STATIC DETECTION OF SOFTWARE ERRORS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Yichen Xie

June 2006

To my wife Ling

Abstract

This thesis describes three novel static analysis techniques for error detection. First, we present Saturn, a general framework for building precise and scalable analysis for real-world software. Saturn encodes program constructs into boolean constraints and uses a satisfiability solver to infer and check program properties. One key advantage of Saturn is precision: it is path sensitive, precise down to the bit level, and models pointers and heap data. Saturn is also highly scalable, which we achieve using two techniques. First, several optimizations compress the size of the boolean formulas that model the control- and data-flow and the heap locations accessed by a function. Second, a concise summary is computed for each function, allowing inter-procedural analysis without a dramatic increase in the size of the boolean constraints to be solved.

We also present a static analysis algorithm for detecting security vulnerabilities in PHP, a popular server-side scripting language for building web applications. Our analysis employs a novel three-tier architecture to capture information at the intrablock, intraprocedural, and interprocedural levels. This architecture enables us to handle dynamic features of scripting languages that have not been adequately modeled by previous techniques.

Finally, we present several lightweight checkers for redundant operations in system software based on the belief that programmers generally attempt to perform useful work. Redundant operations violate this belief and are likely errors. We demonstrate that in fact many redundancies signal mistakes as serious as traditional hard errors such as deadlocks, memory leaks, etc.

We conduct extensive experiments on open-source software to validate each of these approaches. We demonstrate their practicality and effectiveness by finding hundreds of previously unknown errors, most of which have been reported to and fixed by their developers.

Acknowledgments

First and foremost, I would like to thank my advisor Alex Aiken for his guidance and support throughout my Ph.D. career. He has dedicated countless hours in discussions, paper writing, and attending many of my practice talks. He has taught me by example how to become a researcher and presenter. Alex is also a constant source of ideas and insights. His encouragement to take a second look at SAT-based static analysis led directly to the SATURN project, and our frequent discussions resulted in the solution of several key technical problems along the way. I feel greatly privileged to be his first Ph.D. student at Stanford. This thesis would not be possible without his guidance.

I would like to thank Dawson Engler for his advice and guidance. He got me started on my first research paper and has greatly influenced my choice of research topics and style. I also thank members of his research group: Andy Chou, Seth Hallem, Ted Kremenek, and Junfeng Yang for many memorable discussions on research and other topics, and their camaraderie in many late-night coding and paper-writing sessions.

I benefited greatly from courses taught by Monica Lam and David Dill. Andy and I started our SAT-based analysis effort as a course project in Monica's CS343, and David's CS356 gave me the necessary background knowledge for understanding research papers in formal verification.

I would like to thank Sriram Rajamani for his mentorship at Microsoft Research. My summer internship in the Zing group gave me hands-on experience in building a state-of-the-art software model checker. I also benefited greatly from many discussions with Shaz Qadeer, Jakob Rehof, and Jim Larus both during and after my internship.

I would like to thank Mayur Naik and Brian Hackett for sharing not only the same office, but also the discussions, insights, and laughter that made Gates 412 a fun place during the past few years.

Last but not the least, I would like to thank my family for their love and support.

My parents introduced me to computers at a very young age and supported my pursuit of undergraduate and graduate degrees in a faraway country. None of this would have been possible without my loving wife Ling. Among others, her being a professor long before I finished my degree was a constant driving force that kept me motivated and productive throughout my years at Stanford.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Software plays a pivotal role in our society. It has become indispensable in our daily lives and is increasingly deployed in critical systems such as online banking, air traffic control, health care and so on.

One serious problem facing both developers and users today is the poor quality of software. Bugs are a fact of life. Software on average contains 10 to 20 defects per thousand lines of code after compiling and testing [43] and large software systems routinely ship with thousands or even tens of thousands of potential defects [56] upon initial release. These errors are a constant source of frustration and fear. Software companies typically spend more than 80% [74] of their development budget on quality control. Yet despite the countless hours developers pour in, software remains buggy long after it is deployed, as evidenced by the continuous, mandatory patches years after software's initial release. For users, damage due to bugs is equally evident. "Blue screens of death" and application crashes cause innumerable hours of lost productivity. Unpatched software attracts viruses and worms that inflict significant damages to users' computers and their businesses: Code Red [11], Slammer [12], and Blaster [13] are three representative multibillion-dollar worms that exploited unpatched errors in software. According to NIST [74], software errors are estimated to have cost the U.S. economy $59.5 billion in 2002.

Part of the problem lies in the tools programmers use to develop software. Larus *et al* [51] observed that despite the rapid advances in computing technology over the past three decades, software development tools have neither progressed to meet the challenge of complex software nor evolved to exploit faster computers. To address this problem, researchers have recently developed a plethora of techniques for automatic or semi-automatic detection and prevention of bugs.

1.2 Existing Techniques

Existing solutions for software error detection can be classified into dynamic and static techniques, which we briefly describe in the following subsections. Detailed discussion of specific related projects in both categories are presented in the next three chapters.

1.2.1 Dynamic Tools

Dynamic tools instrument the program (source or binary) and detect errors by analyzing information collected from one or more executions. Representative tools include Purify [36], Valgrind [71], and Eraser [68], which are effective at detecting a variety of program errors such as memory leaks, buffer overflows, and race conditions in C programs.

One key advantage of dynamic tools is precision. With a few exceptions (e.g., Eraser [68]), they generate accurate error reports because the information upon which bug reports are based is collected from actual program executions. However, the reliance on executing the program also imposes several limitations on dynamic tools:

1. **Performance**. Dynamic tools typically impose significant time and space overheads due to instrumentation. For example, programs instrumented by Purify [36] run approximately two to five times slower and consume over 50% more memory than their unmodified counterparts. As such, dynamic tools are mostly used during testing and debugging and rarely deployed in production environments.

2. **Coverage**. Dynamic techniques collect information from concrete program executions and are therefore limited by the quality of the test suite. While they are effective at finding errors along commonly executed code paths, bugs hidden in obscure corner cases or those triggered by unexpected program input (e.g., security vulnerabilities) remain undetected.

3. **Limited applicability**. Instrumentation introduces incompatibilities that limit the applicability of dynamic techniques. Programs with stringent timing requirements and

resource constraints (e.g., operating system kernels) are usually not readily checked using dynamic tools.

1.2.2 Static Analysis

Static analysis provides an alternative solution. It finds errors by analyzing the source code and does not require running the program. Compared to dynamic tools, static techniques have the following advantages:

1. **Early Detection of Software Errors**. The earlier program errors are detected, the cheaper it is to fix them. Static analysis works on program source and can be carried out immediately after the code is written. Many modular analysis techniques (e.g., [35, 82, 9]) do not even require a complete program. They are able to analyze independent code modules as they become available, and find bugs long before code-complete.

2. **Coverage**. Static analysis can thoroughly and systematically explore all possible execution scenarios and offer much better coverage than dynamic tools. Static analysis does not require test cases and is particularly effective at finding bugs on obscure code paths.

3. **Cost and applicability**. Static analysis imposes no runtime overhead and can be used to analyze low-level system software that has stringent timing and resource constraints. In addition, the analysis algorithms proposed in this thesis are fully automatic. They require little human effort to run and produce high quality results.

The key challenge in static analysis is the trade-off between precision and scalability. Precise analyses are often expensive and cannot be readily applied to very large code bases, while scalable analyses often have limited analysis power which results in a large number of false warnings. Striking the right balance between the two is the focus of many research efforts in static analysis, including this thesis.

1.3 Thesis Outline and Summary of Contributions

The work described in this thesis lies in the general category of static techniques. In the following chapters, we present three novel static analysis algorithms for automatic detection of errors in large software systems.

Chapter 2 describes the SATURN analysis framework. SATURN leverages recent advances in solving boolean satisfiability (SAT) constraints and serves as a general framework for building precise and scalable static analyses for multimillion-line software systems. We demonstrate that by employing efficient boolean encoding algorithms and choosing appropriate abstraction boundaries, static analysis can achieve a high level of precision without sacrificing scalability. This chapter is based on joint work with Alex Aiken [82, 81] and makes the following contributions:

- We give an encoding from a realistic programming language to SAT that is substantially more efficient than previous approaches. While we model each bit path-sensitively as in [49, 19, 84], several techniques achieve a substantial reduction in the size of the SAT formulas SATURN must solve.

- We describe how to compute a summary, similar to a type signature, for each analyzed function, thereby enabling interprocedural analysis. Computing concise summaries enables SATURN to analyze much larger programs than previous SAT-based error checking systems. In fact, the scaling behavior of SATURN is at least competitive with, if not better than, other, non-SAT approaches to bug finding and verification. In addition, SATURN is able to infer and apply summaries that encode a form of interprocedural path sensitivity, lending itself well to checking complex value-dependent program behaviors.

- We show the flexibility of a SAT-based approach by constructing two property checkers that traditionally required different analysis techniques—a Linux lock checker and a memory leak checker—under the SATURN framework.

- We present significant experimental results for both checkers showing that our approach scales well and achieves significant improvement in precision compared to other state-of-the-art techniques.

- We further demonstrate the parallelizability of our approach by building an analysis framework that harnesses the processing power of multiple, loosely connected CPUs, which resulted in dramatically reduced analysis time for the memory leak checker.

Chapter 3 describes an algorithm for static detection of security vulnerabilities in scripting languages. Scripting languages are widely perceived to be difficult to analyze statically due to their dynamic nature. We propose a three-tier analysis architecture that successfully models the unique features of scripting languages. This chapter is based on joint work with Alex Aiken [83] and makes the following contributions.

- We present an interprocedural static analysis algorithm for PHP. A language as dynamic as PHP presents unique challenges for static analysis: language constructs that allow dynamic inclusion of program code, variables whose types change during execution, operations with semantics that depend on the runtime types of the operands, and pervasive use of hash tables and regular expression matching are just some features that must be modeled well to produce useful results.

- We show how to use our static analysis to find SQL injection vulnerabilities. Once configured, the analysis is fully automatic. Although we focus on SQL injections in this work, the same techniques can be applied to detecting other vulnerabilities such as cross site scripting (XSS) and code injection in web applications.

- We experimentally validate our approach by implementing the analysis algorithm and running it on six popular web applications written in PHP, finding 105 previously unknown security vulnerabilities. We analyze two reported vulnerabilities in PHP-fusion, a mature, widely deployed content management system, and construct exploits for both that allow an attacker to take control of or damage the system.

Chapter 4 presents several lightweight checkers for redundant operations in system software. They are based on the intuition that many high-level conceptual errors map to low-level redundant operations. This chapter is based on joint work with Dawson Engler [86, 87] and makes the following contributions:

- We propose the idea that redundant operations, like type errors, commonly flag serious correctness errors.

- We experimentally validate this idea by writing and applying five redundancy checkers to real code and finding hundreds of errors.

- We demonstrate that redundancies, even when harmless, strongly correlate with the presence of traditional hard errors.

- We show how redundancies provide a way to detect dangerous specification omissions.

Finally, conclusions are presented in Chapter 5.

Chapter 2

A SAT-based Analysis Framework

2.1 Introduction

This chapter presents SATURN,[1], a software error-detection framework based on exploiting recent advances in solving boolean satisfiability (SAT) constraints.

At a high level, SATURN works by transforming commonly used program constructs into boolean constraints and then using a SAT solver to infer and check program properties. Compared to previous error detection tools based on data flow analysis or abstract interpretation, our approach has the following advantages:

1. *Precision:* SATURN's modeling of loop-free code is faithful down to the bit level, and is therefore considerably more precise than approaches where immediate information loss occurs due to abstraction early in the analysis. In the context of error detection, the extra precision translates into added analysis power with less confusion, which we demonstrate by finding many more errors with significantly fewer false positives than previous approaches.

2. *Flexibility:* Traditional techniques rely on a combination of carefully chosen abstractions to focus on a particular class of properties effectively. SATURN, by exploiting the expressive power of boolean constraints, uniformly models many language features and can therefore serve as a general framework for a wider range of analyses. We demonstrate the flexibility of our approach by encoding two property checkers in SATURN that traditionally require distinct sets of techniques.

[1]SATisfiability-based failURe aNalysis.

However, SAT-solving is NP-complete, and therefore incurs a worst-case exponential time cost. Since Saturn aims at checking large programs with millions of lines of code, we employ two techniques to make our approach scale. *Intraprocedurally* (within one procedure), our encoding of program constructs as boolean formulas is substantially more compact than previous approaches (Section 2.2). While we model each bit path sensitively as in [84, 49, 19], several techniques achieve a substantial reduction in the size of the SAT formulas Saturn must solve (Section 2.3).

Interprocedurally (across multiple procedures), Saturn computes a concise *summary*, similar to a type signature, for each analyzed function. The summary-based approach enables Saturn to analyze much larger programs than previous error checking systems based on SAT, and in fact, the scaling behavior of Saturn is at least competitive with, if not better than, other non-SAT approaches to bug finding and verification. In addition, Saturn is able to infer and apply summaries that encode a form of interprocedural path sensitivity, lending itself well to checking complex program behaviors (see Section 2.5.2 for an example).

Summary-based interprocedural analysis also enables parallelization. Saturn processes each function separately and the analysis can be carried out in parallel, subject only to the ordering dependencies of the function call graph. In Section 2.6.8, we describe a simple distributed architecture that harnesses the processing power of a heterogeneous cluster of CPUs (we used approximately 80 in our experiments). Our implementation reduces the running time of our memory leak checker on the Linux kernel (5MLOC) from over 23 hours to 50 minutes.

We present experimental results to validate our approach (Sections 2.5 and 2.6). Section 2.5 describes the encoding of temporal safety properties in Saturn and presents an interprocedural analysis that automatically infers and checks such properties. We show one such specification in detail: checking that a single thread correctly manages locks—i.e., does not perform two lock or unlock operations in a row on any lock (Section 2.5.5). Section 2.6 gives a context- and path-sensitive escape analysis of dynamically allocated objects, which forms the basis of our memory leak checker. Both checkers find more errors than previous approaches with significantly fewer false positives.

One thing that Saturn is not, at least in its current form, is a verification framework. Tools such as CQual [30] are capable of verification (proving the absence of bugs, or at least as close as one can reasonably come to that goal for C programs). In the scope of this chapter, Saturn is used as a bug finding framework in the spirit of MC [35], which means it is designed to find as many bugs as possible with a low false positive rate, potentially at the cost of missing some bugs.

The rest of the chapter is organized as follows: Section 2.2 presents the Saturn language and its encoding into boolean constraints. Section 2.3 discusses a number of key improvements to the encoding that enable efficient checking of open programs. Section 2.4 gives a brief outline of how we use the Saturn framework to build modular checkers for software. Sections 2.5 and 2.6 are two case studies where we present the details of the design and implementation of two property checkers. We describe sources of unsoundness for both checkers in Section 2.7. Related work is discussed in Section 2.8 and we conclude with Section 2.9.

2.2 The Saturn Framework

In this section, we present a low-level programming language and its translation into the Saturn framework. Because our implementation targets C programs, our language models integers, structures, pointers, and handles the arbitrary control flow[2] found in C. We begin with a language and encoding that handles only integer program values (Section 2.2.1) and gradually add features until we have presented the entire framework: intraprocedural control flow including loops (Section 2.2.2), structures (Section 2.2.3), pointers (Section 2.2.4), and finally attributes (Section 2.2.5). In Section 2.3 we consider some techniques that substantially improve the performance of our encoding.

2.2.1 Modeling Integers

Figure 2.1 presents a grammar for a simple imperative language with integers. The parenthesized symbol on the left hand side of each production is a variable ranging over elements of its syntactic category.

The language is statically and explicitly typed; the type rules are completely standard and for the most part we elide types for brevity. There are two base types: booleans (bool) and n-bit signed or unsigned integers (int). Note the base types are syntactically separated in the language as expressions, which are integer-valued, and conditions, which

[2]The current implementation of Saturn handles reducible flow-graphs, which are by far the most common form even in C code. Irreducible flow-graphs can be converted to reducible ones by node-splitting [1].

$\boxed{\text{Language}}$

$$\begin{aligned}
\text{Type } (\tau) &::= (n, \text{signed} \mid \text{unsigned}) \\
\text{Obj } (o) &::= v \\
\text{Expr } (e) &::= \text{unknown}(\tau) \mid \text{const}(n, \tau) \mid o \mid \text{unop } e \mid e_1 \text{ binop } e_2 \mid (\tau)\ e \mid \text{lift}_e(c, \tau) \\
\text{Cond } (c) &::= \text{false} \mid \text{true} \mid \neg\ c \mid e_1 \text{ comp } e_2 \mid c_1 \wedge c_2 \mid c_1 \vee c_2 \mid \text{lift}_c(e) \\
\text{Stmt } (s) &::= o \leftarrow e \mid \text{assert}(c) \mid \text{assume}(c) \mid \text{skip}
\end{aligned}$$

$$\text{comp} \in \{=, >, \geq, <, \leq, \neq\} \qquad \text{unop} \in \{-, !\} \qquad \text{binop} \in \{+, -, *, /, \text{mod}, \text{band}, \text{bor}, \text{xor}, \ll, \gg_l, \gg_a\}$$

$\boxed{\text{Representation}}$

$$\begin{aligned}
\text{Rep } (\beta) &::= [b_{n-1} \dots b_0]_s \quad \text{where } s \in \{\text{signed}, \text{unsigned}\} \\
\text{Bit } (b) &::= 0 \mid 1 \mid x \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \mid \neg b
\end{aligned}$$

Figure 2.1: Modeling integers in SATURN.

are boolean-valued. We use τ to range solely over different types of integer values.

The integer expressions include constants (const), integer variables (v), unary and binary operations, integer casts, and lifting from conditionals. We give the list of operators that we model precisely using boolean formulas (e.g., +, -, bitwise-and, etc.); for other operators (e.g., division, remainder, etc.), we make approximations. We use a special expression unknown to model unknown values (e.g., in the environment) and the result of operations that we do not model precisely.

Objects in the scalar language are n-bit signed or unsigned integers, where n and the signedness are determined by the type τ. As shown at the bottom of Figure 2.1, a separate boolean expression models each bit of an integer and thus tracking the width is important for our encoding. The signed/unsigned distinction is needed to precisely model low-level type casts, bit shift operations, and arithmetic operations.

The class of objects (Obj) ultimately includes variables, pointers, and structures, which encompass all the entities that can be the target of an assignment. For the moment we describe only integer variables.

