Finished processing: " + performance.now());
};
var dispname = "Hoare                                                    ...";
var orderid="3";
var selorderstatus = "3";
video.src = "http://.../zeuscart/admin/?do=disporders&" + "dispname=" + dispname
            + "&" + "orderid=" + orderid + "&" + "selorderstatus="
            + selorderstatus;
</script> </body>
</html>
```

Figure 7.7: Simplified Javascript exploit for ZeusCart's timing side-channel vulnerability

Chapter 8

Related Work[1] [2]

8.1 Performance Bugs

Automated Performance Bug Detection. Several recent projects use program analysis to automatically detect performance bugs. Some of these detect wasteful use of temporary objects [66, 159, 185, 183], others focus on inefficient or incorrect usage of collection data structures [158, 184, 187], and some use dynamic profiling to identify expensive computation that can be memoized [138].

The Toddler tool [139] uses dynamic instrumentation to identify "likely redundant" computation by monitoring repetitive and partially-similar memory access patterns. Like Toddler, our work builds on the observation that repetitive traversal of collections likely constitutes a performance bug. Unlike Toddler, our method is purely static, so it incurs no run-time overhead and does not require that the programmer provide representative performance

[1]O. Olivo, I. Dillig, C. Lin. *Static Detection of Asymptotic Performance Bugs in Collection Traversals.* 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15).

[2]O. Olivo, I. Dillig, C. Lin. *Detecting and Exploiting Second Order Denial-of-Service Vulnerabilities in Web Applications.* 22nd ACM Conference on Computer and Communications Security (CCS '15).

tests.

The PerfChecker tool [124] statically analyzes Android applications to identify common performance bugs. Unlike Clarity, PerfChecker detects performance bugs related to GUI lagging, energy leaks, and memory bloat.

The X-ray tool [10] helps users diagnose performance problems related to configuration settings. X-ray uses a technique called *performance summarization*, which couples performance costs with dynamic information flow analysis. Unlike Clarity, X-ray performs dynamic analysis and focuses on performance problems caused by user rather than developer error.

Trace analysis is a technique for identifying root causes of performance anomalies [192, 65]. For example, the TraceAnalyzer tool [65] constructs performance traces that capture the time-varying performance of program runs. Another approach [192] performs impact and causality analysis on traces to discover patterns that are correlated with performance problems. These techniques can shed light on a wide variety of performance anomalies, but they are not fully automated.

Classification and Impact of Performance Bugs. Jin et al. present a comprehensive study of performance bugs and propose a variety of rules to detect and repair likely performance bugs [104]. These rules, which are inspired from existing patches, perform pattern-matching over syntactic program constructs and require domain-specific knowledge about the classes of performance bugs that exist in a given application. A pattern-matching technique is also proposed in the context of databases [40].

Song and Lu use five open-source applications to study the use of statistical debugging for finding performance bugs [161]. They find that two kinds of statistical models involving branch predicates can help pinpoint root causes of performance problems. The idea is to use existing bug reports to gather similar efficient and inefficient computations and compute statistically significant predicates. While quite general, this approach relies on existing bug reports and on user-provided test parameters.

Zaman et al. find that, for Mozilla Firefox and Google Chrome, developers typically spend more time fixing performance bugs than functionality bugs [195].

Loop-Invariant Code Motion. The removal of redundant traversal bugs bears some similarity to loop invariant code motion (LICM), but the problems are quite different. LICM is typically applied to individual assignment statements, and it uses a low-level notion of loop-invariance that is based on reaching definitions. Thus, LICM is not capable of identifying redundant data structure traversals that are detected by our analysis. Of course, LICM also performs the actual optimization, rather than simply detecting the inefficiency.

Techniques for Estimating Computational Complexity. Recent work uses sophisticated static analyses to automatically estimate worst-case resource usage—such as running time—of programs [84, 83, 112, 95], and the Trend-Profiler tool uses profiling and dynamic analysis to estimate *empirical computational complexity* [75]. While these approaches can help programmers debug and understand performance problems, they do not automatically pinpoint them.

Necessary and Sufficient Preconditions. To detect redundant traversal bugs, our algorithm constructs dual over- and under-approximations of the program. Other static analyses that involve negation (or set complement) also make simultaneous use of necessary and sufficient conditions. In particular, the interplay between over- and under-approximations has been explored in path-sensitive static analysis [60], in precise reasoning for unbounded data structures [61, 62], in the construction of method summaries [52], for analysis of confidentiality properties [33], and in typestate analysis [68].

May and Must Alias Analysis. *May alias* analysis [121, 55, 165, 179, 163] underlies almost any compiler optimization, bug detection, and verification technique. While not quite as common as may-alias analysis, *must-alias* analysis is also considered in several papers [9, 102, 135]. Our work simply utilizes may- and must-alias information and does not make contributions in this area.

8.2 Denial-of-Service Attacks

Defending Against Network-Based DoS Attacks Most mechanisms for defending against DoS attacks are deployed at the network level to monitor network behavior, identify anomalous traffic, and set up firewalls that block the attack [72, 3, 169, 140, 131, 176, 114, 46, 108, 190, 175, 155]. Although these techniques are capable of preventing some DoS (and DDoS) attacks, they are limited to a specific underlying model of anomalous traffic and can suffer from scalability problems. Furthermore, they sometimes raise false alarms that prevent legitimate users from accessing the application. In general, distinguishing between DoS attacks and sudden high-volume user traffic (flash crowds) is an open problem [91]. A more detailed survey of network-layer DoS prevention mechanisms can be found elsewhere [130, 4].

More importantly, network-based defense measures are only activated while the attack is in progress, and they are incapable of diagnosing application-specific performance issues that can be exploited by low-bandwidth attacks, including the second-order DoS attacks considered in this thesis.

Defending Against Application-Level DoS Attacks Typical application-level DoS vulnerabilities cause the software to crash or become unresponsive. Particularly dangerous are the cases where a small amount of data sent by an attacker can cause the application to shut down (*ping of death* [29, 98]), enter an infinite loop, or trigger a super-linear recursion call-stack (*inputs of coma* [35, 162]).