The encoding for a representative selection of constructs is shown in Figure 2.2. The rules for expressions have the form

$$\psi \vdash e \overset{\mathtt{E}}{\Rightarrow} \beta$$

$\boxed{\textsf{Expressions}}$

$$\frac{\beta = \psi(v)}{\psi \vdash v \overset{\mathtt{E}}{\Rightarrow} \beta}\ \textsf{scalar}$$

$$\frac{\begin{array}{c}(n,s) = \tau \\ x_0,\ldots,x_{n-1} \text{ are fresh boolean variables}\end{array}}{\psi \vdash \textsf{unknown}(\tau) \overset{\mathtt{E}}{\Rightarrow} [x_{n-1}\ldots x_0]_s}\ \textsf{unknown}$$

$$\frac{\begin{array}{l}\psi \vdash e \overset{\mathtt{E}}{\Rightarrow} [b_{n-1}\ldots b_0]_x \quad \tau = (m,s) \\ b'_i = \begin{cases} b_i & \text{if } 0 \le i < n \\ 0 & \text{if } s = \textsf{unsigned} \text{ and } n \le i < m \\ b_{n-1} & \text{if } s = \textsf{signed} \text{ and } n \le i < m \end{cases}\end{array}}{\psi \vdash (\tau)\ e \overset{\mathtt{E}}{\Rightarrow} [b'_{m-1}\ldots b'_0]_s}\ \textsf{cast}$$

$$\frac{(n,s) = \tau \quad \psi \vdash c \overset{\mathtt{C}}{\Rightarrow} b}{\psi \vdash \textsf{lift}_e(c,\tau) \overset{\mathtt{E}}{\Rightarrow} [\underbrace{00\cdots0}_{n-1} b]_s}\ \textsf{lift}_e$$

$$\frac{\begin{array}{c}\psi \vdash e \overset{\mathtt{E}}{\Rightarrow} [b_{n-1}\ldots b_0]_s \\ \psi \vdash e' \overset{\mathtt{E}}{\Rightarrow} [b'_{n-1}\ldots b'_0]_s\end{array}}{\psi \vdash e\ \mathtt{band}\ e' \overset{\mathtt{E}}{\Rightarrow} [b_{n-1} \wedge b'_{n-1} \ldots b_0 \wedge b'_0]_s}\ \textsf{and}$$

$\boxed{\textsf{Conditionals}}$

$$\frac{\psi \vdash e \overset{\mathtt{E}}{\Rightarrow} [b_{n-1}\ldots b_0]_s}{\psi \vdash \textsf{lift}_c(e) \overset{\mathtt{C}}{\Rightarrow} \bigvee_i b_i}\ \textsf{lift}_c$$

$\boxed{\textsf{Statements}}$

$$\frac{\psi \vdash e \overset{\mathtt{E}}{\Rightarrow} \beta}{\mathcal{G},\psi \vdash (v \leftarrow e) \overset{\mathtt{S}}{\Rightarrow} \langle \mathcal{G}; \psi[v \mapsto \beta]\rangle}\ \textsf{assign}$$

$$\frac{\psi \vdash c \overset{\mathtt{C}}{\Rightarrow} b}{\mathcal{G},\psi \vdash \textsf{assume}(c) \overset{\mathtt{S}}{\Rightarrow} \langle \mathcal{G} \wedge b; \psi\rangle}\ \textsf{assume}$$

$$\frac{\psi \vdash c \overset{\mathtt{C}}{\Rightarrow} b \quad (\mathcal{G} \wedge \neg b) \text{ not satisfiable}}{\mathcal{G},\psi \vdash \textsf{assert}(c) \overset{\mathtt{S}}{\Rightarrow} \langle \mathcal{G}; \psi\rangle}\ \textsf{assert-ok}$$

Figure 2.2: The translation of integers.

which means that under the environment ψ mapping variables to vectors of boolean expressions (one for each bit in the variable's type), the expression e is encoded as a vector of boolean expressions β.

The encoding scheme for conditionals

$$\psi \vdash c \overset{\mathtt{C}}{\Rightarrow} b$$

is similar, except the target is a single boolean expression b modeling the condition. The most interesting rules are for statements:

$$\mathcal{G},\psi \vdash s \overset{\mathtt{S}}{\Rightarrow} \left\langle \mathcal{G}'; \psi' \right\rangle$$

means that under guard $\mathcal{G}$ and variable environment ψ the statement s results in a new

$$
\begin{array}{l}
\mathsf{MergeScalar}\left(v, \overline{(\mathcal{G}_i, \psi_i)}\right) = [b'_m \dots b'_0]_s \\
\qquad \text{where} \left\{ \begin{array}{l} [b_{im} \dots b_{i0}]_s = \psi_i(v) \\ b'_j = \bigvee_i (\mathcal{G}_i \wedge b_{ij}) \end{array} \right. \\
\mathsf{MergeEnv}\left(\overline{(\mathcal{G}_i, \psi_i)}\right) = \langle \bigvee_i \mathcal{G}_i ; \psi \rangle \\
\qquad \text{where } \psi(v) = \mathsf{MergeScalar}\left(v, \overline{(\mathcal{G}_i, \psi_i)}\right)
\end{array}
$$

Figure 2.3: Merging control-flow paths.

guard/environment pair $\langle \mathcal{G}'; \psi' \rangle$. In our system, guards express path sensitivity; every statement is guarded by a boolean expression expressing the conditions under which that statement may execute. Most statements do not affect guards (the exception is `assume`); the important operations on guards are discussed in Section 2.2.2. We explain the conceptual meaning of a guard using the following example:

```
if (c) {s1;s2} else s3;
s4;
```

Statements s_1 and s_2 are executed if c is true, so the guard for both statements is the boolean encoding of c. Similarly, s_3's guard is the encoding of $\neg c$. Statement s_4 is reached from both branches of the `if` statement and therefore its guard is the disjunction of the guards from the two branches: $(c \vee \neg c) = $ `true`.

A key statement in our language is `assert`, which we use to express points at which satisfiability queries must be checked. A statement `assert(c)` checks that $\neg$`c` cannot be true at that program point by computing the satisfiability of $\mathcal{G} \wedge \neg b$, where $\mathcal{G}$ is the guard of the `assert` and b is the encoding of the condition c.

The overall effect of the encoding is to perform symbolic execution, cast in terms of boolean expressions. Each statement transforms an environment into a new environment (and guard) that captures the effect of the statement. If all bits in the initial environment ψ_0 are concrete 0's and 1's and there are no `unknown` expressions in the program being analyzed, then in fact this encoding is straightforward interpretation and all modeled bits can themselves be reduced to 0's and 1's. However, bits may also be boolean variables (unknowns) and arbitrary boolean expressions over such variables.

2.2.2 Control Flow

We represent function bodies as control-flow graphs, which we define informally. For the purpose of this section, we assume loop-free programs. Loops can be treated in a variety of ways which are described at the end of this section. Each statement s is a node in the control-flow graph, and each edge (s, s') represents an unconditional transfer of control from s to s'. If a statement has multiple successors, then execution may be transferred to any successor non-deterministically.

To model the deterministic semantics of conventional programs, we require that if a node has multiple successors, then each successor is an `assume` statement, and furthermore, that the conditions in those `assume`s are mutually exclusive and that their disjunction is equivalent to `true`. Thus a conditional branch with predicate p is modeled by a statement with two successors: one successor `assumes` p (the true branch) and the other `assumes` $\neg p$ (the false branch).

The other important issue is assigning a guard and environment to each statement s. Assume s has an ordered list of predecessors $\overline{s_i}$.[3] The encoding of s_i produces an environment ψ_i and guard $\mathcal{G}_i$. The initial guard and environment for s is then a combination of the final guards and environments of its predecessors. The desired guard is simply the disjunction of the predecessor guards; as we may arrive at s from any of the predecessors, s may be executed if any predecessor's guard is true. Note that due to the mutual exclusion assumption for branch conditions, at most one predecessor's guard can be true at a time. The desired environment is more complex, as we wish to preserve the path-sensitivity of our analysis down to the bit level. Thus, the value of each bit of each variable in the environment for each predecessor s_i of s must include the guard for s_i as well. This motivates the function MergeScalar in Figure 2.3, which implements a multiplexer circuit that selects the appropriate bits from the input environments ($\psi_i(v)$) based on the predecessor guards ($\mathcal{G}_i$). Finally, MergeEnv combines the two components together to define the initial environment and guard for s.

Preserving path sensitivity for every modeled bit is clearly expensive and it is easy to construct realistic examples where the number of modeled paths is exponential in the size of the control-flow graph. In Section 2.3.3 we present an optimization that enables us to make this approach work in practice.

[3] We use the notation $\overline{X_i}$ as a shorthand for a vector of similar entities: $X_1 \dots X_n$.

$\boxed{\text{Language}}$

$$\begin{aligned} \mathsf{Type}\ (\tau) &::= \{(f_1,\tau_1),\ldots,(f_n,\tau_n)\} \mid \ldots \\ \mathsf{Obj}\ (o) &::= \{(f_1,o_1),\ldots,(f_n,o_n)\} \mid \ldots \end{aligned}$$

$\boxed{\text{Shorthand}}$

$$\frac{o = \{(f_1,o_1),\ldots,(f_n,o_n)\}}{o.f_i \overset{\mathsf{def}}{=} o_i} \quad \textsf{field-access}$$

$\boxed{\text{Representation}}$

$$\mathsf{Rep}\ (\beta) ::= \{(f_1,\beta_1),\ldots,(f_n,\beta_n)\} \mid \ldots$$

$\boxed{\text{Translation}}$

$$\frac{\begin{array}{c} o = \{(f_1,o_1),\ldots,(f_n,o_n)\} \\ \psi \vdash o_i \overset{\mathtt{E}}{\Rightarrow} \beta_i \quad \text{for } i \in 1..n \end{array}}{\psi \vdash o \overset{\mathtt{E}}{\Rightarrow} \{(f_1,\beta_1),\ldots,(f_n,\beta_n)\}} \quad \textsf{object-str}$$

$$\begin{aligned} &\mathsf{RecAssign}(\psi, v, \beta) = \psi[v \mapsto \beta] \\ &\mathsf{RecAssign}(\psi, o, \beta) = \psi_n \\ &\qquad \text{where} \begin{cases} o = \{(f_1,o_1),\ldots,(f_n,o_n)\} \\ \beta = \{(f_1,\beta_1),\ldots,(f_n,\beta_n)\} \\ \psi_0 = \psi \\ \psi_i = \mathsf{RecAssign}(\psi_{i-1}, o_i, \beta_i)\ (\forall\ i \in 1..n) \end{cases} \end{aligned}$$

$$\frac{\begin{array}{c} \psi \vdash e \overset{\mathtt{E}}{\Rightarrow} \beta \\ \psi' = \mathsf{RecAssign}(\psi, o, \beta) \end{array}}{\mathcal{G}, \psi \vdash (o \leftarrow e) \overset{\mathtt{S}}{\Rightarrow} \langle \mathcal{G}; \psi' \rangle} \quad \textsf{assign-struct}$$

Figure 2.4: The translation of structures.

Finally, every control-flow graph has a distinguished entry statement with no predecessors. The guard for this initial statement is `true`. We postpone discussion of the initial environment ψ_0 to Section 2.3.2 where we describe lazy modeling of the external execution environment.

As mentioned in Section 2.1, the two checkers described in this paper treat loops unsoundly. One technique we adopt is to simply unroll a loop a fixed number of times and remove backedges from the control-flow graph. Thus, every function body is represented by an acyclic control-flow graph. Another technique is *havoc'ing*, which we discuss in detail in the context of the memory leak checker (Section 2.6).

While our handling of loops is unsound, we have found it to be useful in practice (see Section 2.5.6 and 2.6.9).

2.2.3 Structures

The program syntax and the encoding of structures is given in Figure 2.4. A structure is a data type with named fields, which we represent as a set of (*field_name*, *object*) pairs. We extend the syntax of types (resp. objects) with sets of types (resp. objects) labeled by field names, and similarly the representation of a `struct` in C is the representation of the fields also labeled by the field names. The shorthand notation $o.f_i$ selects field f_i from object o.

$\boxed{\text{Language}}$

$$\begin{array}{rrl}
\text{Type} & (\tau) & ::= \tau * \mid \text{void*} \mid \ldots \\
\text{Obj} & (o) & ::= p \mid \ldots \\
\text{Deref} & (m) & ::= (*p).f_1.\cdots.f_n \qquad (n \geq 0) \\
\text{Expr} & (e) & ::= \text{null} \mid \&o \mid \&m \mid \ldots \\
\text{Stmt} & (s) & ::= \text{load}(m, o) \mid \text{store}(m, e) \mid \\
 & & \quad\ \text{newloc}(p) \mid \ldots
\end{array}$$

$\boxed{\text{Address}}$

$\text{Addr}\ (\sigma) ::= \hat{1} \mid \hat{2} \mid \ldots$
$\text{AddrOf} : \text{Obj} \mapsto \text{Addr}$
(Constraint: no two objects of the same type share the same address)

$\boxed{\text{Representation}}$

$$\begin{array}{rrl}
\text{Loc} & (l) & ::= \text{null} \mid o \mid \sigma \\
\text{Rep} & (\beta) & ::= \{\!\!|\ (\mathcal{G}_0, l_0), \ldots, (\mathcal{G}_k, l_k)\ |\!\!\} \mid \ldots
\end{array}$$

$\boxed{\text{Translation}}$

$$\frac{\beta = \psi(p)}{\psi \vdash p \overset{\mathsf{E}}{\Rightarrow} \beta}\ \text{pointer}$$

$$\frac{}{\psi \vdash \&o \overset{\mathsf{E}}{\Rightarrow} \{\!\!|\ (\text{true}, o)\ |\!\!\}}\ \text{getaddr-obj}$$

$$\frac{\begin{array}{l} m = (*p).f_1.\cdots.f_n \\ \psi \vdash p \overset{\mathsf{E}}{\Rightarrow} \{\!\!|\ (\mathcal{G}_0, \text{null}), \overline{(\mathcal{G}_i, o_i)}\ |\!\!\} \\ \beta = \{\!\!|\ (\mathcal{G}_0, \text{null}), \overline{(\mathcal{G}_i, o_i.f_1.\cdots.f_n)}\ |\!\!\} \end{array}}{\psi \vdash \&m \overset{\mathsf{E}}{\Rightarrow} \beta}\ \text{getaddr-mem}$$

$$\frac{\psi(p) = \{\!\!|\ (\mathcal{G}_0, \text{null}), \overline{(\mathcal{G}_i, l_i)}\ |\!\!\}}{\psi \vdash \text{lift}_c(p) \overset{\mathsf{C}}{\Rightarrow} \bigvee_{i \neq 0} \mathcal{G}_i}\ \text{lift}_c\text{-pointer}$$

$$\frac{\begin{array}{l} l = \begin{cases} o & \text{if } p \text{ is of type } \tau * \\ \sigma & \text{if } p \text{ is of type void*} \end{cases} \\ \beta = \{\!\!|\ (\text{true}, l)\ |\!\!\} \text{ and } o \text{ or } \sigma \text{ fresh} \end{array}}{\mathcal{G}, \psi \vdash \text{newloc}(p) \overset{\mathsf{S}}{\Rightarrow} \langle \mathcal{G}; \psi[p \mapsto \beta] \rangle}\ \text{newloc}$$

$$\frac{\begin{array}{c} \psi(p) = \{\!\!|\ (\mathcal{G}_0, \text{null}), \overline{(\mathcal{G}_i, \sigma_i)}\ |\!\!\} \\ \text{type of } o_i = \tau \text{ and AddrOf}(o_i) = \sigma_i \end{array}}{\psi \vdash (\tau *)p \overset{\mathsf{E}}{\Rightarrow} \{\!\!|\ (\mathcal{G}_0, \text{null}), \overline{(\mathcal{G}_i, o_i)}\ |\!\!\}}\ \text{cast-from-void*}$$

$$\frac{\begin{array}{c} \psi(p) = \{\!\!|\ (\mathcal{G}_0, \text{null}), \overline{(\mathcal{G}_i, o_i)}\ |\!\!\} \\ \text{AddrOf}(o_i) = \sigma_i \end{array}}{\psi \vdash (\text{void*})p \overset{\mathsf{E}}{\Rightarrow} \{\!\!|\ (\mathcal{G}_0, \text{null}), \overline{(\mathcal{G}_i, \sigma_i)}\ |\!\!\}}\ \text{cast-to-void*}$$

$$\frac{\begin{array}{l} m = (*p).f_1.\cdots.f_n \\ \psi \vdash p \overset{\mathsf{E}}{\Rightarrow} \{\!\!|\ (\mathcal{G}_0, \text{null}), (\mathcal{G}_1, o_1), \ldots, (\mathcal{G}_k, o_k)\ |\!\!\} \\ \mathcal{G}' = \mathcal{G} \wedge \neg \mathcal{G}_0 \\ \mathcal{G}' \wedge \mathcal{G}_i, \psi \vdash (o_i.f_1.\cdots.f_n \leftarrow e) \overset{\mathsf{S}}{\Rightarrow} \langle \mathcal{G}_i; \psi_i \rangle \\ \hfill (\text{for } i \in 1..k) \end{array}}{\mathcal{G}, \psi \vdash \text{store}(m, e) \overset{\mathsf{S}}{\Rightarrow} \text{MergeEnv}\left(\overline{(\mathcal{G}_i; \psi_i)}\right)}\ \text{store}$$

Figure 2.5: Pointers and guarded location sets.

The function RecAssign does the work of structure assignment. As expected, assignment of structures is defined in terms of assignments of its fields. Because structures may themselves be fields of structures, RecAssign is recursively defined.

2.2.4 Pointers

The final and technically most involved construct in our encoding is pointers. The discussion of pointers is divided into three parts: in Section 2.2.4.1, we introduce a concept called *Guarded Location Set* (GLS) to capture path-sensitive points-to information. We extend the representation with type casts and polymorphic locations in Section 2.2.4.2 and discuss the rules in detail in Section 2.2.4.3.

$$\mathsf{AddGuard}\left(\mathcal{G}, \{\!|\ (\mathcal{G}_1, l_1), .., (\mathcal{G}_k, l_k)\ |\!\}\right) = \{\!|\ (\mathcal{G} \wedge \mathcal{G}_1, l_1), .., (\mathcal{G} \wedge \mathcal{G}_k, l_k)\ |\!\}$$

$$\mathsf{MergePointer}\left(p, \overline{(\mathcal{G}_i, \psi_i)}\right) = \bigcup_i \mathsf{AddGuard}(\mathcal{G}_i, \psi_i(p))$$

$$\mathsf{MergeEnv}\left(\overline{(\mathcal{G}_i, \psi_i)}\right) = \langle \bigvee_i \mathcal{G}_i ; \psi \rangle \qquad \text{where} \begin{cases} \psi(v) = \mathsf{MergeScalar}\left(v, \overline{(\mathcal{G}_i, \psi_i)}\right) \\ \psi(p) = \mathsf{MergePointer}\left(p, \overline{(\mathcal{G}_i, \psi_i)}\right) \end{cases}$$

Figure 2.6: Control-flow merges with pointers.

2.2.4.1 Guarded Location Sets

Pointers in SATURN are modeled with *Guarded Location Sets* (GLS). A GLS represents the set of locations a pointer could reference at a particular program point. To maintain path-sensitivity, a boolean guard is associated with each location in the GLS and represents the condition under which the points-to relationship holds. We write a GLS as $\{\!|\ (\mathcal{G}_0, l_0), \ldots, (\mathcal{G}_n, l_n)\ |\!\}$. Special braces ($\{\!|\ \ |\!\}$) distinguish GLSs from other sets. We illustrate GLSs with an example, but delay technical discussion until Section 2.2.4.3.

```
1  if (c) p = &x;      /* p : {| (true, x) |} */
2  else p = &y;        /* p : {| (true, y) |} */
3  *p = 3;          /* p : {| (c, x), (¬c, y) |} */
```

In the true branch, the GLS for p is $\{\!|\ (\mathsf{true}, x)\ |\!\}$, meaning p always points to x. Similarly, $\psi(p)$ evaluates to $\{\!|\ (\mathsf{true}, y)\ |\!\}$ in the false branch. At the merge point, branch guards are added to the respective GLSs and the representation for p becomes $\{\!|\ (c, x), (\neg c, y)\ |\!\}$. Finally, the store at line 3 makes a parallel assignment to x and y under their respective guards (i.e., if (c) x = 3; else y = 3;).