Dynamic analyses for the detection of application-level DoS vulnerabilities try to generate inputs that either exhibit worst-case execution times or enter non-terminating loops [26, 25, 86]. Dynamic analyses can be difficult to scale to large programs and cannot always generate inputs that uncover such worst-case behaviors.

Static analyses have been used to identify high-complexity code whose behavior is dependent on user input and can thus be manipulated by an attacker [35, 162]. Such analyses have been able to uncover some simple classes of algorithmic complexity attacks, but they cannot automatically identify more subtle cases of algorithmic complexity vulnerabilities, such as improperly implemented hash functions [53]. In particular, detection of such vulnerabilities requires knowledge about input distributions, which is hard to reason about statically.

Recent work on static analysis has focused on extreme cases of DoS vulnerabilities: detection of super-linear recursion call-stacks [35] and infinite loops [162]. Our work detects DoS vulnerabilities that stem from polynomial loops that can be manipulated by the attacker, instead of the extreme cases of superlinear recursion and infinite loops. More importantly, our work focuses on high resource usage behavior that is triggered by certain kinds of database queries and where the query result is controlled by the attacker.

Other Static Analysis-Based Defenses Second-order vulnerabilities involving SQL injections (SQLI) and cross-site scripting (XSS) have been known for some time, but a static analysis for detecting these attacks has only recently been proposed [54]. Our work is inspired in part by this recent work and shares a similar overall framework that consists of two connected taint analyses. However, since we target DoS vulnerabilities rather than XSS and SQLI, our notions of taint and sanitization are different; thus, the details of the detection algorithms also differ substantially. First, our analysis must detect multiple potential insertions into database tables and analyze their performance impact. Second, our analysis needs to differentiate between full and conditional sanitizers, whereas conditional sanitization is not relevant in the context of XSS and SQLI vulnerabilities. Third, unlike the method of Dahse and Holz, our technique must perform automated inference to identify conditional sanitizers. Finally, another contribution of this thesis over previous work is a novel symbolic execution algorithm for generating candidate attack vectors.

Xie and Aiken [182] implement a symbolic execution algorithm for SQL injections. The idea is to detect unsanitized variables that reach SQL queries using semantic analysis of path constraints. Their symbolic execution engine is similar to ours, but we must solve the additional problem of relating insertions to extractions. In addition, our engine also generates the attack vectors.

Livshits and Lam describe a flow-analysis for detecting XSS and SQL injection vulnerabilities in Java [125]. Wassermann and Su use static analysis to identify XSS vulnerabilities on code using weak sanitization [177]. These approaches tackle a different security vulnerability and do not reason about database interactions.

Automatic Generation of Attack Vectors Kiezun et al. [110] describe a method of generating attack vectors for XSS attacks and SQL injections. Specifically, they use dynamic symbolic execution to generate concrete inputs that trigger execution paths involving sensitive nodes. Sen et al. [156] and Chaudhuri & Foster [38] use similar dynamic symbolic execution methods for Javascript and Ruby-on-Rails respectively. Since our approach is static, it has the potential to report more false positives, but a key advantage is that it does not depend on an input-generation mechanism or a mutation library.

8.3 Side-Channel Attacks

Side channels in web applications. While side-channel leaks have been known for decades, the first thorough study of side-channel leaks in web applications is presented by Chen et al [39]. Specifically, they study common types of side channels in modern web applications and demonstrate that side channels are a serious and realistic source of privacy problems. They also argue that effective mitigation strategies are application-specific and require detecting underlying vulnerabilities in the application.

The only previous tool for automatically detecting side-channel vulnerabilities in web applications is Sidebuster [197], which consists of a combined static and dynamic analysis. In particular, Sidebuster first uses static taint analysis to determine whether network traffic may depend on sensitive data.

If so, Sidebuster proceeds to perform a dynamic analysis on related network operations to quantify the information leak. Our approach differs from Sidebuster in several ways: First, in addition to identifying which operations are tainted by sensitive data, our approach also statically determines asymptotic differences in resource usage. Second, our static detection framework is more general and can be instantiated for any kind of resource. For example, the timing and response-size side-channel vulnerabilities detected by Scanner fall outside the scope of Sidebuster. Finally, the vulnerabilities discovered by Sidebuster can only be exploited by an attacker who is able to sniff network traffic, which requires the attacker to be in the same network path as the victim. In contrast, our approach detects vulnerabilities that can be exploited remotely through cross-site search attacks.

Web timing attacks. Felten and Schneider present one of the first case studies on web timing attacks [67]. In this work, they infer the browsing history of other users by measuring the loading times of external websites. In later work, Brumley and Boneh show how to extract private keys from web servers running OpenSSL using timing attacks [24]. Bortz and Boneh introduce *cross-site timing attacks*, and describe web timing attacks in which they obtain valid usernames as well as items in users' shopping carts [22].

Gelernter and Herzberg introduce *cross-site search attacks* as a mechanism for exploiting side-channel vulnerabilities [71]. Van Goethem et al. show that multimedia tags in HTML5 can be exploited to estimate response sizes during web timing attacks [172]. Our work uses these two known attack techniques to assess the severity of the vulnerabilities uncovered by our approach. However, we emphasize that our work aims to detect application-specific vulnerabilities rather than to describe a new class of web timing attacks.

Non-interference. One possible security policy for preventing side-channel leaks is *non-interference*, which was originally described by Goguen and Meseguer [74]. As mentioned in Chapter 6, non-inference is a very strict security policy which requires that secret data should never affect the values of publicly observable variables.

In the formal methods and programming languages communities, there has been significant research effort on verifying non-interference. The simplest method for proving non-interference is *self-composition*, which sequentially composes two alpha-renamed copies of the same program and encodes the non-interference property as an assertion, which can then be checked by an off-the-shelf verifier [18]. Since self-composition does not work well in practice, subsequent papers try to improve analysis precision by using more sophisticated techniques (e.g., involving *product programs*) [17, 170]. As explained in Section 6.3, we could, in principle, use similar techniques to detect resource side-channel vulnerabilities by first instrumenting the program with ghost variables to track resource usage. However, the resulting approach would yield too many false positives, since small differences in resource usage are unlikely to be observable. In contrast, our approach focuses on asymptotic resource side-channel vulnerabilities and detects them using lightweight static analysis rather than heavy-weight verification techniques that require the inference of precise loop invariants.

There have also been other proposals for statically checking security policies like non-interference. For instance, Jif [134] is a security-typed extension of Java that checks information flow and access control. Other techniques [13, 41, 126] use static analysis to quantify leakage in terms of Shannon's information theory. However, none of these techniques addresses side channels that arise due to asymptotic imbalances in resource usage.

Mitigation of side-channel leaks. To mitigate side-channel leaks , mechanisms such as introducing random delays or fixing server response times have been proposed. Kocher shows how adding random delays can be overcome by an attacker by collecting more timing measurements [113]. Bortz and Boneh describe how making inter-chunk transmission times constant can defend against their timing attacks [22], but cross-site search attacks allow an attacker to inflate computation times, minimizing the effects of this defense. Disallowing cross-site requests is impractical, given their widespread use in the modern web. Anti-CSRF defenses such as requiring the submission of secret tokens or checking the origin header of the request [16] are typically sufficient to prevent unintended state-changing requests, but they have not been widely deployed against cross-site attacks [71]. However, even in the presence of these defenses, the underlying side-channel vulnerabilities can leak private information in the presence of man-in-the-middle attacks.

Chapter 9

Conclusion

In this thesis, we have presented formal definitions for performance-related bugs and security vulnerabilities, and static analyses for detecting them. We have used our tools to uncover 92 performance bugs, 37 denial-of-service vulnerabilities, and 10 side-channel vulnerabilities, in mature, widely used, open source applications with over hundreds of thousands of lines of code.

Our definitions capture the subtle symptoms of problematic performance, beyond the traditional notion of slow running times. Our static analyses compute lightweight approximations of a program's asymptotic complexity, by leveraging traditional techniques from static analysis such as weakest-precondition computation and taint analysis.

Our work opens a new avenue of research in software quality: problematic performance can have different symptoms beyond slow program execution, and as these subtle manifestations of bad performance continue to escape existing techniques and become widespread in software, there will be a growing need to implement rigorous analysis techniques to achieve effective performance understanding.

Appendices

Appendix A

Web Applications in PHP

In this appendix, we provide relevant background on PHP and relational databases.

A.0.1 Background on PHP Scripts

PHP is one of the most popular programming languages for web development, with approximately 82% of server-side scripts implemented using PHP [160]. Specifically, PHP makes it convenient to create *dynamic web pages* that interact with the user and customize page content based on user preferences. There are two key reasons why PHP is so popular for web application development: (1) HTML integration, and (2) native support for relational databases.

HTML integration. One of the most useful features of PHP is the way it allows programmers to handle HTML forms. Specifically, when a user fills out an HTML form, it is possible to invoke a PHP script with the user data stored in so-called *superglobals* that are available in every scope. Two of the most commonly used PHP superglobals are `$_GET` and `$_POST`, both of which are mappings from keys to values (called *arrays* in PHP terminology). If a web form contains an input field named x which is sent using the HTTP POST (resp. GET) method, then `$_POST["x"]` (resp. `$_GET["x"]`) holds the value of the user input for field x. For instance, consider the following HTML form:

```
<form action="example.php" method="get">
Name: <input type="text" name="name">
<input type="submit"> </form>
```

and the corresponding PHP script called "example.php":

```
echo $_GET["name"];
```

Here, when the user enters their name into the `name` field of the above HTML form and clicks submit, a PHP script called "example.php" gets executed. Furthermore, since the web form specifies the data submission method to be HTTP GET, the user input is stored in the superglobal variable `$_GET["name"]`. Thus, the net effect of the above script is to simply print out the name entered into the web form.

Database support. Another attractive feature of PHP is its native support for databases. The most popular database system used with PHP is MySQL, which is an open-source, cross-platform relational database. Certain built-in PHP commands allow scripts to connect to a MySQL database and request content that is typically displayed on a webpage. For the purposes of this paper, the most relevant command is the `mysqli_query` function, which takes as input a string corresponding to the database query. For instance, consider the following PHP code:

```
$name = $_GET["name"];
$query = "SELECT * FROM Customers WHERE Name=$name";
$result = $mysqli->query($query);
```

This code selects from the `Customers` database exactly those people whose name matches the user input. Insertions into the database are performed in a similar way, also using the `mysqli_query` function.

A.0.2 Relational Databases

In relational databases, such as MySQL, data is organized into *tables* (or *relations*) of rows and columns where there is a unique key associated with each row. In this model, each row is a *tuple* representing a single data item, and each column is a labeled *attribute* of the tuple. Given a relation R with set of attributes A and a formula φ such that $\text{vars}(\varphi) \subseteq A$, a selection operation $\sigma_\varphi(R)$ selects all tuples in R that satisfy condition φ. Similarly, given a set A' and a relation R with attributes A such that $A' \subseteq A$, a projection operation $\pi_{A'}(R)$ yields a new relation R' that includes all rows of R but only those columns whose name is in A'. Hence, the following MySQL query:

```
SELECT Author FROM Papers
WHERE title = "X" AND author="Y"
```

corresponds to the operation $\pi_{author}(\sigma_\varphi(papers))$ where φ represents the formula $title = \text{"}X\text{"} \wedge author = \text{"}Y\text{"}$.

In the remainder of this paper, we refer to any set of tuples retrieved from relation R as a *view* of R. Hence, the result of any MySQL query of the form:

```
SELECT ... FROM MyTable WHERE ...
```

corresponds to a view of `MyTable`. Note that every relation R is also a view of itself. Given some view R, we write $|R|$ to denote the number of tuples in R.

Index

Bibliography

[1] Usage statistics and market share of php for websites. `http://w3techs.com/technologies/details/pl-php/all/all`.

[2] *CAV*, volume 3114. Springer, 2004.