To simplify technical discussion, we assume locations in a GLS occur at most once—redundant entries $(\mathcal{G}, l)$ and $(\mathcal{G}', l)$ are merged into $(\mathcal{G} \vee \mathcal{G}', l)$. Also, we assume the first location l_0 in a GLS is always null (we use the false guard for $\mathcal{G}_0$ if necessary).

2.2.4.2 Polymorphic Locations and Type Casts

The GLS representation models pointers to concrete objects with a single known type. However, it is common for heap objects to go through multiple types in C. For example, in the following code,

```
1 void *malloc(int size);
2 p = (int *)malloc(len);
3 q = (char *)p;
4 return q;
```

the memory block allocated at line 2 goes through three different types. These types all have different representations (i.e., different numbers of bits) and thus need to be modeled separately, but the analysis must understand that they refer to the same location. We need to model: 1) the polymorphic pointer type void*, and 2) cast operations to and from void*. Casts between incompatible pointer types (e.g., from int* to char*) can then be modeled via an intermediate cast to void*.

We solve this problem by introducing *addresses* (Addr), which are symbolic identifiers associated with each unique memory location. We use a mapping $\mathsf{AddrOf} : \mathsf{Obj} \rightarrow \mathsf{Addr}$ to record the addresses of objects. Objects of different types share the same address if they start at the same memory location. In the example above, p and q point to different objects, say o_1 of type int and o_2 of type char, and o_1 and o_2 must share the same address (i.e., $\mathsf{AddrOf}(o_1) = \mathsf{AddrOf}(o_2)$). Furthermore, an address may have no associated concrete objects if it is referenced only by a pointer of type void* and never dereferenced at any other types. In other words, the inverse mapping AddrOf^{-1} may not be defined for some addresses. Using guarded location sets and addresses, we can now describe the encoding of pointers in detail.

2.2.4.3 Encoding Rules

Figure 2.5 defines the language and encoding rules for pointers. Locations in the GLS can be 1) null, 2) a concrete object o, or 3) an address σ of a polymorphic pointer (void*). We maintain a global mapping AddrOf from objects to their addresses and use it in the cast rules to convert pointers to and from void*.

The rules work as follows. Taking the address of an object (get-addr-{obj,mem}) constructs a GLS with a single entry–the object itself with guard true. The newloc rule creates a fresh object or address depending on the type of the target pointer and binds the GLS containing that location to the target pointer in the environment ψ. Notice that Saturn does not have a primitive modeling explicit deallocation. Type casts to void* lift entries in the GLS to their addresses using the AddrOf mapping, and casts from void* find the concrete object of the appropriate type in the AddrOf mapping to replace addresses in the

GLS. Finally, the store rule models indirect assignment through a pointer, possibly involving field dereferences, by combining the results for each possible location the pointer could point to. The pointer is assumed to be non-null by adding $\neg\mathcal{G}_0$ to the current guard (recall $\mathcal{G}_0$ is the guard of null in every GLS). Notice that the store rule requires concrete locations in the GLS as one cannot assign through a pointer of type void*. Loading from a pointer is similar.

2.2.4.4 Pointer Arithmetic and Arrays

Due to its computational cost, the Saturn language and its transformation to Boolean formulas do not explicitly model pointer arithmetic and array operations in C. This is a potential source of both false positives and false negatives when analyzing properties that interact with such features. In the implementation of the Linux lock checker and the memory leak checker (Sections 2.5 and 2.6), we use a simple approximation which assumes that modifications to existing pointers yield fresh locations. Our experiments show that the approximation does not have a meterial impact on precision in practice. However, we expect that a more precise (and therefore expensive) algorithm would be needed to analyze pointer dependent properties such as buffer overruns.

2.2.5 Attributes

Another feature in Saturn is *attributes*, which are simply annotations associated with non-null Saturn locations (i.e., structs, integer variables, pointers, and addresses). We use the syntax o#attrname to denote the attrname attribute of object o.

The definition and encoding of attributes is the same as `struct` fields except that it does not require predeclaration, and attributes can be added during the analysis as needed. Similar to `struct` fields, attributes can also be accessed indirectly through pointers.

We use an example to illustrate attribute usage in analysis.

```
1  (*p)#escaped <- true;
2  q <- (void *) p;
3  assert ((*q)#escaped == (*p)#escaped);
```

In the example above, we use the store statement at line 1 to model the fact that the location pointed to by p has escaped. The advantage of using attributes here is that they are attached to addresses and preserved through pointer casts—thus the assertion at line 3 holds.

2.3 Discussions and Improvements

In this section, we discuss how our encoding reduces the size of satisfiability queries by achieving a form of program slicing (Section 2.3.1). We also discuss two improvements to our approach. The first (Section 2.3.2) concerns how we treat inputs of unknown shape to functions and the second (Section 2.3.3) is an optimization that greatly reduces the cost of guards.

2.3.1 Automatic Slicing

Program slicing is a technique to simplify a program by removing the parts that are irrelevant to the property of concern. Slicing is commonly done by computing control and data dependencies and preserving only the statements that the property depends on. We show that our encoding automatically slices a program and only uses clauses that the current SAT query requires.

Consider the following program snippet below:

```
if (x) y = a;
else y = b;
z = /* complex computation here */;
if (z) ... else ...;
assert(y < 5);
```

The computation of z is irrelevant to the property we are checking ($y < 5$). The variable y is data dependent on a and b and control dependent on x. Using the encoding rules in Section 2.2, we see that the encoding of $y < 5$ only involves the bits in x, a, and b, but not z, because the assign rule accounts for the data dependencies and the merge rule pulls in the control dependency. No extra constraints are included in the representation of y. In large programs, properties of interest often depend on a small portion of the code analyzed, therefore this design helps keep the size of SAT queries under control.

2.3.2 Lazy Construction of the Environment

A standard problem in modular program analysis systems is the modeling of the external environment. In particular, we need a method to model and track data structures used, but not created, by the code fragment being analyzed.

There is no consensus on the best solution to this problem. To the best of our knowledge, SLAM [7] and Blast [40] require manual construction of the environment. For example, to analyze a module that manipulates a linked list of locks defined elsewhere, these systems likely require a harness that populates an input list with locks. The problem is reduced as the target code-bases (e.g., Windows drivers in the case for SLAM) can often share a carefully crafted harness (e.g., a model for the Windows kernel) [6]. Nevertheless, the need to "close" the environment represents a substantial manual effort in the deployment of such systems.

Because we achieve scalability by computing function summaries, we must analyze a function independent of its calling context and still model its arguments. Our solution is similar in spirit to the lazy initialization algorithm described in [46] and, conceptually, to lazy evaluation in languages such as Haskell. Recall in Section 2.2, values of variables referenced but not created in the code, i.e., those from the external environment, are defined in the initial evaluation environment ψ_0. SATURN lazily constructs ψ_0 by calling a special function DefVal, which is supplied by the analysis designer and maps all external objects to a checker-specific estimation of their default values; ψ_0 is then defined as $\mathsf{DefVal}(v)$ for all v. Operationally, DefVal is applied on demand, when uninitialized objects are first accessed during symbolic evaluation. This allows us to model potentially unbounded data structures in the environment. Besides its role in defining the initial environment ψ_0, DefVal is also used to provide an approximation of the return values and side-effects of function calls (Section 2.5.3).

In our implementation, we model integers from the environment with a vector of unconstrained boolean variables. For pointers, we use the common assumption that distinct pointers from the environment do not alias each other. This can be modeled by a DefVal that returns a fresh location for each previously unseen pointer dereference.[4] A sound alternative would be to use a separate global alias analysis as part of the definition of ψ_0. Note once a pointer is initialized, SATURN performs an accurate path-sensitive intraprocedural analysis, including respecting alias relationships, on that pointer.

2.3.3 Using BDDs for Guards

Consider the following code fragment:

```
if (c) { ... } else { ... }
```

[4]In the implementation, $\mathsf{DefVal}(\mathsf{p})$ returns $\{\!|\ (\mathcal{G}, \mathsf{null}), (\neg\mathcal{G}, o)\ |\!\}$, where $\mathcal{G}$ is an unconstrained boolean variable, and o is a fresh object of the appropriate type. This allows us to model common data structures like linked lists and trees of arbitrary length or depth. A slightly smarter variant handles doubly linked lists and trees with parent pointers knowing one node in such a data structure.

```
    s;
```

After conversion to a control-flow graph, there are two paths reaching the statement `s` with guards `c` and ¬`c`. Thus the guard of `s` is `c` ∨ ¬`c`. Since guards are attached to every bit of every modeled location at every program point, it is important to avoid growth in the size of guards at every control-flow merge. One way to accomplish this task is to decompile the unrolled control flow graph into structured programs with only `if` statements, so that we know exactly where branch conditionals cancel. However, this approach requires code duplication in the presence of `goto`, `break`, and `continue` statements commonly found in C.

Our solution is to introduce an intermediate representation of guards using binary decision diagrams [8]. We give each condition (which may be a complex expression) a name and use a BDD to represent the boolean combination of all condition names that enable a program path. At control-flow merges we join the corresponding BDDs. The BDD join operation can simplify the representation of the boolean formula to a canonical form; for example, the join of the BDDs for `c` and ¬`c` is represented by `true`. In our encoding of a statement, we convert the BDD representing the set of conditions at that program point to the corresponding Boolean guard.

The simplification of guards also eliminates trivial control dependencies in the automatic slicing scheme described in Section 2.3.1. In the small example in that section, had we not simplified guards, the assertion would have been checked under the guard $(x \vee \neg x) \wedge (z \vee \neg z)$, which pulls in the otherwise irrelevant computation of z.

2.4 Building Modular Property Checkers with Saturn

The Saturn framework we have described so far can be applied directly to checking properties such as assertions. While other program behavior can be encoded and checked under the current scheme, there are two main limitations that prevent it from being applied to complex properties in large systems.

1. *Function calls.* Saturn, like many other SAT-based techniques, does not directly model function calls. A common solution among SAT-based assertion checkers is inlining. However, although we employ a number of optimizations in our transformation such as slicing, the exponential time cost of SAT-solving means that inlining will not be practical for large software systems.

2. *Execution environment.* Assertion checking commonly requires a closed program. However, many software systems are open programs whose environment is a complex combination of user input and component interdependencies. Modeling the environment for such programs often requires extensive manual effort that is both costly and error prone.

Our solution is based on Saturn's ability to not only *check* program properties, but also *infer* them by formulating SAT queries that can be solved efficiently. The latter ability solves the two problems mentioned above.

First, inference enables modular analyses[5] that scale. With appropriate abstractions, the checker can summarize a function's behavior with respect to a property into a concise summary. This summary, in turn, can be used in lieu of the full function body at the function's call sites, which prevents exponential growth in the cost of the analysis.

Secondly, by making general enough assumptions about the execution environment, summaries capture the behavior of a function under all (or, for error detection purposes, a common subset of) runtime scenarios, which alleviates the requirement of having to close the environment.

An added benefit of the modular approach is that it enables local reasoning during error inspection. Instead of following long error traces which may involve multiple function calls, human-readable function summaries give information about the assumptions made for each of the callees in the current function. Therefore, the user can focus on one function at a time when inspecting error reports. In our experience, we find it much easier to confirm errors and identify false positives with the help of function summaries.

Using the modular approach to analyzing functions, we briefly outline a four step process by which we construct property checkers under Saturn:

1. We model the property we intend to check with program constructs in Saturn. For example, finite state machines (FSM) can be encoded by attaching integer state fields to program objects to track their current states. State transitions are subsequently modeled with conditional assignments, and checking is done by ensuring that the error state is not reached at the end of the program—a task easily accomplished with SAT queries on the final program state.

[5]Here, modular analysis is defined in two senses: 1) the ability to infer and check open program modules independent of their usage; and 2) the ability to summarize results of analyzed modules so as to avoid redundant analysis.

2. The next step is to design the function summary representation. A good summary is one that is both concise for scalability and expressive enough to adequately describe the relevant properties of function behavior. Striking the right balance often takes several iterations of design. For example, in designing the FSM checking framework, we started with a simple summary that records the set of feasible state transitions across the function, but found it to be inadequate for Linux lock checking due to its interprocedural path-insensitivity. We observed that the output lock state often correlates with the return value of the function and remedied the situation by simply including the return value in our summary design.

3. The third step is to design an algorithm that infers and applies function summaries. As mentioned above, inference is done by automatically inserting SAT queries at appropriate program points. For example, we can infer the set of possible state transitions by querying, at the end of each function, the satisfiability of all possible combinations of input and output states. The feasible (i.e., satisfiable) subset of state transitions is included in the function summary.[6]

4. Finally, we run the checker on a number of real-world applications, and inspect the analysis results. During early design iterations, the results often point to inaccuracies in the property encoding (Step 1), inadequacies in the summary design (Step 2), or inefficiencies in the inference algorithm (Step 3). We use that as feedback to improve the checker in the next iteration.

Following this four step process, we designed and implemented two property checkers for large open source software: a Linux lock checker and a memory leak checker. We present the details of the checkers and experiments in the following two sections.

2.5 Case Study I: Checking Finite State Properties

Finite state properties are a class of specifications that can be described as certain program values passing through a finite set of states, over time, under specific conditions. Locking, where a lock can legally only go from the unlocked state to the locked state and then back to

[6]This is a simplification of the actual summary inference algorithm, which takes into account function side-effects and return-value state-transition correlations. We describe the full algorithm in Section 2.5.2.

the unlocked state, is a canonical example. These properties are also referred to as *temporal safety properties.*

In this section, we focus on finite state properties, and describe a summary-based interprocedural analysis that uses the Saturn framework to automatically check such properties. We start by defining a common naming scheme for shared objects between the caller and the callee (Section 2.5.1), which we use to define a general summary representation for finite state properties (Section 2.5.2). We then describe algorithms for applying (Section 2.5.3) and inferring (Section 2.5.4) function summaries in the Saturn framework. We describe our implementation of an interprocedural lock checker (Section 2.5.5) and end with experimental results (Section 2.5.6).

2.5.1 Interface Objects

In C, the two sides of a function invocation share the global name space but have separate local name spaces. Thus we need a common name space for objects referred to in the summary. Barring external channels and unsafe memory accesses, the two parties share values through global variables, parameters, and the function's result. Therefore, shared objects can be named using a path from one of these three roots.

We formalize this idea using *interface objects* (IObj) as common names for objects shared between caller and callee:

$$\mathsf{IObj}\ (l) ::= \mathsf{param}_i \mid \mathsf{global\_var} \mid \mathsf{ret\_val} \mid \ *l \mid l.f$$

Dependencies across function calls are expressed by interface expressions (IExpr) and conditions (ICond), which are defined respectively by replacing references to objects with interface objects in the definition of Expr and Cond (as defined in Figure 2.1 and extended in Figure 2.5).

To perform interprocedural analysis, we must map input interface objects to the names used in the function body, perform symbolic evaluation of the function, and map the final function state to the final state of the interface objects. Thus, we need two mappings to convert between interface objects and those in the native name space of a function:

$$[\![\cdot]\!]_{\mathsf{args}} : \mathsf{IObj} \rightarrow \mathsf{Obj}^{\mathsf{ext}} \text{ and } [\![\cdot]\!]^{-1}_{\mathsf{args}} : \mathsf{Obj} \rightarrow \mathsf{IObj}$$

Converting IObj's to native objects is straightforward. For function call $r = f(a_0, ..., a_n)$,

$$\begin{aligned}
[\![\mathsf{global}]\!]_{a_0 \ldots a_n} &= \mathsf{global} \\
[\![\mathsf{param}_i]\!]_{a_0 \ldots a_n} &= a_i \\
[\![\mathsf{ret\_val}]\!]_{a_0 \ldots a_n} &= r \\
[\![*l]\!]_{a_0 \ldots a_n} &= *([\![l]\!]_{a_0 \ldots a_n}) \\
[\![l.f]\!]_{a_0 \ldots a_n} &= ([\![l]\!]_{a_0 \ldots a_n}).f
\end{aligned}$$

Note that the result of the conversion is in $\mathsf{Obj}^{\mathsf{ext}}$, which is defined as Obj (Section 2.2) extended with pointer dereferences. The extra dereference operations can be transformed away by introducing temporary variables and explicit load/store operations.

The inverse conversion is more involved, since there may be multiple aliases of the same object in the program. We incrementally construct the $[\![\cdot]\!]^{-1}_{\mathsf{args}}$ mapping for objects accessed through global variables and parameters. For example, in

```
void f(struct str *p) {
  spin_lock(&p–>lock);
}
```

the corresponding interface object for p is param_0, since it is defined as the first formal parameter of f. Recall that the object pointed to by p→lock is lazily instantiated when p is dereferenced by calling DefVal(p) (see Section 2.3.2). As part of the instantiation, we initialize every field of the struct (*p), and compute the appropriate IObj for each field at that time. Specifically, the interface object for p→lock is $(*\mathsf{param}_0).\mathsf{lock}$.

The conversion operations extend to interface expressions and conditionals. For brevity, name space conversions for objects, expressions, and conditionals are mostly kept implicit in the discussion below.

2.5.2 Function Summary Representation

The language for expressing finite state summaries is given in Figure 2.7. Each function summary is a four-tuple consisting of:

- a set of input propositions P_{in},
- a set of output propositions P_{out},
- a set of interface objects M, which may be modified during the function call, and

$$\begin{array}{rl}
\textsf{FSM States} & \mathcal{S} = \{\textsf{Error}, s_1, \ldots, s_n\} \\
\textsf{Summaries} & \Sigma = \langle P_{in}, P_{out}, M, R\rangle \\
\text{where} & P_{in} = \{p_1, \ldots, p_n\} \quad p_i \in \textsf{ICond}, \\
 & P_{out} = \{q_1, \ldots, q_n\} \quad q_i \in \textsf{ICond}, \\
 & M \subseteq \textsf{IObj}, \text{ and} \\
 & R \subseteq \textsf{IObj} \times 2^{|P_{in}|} \times \mathcal{S} \times 2^{|P_{out}|} \times \mathcal{S}
\end{array}$$

Figure 2.7: Function summary representation.

- a relation R summarizing the FSM behavior of the function.

The checker need only supply the set of FSM states and the set of input and output propositions (i.e. $\mathcal{S}$, P_{in}, and P_{out}); both M and R are computed automatically for each function by SATURN (see Section 2.5.4).

The FSM behavior of a function call is modeled as a set of state transitions of one or more interface objects. These transitions map input states to output states based on the values of a set of input (P_{in}) and output (P_{out}) propositions. The state transitions are given in the set R. Each element in R is a five tuple: $(\textsf{sm}, \textsf{incond}, s, \textsf{outcond}, s')$, which we describe below:

- $\textsf{sm} \in \textsf{IObj}$ is the object whose state is affected by the transition relationship. In the lock checker, sm identifies the accessed lock objects, as a function may access more than one lock during its execution.

- $\textsf{incond} \in 2^{|P_{in}|}$ denotes the pre-condition of the FSM transition: $(\bigwedge_{i \in \textsf{incond}} p_i) \wedge (\bigwedge_{i \notin \textsf{incond}} \neg p_i)$ where $\{p_1, \ldots, p_n\} = P_{in}$. It specifies one of the 2^n possible valuations of the input propositions, and is evaluated on entry to the function.