[3] S. Abdelsayed, D. Glismsholt, C. Leckie, S. Ryan, and S. Shami. An efficient filter for denial-of-service bandwidth attacks. In *Global Telecommunications Conference (GLOBECOM)*, pages 1353–1357. IEEE, 2003.

[4] M. Abliz. Internet denial of service attacks and defense mechanisms. In *Technical Report No. TR-11-178*, pages 1–50. University of Pittsburgh, 2011.

[5] Johan Agat. Transforming out timing leaks. In *Principles of Programming Languages*, pages 40–53. ACM, 2000.

[6] A. Aiken, S. Bugrara, I. Dillig, T. Dillig, B. Hackett, and P. Hawkins. An overview of the Saturn Project. In *PASTE*, pages 43–48, 2007.

[7] Xavier Allamigeon. Non-disjunctive numerical domain for array predicate abstraction. In *ESOP*, pages 163–177, 2008.

[8] Erik Altman, Matthew Arnold, Stephen Fink, and Nick Mitchell. Perfor-

mance analysis of idle programs. In *ACM SIGPLAN Notices*, volume 45, pages 739–753. ACM, 2010.

[9] Rita Z Altucher and William Landi. An extended form of must alias analysis for dynamic allocation. In *PLDI*, pages 74–84. ACM, 1995.

[10] Mona Attariyan, Michael Chow, and Jason Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *OSDI*, pages 307–320, 2012.

[11] A. Avizienis, J-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. In *Transactions on Dependable and Secure Computing*, volume 1, pages 11–33. IEEE, 2004.

[12] Michael Backes and Boris Köpf. Formally bounding the side-channel leakage in unknownmessage attacks. In *ESORICS '08*. Springer, 517–532.

[13] Michael Backes, Boris Köpf, and Andrey Rybalchenko. Automatic discovery and quantification of information leaks. In *Symposium on Security and Privacy*, pages 141–153. IEEE Computer Society, 2009.

[14] T. Ball and S.K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL*, pages 1–3, 2002.

[15] Thomas Ball and James R Larus. Efficient path profiling. In *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 46–57. IEEE Computer Society, 1996.

[16] Adam Barth, Collin Jackson, and John C. Mitchell. Robust defenses for cross-site request forgery. In *Computer and Communications Security*, pages 75–88. ACM, 2008.

[17] Gilles Barthe, Juan Manuel Crespo, and César Kunz. Relational verification using product programs. In *FM 2011: Formal Methods*, pages 200–214. Springer, 2011.

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

[19] P. Bisht, T. Hinricks, N. Skrupsky, and V. N. Venkatakrishnan. Waptec: Whitebox analysis of web applications for parameter tampering exploit construction. In *18th Conference on Computer and Communications Security (CCS)*, pages 575–586. ACM, 2011.

[20] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software, 2002.

[21] I. Bogudlov, T. Lev-Ami, T. Reps, and M. Sagiv. Revamping TVLA: Making parametric shape analysis competitive. *Lecture Notes in Computer Science*, 4590:221, 2007.

[22] Andrew Bortz and Dan Boneh. Exposing private information by timing web applications. In *World Wide Web*, pages 621–628. ACM, 2007.

[23] A.R. Bradley, Z. Manna, and H.B. Sipma. What's Decidable About Arrays? *Lecture notes in computer science*, 3855:427, 2006.

[24] David Brumley and Dan Boneh. Remote timing attacks are practical. In *USENIX Security Symposium*. USENIX Association, 2003.

[25] J. Burnim, N. Jalbert, C. Stergiou, and K. Sen. Looper: Lightweight detection of infinite loops at runtime. In *ASE*, 2009.

[26] J. Burnim, S. Juvekar, and K. Sen. Wise: Automated test generation for worst-case complexity. In *ICSE*, 2009.

[27] W.R. Bush, J.D. Pincus, and D.J. Sielaff. A static analyzer for finding dynamic programming errors. *Software: Practice and Experience*, 30(7):775–802, 2000.

[28] Cristian Cadar, Daniel Dunbar, and Dawson R Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[29] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, pages 322–335, New York, NY, USA, 2006. ACM.

[30] C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. *POPL*, pages 289–300, 2009.

[31] The Official CAPTCHA. `http://www.captcha.net/`.

[32] Quentin Carbonneaux, Jan Hoffmann, and Zhong Shao. Compositional certified resource bounds. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 467–478. ACM, 2015.

[33] Pavol Černỳ and Rajeev Alur. Automated analysis of Java methods for confidentiality. In *CAV*, pages 173–187. Springer, 2009.

[34] S. Chandra and T. Reps. Physical type checking for c. *SIGSOFT*, 24(5):66–75, 1999.

[35] R. Chang, G. Jiang, F. Ivanĉić, S. Sankaranarayanan, and V. Shmatikov. Inputs of coma: Static detection of denial-of-service vulnerabilities. In *22nd Computer Security Foundations Symposium (CSF)*, pages 186–199. IEEE, July 2009.

[36] D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *PLDI*, pages 296–310, NY, USA, 1990. ACM.

[37] R. Chatterjee, B.G. Ryder, and W.A. Landi. Relevant context inference. In *POPL*, pages 133–146. ACM, 1999.

[38] Avik Chaudhuri and Jeffrey Foster. Symbolic security analysis of ruby-on-rails web applications. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS'10)*, pages 585–594. ACM, 2010.

[39] Shuo Chen, Rui Wang, XiaoFeng Wang, and Kehuan Zhang. Side-channel leaks in web applications: A reality today, a challenge tomorrow. In *Symposium on Security and Privacy*, pages 191–206. IEEE Computer Society, 2010.

[40] T.H. Chen, W. Shang, Z.M. Jiang, A.E. Hassan, M. Nasser, and P. Flora. Detecting performance anti-patterns for applications developed using object-relational mapping. In *ICSE*, pages 1013–1024. ACM, 2014.

[41] David Clark, Sebastian Hunt, and Pasquale Malacaria. A static analysis for quantifying information flow in a simple imperative language. *J. Comput. Secur.*, 15(3):321–371, 2007.

[42] Malcolm Clark. Post congress tristesse. In *TeX90 Conference Proceedings*, pages 84–89. TeX Users Group, March 1991.

[43] Michael R. Clarkson, Andrew C. Myers, and Fred B. Schneider. Quantifying information flow with beliefs. *J. Comput. Secur.*, 17(5):655–701, 2009.

[44] M. Colón and H. Sipma. Synthesis of linear ranking functions. In *TACAS*, 2001.

[45] William R. Cook. Safe query objects: statically typed objects as remotely executable queries. In *In Proceedings of the 27th International Conference on Software Engineering (ICSE*, pages 97–106. ACM Press, 2005.

[46] SYN cookies. `http://cr.yp.to/syncookies.html`.

[47] D.C. Cooper. Theorem proving in arithmetic without multiplication. *Machine Intelligence*, 7:91–100, 1972.

[48] Emilio Coppa, Camil Demetrescu, and Irene Finocchi. Input-sensitive profiling. In *ACM SIGPLAN Notices*, volume 47, pages 89–98. ACM, 2012.

[49] P. Cousot. Verification by abstract interpretation. *Lecture notes in computer science*, pages 243–268, 2003.

[50] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, pages 84–96, NY, USA, 1978. ACM.

[51] Patrick Cousot and Radhia Cousot. Modular static program analysis. In *CC*, pages 159–178, 2002.

[52] Patrick Cousot, Radhia Cousot, Manuel Fähndrich, and Francesco Logozzo. Automatic inference of necessary preconditions. In *Verification, Model Checking, and Abstract Interpretation*, pages 128–148. Springer, 2013.

[53] S. Crosby and D. Wallach. Denial of service via algorithmic complexity attacks. In *USENIX Security*, 2003.

[54] D. Dahse and T. Holz. Static detection of second-order vulnerabilities in web applications. In *23rd USENIX Security Symposium*, pages 989–1003, August 2014.

[55] A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *PLDI*, pages 230–241. ACM, 1994.

[56] I. Dillig, T. Dillig, and A. Aiken. Sound, complete and scalable path-sensitive analysis. In *PLDI*, pages 270–280, 2008.

[57] I. Dillig, T. Dillig, and A. Aiken. Cuts from proofs: A complete and practical technique for solving linear inequalities over integers. In *In CAV*. Springer, 2009.

[58] I. Dillig, T. Dillig, and A. Aiken. Fluid updates: Beyond strong vs. weak updates. In *ESOP*, 2010.

[59] I. Dillig, T. Dillig, and A. Aiken. Small Formulas for Large Programs: On-line Constraint Simplification in Scalable Static Analysis. In *SAS*, 2010.

[60] Isil Dillig, Thomas Dillig, and Alex Aiken. Sound, complete and scalable path-sensitive analysis. In *PLDI*, volume 43, pages 270–280, 2008.

[61] Isil Dillig, Thomas Dillig, and Alex Aiken. Fluid updates: Beyond strong vs. weak updates. In *ESOP*, pages 246–266. 2010.

[62] Isil Dillig, Thomas Dillig, and Alex Aiken. Precise reasoning for programs using containers. In *ACM SIGPLAN Notices*, volume 46, pages 187–200. ACM, 2011.

[63] Isil Dillig, Thomas Dillig, Alex Aiken, and Mooly Sagiv. Precise and compact modular procedure summaries for heap manipulating programs. In *PLDI*, pages 567–577. ACM, 2011.

[64] D. Distefano, P.W. O Hearn, and H. Yang. A local shape analysis based on separation logic. *Lecture Notes in Comp. Sc.*, 3920:287, 2006.

[65] Amer Diwan, Matthias Hauswirth, Todd Mytkowicz, and Peter F Sweeney. Traceanalyzer: a system for processing performance traces. *Software: Practice and Experience*, 41(3):267–282, 2011.

[66] Bruno Dufour, Barbara G Ryder, and Gary Sevitsky. A scalable technique for characterizing the usage of temporaries in framework-intensive Java applications. In *FSE*, pages 59–70, 2008.

[67] Edward W. Felten and Michael A. Schneider. Timing attacks on web privacy. In *Computer and Communications Security*, pages 25–32. ACM, 2000.

[68] Stephen J Fink, Eran Yahav, Nurit Dor, G Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of alias-

ing. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 17(2):9, 2008.

[69] C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *POPL*, pages 191–202. ACM, 2002.

[70] Y. Ge and L. de Moura. Complete instantiation for quantified formulas in Satisfiabiliby Modulo Theories. In *CAV*, page 320. Springer, 2009.

[71] Nethanel Gelernter and Amir Herzberg. Cross-site search attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1394–1405. ACM, 2015.

[72] Thomer M. Gil and Massimiliano Poletto. Multops: A data-structure for bandwidth attack detection. In *Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10*, SSYM'01, pages 3–3, 2001.

[73] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *ACM SIGPLAN Notices*, volume 40, pages 213–223. ACM, 2005.

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

[75] Simon F Goldsmith, Alex S Aiken, and Daniel S Wilkerson. Measuring empirical computational complexity. In *FSE*, pages 395–404, 2007.

[76] D. Gopan, T. Reps, and M. Sagiv. A framework for numeric analysis of array operations. In *POPL*, pages 338–350, NY, USA, 2005. ACM.

[77] B. Gulavani, S. Chakraborty, G. Ramalingam, and A. Nori. Bottom-up shape analysis. *SAS*, pages 188–204, 2009.

[78] B. Gulavani and S. Gulwani. A numerical abstract domain based on expression abstraction and max operator with application in timing analysis. In *CAV*, 2008.

[79] S. Gulwani, B. McCloskey, and A. Tiwari. Lifting abstract interpreters to quantified logical domains. In *POPL*, pages 235–246. ACM New York, NY, USA, 2008.

[80] S. Gulwani, K.K. Mehra, and T. Chilimbi. SPEED: precise and efficient static estimation of program computational complexity. In *POPL*, pages 127–139, 2009.

[81] S. Gulwani and M. Musuvathi. Cover algorithms. In *ESOP*, pages 193–207, 2008.