- $s \in S$ is the initial state of sm in the state transition.

- $\textsf{outcond} \in 2^{|P_{out}|}$ is similarly defined as incond and denotes the output condition of the transition. outcond is evaluated on exit.

- $s' \in S$ is the state of sm after the transition.

Figure 2.8 presents the summary of three sample locking functions: spin_lock, spin_trylock, and complex_wrapper. The function complex_wrapper captures some of the more complicated locking behavior in Linux. Nevertheless, given appropriate input and output propositions

```
void complex_wrapper(spinlock_t *l, int flag, int *success)
{
    /* spin_trylock returns non-zero on successful
       acquisition of the lock; 0 otherwise. */
    if (flag) *success = spin_trylock(l);
    else { spin_unlock(l); *success = 1; }
}
```

States: $S = \{\mathsf{Error} = 0, \mathsf{Locked} = 1, \mathsf{Unlocked} = 2\}$
Summary: $\Sigma = \langle M, R, P_{in}, P_{out}\rangle$

spin_lock :
Input: $P_{in} = \{\}$ $\quad P_{out} = \{\}$
Output: $M = \{*\mathsf{param}_0\}$
$R = \{\ (*\mathsf{param}_0, \_\,, \mathsf{Unlocked}, \_\,, \mathsf{Locked}),$
$\quad (*\mathsf{param}_0, \_\,, \mathsf{Locked}, \_\,, \mathsf{Error})\}$

spin_trylock :
Input: $P_{in} = \{\}$ $\quad P_{out} = \{\mathsf{lift}_c(\mathsf{ret\_val})\}$
Output: $M = \{*\mathsf{param}_0, \mathsf{ret\_val}\}$
$R = \{\ (*\mathsf{param}_0, \_\,, \mathsf{Unlocked}, \mathsf{true}, \mathsf{Locked}),$
$\quad (*\mathsf{param}_0, \_\,, \mathsf{Unlocked}, \mathsf{false}, \mathsf{Unlocked}),$
$\quad (*\mathsf{param}_0, \_\,, \mathsf{Locked}, \_\,, \mathsf{Error})\}$

complex_wrapper :
Input: $P_{in} = \{\mathsf{lift}_c(\mathsf{param}_1)\}$ $\quad P_{out} = \{\mathsf{lift}_c(*\mathsf{param}_2)\}$
Output: $M = \{*\mathsf{param}_0, *\mathsf{param}_2\}$
$R = \{\ (*\mathsf{param}_0, \mathsf{true}, \mathsf{Locked}, \_\,, \mathsf{Error})$
$\quad (*\mathsf{param}_0, \mathsf{true}, \mathsf{Unlocked}, \mathsf{true}, \mathsf{Locked})$
$\quad (*\mathsf{param}_0, \mathsf{true}, \mathsf{Unlocked}, \mathsf{false}, \mathsf{Unlocked})$
$\quad (*\mathsf{param}_0, \mathsf{false}, \mathsf{Unlocked}, \mathsf{true}, \mathsf{Error})$
$\quad (*\mathsf{param}_0, \mathsf{false}, \mathsf{Locked}, \mathsf{true}, \mathsf{Unlocked})\}$

Figure 2.8: Sample function summaries for the locking property.

(i.e., P_{in} and P_{out}), we are able to express (and automatically infer) its behavior using our summary representation (i.e., M and R). We describe how function summaries are inferred and used in the following subsections.

2.5.3 Summary Application

This subsection describes how the summary of a function is used to model its behavior at a call site. For a given function invocation $f(a_0, \ldots, a_n)$, we encode the call into a set of statements simulating the observable effects of the function. The encoding, given in Figure 2.9, is composed of two stages:

Assumptions

$\Sigma(f) = \langle P_{in}, P_{out}, M, R\rangle$

$$\text{where } \begin{cases} P_{in} = \{p_1, \ldots, p_m\} \\ P_{out} = \{q_1, \ldots, q_n\} \\ M = \{o_1, \ldots, o_k\} \\ R = \{(\mathsf{sm}_1, \mathsf{incond}_1, s_1, \mathsf{outcond}_1, s'_1), \ldots, \\ \qquad (\mathsf{sm}_l, \mathsf{incond}_l, s_l, \mathsf{outcond}_l, s'_l)\} \end{cases}$$

Instrumentation

```
1: (* Stage 1: Preparation *)
2: (* save useful program states *)
3: $\widehat{p}_1 \leftarrow p_1; \ldots; \widehat{p}_n \leftarrow p_m;$
4: $\widehat{\mathsf{sm}}_1 \leftarrow \mathsf{sm}_1; \ldots; \widehat{\mathsf{sm}}_l \leftarrow \mathsf{sm}_l;$
5: (* account for the side-effects of $f$ *)
6: $o_1 \leftarrow \mathsf{unknown}(\tau_{o_1}); \ldots; o_k \leftarrow \mathsf{unknown}(\tau_{o_k});$
7: (* save the values of output propositions *)
8: $q'_1 \leftarrow q_1; \ldots; q'_n \leftarrow q_n;$
9: (* rule out infeasible comb. of incond and outcond *)
10: $\mathsf{assume}(\bigvee_i(\mathsf{sm}_i = s_i \wedge \mathsf{incond}_i \wedge \mathsf{outcond}_i));$
11:
12: (* Stage 2: Transitions *)
13: (* record state transitions after the function call *)
14: if $(\widehat{\mathsf{sm}}_1 = s_1 \wedge \mathsf{incond}_1 \wedge \mathsf{outcond}_1)$ $\mathsf{sm}_1 \leftarrow s'_1;$
15:     ...
16: if $(\widehat{\mathsf{sm}}_l = s_l \wedge \mathsf{incond}_l \wedge \mathsf{outcond}_l)$ $\mathsf{sm}_l \leftarrow s'_l;$
```

Figure 2.9: Summary application.

1. In the first stage, we save the values of relevant program states before and after the call (lines 3-4 and 8 in Figure 2.9), and account for the side effects of the function by conservatively assigning unknown values to objects in the modified set M (line 6). Relevant values before the call include all input propositions p_i, and the current states (sm_i) of the interface objects mentioned in the transition relation R. Relevant values after the call include all output states q_i. We then use an assume statement to rule out impossible combinations of input and output propositions (line 10; e.g., some functions always return a non-NULL pointer).

2. In the second stage, we process the state transitions in R by first testing their activation conditions, and, when satisfied, carrying out the transitions (line 14-16). The proposition incond denotes the condition $(\bigwedge_{i \in \mathsf{incond}} \widehat{p}_i) \wedge (\bigwedge_{i \notin \mathsf{incond}} \neg\widehat{p}_i)$; the condition

$$
\begin{aligned}
&P_{in} = \{p_1, \ldots, p_m\}\\
&P_{out} = \{q_1, \ldots, q_n\}\\
&M = \{v \mid \mathsf{is\_satisfiable}(\psi_0(v) \neq \psi(v))\}\\
&R = \{\ (\mathsf{sm}, \mathsf{incond}, s, \mathsf{outcond}, s') \mid \mathsf{is\_satisfiable}(\psi_0(\mathsf{sm} = s) \wedge \psi_0(\mathsf{incond}) \wedge\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad \psi(\mathsf{outcond}) \wedge \psi(\mathsf{sm} = s'))\ \}
\end{aligned}
$$

Figure 2.10: Summary inference.

for outcond is symmetric. Notice that since incond and outcond are a valuation of all input and output propositions, no two transitions on the same state machine should be enabled simultaneously. Violations of this property can be attributed to either an inadequate choice of input and output propositions, or a bug in the program (e.g., Type B errors in the Linux lock checker–Section 2.5.5).

There is one aspect of the encoding that is left unspecified in the description, which is the unknown values used to model the side-effects of the function call. For integer values, we use the rule for unknown and conservatively model these values with a set of unconstrained boolean variables. For pointers, we extend the DefVal operator described in Section 2.3.2 to obtain a checker-specified estimation of the shape of the object being pointed to. The current implementation uses fresh locations for modified pointers.

2.5.4 Summary Inference

This section describes how we compute the summary of a function after analysis. Before we proceed, we first state two assumptions about the translation from C to Saturn's intermediate language:

1. We assume that each function has one unique exit block. In case the function has multiple return statements, we add a dummy exit block linked to all return sites. The exit block is analyzed last (see Section 2.2) and the environment ψ at that point encodes all paths from function entry to exit. Summary inference is carried out after analyzing the exit block.

2. We model return statements in C by assigning the return value to a special object rv, and $[\![\mathsf{rv}]\!]^{-1}_{\mathsf{args}} = \mathsf{ret\_val}$.

Figure 2.10 gives the summary inference algorithm. The input to the algorithm is a set of input (P_{in}) and output (P_{out}) propositions. The inference process involves a series of queries to the SAT solver based on the initial (ψ_0) and final state (ψ) to determine: (1) the set of modified objects M, and (2) the set of transition relationships R. In computing M and R, we use a shorthand $\psi(x)$ to denote the valuation of x under environment ψ.

The summary inference algorithm proceeds as follows. Intuitively, modified objects are those whose valuation may be different under the initial environment ψ_0 and the final environment ψ. We compute M by iterating over all interface objects v and use the SAT solver to determine whether the values may be different or not.

The transition set R is computed by enumerating all relevant interface objects (e.g., locks in the lock checker) in the function and all combinations of input and output propositions. We again use the SAT solver to determine whether a transition under a particular set of input and output propositions is feasible.

As the reader may notice, summary inference requires many SAT queries and can be computationally expensive when solved individually. Fortunately, these queries share a large set of common constraints encoding the function control and data flow. In fact, the only difference among the queries are constraints that describe the different combinations of input/output propositions and initial/final state pairs for each state machine. We exploit this fact by taking advantage of incremental solving capabilities in modern SAT solvers. Incremental solving algorithms share and reuse information learned (e.g., using conflict clauses) in the common parts of the queries and can considerably speed up SAT solving time for similar queries. In practice, we observe that SAT queries in Saturn typically complete in under one second.

2.5.5 A Linux Lock Checker

In this section, we use the FSM checking framework described above to construct a lock checker for the Linux kernel. We start with some background information, and list the challenges we encountered in trying to detect locking bugs in Linux. We then describe the lock checker we have implemented in the Saturn framework.

The Linux kernel is a widely deployed and well-tested core of the Linux operating system. The kernel is designed to scale to an array of multiprocessor platforms and thus is inherently concurrent. It uses a variety of locking mechanisms (e.g., spin locks, semaphores, read/write locks, primitive compare and swap instructions, etc.) to coordinate concurrent accesses of kernel data structures. For efficiency reasons, most of the code in the kernel runs in supervisor mode, and synchronization bugs can thus cause crashes or hangs that result in data loss and system down time. For this reason, locking bugs have received the attention of a number of research and commercial checking and verification efforts.

The behavior of locks (a.k.a. mutexes) is naturally expressed as a finite state property with three states: Locked, Unlocked, and Error. The lock operation can be modeled as two transitions: from Unlocked to Locked, and Locked to Error (unlock is similar). There are a few challenges that a checker must overcome to model locking behavior in Linux:

- **Aliasing**. In Linux, locks are passed by reference (i.e., by pointers in C). One immediate problem is the need to deal with pointer aliasing. Previous projects have taken a wide variety of approaches to dealing with aliasing. For example, CQual employs a number of techniques to infer non-aliasing relationships to help refine the results from the alias analysis [3]. MC [35] assumes non-aliasing among all pointers, which helps reduce false positives, but also limits the checking power of the tool.

- **Heap Objects**. In fine grained locking, locks are often embedded in heap objects. These objects are stored in the heap and passed around by reference. To detect bugs involving heap objects, a reasonable model of the heap needs to be constructed (recall Section 2.3.2). The need to write "harnesses" that construct the checking environment has proven to be a non-trivial task in traditional model checkers [6].

- **Path Sensitivity**. The state machine for locks becomes more complex when we consider *trylocks*. Trylocks are lock operations that can fail; the caller must check the return value of trylock operations to determine whether the operations have succeeded or not. Besides trylocks, some functions intentionally exit with locks held on error paths and expect their callers to carry out error recovery and cleanup work. These constructs are used extensively in Linux. In addition, one common usage scenario in Linux is the following:

  ```
  if (x) spin_lock(&l);
  ...;
  if (x) spin_unlock(&l);
  ```

  Some form of path sensitivity is necessary to handle these cases.

- **Interprocedural Analysis**. As we show in Section 2.5.6, a large portion of synchronization errors arise from misunderstanding of function interface constraints. The presence of more than 600 lock/unlock/trylock wrappers further complicates the analysis. Imprecision in the intraprocedural analysis is amplified in the interprocedural phase, so we believe a precise interprocedural analysis is important in the construction of a lock checker.

Our lock checker is based on the framework described above. States are defined as usual: $\{\mathsf{Locked}, \mathsf{Unlocked}, \mathsf{Error}\}$. To accurately model trylocks, we define $P_{out} = \{\mathsf{lift}_c(\mathsf{ret\_val})\}$ for functions that return integers or pointers. Tracking this proposition in summaries is also adequate for modeling functions that exit in different lock states depending on whether the return value is 0 (null) or not. We define P_{out} to be the empty set for functions of type void; P_{in} is defined to be the empty set.

We detect two types of locking errors in Linux:

- **Type A: double locking/unlocking**. These are functions that may acquire or release the same lock twice in a row. The summary relationship R of such functions contains two transitions on the same lock: one leads from the Locked state to Error, and the other from the Unlocked state to Error. This signals an internal inconsistency in the function—no matter what state the lock is in on entry to the function, there is a path leading to the error state.
- **Type B: ambiguous return state**. These are functions that may exit in both Locked and Unlocked states with no observable difference (w.r.t. P_{out}, which is $\mathsf{lift}_c(\mathsf{ret\_val})$) in the return value. These bugs are commonly caused by missed operations to restore lock states on error paths.[7]

2.5.6 Experimental Results

We have implemented the lock checker described in Section 2.5.5 in the SATURN framework. The checker models locks in Linux (e.g., objects of type `spinlock_t, rwlock_t, rw_semaphore`, and `semaphore`) using the state machines defined in Section 2.5. When analyzing a function, we retrieve the lock summaries of its callees and use the algorithm described in Section 2.5.3 to simulate their observable effects. At the end of the analysis, we compute a summary for the current function using the algorithm described in Section 2.5.4 and store it in the summary database for future use.

[7] One can argue that Type B errors are rather a manifestation of the restricted sets of predicates used for the analysis; a more precise way of detecting these bugs is to allow ambiguous output states in the function summary, and report bugs in calling contexts where only one of the output states is legal. Practically, however, we find that this restriction is a desirable feature that allows us to exploit domain knowledge about lock usage in Linux, and thus helps the analysis to pinpoint more accurately the root cause of a bug.

Type	**Count**
Num. of Files	12455
Total Line Count	4.8 million LOC
Total Num. Func.	63850
Lock Related Func.	23458
Running time	19h40m CPU time
Approx. LOC/sec	67

Table 2.1: Performance statistics on a single processor Pentium IV 3.0G desktop with 1GB memory.

The order of analysis for functions in Linux is determined by topologically sorting the static call graph of the Linux kernel. Recursive function calls are represented by strongly connected components (SCC) in the call graph. During the bottom up analysis, functions in SCCs are analyzed once in an arbitrary order, which might result in imprecision in inferred summaries. A more precise approach would require unwinding the recursion as we do for loops, until a fixed point is reached for function summaries in the SCC. However, our experiments indicate that recursion does not seem to impact the precision of inferred lock summaries, and therefore we adopt the simpler approach in our implementation.

We start the analysis by seeding the lock summary database with manual specifications of around 40 lock, unlock and trylock primitives in Linux. Otherwise the checking process is fully automatic: our tool works on the unmodified source tree and requires no human guidance during the analysis.

We ran our lock checker on the then latest release of the Linux kernel source tree (v2.6.5). Performance statistics of the experiment are tabulated in Table 2.1. All experiments were done on a single processor 3.0GHz Pentium IV computer with 1G of memory. Our tool parsed and analyzed around 4.8 million lines of code in 63,850 functions in under 20 hours. Function side-effect computation is not implemented in the version of the checker reported here. Loops are unrolled a maximum of two iterations based on the belief that most double lock errors manifest themselves by the second iteration. We implemented an optimization that skips functions that have no lock primitives and do not call any other functions with non-trivial lock summaries. These functions are automatically given the trivial “No-Op” summary. We analyzed the remaining 23,927 lock related functions, and stored their summaries in a GDBM database.

We set the memory limit for each function to 700MB to prevent thrashing and the

Type	Bugs	FP	Warnings	Accuracy (Bug/Warning)
A	134	99	233	57%
B	45	22	67	67%
Total	179	121	300	60%

Table 2.2: Number of bugs found in each category.

Type	A	B	Total
Interprocedural	108	27	135
Intraprocedural	26	18	44
total	134	45	179

Table 2.3: Breakdown of intra- and inter-procedural bugs.

CPU time limit to 90 seconds. Our tool failed to analyze 27 functions, some of which were written in assembly; the rest failed due to internal failures of the tool. The tool also failed to terminate on 442 functions in the kernel, largely due to resource constraints, with a small number of them due to implementation bugs in our tool. In every case we have investigated, resource exhaustion is caused by exceeding the capacity of an internal cache in SATURN itself and not in the SAT solver. This represents a failure rate of $< 2\%$ on the lock-related functions.

The result of the analysis consists of a bug report of 179 previously unknown errors and a lock summary database for the entire kernel, which we describe in the subsections below.

2.5.6.1 Errors and False Positives

As described in Section 2.5.5, we detect two types of locking errors in Linux: double lock/unlock (Type A) and ambiguous output states (Type B). We tabulate the bug counts in Table 2.2.

The bugs and false positives are classified by manually inspecting the error reports generated by the tool. One caveat of this approach is that errors we diagnose may not be actual errors. To counter this, we only flag reports as errors that we are reasonably sure about. We have several years of experience examining Linux bugs, so the number of misdiagnosed errors is expected to be low.

	Type A	Type B	Total
Propositions	26	16	42
Lock Assertions	21	4	25
Semaphores	22	0	22
Saturn Lim.	18	1	19
Readlocks	7	0	7
Others	5	1	7
Total	99	22	121

Table 2.4: Breakdown of false positives.

Table 2.3 further breaks down the 179 bugs into intraprocedural versus interprocedural errors. We observe that more than three quarters of diagnosed errors are caused by misunderstanding of function interface constraints.

Table 2.4 classifies the false positives into six categories. The biggest category of false positives is caused by inadequate choice of propositions P_{in} and P_{out}. In a small number of widely called utility functions, input and output lock states are correlated with values passed in/out through a parameter, instead of the return value. To improve this situation, we need to detect the relevant propositions either by manual specification or by using a predicate abstraction algorithm similar to that used in SLAM [7] or BLAST [39]. Another large source of false positives is an idiom that uses trylock operations as a way of querying the current state of the lock. This idiom is commonly used in assertions to ensure that a lock is held at a certain point. We believe a better way to accomplish this task is to use the lock querying functions, which we model precisely in our tool. Fortunately, this usage pattern only occurs in a few macros, and can be easily identified during inspection. The third largest source of false positives is counting semaphores. Depending on the context, semaphores can be used in Linux either as locks (with `down` being lock and `up` being unlock) or resource counters. Our tool treats all semaphores as locks, and therefore may misflag consecutive down/up operations as double lock/unlock errors. The remaining false positives are due to readlocks (where double locks are OK), and unmodeled features such as arrays.