[82] S. Gulwani and A. Tiwari. Computing procedure summaries for interprocedural analysis. *ESOP*, pages 253–267, 2007.

[83] Sumit Gulwani. Speed: Symbolic complexity bound analysis. In *Computer Aided Verification*, pages 51–62. Springer, 2009.

[84] Sumit Gulwani, Krishna K Mehra, and Trishul Chilimbi. Speed: precise and efficient static estimation of program computational complexity. In *ACM SIGPLAN Notices*, volume 44, pages 127–139. ACM, 2009.

[85] Sumit Gulwani, Ivan Radiček, and Florian Zuleger. Feedback generation for performance problems in introductory programming assignments. *FSE*, 2014.

[86] A. Gupta, T. Henzinger, R. Majumdar, A. Rybalchenko, and R. Xu. Proving non-termination. In *POPL*, 2008.

[87] P. Habermehl, R. Iosif, and T. Vojnar. What else is decidable about integer arrays? *Lecture Notes in Computer Science*, 4962:474, 2008.

[88] N. Halbwachs and M. Péron. Discovering properties about arrays in simple programs. In *PLDI*, pages 339–348, NY, USA, 2008. ACM.

[89] W. G. J. Halfond, A. Orso, and P. Manolios. Wasp: Protecting web applications using positive tainting and syntax-aware evaluation. In *Transactions on Software Engineering (TSE)*, volume 34(1), pages 65–81. IEEE, 2008.

[90] Shi Han, Yingnong Dang, Song Ge, Dongmei Zhang, and Tao Xie. Performance debugging in the large via mining millions of stack traces. In *Proceedings of the 2012 International Conference on Software Engineering*, pages 145–155. IEEE Press, 2012.

[91] M. Handley, E. Rescorla, and IAB. Internet denial-of-service considerations. In *RFC 4732*, 2006.

[92] Matthias Hauswirth, Amer Diwan, Peter F Sweeney, and Michael C Mozer. Automating vertical profiling. In *ACM SIGPLAN Notices*, volume 40, pages 281–296. ACM, 2005.

[93] Thomas A Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Software verification with BLAST. In *Model Checking Software*, pages 235–239. Springer, 2003.

[94] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Multivariate amortized resource analysis. In *ACM SIGPLAN Notices*, volume 46, pages 357–370. ACM, 2011.

[95] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Multivariate amortized resource analysis. In *ACM SIGPLAN Notices*, volume 46, pages 357–370. ACM, 2011.

[96] Pieter Hooimeijer, Prateek Saxena, Benjamin Livshits, Margus Veanes, and David Molnar. Fast and precise sanitizer analysis with bek. In *In 20th USENIX Security Symposium*, 2011.

[97] http://gcc.gnu.org/. Gcc 4.3.0.

[98] http://insecure.org/sploits/ping-o death.html. Ping of death.

[99] http://www.gnu.org/software/coreutils/. Unix coreutils.

[100] http://www.openssh.com/. Openssh 5.3p1.

[101] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th International Conference on World Wide Web*, WWW '04, pages 40–52, New York, NY, USA, 2004. ACM.

[102] Suresh Jagannathan, Peter Thiemann, Stephen Weeks, and Andrew Wright. Single and loving it: Must-alias analysis for higher-order languages. In *PLDI*, pages 329–341. ACM, 1998.

[103] R. Jhala and K. L. Mcmillan. Array abstractions from proofs. In *CAV*, 2007.

[104] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and detecting real-world performance bugs. *ACM SIGPLAN Notices*, 47(6):77–88, 2012.

[105] N.D. Jones and S.S. Muchnick. Flow analysis and optimization of LISP-like structures. In *POPL*, pages 244–256. ACM NY, USA, 1979.

[106] Milan Jovic, Andrea Adamoli, and Matthias Hauswirth. Catch me if you can: performance bug detection in the wild. In *ACM SIGPLAN Notices*, volume 46, pages 155–170. ACM, 2011.

[107] Vineet Kahlon. Bootstrapping: a technique for scalable flow and context-sensitive pointer alias analysis. In *ACM SIGPLAN Notices*, volume 43, pages 249–259. ACM, 2008.

[108] S. Kandula, D. Katabi, M. Jacob, and A. Berger. Botz-4-sale: Surviving organized DDoS attacks that mimic flash crowds. In *NSDI*, 2005.

[109] M. Karr. Affine relationships among variables of a program. *A.I.*, pages 133–151, 1976.

[110] Adam Kieżun, Philip J. Guo, Karthick Jayaraman, and Michael D. Ernst. Automatic creation of SQL injection and cross-site scripting attacks. In *31st International Conference on Software Engineering (ICSE)*, pages 199–209, May 2009.

[111] Charles Killian, Karthik Nagaraj, Salman Pervez, Ryan Braud, James W Anderson, and Ranjit Jhala. Finding latent performance bugs in systems implementations. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 17–26. ACM, 2010.

[112] Jens Knoop, Laura Kovács, and Jakob Zwirchmayr. Symbolic loop bound computation for WCET analysis. In *Perspectives of Systems Informatics*, pages 227–242. Springer, 2012.

[113] Paul C. Kocher. Timing attacks on implementations of diffie-hellman,

rsa, dss, and other systems. In *Advances in Cryptology*, pages 104–113. Springer-Verlag, 1996.

[114] R.R. Kompella, S. Singh, and G. Varghese. On scalable attack detection in the network. In *Proceedings of Internet Measurement Conference (SIGCOMM)*. ACM, 2004.

[115] Boris Köpf and David Basin. An information-theoretic model for adaptive side-channel attacks. In *Conference on Computer and Communications Security*, pages 286–296. ACM, 2007.

[116] L. Kovacs and A. Voronkov. Finding loop invariants for programs over arrays using a theorem prover. In *FASE 2009*, pages 470–485. Springer, 2009.

[117] D. Kroening and O. Strichman. *Decision procedures: an algorithmic point of view.* Springer-Verlag New York Inc, 2008.