Figure 2.11 shows a sample interprocedural Type A error found by Saturn, where `sscape_coproc_close` calls `sscape_write` with &devc→lock held. However, the first thing `sscape_write` does is to acquire that lock again, resulting in a deadlock on multiprocessor systems. Figure 2.12 gives a sample intraprocedural Type B error. There are two places where the function exits with return value -EBUSY: one with the lock held, and the other

```
1  static void sscape_coproc_close(void *dev_info, int sub_device)
2  {
3          spin_lock_irqsave(&devc->lock,flags);
4          if (devc->dma_allocated) {
5                  sscape_write(devc, GA_DMAA_REG, 0x20); // bug here
6                  ...
7          ...
8  }
9
10 static void sscape_write(struct sscape_info *devc, int reg, int data)
11 {
12         ...
13         spin_lock_irqsave(&devc->lock,flags); // acquires the lock
14 }
```

Figure 2.11: An interprocedural Type A error found in `sound/oss/sscape.c`.

with the lock not held. The programmer has forgotten to release the lock before returning at line 13.

We have filed the bug reports to the Linux Kernel Mailing List (LKML) and received confirmations and patches for a number of reported errors. To the best of our knowledge, Saturn is by far the most effective bug detection tool for Linux locking errors.

2.5.6.2 The Lock Summary Database

Synchronization errors are known to be difficult to reproduce and debug dynamically. To help developers diagnose reported errors, and also to better understand the often subtle locking behavior in the kernel (e.g., lock states under error conditions), we built a web interface for the Linux lock summary database generated during the analysis.

Our own experience with the summary database has been pleasant. During inspection, we use the database extensively to match up the derived summary with the implementation code to confirm errors and identify false positives. For each summary query, we enter the name of a specific function. The system automatically retrieves the summary of that function and outputs it in a human readable form. An example is given below:

```
Function summary for 'sound/oss/gus_midi.c:dump_to_midi'
{
  gus_lock(global): [unlocked -> unlocked]
}
```

```
int i2o_claim_device(struct i2o_device *d,
                     struct i2o_handler *h)
{
        down(&i2o_configuration_lock);
        if (d->owner) {
                ...
                up(&i2o_configuration_lock);
                return -EBUSY;
        }
        ...
        if(...) {
                ...
                return -EBUSY;
        }
        up(&i2o_configuration_lock);
        return 0;
}
```

Figure 2.12: An intraprocedural Type B error found in `drivers/message/i2o/i2o_core.c`.

In our experience the generated summaries accurately model the locking behavior of the function being analyzed. In fact, shortly after we filed these bugs, we logged more than a thousand queries to the summary database from the Linux community.

The summary database also reveals interesting facts about the Linux kernel. To our surprise, locking behavior is far from simple in Linux. More than 23,000 of the ∼63,000 functions in Linux directly or indirectly operate on locks. In addition, 8873 functions access more than one lock. There are 193 lock wrappers, 375 unlock wrappers, and 36 functions where the output state correlates with the return value. Furthermore, more than 17,000 functions directly or indirectly require locks to be in a particular state on entry.

We believe Saturn is the first automatic tool that successfully understands and documents any aspect of locking behavior in code the size of Linux.

2.6 Case Study II: The Leak Detector

In this section, we present a static memory leak detector based on the path sensitive pointer analysis in Saturn. We target one important class of leaks, namely neglecting to free a newly allocated memory block before all its references go out of scope. These bugs are commonly found in error handling paths, which are less likely to be covered during testing.

This second study is interesting in its own right as an effective memory leak detector, and as evidence that Saturn can be used to analyze a variety properties.

The rest of the section is organized as follows: Section 2.6.1 gives examples illustrating the targeted class of bugs and the analysis techniques required. We briefly outline the detection algorithm in Section 2.6.2 and give details in Sections 2.6.3, 2.6.4, and 2.6.5. Handling the unsafe features of C is described in Section 2.6.7. Section 2.6.8 describes a parallel client/server architecture that dramatically improves analysis speed. We end with experimental results in Section 2.6.9.

2.6.1 Motivation and Examples

Below we show a typical memory leak found in C code:

```
p = malloc(...); ...
if (error_condition) return NULL;
return p;
```

Here, the programmer allocates a block of memory and stores the reference in p. Under normal conditions p is returned to the caller, but in case of an error, the function returns NULL and the new location is leaked. The problem is fixed by inserting the statement free(p) immediately before the error return.

Our goal is to find these errors automatically. We note that leaks are always a flow-sensitive property, but sometimes are path-sensitive as well. The following example shows a common usage where a memory block is freed when its reference is non-NULL.

```
if (p != NULL)
    free(p);
```

To avoid false positives in their path insensitive leak detector, Heine *et. al.* [38] transform this code into:

```
if (p != NULL)
    free(p);
else
    p = NULL;
```

The transformation handles the idiom with a slight change of program semantics (i.e., the extra NULL assignment to p). However, syntactic manipulations are unlikely to succeed in more complicated examples:

```
char fastbuf[10], *p;
if (len < 10)
    p = fastbuf;
else
    p = (char *)malloc(len);
...
if (p != fastbuf)
    free(p);
```

In this case, depending on the length of the required buffer, the programmer chooses between a smaller but more efficient stack-allocated buffer and a larger but slower heap-allocated one. This optimization is common in performance critical code such as Samba and the Linux kernel and a fully path sensitive analysis is desirable in analyzing such code.

Another challenge to the analysis is illustrated by the following example:

```
p->name = strdup(string);
push_on_stack(p);
```

To correctly analyze this code, the analysis must infer that strdup allocates new memory and that push_on_stack adds an external reference to its first argument p and therefore causes (*p).name to escape. Thus, an interprocedural analysis is required. Without abstraction, interprocedural program analysis is prohibitively expensive for path sensitive analyses such as ours. As with the lock checker, we use a summary-based approach that exploits the natural abstraction boundary at function calls. For each function, we use SAT queries to infer information about the function's memory behavior and construct a summary for that function. The summary is designed to capture the following two properties:

1. whether the function is a memory allocator, and

2. the set of escaping objects that are reachable from the function's parameters.

We show how we infer and use such function summaries in Section 2.6.5.

2.6.2 Outline of the Leak Checker

This subsection discusses several key ideas behind the leak checker. First of all, we observe that pointers are not all equal with respect to memory leaks. Consider the following example:

```
(*p).data = malloc(...);
return;
```

The code contains a leak if p is a local variable, but not if p is a global or a parameter. In the case where *p itself is newly allocated in the current procedure, (*p).data escapes only if object *p escapes (except for cases involving cyclic structures; see below). In order to distinguish between these cases, we need a concept called *access paths* (Section 2.6.3) to track the paths through which objects are accessed from both inside and outside (if possible) the function body. We describe details about how we model object accessibility in Section 2.6.4.

References to a new memory location can also escape through means other than pointer references:

1. memory blocks may be deallocated;

2. function calls may create external references to newly allocated locations;

3. references can be transferred via program constructs in C that currently are not modeled in SATURN (e.g., by decomposing a pointer into a page number and a page offset, and reconstructing it later).

To model these cases, we instrument every allocated memory block with a boolean escape attribute whose default value is false. We set the escape attribute to true whenever we encounter one of these three situations. A memory block is not considered leaked when its escape attribute is set.

One final issue that requires explicit modeling is that malloc functions in C might fail. When it does, malloc returns null to signal a failed allocation. This situation is illustrated in Section 2.6.1 and requires special-case handling in path insensitive analyses. We use a boolean valid attribute to track the return status of each memory allocation. The attribute is non-deterministically set at each allocation site to model both success and failure scenarios. For a leak to occur, the corresponding allocation must originate from a successful allocation and thus have its valid attribute set to true.

2.6.3 Access Paths and Origins

This subsection extends the interface object concept introduced in Section 2.5.1 to track and manipulate the path through which objects are first accessed. Following standard literature on alias and escape analysis, we call the revised definition *access paths*. As shown

$$\begin{aligned}
\mathsf{Params} &= \{\mathsf{param}_0, \ldots, \mathsf{param}_{n-1}\} \\
\mathsf{Origins}\ (r) &::= \{\mathsf{ret\_val}\}\ \cup \mathsf{Params} \cup \\
&\qquad \mathsf{Globals} \cup \mathsf{NewLocs} \cup \mathsf{Locals} \\
\mathsf{AccPath}\ (\pi) &::= r \mid \pi.f \mid\ * \pi \\
\\
\mathsf{PathOf} &: \mathsf{Loc} \rightarrow \mathsf{AccPath} \\
\mathsf{RootOf} &: \mathsf{AccPath} \rightarrow \mathsf{Origins}
\end{aligned}$$

Figure 2.13: Access paths.

in the Section 2.6.2, access path information is important in defining the escape condition for memory locations.

Figure 2.13 defines the representation and operations on access paths, which are interface objects (see Section 2.5.1) extended with Locals and NewLocs. Objects are reached by field accesses or pointer dereferences from five origins: global and local variables, the return value, function parameters, and newly allocated memory locations. We represent the path through which an object is accessed first with AccPath.

PathOf maps objects (and polymorphic locations) to their access paths and access path information is computed by recording object access paths used during the analysis. The RootOf function takes an access path and returns the object from which the path originates.

We illustrate these concepts using the following example:

```
struct state { void *data; };
void *g;
void f(struct state *p)
{
  int *q;
  g = p->data;
  q = g;
  return q; /* rv = q */
}
```

Table 2.5 summarizes the objects reached by the function, their access paths and origins. The origin and path information indicates how these objects are first accessed and is used in defining the leak conditions in Section 2.6.4.

Object	AccPath	RootOf
p	param_0	param_0
$*p$	$*\mathsf{param}_0$	param_0
$(*p).data$	$(*\mathsf{param}_0).data$	param_0
$*(*p).data$	$*(*\mathsf{param}_0).data$	param_0
g	global_g	global_g
q	local_q	local_q
rv	ret_val	ret_val

Table 2.5: Objects, access paths, and access origins in the sample program.

$$\frac{\psi(p) = \{\!|\ (\mathcal{G}_0, l_0), \ldots, (\mathcal{G}_{n-1}, l_{n-1})\ |\!\}}{\mathsf{PointsTo}(p, l) = \begin{cases} \mathcal{G}_i & \text{if } \exists i \text{ s.t. } \mathsf{AddrOf}(l)\text{=}\mathsf{AddrOf}(l_i) \\ \mathsf{false} & \text{otherwise} \end{cases}} \quad \textsf{points-to}$$

Excluded Set: $\mathcal{X} \subseteq \mathsf{Origins} - (\mathsf{Globals} \cup \mathsf{Locals})$

$$\frac{\mathsf{RootOf}(p) \in \mathsf{Locals} \cup \mathcal{X}}{\mathsf{EscapeVia}(l, p, \mathcal{X}) = \mathsf{false}} \quad \textsf{via-local}$$

$$\frac{\mathsf{RootOf}(p) \in \mathsf{Globals}}{\mathsf{EscapeVia}(l, p, \mathcal{X}) = \mathsf{PointsTo}(p, l)} \quad \textsf{via-global}$$

$$\frac{\mathsf{RootOf}(p) = (\mathsf{Params} \cup \{\mathsf{ret\_val}\}) - \mathcal{X}}{\mathsf{EscapeVia}(l, p, \mathcal{X}) = \mathsf{PointsTo}(p, l)} \quad \textsf{via-interface}$$

$$\frac{l' = \mathsf{RootOf}(p) \quad l' \in (\mathsf{NewLocs} - \mathcal{X})}{\mathsf{EscapeVia}(l,\ p, \mathcal{X}) = \mathsf{PointsTo}(p, l) \wedge \mathsf{Escaped}(l', \mathcal{X} \cup \{l\})} \quad \textsf{via-newloc}$$

$$\mathsf{Escaped}(l, \mathcal{X}) = [\![l\#\mathsf{escaped}]\!]_\psi \vee \bigvee_p \mathsf{EscapeVia}(l, p, \mathcal{X}) \quad \textsf{escaped}$$

$$\mathsf{Leaked}(l, \mathcal{X}) = [\![l\#\mathsf{valid}]\!]_\psi \wedge \neg\mathsf{Escaped}(l, \mathcal{X}) \quad \textsf{leaked}$$

* $\mathsf{RootOf}(p)$ is a shorthand for $\mathsf{RootOf}(\mathsf{PathOf}(p))$.

Figure 2.14: Memory leak detection rules.

2.6.4 Escape and Leak Conditions

Figure 2.14 defines the rules we use to find memory leaks and construct function summaries. As discussed in Section 2.5.4, we assume that there is one unique exit block in each function's control flow graph. We apply the leak rules at the end of the exit block, and the implicitly defined environment ψ in the rules refers to the exit environment.

In Figure 2.14, the $\mathsf{PointsTo}(p, l)$ function gives the condition under which pointer p points to location l. The result is simply the guard associated with l if it occurs in the GLS of p and false otherwise. Using the PointsTo function, we are ready to define the escape relationships Escaped and EscapeVia.

Ignoring the exclusion set $\mathcal{X}$ for now, $\mathsf{EscapeVia}(l, p, \mathcal{X})$ returns the condition under which location l escapes through pointer p. Depending on the origin of p, EscapeVia is defined by four rules via-* in Figure 2.14. The simplest of the four rules is via-local, which stipulates that location l cannot escape through p if p's origin is a local variable, since the reference is lost when p goes out of scope at function exit.

The rule via-global handles the case where p is accessible through a global variable. In this case, l escapes when p points to l, which is described by the condition $\mathsf{PointsTo}(p, l)$. The case where a location escapes through a function parameter is treated similarly in the via-interface rule.

The rule via-newloc handles the case where p is a newly allocated location. Again ignoring the exclusion set $\mathcal{X}$, the rule stipulates that a location l escapes if p points to l and the origin of p, which is itself a new location, in turn escapes.

However, the above statement is overly generous in the following situation:

```
s = malloc(...); /* creates new location l' */
s->next = malloc(...);          /* creates l */
s->next->prev = s;   /* circular reference */
```

The circular dependency that l escapes if l' does, and vice versa, can be satisfied by the constraint solver by assuming both locations escape. To find this leak, we prefer a solution where neither escapes. We solve this problem by adding an *exclusion set* $\mathcal{X}$ to the leak rules to prevent circular escape routes. In the via-newloc rule, the location l in question is added to the exclusion set, which prevents l' from escaping through l.

The $\mathsf{Escaped}(l, \mathcal{X})$ function used by the via-newloc rule computes the condition under which l escapes through a route that does not intersect with $\mathcal{X}$. It is defined by considering escape routes through all pointers and other means such as function calls (modeled by the

$$\begin{array}{l}\mathsf{Escapee}(\epsilon) ::= \mathsf{param}_i \mid \epsilon.f \mid \ *\epsilon \\ \mathsf{Summary} : \Sigma \in \mathsf{bool} \times 2^{\mathsf{Escapee}}\end{array}$$

Figure 2.15: The definition of function summaries.

attribute $l\#\mathsf{escaped}$).

Finally, $\mathsf{Leaked}(l, \mathcal{X})$ computes the condition under which a new location l is leaked through some route that does not intersect with $\mathcal{X}$. It takes into consideration the *validity* of l, which models whether the initial allocation is successful or not (see Section 2.6.1 for an example).

Using these definitions, we specify the condition under which a leak error occurs:

$$\exists l \text{ s.t. } (l \in \mathsf{NewLocs}) \text{ and } (\mathsf{Leaked}(l, \{\}) \text{ is satisfiable})$$

We issue a warning for each location that satisfies this condition.

2.6.5 Interprocedural Analysis

This subsection describes the summary-based approach to interprocedural leak detection in Saturn. We start by defining the summary representation in Section 2.6.5.1 and discuss summary generation and application in Sections 2.6.5.2 and 2.6.5.3.

2.6.5.1 Summary Representation

Figure 2.15 shows the representation of a function summary. In leak analysis we are interested in whether the function returns newly allocated memory (i.e. allocator functions), and whether it creates any external reference to objects passed via parameters (recall Section 2.6.1). Therefore, a summary Σ is composed of two components: 1) a boolean value that describes whether the function returns newly allocated memory, and 2) a set of escaped locations (*escapees*). Since caller and callee have different names for the formal and actual parameters, we use access paths (recall Section 2.6.3) to name escaped objects. These paths, called $\mathsf{Escapees}$ in Figure 2.15, are defined as a subset of access paths whose origin is a parameter.

Consider the following example:

$\boxed{\textsf{IsMalloc:}}$

$$\begin{array}{l}
\psi(\mathsf{rv}) = \{\!|\ (\mathcal{G}_0, \mathsf{null}), \overline{(\mathcal{G}_i, l_i)}, \overline{(\mathcal{G}'_j, l'_j)}\ |\!\} \\
\qquad \text{where } l_i \in \mathsf{NewLocs} \text{ and } l'_j \notin \mathsf{NewLocs} \\
\bullet\ \bigvee_i \mathcal{G}_i \text{ is satisfiable and } \bigvee_j \mathcal{G}'_j \text{ is not satisfiable} \\
\bullet\ \forall l_i \in \mathsf{NewLocs}, (\mathcal{G}_i \implies \mathsf{Leaked}(l_i, \{\mathsf{ret\_val}\})) \\
\qquad\qquad\qquad\qquad \text{is a tautology}
\end{array}$$

$\boxed{\textsf{Escapees:}}$

$$\begin{array}{l}
\mathsf{EscapedSet}(f) = \{\ \mathsf{PathOf}(l) \mid \mathsf{RootOf}(l) = \mathsf{param}_i \text{ and} \\
\qquad\qquad\qquad\qquad \mathsf{Escaped}(l, \{\mathsf{param}_i\}) \text{ is satisfiable } \}
\end{array}$$

Figure 2.16: Summary generation.

```
1 void *global;
2 void *f(struct state *p) {
3   global = p->next->data;
4   return malloc(5);
5 }
```

The summary for function f is computed as

$$\langle \mathsf{isMalloc{:}\ true;\ escapees{:}}\ \{(*(*\mathsf{param}_0).\mathsf{next}).\mathsf{data}\}\rangle$$

because f returns newly allocated memory at line 4 and adds a reference to p->next->data from global and therefore escapes that object.

Notice that the summary representation focuses on common leak scenarios. It does not capture all memory allocations. For example, functions that return new memory blocks via a parameter (instead of the return value) are not considered allocators. Likewise, aliasing relationships between parameters are not captured by the summary representation.

2.6.5.2 Summary Generation

Figure 2.16 describes the rules for function summary generation. When the return value of a function is a pointer, the IsMalloc rule is used to decide whether a function returns a newly allocated memory block. A function qualifies as a *memory allocator* if it meets the following two conditions:

1. The return value can only point to null or newly allocated memory locations. The possibility of returning any other pre-existing locations disqualifies the function as a memory allocator.

2. The return value is the only externally visible reference to new locations that might be returned. This prevents false positives from region-based memory management schemes where a reference is retained by the allocator to free all new locations in a region together.