[118] Marc Kührer, Thomas Hupperich, Christian Rossow, and Thorsten Holz. Exit from hell? reducing the impact of amplification DDoS attacks. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 111–125, San Diego, CA, August 2014. USENIX Association.

[119] S.K. Lahiri and S. Qadeer. Verifying properties of well-founded linked lists. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 115–126, 2006.

[120] W. Landi and B. G. Ryder. A safe approximate algorithm for interprocedural aliasing. *SIGPLAN Not.*, 27(7):235–248, 1992.

[121] William Landi and Barbara G Ryder. A safe approximate algorithm for interprocedural aliasing. *ACM SIGPLAN Notices*, 27(7):235–248, 1992.

[122] S. Lee and D. Cho. Packet-scheduling algorithm based on priority of separate buffers for unicast and multicast services. *Electronics Letters*, 39(2):259–260, Jan 2003.

[123] Tongping Liu and Emery D Berger. Sheriff: precise detection and automatic mitigation of false sharing. In *ACM SIGPLAN Notices*, volume 46, pages 3–18. ACM, 2011.

[124] Y. Liu, C. Xu, and S.C. Cheung. Characterizing and detecting performance bugs for smartphone applications. In *ICSE*, pages 1013–1024, 2014.

[125] V. Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in java applications with static analysis. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, 2005.

[126] Pasquale Malacaria. Assessing security threats of looping constructs. In *Principles of Programming Languages*, pages 225–235. ACM, 2007.

[127] Darko Marinov and Sarfraz Khurshid. TestEra: a novel framework for automated testing of Java programs. In *16th IEEE Conference on Automated Software Engineering*, page 22, 2001.

[128] J. Mccarthy. Towards a mathematical science of computation. In *IFIP*, 1962.

[129] Yasuhiko Minamide. Static approximation of dynamically generated web pages. In *Proceedings of the 14th International Conference on World Wide Web*, WWW '05, pages 432–441, New York, NY, USA, 2005. ACM.

[130] J. Mirkovic, S. Dietrich, D. Dittrich, and P. Reiher. *Internet of Service: Attack and Defense Mechanisms*. Prentice Hall, 2005.

[131] J. Mirkovic, G. Prier, and P. Reiher. Attacking DDoS at the source. In *ICNP*, pages 312–321, 2002.

[132] R. Monavich. *Partially Disjunctive Shape Analysis*. PhD thesis, Tel Aviv University, 2009.

[133] Maliheh Monshizadeh, Prasad Naldurg, and V. N. Venkatakrishnan. Mace: Detecting privilege escalation vulnerabilities in web applications. In *Conference on Computer and Communications Security*, pages 690–701, 2014.

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

[135] Mayur Naik and Alex Aiken. Conditional must not aliasing for static race detection. *ACM SIGPLAN Notices*, 42(1):327–338, 2007.

[136] Mayur Naik, Alex Aiken, and John Whaley. *Effective static race detection for Java*, volume 41. ACM, 2006.

[137] K. Nguyen, T. Nguyen, and S. Cheung. P2p streaming with hierarchical network coding, July 2007.

[138] Khanh Nguyen and Guoqing Xu. Cachetor: Detecting cacheable data to remove bloat. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 268–278. ACM, 2013.

[139] Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. Toddler: Detecting performance problems via similar memory-access patterns. In *ICSE*, pages 562–571. IEEE, 2013.

[140] T. Peng, C. Lecking, and K. Ramamohanarao. Proactively detecting distributed denial of service attacks using source ip address monitoring. In *Networking*, pages 771–782. Springer-Verlag, 2004.

[141] Paul Petefish, Eric Sheridan, and Dave Wichers. Cross-site request forgery (csrf) prevention cheat sheet. In *OWASP*, pages 225–235. ACM, 2007.

[142] PHP-Parser. `https://github.com/nikic/PHP-Parser`.

[143] Tadeusz Pietraszek and Chris Vanden Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Proceedings of the 8th International Conference on Recent Advances in Intrusion*

Detection, RAID'05, pages 124–145, Berlin, Heidelberg, 2006. Springer-Verlag.

[144] M.S.A. Pnueli. Two approaches to interprocedural data flow analysis. *Program Flow Analysis: Theory and Applications*, pages 189–234, 1981.

[145] W. Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. In *ACM conference on Supercomputing*, pages 4–13, 1991.

[146] T. W. Reps, S. Sagiv, and R. Wilhelm. Static program analysis via 3-valued logic. In *CAV* [2], pages 15–30.

[147] Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, pages 49–61, 1995.

[148] J.C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, 2002.

[149] Dorothy Elizabeth Robling Denning. *Cryptography and Data Security*. Addison-Wesley Longman Publishing Co., Inc., 1982.

[150] Shmuel Sagiv, Thomas W. Reps, and Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theor. Comput. Sci.*, 167(1&2):131–170, 1996.

[151] A. Salcinau. *Pointer Analysis for Java Programs: Novel Techniques and Applications*. PhD thesis, MIT, 2006.

[152] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for javascript. In *Symposium on Security and Privacy (SP)*. IEEE, 2010.

[153] D. A. Schmidt. A calculus of logical relations for over- and underapproximating static analyses. *Sci. Comput. Program.*, 64(1):29–53, 2007.

[154] M.N. Seghir, A. Podelski, and T. Wies. Abstraction Refinement for Quantified Array Assertions. In *SAS*, page 3. Springer-Verlag, 2009.

[155] V. Sekar, N. Duffield, K. van der Merwe, O. Spatscheck, and H. Zhang. Lads: Large-scale automated DDoS detection system. In *USENIX*, 2006.

[156] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for javascript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 488–498. ACM, 2013.

[157] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference*, pages 263–272, 2005.

[158] Ohad Shacham, Martin Vechev, and Eran Yahav. Chameleon: adaptive selection of collections. In *ACM SIGPLAN Notices*, volume 44, pages 408–418. ACM, 2009.

[159] Ajeet Shankar, Matthew Arnold, and Rastislav Bodik. Jolt: lightweight dynamic analysis and removal of object churn. In *ACM SIGPLAN Notices*, volume 43, pages 127–142. ACM, 2008.