The set of escaped locations is computed by iterating through all parameter accessible objects (i.e., objects whose access path origin is a parameter p), and testing whether the object can escape through a route that does not go through p, i.e., if $\mathsf{Escaped}(l, \{\mathsf{param}_i\})$ is satisfiable.

Take the following code as an example:

```
void insert_after(struct node *head, struct node *new) {
  new->next = head->next;
  head->next = new;
}
```

The escapee set of insert_after includes: (*head).next, since it can be reached by the pointer (*new).next; and *new, since it can be reached by the pointer (*head).next. The object *head is not included, because it is only accessible through the pointer head, which is excluded as a possible escape route. (For clarity, we use the more mnemonic names head and next instead of param_0 and param_1 in these access paths.)

2.6.5.3 Summary Application

Function calls are replaced by code that simulates their memory behavior based on their summary. The following pseudo-code models the effect of the function call r = f(e_1, e_2, ..., e_n), assuming f is an allocator function with escapee set escapees:

```
1  /* escape the escapees */
2  foreach (e) in escapees do
3      (*e)#escaped = true;
4
5  /* allocate new memory, and store it in r */
6  if (*) {
7      newloc(r);
8      (*r)#valid <- true;
9  } else
10     r <- null;
```

Lines 1-3 set the `escaped` attribute for f's escapees. Note that `e` at line 3 is an access path from a parameter. Thus `(*e)` is not strictly a valid Saturn object and must be transformed into one using a series of assignments.

Lines 5-10 simulate the memory allocation performed by `f`. We non-deterministically assign a new location to `r` and set the valid bit of the new object to `true`. To simulate a failed allocation, we assign `null` to `r` at line 10.

In the case where `f` is not an allocation function, lines 5-10 are replaced by the statement `r` $\leftarrow$ `unknown`.

2.6.6 Loops and Recursion

For the leak detector Saturn uses a two-pass algorithm for loops. In the first pass the loop is unrolled a small number of times (three in our implementation) and the backedges discarded; thus, just the first three iterations are analyzed. For leak detection this strategy works well except for loops such as

```
for (i = 0; i < 10000; i++)
    ;
```

The problem here is that the loop exit condition is never true in the first few iterations of the loop. Thus path sensitive analysis of just the first few iterations concludes that the exit test is never satisfied and the code after the loop appears to be unreachable. In the first pass, if the loop can terminate within the number of unrolled iterations, the analysis of the loop is just the result of the first pass. Otherwise, we discard results from the first pass and a second, more conservative analysis is used. In the second pass, we replace the right-hand side of all assignments in the loop body by `unknown` expressions and the loop is analyzed once (havoc'ing). Intuitively, the second pass analyzes the last iteration of the loop; we model the fact that we do not know the state of modified variables after an arbitrary number of earlier iterations by assigning them `unknown` values [84]. Integer `unknown`s are represented using unconstrained Boolean variables, and are thus conservative approximations of the actual runtime values. For pointers, however, `unknown` evaluates to a fresh location and therefore is unsound. The motivation for this two-pass analysis is that the first pass yields more precise results when the loop can be shown to terminate; however, if the unrolled loop iterations cannot reach the loop exit, then the second pass is preferable because it is more important to at least reach the code after the loop than to have precise information for the loop itself.

Recursion is handled in a similar manner as in the Linux lock checker: we analyze mutually recursive functions once in arbitrary order. Theoretically, this is a source of both false positives (missed escape routes through recursive function calls) and false negatives (missed allocation functions). Practically, however, in our experiments recursion seems to be rare and does not impact the precision of the results.

2.6.7 Handling Unsafe Operations in C

The C type system allows constructs (i.e., unsafe type casts and pointer arithmetic) not directly modeled by Saturn. We have identified several common idioms that use such operations, motivating some extensions to our leak detector.

One extension handles cases similar to the following, which emulates a form of inheritance in C:

```
struct sub { int value; struct super super; }
struct super *allocator(int size)
{
    struct sub *p = malloc(...);
    p->value = ...;
    return (&p->super);
}
```

The allocator function returns a reference to the super field of the newly allocated memory block. Technically, the reference to sub is lost on exit, but it is not considered an error because it can be recovered with pointer arithmetic. Variants of this idiom occur frequently in the projects we examined. Our solution is to consider a structure escaped if any of its components escape.

Another extension recognizes common address manipulation macros in Linux such as virt_to_phys and bus_to_virt, which add or subtract a constant page offset to arrive at the physical or virtual equivalent of the input address. Our implementation matches such operations and treats them as identity functions.

2.6.8 A Distributed Architecture

The leak analysis uses a path sensitive algorithm to track every incoming and newly allocated memory location in a function. Compared to the lock checker in Section 2.5, the higher number of tracked objects (and thus SAT queries) means the leak analysis is considerably more computationally intensive.

However, SATURN is highly parallelizable, because it analyzes each function separately, subject only to the ordering dependencies of the function call graph. We have implemented a distributed client/server architecture to exploit this parallelism in the memory leak checker.

The server side consists of a scheduler, dispatcher, and database server. The scheduler computes the dependence graph between functions and determines the set of functions ready to be analyzed. The dispatcher sends ready tasks to idle clients. When the client receives a new task, it retrieves the function's abstract syntax tree and summaries of its callees from the database server. The result of the analysis is a new summary for the analyzed function, which is sent to the database server for use by the function's callers.

We employ caching techniques to avoid congestion at the server. Our implementation scales to hundreds of CPUs and is highly effective: the analysis time for the Linux kernel, which requires nearly 24 hours on a single fast machine, is analyzed in 50 minutes using around 80 unloaded CPUs.[8] The speedup is sublinear in the number of processors because there is not always enough parallelism to keep all processors busy, particularly near the root of a call graph.

Due to the similarity of the analysis architecture between the Linux lock checker and the memory leak detector, we expect that the former would also benefit from a distributed implementation and achieve similar speed up.

	LOC	**Single Proc.**		**Distributed**	
		Time	**LOC/s**	**P.Time**	**P.LOC/s**
User-space App.					
Samba	403,744	3h22m52s	33	10m57s	615
OpenSSL	296,192	3h33m41s	23	11m09s	443
Postfix	137,091	1h22m04s	28	12m00s	190
Binutils	909,476	4h00m11s	63	16m37s	912
OpenSSH	36,676	27m34s	22	6m00s	102
Sub-total	1,783,179	12h46m22s	39	56m43s	524
Linux Kernel					
v2.6.10	5,039,296	23h13m27s	60	50m34s	1661
Total	6,822,475	35h59m49s	53	1h47m17s	1060

LOC: total number of lines of code; **Time**: analysis time on a single processor (2.8G Xeon); **P.Time**: parallel analysis time on a heterogeneous cluster of around 80 unloaded CPUs.

(a) Performance Statistics.

	Fn	**Failed** (%)	**Alloc**	**Bugs**	**FP** (%)
User-space App.					
Samba	7,432	24 (0.3%)	80	83	8 (8.79%)
OpenSSL	4,181	60 (1.4%)	101	117	1 (0.85%)
Postfix	1,589	11 (0.7%)	96	8	0 (0%)
Binutils	2,982	36 (1.2%)	91	136	5 (3.55%)
OpenSSH	607	5 (0.8%)	19	29	0 (0%)
Sub-total	16,791	136 (0.8%)	387	373	14 (3.62%)
Linux Kernel					
v2.6.10	74,367	792 (1.1%)	368	82	41 (33%)
Total	91,158	928 (1.0%)	755	455	55 (10.8%)

Fn: number of functions in the program;
Alloc: number of memory allocators detected;
FP: number of false positives.

(b) Analysis results.

Table 2.6: Experimental results for the memory leak checker.

```
/* Samba – libads/ldap.c:ads_leave_realm */
host = strdup(hostname);
if (...) {
  ...;
  return ADS_ERROR_SYSTEM(ENOENT);
}
...
```

(a) The programmer forgot to free host on an error exit path.

```
/* Samba – client/clitar.c:do_tarput */
longfilename = get_longfilename(finfo);
...
return;
```

(b) The programmer apparently is not aware that get_longfilename allocates new memory, and forgets to de-allocate longfilename on exit.

```
/* Samba – utils/net_rpc.c:rpc_trustdom_revoke */
domain_name = smb_xstrdup(argv[0]);
...
if (!trusted_domain_password_delete(domain_name))
    ...
return ..;
```

(c) trusted_domain_password_delete does not de-allocate memory, as its name might suggest. Memory referenced by domain_name is thus leaked on exit.

Figure 2.17: Three representative errors found by the leak checker.

```
/* OpenSSL – crypto/bn/bn_lib.c:BN_copy */
t = BN_new();
if (t == NULL) return (NULL);
r = BN_copy(t, a);
if (r == NULL)
    BN_free(t);
return r;
```

Figure 2.18: A sample false positive.

2.6.9 Experimental Results

We have implemented the leak checker in the SATURN analysis framework and applied it to five user space applications and the Linux kernel.

2.6.9.1 User Space Applications

We checked five user space software packages: Samba, OpenSSL, PostFix, Binutils, and OpenSSH. We analyzed the latest release of the first three, while we used older versions of the last two to compare with results reported for other leak detectors [38, 34]. All experiments were done on a lightly loaded dual Xeon™ 2.8G server with 4 gigabytes of memory as well as on a heterogeneous cluster of around 80 idle workstations. For each function, the resource limits were set to 512MB of memory and 90 seconds of CPU time.

The top portions of Tables 2.6(a) and (b) give the performance statistics and bug counts of the leak checker on the five user-space applications. Note that we miss any bugs in the small percentage of functions where resource limits are exceeded. The 1.8 million lines of code were analyzed in under 13 hours using a single processor and in under 1 hour using a cluster of about 80 CPUs. The parallel speedups increase significantly with project size, indicating larger projects have relatively fewer call graph dependencies than smaller ones. Note that the sequential scaling behavior (measured in lines of code per second) remains stable across projects ranging from 36K up to 909K lines of unpreprocessed code.

The tool issued 379 warnings across these applications. We have examined all the

[8]As courtesy to the generous owners of these machines, we constantly monitor CPU load and user activity on these machines, and turn off clients that have active users or tasks. Furthermore, these 80 CPUs range from low-end Pentium 4 1.8G workstations to high-end Xeon 2.8G servers in dual- and quad-processor configurations. Thus, performance statistics for distributed runs reported here only provide an approximate notion of speed-up when compared to single processor analysis runs.

warnings and believe 365 of them are bugs. (Warnings are per allocation site to facilitate inspection.) Besides bug reports, the leak checker generates a database of function summaries documenting each function's memory behavior. In our experience, the function summaries are highly accurate, and that, combined with path-sensitive intraprocedural analysis, explains the exceptionally low false positive rate. The summary database's function level granularity enabled us to focus on one function at a time during inspection, which facilitated bug confirmation.

Most of the bugs we found can be classified into three main categories:

1. *Missed deallocation on error paths.* This case is by far the most common, often happening when the procedure has multiple allocation sites and error conditions. Errors are common even when the programmer has made an effort to clean-up orphaned memory blocks. Figure 2.17a gives an example.

2. *Missed allocators.* Not all memory allocators have names like OPENSSL_malloc. Programmers sometimes forget to free results from less obvious allocators such as get_longfilename (`samba/client/clitar.c`, Figure 2.17b).

3. *Non-escaping procedure calls.* Despite the suggestive name, trusted_domain_password_delete (`samba/passdb/secrets.c`) does not free its parameter (Figure 2.17c).

Figure 2.18 shows a false positive caused by a limitation of our choice of function summaries. At line 4, BN_copy returns a copy of t on success and null on failure, which is not detected, nor is it expressible by the function summary.

2.6.9.2 The Linux Kernel

The bottom portions of Tables 2.6(a) and (b) summarize statistics of our experiments on Linux 2.6.10. Using the parallel analysis framework (recall Section 2.6.8) we distributed the analysis workload on 80 CPUs. The analysis completed in 50 minutes, processing 1661 lines per second. We are not aware of any other analysis algorithm that achieves this level of parallelism.

The bug count for Linux is considerably lower than for the other applications relative to the size of the source code. The Linux project has made a conscious effort to reduce memory leaks, and, in most cases, they try to recover from error conditions, where most of the leaks occur. Nevertheless, the tool found 82 leak errors, some of which were surrounded by error handling code that frees a number of other resources. Two errors were confirmed by the developers as exploitable and could potentially enable denial of service attacks against the system. These bugs were immediately fixed when reported.

The false positive rate is higher in the kernel than user space applications due to widespread use of function pointers and pointer arithmetic. Of the 41 false positives, 16 are due to calls via function pointers and 9 due to pointer arithmetic. Application specific logic accounted for another 12, and the remaining 4 are due to Saturn's current limitations in modeling constructs such as arrays and unions.

2.7 Unsoundness

One theoretical weakness of the two checkers, as described above, is unsoundness. In this section, we briefly summarize the sources of unsoundness for both the finite-state machine (FSM) checker and the memory leak analysis:

1. **Handling of loops.** We introduced two techniques to handle loops in Saturn: unrolling and havoc'ing, both of which are unsound. The former might miss bugs that occur only in a long-running loop, and the latter is unsound in its treatment of modified pointers in the loop body (see Section 2.6.6).

2. **Handling of recursion.** Recursive function calls are not handled in the two checkers, so bugs could remain undetected due to inaccurate function summaries. Handling recursion is no different than handling loops; however, the two are treated separately in Saturn for engineering reasons.

3. **Interprocedural aliasing.** Both checkers use the heuristic that distinct pointers from the external environment (e.g., function parameters, global variables) point to distinct objects. Although effective in practice, this heuristic may prevent our analysis from detecting bugs caused by interprocedural aliasing.

4. **Summary representation.** The function summary representations for both checkers leave several aspects of a function's behavior unspecified. Examples include interprocedural side-effects (e.g., modification of global variables) and aliasing, both of which may lead to false negatives.

5. **Unhandled C constructs.** For efficiency reasons, constructs such as unions, arrays, and pointer arithmetic are not directly modeled by the SATURN framework. Rather, they are handled by specific checkers during translation from C to the SATURN intermediate language. For example, in the leak checker, memory blocks stored in arrays are considered to be escaped, which is a source of unsoundness.

It is worth noting that unsoundness is not a fundamental limitation of the SATURN framework. Sound analyses can be constructed in SATURN by using appropriate summaries for both loops and functions and by iterating the analyses to reach a fixed point. For example, [33] describes the design and implementation of a sound and precise pointer alias analysis in SATURN.

2.8 Related Work

In this section we discuss the relationship of SATURN to several other systems for error detection and program verification.

2.8.1 FSM Checking

Several previous systems have been successfully applied to checking finite state machine properties in system code. SATURN was partly inspired by previous work on *Meta Compilation* (MC) [25, 35] and the project is philosophically aligned with MC in that it is a bug detection, rather than a verification, system. In fact, SATURN began as an attempt to improve the accuracy of MC's flow sensitive but path insensitive analysis [35].

Under the hood, MC attaches finite state machines (FSM) to syntactic program objects (e.g., variables, memory locations, etc.) and uses an interprocedural data flow analysis to compute the reachability of the error state. Because conservative pointer analysis is often a source of false positives for bug finding purposes [30], MC simply chooses not to model pointers or the heap, thereby preventing false positives from spurious alias relationships by fiat. MC checkers use heuristics (e.g., separate FSM transitions for the true and false branches of relevant `if` statements) and statistical methods to infer some of the lost information. These techniques usually dramatically reduce false positive rates after several rounds of trial and error. However, they cannot fully compensate for the information lost during the analysis. For example, in the code below,

```
/* 1: data correlation */
if (x) spin_lock(&lock);
if (x) spin_unlock(&lock);

/* 2: aliasing */
l = &p->lock;
spin_lock(&p->lock);
spin_lock(l);
```

MC emits a spurious warning in the first case, and misses the error in the second. The first scenario occurs frequently in Linux, and an interprocedural version of the second is also prevalent.

Saturn can be viewed as both a generalization and simplification of MC because it uniformly relies on boolean satisfiability to model all aspects without special cases. The lock checker presented in Section 2.5.5 naturally tracks locks that are buried in the heap, or conditionally manipulated based on the values of certain predicates. In designing this checker, we focused on two kinds of Linux mutex errors that exhibited high rates of false positives in MC: double locking and double unlocking (2 errors and 23 false positives [25]). Our experiments show that Saturn's improved accuracy and summary-based interprocedural analysis allow it to better capture locking behavior in the Linux kernel and thus find more errors at a lower false positive rate.

While SLAM, BLAST, and other software model checking projects have made progress in handling hundreds of thousands of lines of code [7, 40, 39], these are whole-program analyses. ESP, a lower-complexity approach based on context-free reachability, is similarly whole-program [21]. In contrast, Saturn analyzes open programs and computes summaries for functions independent of their calling context. In our experiments, Saturn scales to millions of lines of code and should in fact be able to scale arbitrarily, at least for checking properties that lend themselves to concise function summaries. In addition, Saturn has the precision of path-sensitive bit-level analysis within function bodies, which makes handling normally difficult-to-model constructs, such as type casts, easy. In fact, Saturn's code size is only about 25% of the comparable part of BLAST (the most advanced software model checker available to us), which supports our impression that a SAT-based checker is easier to engineer.

CQual is a quite different, type-based approach to program checking [30, 3]. CQual's primary limitation is that it is path insensitive. In the locking application path sensitivity is not particularly important for most locks, but we have found that it is essential for uncovering the numerous `trylock` errors in Linux. CQual's strength is in sophisticated global alias analysis that allows for sound reasoning and relatively few false positives due to spurious aliases.

2.8.2 Memory Leak Detection

Memory leak detection using dynamic tools has been a standard part of the working programmer's toolkit for more than a decade. One of the earliest and best known tools is *Purify* [36]; see [15] for a recent and significantly different approach to dynamic leak detection. Dynamic memory leak detection is limited by the quality of the test suite; unless a test case triggers the memory leak it cannot be found.

More recently there has been work on detecting memory leaks statically, sometimes as an application of general shape or heap analysis techniques, but in other cases focusing on leak detection as an interesting program analysis problem in its own right. One of the earliest static leak detectors was LCLint [27], which employs an intraprocedural dataflow analysis to find likely memory errors. The analysis depends heavily on user annotation to model function calls, thus requiring substantial manual effort to use. The reported false positive rate is high mainly due to path insensitive analysis.

PREfix [9] detects memory leaks by symbolic simulation. Like Saturn, PREfix uses function summaries for scalability and is path sensitive. However, PREfix explicitly explores paths one at a time, which is expensive for procedures with many paths. Heuristics limit the search to a small set of "interesting" paths. In contrast, Saturn represents all paths using boolean constraints and path exploration is implicit as part of boolean constraint solving.

Chou [16] describes a path-sensitive leak detection system based on static reference counting. If the static reference count (which over-approximates the dynamic reference count) becomes zero for an object that has not escaped, that object is leaked. Chou reports finding hundreds of memory leaks in an earlier Linux kernel using this method, most of which have since been patched. The analysis is quite conservative in what it considers escaping; for example, saving an address in the heap or passing it as a function argument both cause the analysis to treat the memory at that address as escaped (i.e., not leaked). The interprocedural aspect of the analysis is a conservative test to discover `malloc` wrappers. Saturn's path- and context-sensitive analysis is more precise both intra- and inter-procedurally.

We know of two memory leak analyses that are sound and for which substantial experimental data is available. Heine and Lam use *ownership types* to track an object's owning reference (the reference responsible for deallocating the object) [38]. A recent extension handles polymorphic containers which substantially improves the precision of the tool [37]. Their algorithm tracks the full life-cycle of objects from allocation to deletion and thus is able to find all leaks. In contrast, we focus on precisely modeling references to newly allocated objects before they reach the heap. Hackett and Rugina describe a hybrid region and shape analysis (where the regions are given by the equivalence classes defined by an underlying points-to analysis) [34]. It similarly aims to track the full life-cycle of allocated memory blocks, and can potentially find errors besides memory leaks such as dangling references and double free errors. In both cases, on the same inputs SATURN finds more bugs with a lower false positive rate. While SATURN's lower false positive is not surprising (soundness usually comes at the expense of more false positives), the higher bug counts for SATURN are surprising (because sound tools should not miss any bugs). For example, for `binutils` SATURN found 136 bugs compared with 66 found by Heine and Lam [38]. We compared bug reports issued by both tools and the reason appears to be that Heine and Lam inspected 279 of 1106 warnings generated by their system; the other 727 warnings were considered likely to be false positives. (SATURN did miss one bug reported by Heine and Lam due to exceeding the CPU time limit for the function containing the bug.) Hackett and Rugina report 10 bugs in `OpenSSH` out of 26 warnings. Here there appear to be two issues. First, the abstraction for which the algorithm is sound does not model some common features of C, causing the implementation for C to miss some bugs. Second, the implementation does not always finish (just as SATURN does not).

There has been extensive prior research in points-to and escape analysis. IPSSA [54] is a summary-based path- and context-sensitive pointer alias analysis for bug detection. Access paths were first used by Landi and Ryder [50] as symbolic names for memory locations accessed in a procedure. Several later algorithms (e.g., [23, 79, 53]) also make use of parameterized pointer information to achieve context sensitivity. Escape analysis (e.g., [78, 67]) determines the set of objects that do not escape a certain region. The result is traditionally used in program optimizers to remove unnecessary synchronization operations (for objects that never escape a thread) or enable stack allocation (for ones that never escape a function call). Leak detection benefits greatly from path-sensitivity, which is not a property of traditional escape analyses.

2.8.3 Other SAT-based Checking and Verification Tools

Rapid improvements in algorithms for SAT (e.g., zChaff [89, 61], which we use in Saturn) have led to its use in a variety of applications, including recently in program verification.

Jackson and Vaziri were apparently the first to consider finding bugs via reducing program source to boolean formulas [44]. Subsequently there has been significant work on a similar approach called *bounded model checking* [49]. In [20], Clarke et. al. has further explored the idea of SAT-based predicate abstraction of ANSI-C programs. While there are many low-level algorithmic differences between Saturn and these other systems, the primary conceptual difference is our emphasis on scalability (e.g., function summaries) and focus on fully automated inference, as well as checking, of properties without separate programmer-written specifications.

2.9 Conclusion

In this chapter, we have presented Saturn, a scalable and precise error detection framework based on boolean satisfiability. Our system has a novel combination of features: it models all values, including those in the heap, path sensitively down to the bit level, it computes function summaries automatically, and it scales to millions of lines of code. We have experimentally validated our approach by conducting two case studies involving a Linux lock checker and a memory leak checker. Results from the experiments show that our system scales well, parallelizes well, and finds more errors with less false positives than previous error detection systems.

Chapter 3

Static Analysis of Scripting Languages

3.1 Introduction

Web-based applications have proliferated rapidly in recent years and have become the *de facto* standard for delivering online services ranging from discussion forums to security sensitive areas such as banking and retailing. As such, security vulnerabilities in these applications represent an increasing threat to both the providers and the users of such services. During the second half of 2004, Symantec cataloged 670 vulnerabilities affecting web applications, an 81% increase over the same period in 2003 [73]. This trend is likely to continue for the foreseeable future.

According to the same report, these vulnerabilities are typically caused by programming errors in input validation and improper handling of submitted requests [73]. Since vulnerabilities are usually deeply embedded in the program logic, traditional network-level defense (e.g., firewalls) does not offer adequate protection against such attacks. Testing is also largely ineffective because attackers typically use the least expected input to exploit these vulnerabilities and compromise the system.

A natural alternative is to find these errors using static analysis. This approach has been explored in WebSSARI [42] and by Minamide [58]. WebSSARI has been used to find a number of security vulnerabilities in PHP scripts, but has a large number of false positives and negatives due to its intraprocedural type-based analysis. Minamide's system checks syntactic correctness of HTML output from PHP scripts and does not seem to be effective for finding security vulnerabilities. The main message of this chapter is that analysis of scripting languages need not be significantly more difficult than analysis of conventional languages. While a scripting language stresses different aspects of static analysis, an analysis suitably designed to address the important aspects of scripting languages can identify many serious vulnerabilities in scripts reliably and with a high degree of automation. Given the importance of scripting in real world applications, we believe there is an opportunity for static analysis to have a significant impact in this new domain.

In this chapter, we apply static analysis to finding security vulnerabilities in PHP, a server-side scripting language that has become one of the most widely adopted platforms for developing web applications.[1] Our goal is a bug detection tool that automatically finds serious vulnerabilities with high confidence. This work, however, does not aim to verify the absence of bugs.

The techniques described in this chapter make the following contributions:

- We present an interprocedural static analysis algorithm for PHP. A language as dynamic as PHP presents unique challenges for static analysis: language constructs (e.g., include) that allow dynamic inclusion of program code, variables whose types change during execution, operations with semantics that depend on the runtime types of the operands (e.g., $<$), and pervasive use of hash tables and regular expression matching are just some features that must be modeled well to produce useful results.

  To faithfully model program behavior in such a language, we use a three-tier analysis that captures information at decreasing levels of granularity at the intrablock, intraprocedural, and interprocedural levels. This architecture allows the analysis to be precise where it matters the most–at the intrablock and, to a lesser extent, the intraprocedural levels–and use agressive abstraction at the natural abstraction boundary along function calls to achieve scalability. We use symbolic execution to model dynamic features inside basic blocks and use block summaries to hide that complexity from intra- and inter-procedural analysis. We believe the same techniques can be applied easily to other scripting languages (e.g., Perl).

- We show how to use our static analysis algorithm to find SQL injection vulnerabilities. Once configured, the analysis is fully automatic. Although we focus on SQL injections

[1]Installed on over 23 million Internet domains [66], and is ranked fourth on the TIOBE programming community index [75].

in this work, the same techniques can be applied to detecting other vulnerabilities such as cross site scripting (XSS) and code injection in web applications.

- We experimentally validate our approach by implementing the analysis algorithm and running it on six popular web applications written in PHP, finding 105 previously unknown security vulnerabilities. We analyzed two reported vulnerabilities in PHP-fusion, a mature, widely deployed content management system, and construct exploits for both that allow an attacker to control or damage the system.[2]

The rest of the chapter is organized as follows. We start with a brief introduction to PHP and show examples of SQL vulnerabilities in web application code (Section 3.2). We then present our analysis algorithm and show how we use it to find SQL injection vulnerabilities (Section 3.3). Section 3.4 describes the implementation, experimental results, and two case studies of exploitable vulnerabilities in PHP-fusion. Section 3.5 discusses related work and Section 3.6 concludes.

3.2 Background

This section briefly introduces the PHP language and shows examples of SQL injection vulnerabilities in PHP.

PHP was created a decade ago by Rasmus Lerdorf as a simple set of Perl scripts for tracking accesses to his online resume. It has since evolved into one of the most popular server-side scripting languages for building web applications. According to a recent Security Space survey, PHP is installed on 44.6% of Apache web servers [70], adopted by millions of developers, and used or supported by Yahoo, IBM, Oracle, and SAP, among others [66].

Although the PHP language has undergone two major re-designs over the past decade, it retains a Perl-like syntax and dynamic (interpreted) nature, which contributes to its most frequently claimed advantage of being simple and flexible.

PHP has a suite of programming constructs and special operations that ease web development. We give three examples:

1. **Natural integration with SQL:** PHP provides nearly native support for database operations. For example, using inline variables in strings, most SQL queries can be concisely expressed with a simple function call

[2]Both vulnerabilities have been reported to and fixed by the PHP-fusion developers.

```
$rows=mysql_query("UPDATE users SET pass='$pass' WHERE userid='$userid'");
```

Contrast this code with Java, where a database is typically accessed through *prepared statements*: one creates a statement template and fills in the values (along with their types) using *bind variables*:

```
PreparedStatement s = con.prepareStatement
    ("UPDATE users SET pass = ? WHERE userid = ?");
s.setString(1, pass);
s.setInt(2, userid);
int rows = s.executeUpdate();
```

2. **Dynamic types and implicit casting to and from strings:** PHP, like other scripting languages, has extensive support for string operations and automatic conversions between strings and other types. These features are handy for web applications because strings serve as the common medium between the browser, the web server, and the database backend. For example, we can convert a number into a string without an explicit cast:

```
if ($userid < 0) exit;
$query = "SELECT * from users WHERE userid = '$userid'";
```

3. **Variable scoping and the environment:** PHP has a number of mechanisms that minimize redundancy when accessing values from the execution environment. For example, HTTP *get* and *post* requests are automatically imported into the global name space as hash tables $_GET and $_POST. To access the "name" field of a submitted form, one can simply use $_GET['name'] directly in the program.

   If this still sounds like too much typing, PHP provides an extract operation that automatically imports all key-value pairs of a hash table into the current scope. In the example above, one can use extract(_GET, EXTR_OVERWRITE) to import data submitted using the HTTP get method. To access the $name field, one now simply types $name, which is preferred by some to $_GET['name'].

However, these conveniences come with security implications:

1. **SQL injection made easy:** Bind variables in Java have the benefit of assuring the programmer that any data passed into an SQL query remains data. The same cannot be said for the PHP example where malformed data from a malicious attacker may

change the meaning of an SQL statement and cause unintended operations to the database. These are commonly called *SQL injection* attacks.

In the example above (case 1), suppose $userid is controlled by the attacker and has value

```
' OR '1' = '1
```

The query string becomes

```
UPDATE users SET pass='...' WHERE userid='' OR '1'='1'
```

which has the effect of updating the password for all users in the database.

2. **Unexpected conversions:** Consider the following code:

   ```
   if ($userid == 0) echo $userid;
   ```

   One would expect that if the program prints anything, it should be "0". Unfortunately, PHP implicitly casts string values into numbers before comparing them with an integer. Non-numerical values (e.g., "abc") convert to 0 without a complaint, so the code above can print anything other than a non-zero number. We can imagine a potential SQL injection vulnerability if $userid is subsequently used to construct an SQL query as in the previous case.

3. **Uninitialized variables under user control:** In PHP, uninitialized variables default to null. Some programs rely on this fact for correct behavior; consider the following code:

   ```
   1 extract($_GET, EXTR_OVERWRITE);
   2 for ($i=0;$i<=7;$i++)
   3   $new_pass .= chr(rand(97, 122)); // append one char
   4 mysql_query("UPDATE ... $new_pass ...");
   ```

   This program generates a random password and inserts it into the database. However, due to the extract operation on line 1, a malicious user can introduce an arbitrary initial value for $new_pass by adding an unexpected new_pass field into the submitted HTTP form data.

```
function analyze_function(AbstractSyntaxTree ast) : FunctionSummary
{
    CFG := build_control_flow_graph(ast);
    foreach (basic_block b in CFG)
        summaries[b] := simulate_block(b);
    return make_function_summary(CFG, summaries);
}
```

Figure 3.1: Pseudo-code for the analysis of a function.

3.3 Analysis

Given a PHP source file, our tool carries out static analysis in the following steps:

- We parse the PHP source into abstract syntax trees (ASTs). Our parser is based on the standard open-source implementation of PHP 5.0.5 [65]. Each PHP source file contains a *main* section (referred to as the *main* function hereafter although it is technically not part of any function definition) and zero or more user-defined functions. We store the user-defined functions in the environment and start the analysis from the main function.

- The analysis of a single function is summarized in Figure 3.1. For each function in the program, the analysis performs a standard conversion from the abstract syntax tree (AST) of the function body into a control flow graph (CFG). The nodes of the CFG are *basic blocks*: maximal single entry, single exit sequences of statements. The edges of the CFG are the jump relationships between blocks. For conditional jumps, the corresponding CFG edge is labeled with the branch predicate.

- Each basic block is simulated using symbolic execution. The goal is to understand the collective effects of statements in a block on the global state of the program and summarize them into a concise *block summary* (which describes, among other things, the set of variables that must be sanitized[3] before entering the block). We describe the simulation algorithm in Section 3.3.1.

- After computing a summary for each basic block, we use a standard reachability analysis to combine block summaries into a *function summary*. The function summary

[3]Sanitization is an operation that ensures that user input can be safely used in an SQL query (e.g., no unescaped quotes or spaces).

```
function simulate_block(BasicBlock b) : BlockSummary
{
    state := init_simulation_state();
    foreach (Statement s in b) {
        state := simulate(s, state);
        if (state.has_returned || state.has_exited)
            break;
    }
    summary := make_block_summary(state);
    return summary;
}
```

Figure 3.2: Pseudo-code for intra-block simulation.

describes the pre- and post-conditions of a function (e.g., the set of sanitized input variables after calling the current function). We discuss this step in Section 3.3.2.

- During the analysis of a function, we might encounter calls to other user-defined functions. We discuss modeling function calls, and the order in which functions are analyzed, in Section 3.3.3.

3.3.1 Simulating Basic Blocks

3.3.1.1 Outline

Figure 3.2 gives pseudo-code outlining the symbolic simulation process. Recall each basic block contains a linear sequence of statements with no jumps or jump targets in the middle. The simulation starts in an *initial state*, which maps each variable x to a symbolic initial value x_0. It processes each statement in the block in order, updating the simulator state to reflect the effect of that statement. The simulation continues until it encounters any of the following:

1. the end of the block;

2. a return statement. In this case, the current block is marked as a *return* block, and the simulator evaluates and records the return value;

3. an exit statement. In this case the current block is marked as an *exit* block;

$$\begin{array}{rcl}
\mathsf{Type}\ (\tau) & ::= & \mathsf{str} \mid \mathsf{bool} \mid \mathsf{int} \mid \top \\
\mathsf{Const}\ (c) & ::= & \mathsf{string} \mid \mathsf{int} \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{null} \\
\mathsf{L\text{-}val}\ (lv) & ::= & x \mid \mathtt{Arg}\#\mathsf{i} \mid lv[e] \\
\mathsf{Expr}\ (e) & ::= & c \mid lv \mid e\ \mathsf{binop}\ e \mid \mathsf{unop}\ e \mid (\tau)e \\
\mathsf{Stmt}\ (S) & ::= & lv \leftarrow e \mid lv \leftarrow f(e_1, \ldots, e_n) \\
& & \mid \mathsf{return}\ e \mid \mathsf{exit} \mid \mathsf{include}\ e \\
\\
\mathsf{binop} & \in & \{+, -, \mathsf{concat}, ==, !=, <, >, \ldots\} \\
\mathsf{unop} & \in & \{-, \neg\}
\end{array}$$

Figure 3.3: Language Definition

4. a call to a user-defined function that exits the program. This condition is automatically determined using the function summary of the callee (see Sections 3.3.2 and 3.3.3).

Note that in the last case execution of the program has also terminated and therefore we remove any ensuing statements and outgoing CFG edges from the current block.

After a basic block is simulated, we use information contained in the final state of the simulator to summarize the effect of the block into a *block summary*, which we store for use during the intraprocedural analysis (see Section 3.3.2). The state itself is discarded after simulation.

The following subsections describe the simulation process in detail. We start with a definition of the subset of PHP that we model (Section 3.3.1.2) and discuss the representation of the simulation state and program values (Section 3.3.1.3, Section 3.3.1.4) during symbolic execution. Using the value representation, we describe how the analyzer simulates expressions (Section 3.3.1.5) and statements (Section 3.3.1.6). Finally, we describe how we represent and infer block summaries (Section 3.3.1.7).

3.3.1.2 Language

Figure 3.3 gives the definition of a small imperative language that captures a subset of PHP constructs that we believe is relevant to SQL injection vulnerabilities. Like PHP, the language is dynamically typed. We model three basic types of PHP values: strings, booleans and integers. In addition, we introduce a special $\top$ type to describe objects whose

$\boxed{\textsf{Value Representation}}$

$$\begin{aligned}
\textsf{Loc}\ (l) &::= x \mid l[\textsf{string}] \mid l[\top] \\
\textsf{Init-Values}\ (o) &::= l_0 \\
\textsf{Segment}\ (\beta) &::= \textsf{string} \mid \textsf{contains}(\sigma) \mid o \mid \bot \\
\textsf{String}\ (s) &::= \langle \beta_1, \ldots, \beta_n \rangle \\
\textsf{Boolean}\ (b) &::= \textsf{true} \mid \textsf{false} \mid \textsf{untaint}(\sigma_0, \sigma_1) \\
\textsf{Loc-set}(\sigma) &::= \{l_1, \ldots, l_n\} \\
\textsf{Integer}\ (i) &::= k \\
\textsf{Value}\ (v) &::= s \mid b \mid i \mid o \mid \top
\end{aligned}$$

$\boxed{\textsf{Simulation State}}$

$\textsf{State}\ (\Gamma) : \textsf{Loc} \rightarrow \textsf{Value}$

(a) Value representation and simulation state.

$\boxed{\textsf{Locations}}$

$$\frac{}{\Gamma \vdash x \overset{\texttt{Lv}}{\Rightarrow} x}\ \textsf{var}$$

$$\frac{}{\Gamma \vdash \texttt{Arg\#n} \overset{\texttt{Lv}}{\Rightarrow} \texttt{Arg\#n}}\ \textsf{arg}$$

$$\frac{\begin{array}{l}\Gamma \vdash e \overset{\texttt{E}}{\Rightarrow} l \\ \Gamma \vdash e' \overset{\texttt{E}}{\Rightarrow} v' \quad v'' = \textsf{cast}(v', \textsf{str})\end{array}}{\Gamma \vdash e[e'] \overset{\texttt{Lv}}{\Rightarrow} \begin{cases} l[\alpha] & \text{if } v'' = \langle \text{“}\alpha\text{”} \rangle \\ l[\top] & \text{otherwise} \end{cases}}\ \textsf{dim}$$

(b) L-values.

$\boxed{\textsf{Expressions}}$

Type casts:

$$\textsf{cast}(k, \textsf{bool}) = \begin{cases} \textsf{true} & \text{if } k \neq 0 \\ \textsf{false} & \text{otherwise} \end{cases}$$

$$\textsf{cast}(\textsf{true}, \textsf{str}) = \langle \text{“1”} \rangle$$

$$\textsf{cast}(\textsf{false}, \textsf{str}) = \langle \rangle$$

$$\textsf{cast}(v = \langle \beta_1, \ldots, \beta_n \rangle, \textsf{bool})$$

$$= \begin{cases} \textsf{true} & \text{if } (v \neq \langle \text{“0”} \rangle) \wedge \bigvee_{i=1}^{n} \neg \textsf{is\_empty}(\beta_i) \\ \textsf{false} & \text{if } (v = \langle \text{“0”} \rangle) \vee \bigwedge_{i=1}^{n} \textsf{is\_empty}(\beta_i) \\ \top & \text{otherwise} \end{cases}$$

...