[160] Market share of PHP-based websites. `http://w3techs.com/technologies/details/pl-php/all/all`.

[161] L. Song and S. Lu. Statistical debugging for real-world performance problems. In *OOPSLA*, NY, USA, 2014. ACM.

[162] S. Song and V. Shmatikov. Saferphp: Finding semantic vulnerabilities in php applications. In *6th Workshop on Programming Languages and Analysis for Security (PLAS)*. ACM, November 2011.

[163] Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for Java. *ACM SIGPLAN Notices*, 41(6):387–400, 2006.

[164] Francois-Xavier Standaert, Tal G. Malkin, and Moti Yung. A unified framework for the analysis of side-channel key recovery attacks. In *Advances in Cryptology: The Theory and Applications of Cryptographic Techniques*, pages 443–461. Springer-Verlag, 2009.

[165] Bjarne Steensgaard. Points-to analysis in almost linear time. In *PLDI*, pages 32–41. ACM, 1996.

[166] A. Stump, C.W. Barrett, D.L. Dill, and J. Levitt. A decision procedure for an extensional theory of arrays. In *IEEE Symposium on Logic in Computer Science*, pages 29–37, 2001.

[167] Zhendong Su and Gary Wassermann. The essence of command injection attacks in web applications. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, pages 372–382. ACM, 2006.

[168] Symantec. Symantec internet security threat report, 2014.

[169] R. Talpade, G. Kim, and S. Khurana. Nomad: Traffic-based network monitoring framework for anomaly detection. In *International Symposium on Computers and Communications*, pages 442–451. IEEE, 1999.

[170] Tachio Terauchi and Alex Aiken. Secure information flow as a safety problem. In *International Conference on Static Analysis*, pages 352–367. Springer-Verlag, 2005.

[171] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot-a Java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999.

[172] Tom Van Goethem, Wouter Joosen, and Nick Nikiforakis. The clock is still ticking: Timing attacks in the modern web. In *Computer and Communications Security*, pages 1382–1393. ACM, 2015.

[173] Kapil Vaswani, Aditya V Nori, and Trishul M Chilimbi. Preferential path profiling: compactly numbering interesting paths. In *ACM SIGPLAN Notices*, volume 42, pages 351–362. ACM, 2007.

[174] Dennis Volpano and Geoffrey Smith. Probabilistic noninterference in a concurrent language. *J. Comput. Secur.*, 7(2-3):231–253, 1999.

[175] M. Walfish, M. Vutukuru, H. Balakrishnan, D. Karger, and S. Shenker. DDoS defense by offense. In *SIGCOMM*, 2006.

[176] H. Wang, D. Zhang, and K.G. Shin. Detecting syn flooding attacks. In *21st Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, pages 1530–1539. IEEE, 2002.

[177] Gary Wassermann and Zhendong Su. Static detection of cross-site scripting vulnerabilities. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 171–180, New York, NY, USA, 2008. ACM.

[178] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *OOPSLA*, pages 187–206. ACM, 1999.

[179] Robert P Wilson and Monica S Lam. Efficient context-sensitive pointer analysis for c programs. *ACM SIGPLAN Notices*, 30(6):1–12, 1995.

[180] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for c programs. In *PLDI*, 1995.

[181] www.garmin.com/support/pdf/iop_spec.pdf. Interface specification.

[182] Yichen Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, 2006.

[183] G. Xu, M. Arnold, A. Rountev, and G. Sevitsky. Finding low utility data structures. In *PLDI*, pages 174–186. ACM, 2010.

[184] Guoqing Xu. CoCo: sound and adaptive replacement of Java collections. In *ECOOP 2013–Object-Oriented Programming*, pages 1–26. Springer, 2013.

[185] Guoqing Xu, Matthew Arnold, Nick Mitchell, Atanas Rountev, and Gary Sevitsky. Go with the flow: profiling copies to find runtime bloat. In *ACM SIGPLAN Notices*, volume 44, pages 419–430. ACM, 2009.

[186] Guoqing Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, Edith Schonberg, and Gary Sevitsky. Scalable runtime bloat detection using abstract dynamic slicing. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(3):23, 2014.

[187] Guoqing Xu and Atanas Rountev. Detecting inefficiently-used containers to avoid bloat. *ACM SIGPLAN Notices*, 45(6):160–173, 2010.

[188] Guanhua Yan, Ritchie Lee, Alex Kent, and David Wolpert. Towards a bayesian network game framework for evaluating DDoS attacks and

defense. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 553–566, New York, NY, USA, 2012. ACM.

[189] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. O'Hearn. Scalable shape analysis for systems code. *CAV*, pages 385–398, 2008.

[190] X. Yang, D. Wetherall, and T. T. Anderson. A DoS-limiting network architecture. In *SIGCOMM*, 2005.

[191] G. Yorsh, E. Yahav, and S. Chandra. Generating precise and concise procedure summaries. *POPL*, 43(1):221–234, 2008.

[192] Xiao Yu, Shi Han, Dongmei Zhang, and Tao Xie. Comprehending performance from real-world execution traces: A device-driver case. In *ASPLOS*, pages 193–206. ACM, 2014.

[193] Z3. `https://github.com/Z3Prover/z3`.

[194] Z3-str2. `https://sites.google.com/site/z3strsolver/`.

[195] S. Zaman, B. Adams, and A. E. Hassan. A qualitative study on performance bugs. In *Mining Software Repositories*. ACM, 2012.

[196] Dmitrijs Zaparanuks and Matthias Hauswirth. Algorithmic profiling. *ACM SIGPLAN Notices*, 47(6):67–76, 2012.

[197] Kehuan Zhang, Zhou Li, Rui Wang, XiaoFeng Wang, and Shuo Chen. Sidebuster: Automated detection and quantification of side-channel leaks in web application development. In *Computer and Communications Security*, pages 595–606. ACM, 2010.

[198] Xin Zheng and Radu Rugina. Demand-driven alias analysis for c. *ACM SIGPLAN Notices*, 43(1):197–208, 2008.