Evaluation Rules:

$$\frac{\Gamma \vdash lv \overset{\texttt{Lv}}{\Rightarrow} l}{\Gamma \vdash lv \overset{\texttt{E}}{\Rightarrow} \Gamma(l)}\ \textsf{L-val}$$

$$\frac{\begin{array}{ll}\Gamma \vdash e_1 \overset{\texttt{E}}{\Rightarrow} v_1 & \textsf{cast}(v_1, \textsf{str}) = \langle \beta_1, \ldots, \beta_n \rangle \\ \Gamma \vdash e_2 \overset{\texttt{E}}{\Rightarrow} v_2 & \textsf{cast}(v_2, \textsf{str}) = \langle \beta_{n+1}, \ldots, \beta_m \rangle \end{array}}{\Gamma \vdash e_1\ \textsf{concat}\ e_2 \overset{\texttt{E}}{\Rightarrow} \langle \beta_1, \ldots, \beta_m \rangle}\ \textsf{concat}$$

$$\frac{\Gamma \vdash e \overset{\texttt{E}}{\Rightarrow} v \quad \textsf{cast}(v, \textsf{bool}) = v'}{\Gamma \vdash \neg e \overset{\texttt{E}}{\Rightarrow} \begin{cases} \textsf{true} & \text{if } v' = \textsf{false} \\ \textsf{false} & \text{if } v' = \textsf{true} \\ \textsf{untaint}(\sigma_1, \sigma_0) & \text{if } v' = \textsf{untaint}(\sigma_0, \sigma_1) \\ \top & \text{otherwise} \end{cases}}\ \textsf{not}$$

(c) Expressions.

Figure 3.4: Intrablock simulation algorithm.

static types are undetermined (e.g., input parameters).[4]

Expressions can be *constants*, *l-values*, *unary* and *binary operations*, and *type casts*. The definition of l-values is worth mentioning because in addition to variables and function parameters, we include a named subscript operation to provide limited support to the array and hash table accesses used extensively in PHP programs.

A statement can be an *assignment*, *function call*, *return*, *exit*, or *include*. The first four statement types require no further explanation. The `include` statement is a commonly used feature unique to scripting languages, which allows programmers to dynamically insert code into the program. In our language, `include` evaluates its string argument, and executes the program file designated by the string as if it is inserted at that program point (e.g., it shares the same scope). We describe how we simulate such behavior in Section 3.3.1.6.

3.3.1.3 State

Figure 3.4(a) gives the definition of values and states during simulation. The simulation state maps memory locations to their value representations, where a memory location is either a program variable (e.g. x), or an entry in a hash table accessed via another location (e.g. $x[key]$). Note the definition of locations is recursive, so multi-level hash dereferences are supported in our algorithm.

On entry to the function, each location l is implicitly initialized to a symbolic initial value l_0, which makes up the initial state of the simulation. The values we represent in the state can be classified into three categories based on type:

Strings: Strings are the most fundamental type in many scripting languages, and precision in modeling strings directly determines analysis precision. Strings are typically constructed through concatenation. For example, user inputs (via HTTP `get` and `post` methods) are often concatenated with a pre-constructed skeleton to form an SQL query. Similarly, results from the query can be concatenated with HTML templates to form output. Modeling concatenation well enables an analysis to better understand information flow in a script. Thus, our string representation is based on concatenation. String values are represented as an ordered concatenation of string *segments*, which can be one of the following: a string constant, the initial value of a memory location on entry to the current block (l_0), or a

[4]In general, in a dynamically typed language, a more precise static approximation in this case would be a sum (aka. soft typing) [4, 80]. We have not found it necessary to use type sums.

string that contains initial values of zero or more elements from a set of memory locations (contains(σ)). We use the last representation to model return values from function calls, which may non-deterministically contain a combination of global variables and input parameters. For example, in

```
1 function f($a, $b) {
2    if (...) return $a;
3    else return $b;
4 }
5 $ret = f($x.$y, $z);
```

we represent the return value on line 5 as contains($\{x, y, z\}$) to model the fact that it may contain any element in the set as a sub-string.

The string representation described above has the following benefits:

- First, we get automatic constant folding for strings within the current block, which is often useful for resolving hash keys and distinguishing between hash references (e.g., in $key = "key"; return $hash[$key];).
- Second, we can track how the contents of one input variable flow into another by finding occurrences of initial values of the former in the final representation of the latter. For example, in: $a = $a . $b, the final representation of $a is $\langle a_0, b_0\rangle$. We know that if either $a or $b contains unsanitized user input on entry to the current block, so does $a upon exit.
- Finally, interprocedural dataflow is possible by tracking function return values based on function summaries using contains(σ). We describe this aspect in more detail in Section 3.3.3.

Booleans: In PHP, a common way to perform input validation is to call a function that returns true or false depending on whether the input is well-formed or not. For example, the following code sanitizes $userid:

```
$ok = is_safe($userid);
if (!$ok) exit;
```

The value of Boolean variable $ok after the call is undetermined, but it is correlated with the safety of $userid. This motivates untaint(σ_0, σ_1) as a representation for such Booleans: σ_0 (resp. σ_1) represents the set of validated l-values when the Boolean is false (resp. true). In the example above, $ok has representation untaint($\{\}, \{\mathsf{userid}\}$).

Besides untaint, representation for Booleans also include constants (true and false) and unknown ($\top$).

Integers: Integer operations are less emphasized in our simulation. We track integer constants and binary and unary operations between them. We also support type casts from integers to Boolean and string values.

3.3.1.4 Locations and L-values

In the language definition in Figure 3.3, hash references may be aliased through assignments and l-values may contain hash accesses with non-constant keys. The same l-value may refer to different memory locations depending on the value of both the host and the key, and therefore, l-values are not suitable as memory locations in the simulation state.

Figure 3.4(b) gives the rules we use to resolve l-values into memory locations. The var and arg rules map each program variable and function argument to a memory location identified by its name, and the dim rule resolves hash accesses by first evaluating the hash table to a location and then appending the key to form the location for the hash entry.

These rules are designed to work in the presence of simple aliases. Consider the following program:

```
1  $hash = $_POST;
2  $key = 'userid';
3  $userid = $hash[$key];
```

The program first creates an alias ($hash) to hash table $_POST and then accesses the userid entry using that alias. On entry to the block, the initial state maps every location to its initial value:

$$\Gamma = \{\mathsf{hash} \Rightarrow \mathsf{hash}_0, \mathsf{key} \Rightarrow \mathsf{key}_0, \mathsf{\_POST} \Rightarrow \mathsf{\_POST}_0, \\ \mathsf{\_POST[userid]} \Rightarrow \mathsf{\_POST[userid]}_0\}$$

According to the var rule, each variable maps to its own unique location. After the first two assignments, the state is:

$$\Gamma = \{\mathsf{hash} \Rightarrow \mathsf{\_POST}_0, \mathsf{key} \Rightarrow \langle \text{`userid'} \rangle, \ldots\}$$

We use the dim rule to resolve $hash[$key] on line 3: $hash evaluates to $\mathsf{\_POST}_0$, and $key evaluates to constant string 'userid'. Therefore, the l-value $hash[$key] evaluates to location _POST[userid], and thus the analysis assigns the desired value $\mathsf{\_POST[userid]}_0$ to $userid.

3.3.1.5 Expressions

We perform abstract evaluation of expressions based on the value representation described above. Because PHP is a dynamically typed language, operands are implicitly cast to appropriate types for operations in an expression. Figure 3.4(c) gives a representative sample of cast rules simulating cast operations in PHP. For example, the Boolean value true, when used in a string context, evaluates to “1”. The value false, on the other hand, is converted to the empty string instead of “0”. In cases where exact representation is not possible, the result of the cast is unknown ($\top$).

Figure 3.4(c) also gives three representative rules for evaluating expressions. The first rule handles l-values, and the result is obtained by first resolving the l-value into a memory location, and then looking up the location in the evaluation context (recall that $\Gamma(l) = l_0$ on entry to the block).

The second rule models string concatenation. We first cast the value of both operands to string values, and the result is the concatenation of both.

The final rule handles Boolean negation. The interesting case involves untaint values. Recall that untaint(σ_0, σ_1) denotes an unknown Boolean value that is false (resp. true) if l-values in the set σ_0 (resp. σ_1) are sanitized. Given this definition, the negation of untaint(σ_0, σ_1) is untaint(σ_1, σ_0).

The analysis of an expression is $\top$ if we cannot determine a more precise representation, which is a potential source of false negatives.

3.3.1.6 Statements

We model assignments, function calls, return, exit, and include statements in the program. The assignment rule resolves the left-hand side to a memory location l, and evaluates the right-hand side to a value v. The updated simulation state after the assignment maps l to the new value v:

$$\frac{\Gamma \vdash lv \overset{\texttt{Lv}}{\Rightarrow} l \qquad \Gamma \vdash e \overset{\texttt{E}}{\Rightarrow} v}{\Gamma \vdash lv \leftarrow e \overset{\texttt{S}}{\Rightarrow} \Gamma[l \mapsto v]}\textsf{assignment}$$

Function calls are similar. The return value of a function call $f(e_1, \ldots, e_n)$ is modeled using either contains(σ) (if f returns a string) or untaint(σ_0, σ_1) (if f returns a Boolean) depending on the inferred summary for f. We defer discussion of the function summaries and the return value representation to Sections 3.3.2 and 3.3.3. For the purpose of this section, we use the uninterpreted value $f(v_1, \ldots, v_n)$ as a place holder for the actual representation of the return value:

$$\cfrac{\Gamma \vdash lv \overset{\texttt{Lv}}{\Rightarrow} l \quad \Gamma \vdash e_1 \overset{\texttt{E}}{\Rightarrow} v_1 \ \ldots\ \Gamma \vdash e_n \overset{\texttt{E}}{\Rightarrow} v_n}{\Gamma \vdash lv \leftarrow f(e_1, \ldots, e_n) \overset{\texttt{S}}{\Rightarrow} \Gamma[l \mapsto f(v_1, \ldots, v_n)]}\,\textsf{fun}$$

In addition to the return value, certain functions have pre- and post-conditions depending on the operation they perform. Pre- and post-conditions are inferred and stored in the callee's summary, which we describe in detail in Sections 3.3.2 and 3.3.3. Here we show two examples to illustrate their effects:

```
1  function validate($x) {
2   if (!is_numeric($x)) exit;
3   return;
4  }
5  function my_query($q) {
6    global $db;
7    mysql_db_query($db, $q);
8  }
9  validate($a.$b);
10 my_query("SELECT . . . WHERE a = '$a' AND c = '$c'");
```

The validate function tests whether the argument is a number (and thus safe) and aborts if it is not. Therefore, line 9 sanitizes both $a and $b. We record this fact by inspecting the value representation of the actual parameter (in this case $\langle a_0, b_0 \rangle$), and remembering the set of non-constant segments that are sanitized.

The second function my_query uses its argument as a database query string by calling mysql_db_query. To prevent SQL injection attacks, any user input must be sanitized before it becomes part of the first parameter. Again, we enforce this requirement by inspecting the value representation of the actual parameter. We record any unsanitized non-constant segments (in this case $c, since $a is sanitized on line 9) and require they be sanitized as part of the pre-condition for the current block.

Sequences of assignments and function calls are simulated by using the output environment of the previous statement as the input environment of the current statement:

$$\cfrac{\Gamma \vdash s_1 \overset{\texttt{S}}{\Rightarrow} \Gamma' \quad \Gamma' \vdash s_2 \overset{\texttt{S}}{\Rightarrow} \Gamma''}{\Gamma \vdash (s_1; s_2) \overset{\texttt{S}}{\Rightarrow} \Gamma''}\,\textsf{seq}$$

The final simulation state is the output state of the final statement.

The return and exit statements terminate control flow[5] and require special treatment. For a return, we evaluate the return value and use it in calculating the function summary. In case of an exit statement, we mark the current block as an *exit block*.

Finally, include statements are a commonly used feature unique to scripting languages allowing programmers to dynamically insert code and function definitions from another script. In PHP, the included code inherits the variable scope at the point of the include statement. It may introduce new variables and function definitions, and change or sanitize existing variables before the next statement in the block is executed.

We process include statements by first parsing the included file, and adding any new function definitions to the environment. We then splice the control flow graph of the included main function at the current program point by a) removing the include statement, b) breaking the current basic block into two at that point, c) linking the first half of the current block to the start of the main function, and all return blocks (those containing a return statement) in the included CFG to the second half, and d) replacing the return statements in the included script with assignments to reflect the fact that control flow resumes in the current script.

3.3.1.7 Block summary

The final step for the symbolic simulator is to characterize the behavior of a CFG block into a concise summary. A block summary is represented as a six-tuple $\langle \mathcal{E}, \mathcal{D}, \mathcal{F}, \mathcal{T}, \mathcal{R}, \mathcal{U} \rangle$:

- **Error set ($\mathcal{E}$):** the set of input variables that must be sanitized before entering the current block. These are accumulated during simulation of function calls that require sanitized input.

- **Definitions ($\mathcal{D}$):** the set of memory locations defined in the current block. For example, in

  ```
  $a = $a.$b; $c = 123;
  ```

  we have $\mathcal{D} = \{a, c\}$.

- **Value flow ($\mathcal{F}$):** the set of pairs of locations (l_1, l_2) where the string value of l_1 on entry becomes a sub-string of l_2 on exit. In the example above, $\mathcal{F} = \{(a, a), (b, a)\}$.

[5]So do function calls that exits the program, in which case we remove any ensuing statements and outgoing edges from the current CFG block. See Section 3.3.3.

- **Termination predicate ($\mathcal{T}$):** true if the current block contains an exit statement, or if it calls a function that causes the program to terminate.
- **Return value ($\mathcal{R}$):** records the representation for the return value if any, undefined otherwise. Note that if the current block has no successors, either $\mathcal{R}$ has a value or $\mathcal{T}$ is true.
- **Untaint set ($\mathcal{U}$):** for each successor of the current CFG block, we compute the set of locations that are sanitized if execution continues onto that block. Sanitization can occur via function calls, casting to safe types (e.g., int, etc), regular expression matching, and other tests. The untaint set for different successors might differ depending on the value of branch predicates. We show an example below.

```
validate($a);
$b = (int) $c;
if (is_numeric($d))
        ...
```

  As mentioned earlier, validate exits if $a is unsafe. Casting to integer also returns a safe result. Therefore, the untaint set is $\{a, b, d\}$ for the true branch, and $\{a, b\}$ for the false branch.

3.3.2 Intraprocedural Analysis

Based on block summaries computed in the previous step, the intraprocedural analysis computes the following summary $\langle \mathcal{E}, \mathcal{R}, \mathcal{S}, \mathcal{X} \rangle$ for each function:

1. **Error set ($\mathcal{E}$):** the set of memory locations (variables, parameters, and hash accesses) whose value may flow into a database query, and therefore must be sanitized before invoking the current function. For the main function, the error set must not include any user-defined variables (e.g. $_GET['...'] or $_POST['...'])—the analysis emits an error message for each such violation.

   We compute $\mathcal{E}$ by a backwards reachability analysis that propagates the error set of each block (using the $\mathcal{E}, \mathcal{D}, \mathcal{F}$, and $\mathcal{U}$ components in the block summaries) to the start block of the function.

2. **Return set ($\mathcal{R}$):** the set of parameters or global variables whose value may be a substring of the return value of the function. $\mathcal{R}$ is only computed for functions that may

return string values. For example, in the following code, the return set includes both function arguments and the global variable $table (i.e., $\mathcal{R} = \{\mathsf{table}, \mathtt{Arg\#1}, \mathtt{Arg\#2}\}$).

```
function make_query($user, $pass) {
  global $table;
  return "SELECT * from $table where user = $user and pass = $pass";
}
```

We compute the function return set by using a forward reachability analysis that expresses each return value (recorded in the block summaries as $\mathcal{R}$) as a set of function parameters and global variables.

3. **Sanitized values ($\mathcal{S}$):** the set of parameters or global variables that are sanitized on function exit. We compute the set by using a forward reachability analysis to determine the set of sanitized inputs at each return block, and we take the intersection of those sets to arrive at the final result.

   If the current function returns a Boolean value as its result, we distinguish the sanitized value set when the result is true versus when it is false (mirroring the untaint representation for Boolean values above). The following example motivates this distinction:

   ```
   function is_valid($x) {
     if (is_numeric($x)) return true;
     return false;
   }
   ```

   The parameter is sanitized if the function returns true, and the return value is likely to be used by the caller to determine the validity of user input. In the example above,

   $$\mathcal{S} = (\mathsf{false} \Rightarrow \{\}, \mathsf{true} \Rightarrow \{\mathtt{Arg\#1}\})$$

   For comparison, the validate function defined previously has $\mathcal{S} = (* \Rightarrow \{\mathtt{Arg\#1}\})$. In the next section, we describe how we make use of this information in the caller.

4. **Program Exit ($\mathcal{X}$):** a Boolean which indicates whether the current function terminates program execution on all paths. Note that control flow can leave a function either by returning to the caller or by terminating the program. We compute the exit predicate by enumerating over all CFG blocks that have no successors, and identify them as either return blocks or exit blocks (the $\mathcal{T}$ and $\mathcal{R}$ component in the block

summary). If there are no return blocks in the CFG, the current function is an exit function.

The dataflow algorithms used in deriving these facts are standard fix-point computations [1].

3.3.3 Interprocedural Analysis

This section describes how we conduct interprocedural analysis using summaries computed in the previous step. Assuming f has summary $\langle \mathcal{E}, \mathcal{R}, \mathcal{S}, \mathcal{X} \rangle$, we process a function call $f(e_1, \ldots, e_n)$ during intrablock simulation as follows:

1. **Pre-conditions:** We use the error set ($\mathcal{E}$) in the function summary to identify the set of parameters and global variables that must be sanitized before calling this function. We substitute actual parameters for formal parameters in $\mathcal{E}$ and record any unsanitized non-constant segments of strings in the error set as the sanitization pre-condition for the current block.

2. **Exit condition:** If the callee is marked as an exit function (i.e., $\mathcal{X}$ is true), we remove any statements that follow the call and delete all outgoing edges from the current block. We further mark the current block as an exit block.

3. **Post-conditions:** If the function unconditionally sanitizes a set of input parameters and global variables, we mark this set of values as safe in the simulation state after substituting actual parameters for formal parameters.

   If sanitization is conditional on the return value (e.g., the is_valid function defined above), we record the intersection of its two component sets as being unconditionally sanitized (i.e., $\sigma_0 \cap \sigma_1$ if the untaint set is $(\mathsf{false} \Rightarrow \sigma_0, \mathsf{true} \Rightarrow \sigma_1)$).

4. **Return value:** If the function returns a Boolean value and it conditionally sanitizes a set of input parameters and global variables, we use the untaint representation to

model that correlation:

$$\frac{\begin{array}{l}\Gamma \vdash lv \overset{\texttt{Lv}}{\Rightarrow} l \quad \Gamma \vdash e_1 \overset{\texttt{E}}{\Rightarrow} v_1 \ \ldots \ \Gamma \vdash e_n \overset{\texttt{E}}{\Rightarrow} v_n \\ \texttt{Summary}(f) = \langle \mathcal{E}, \mathcal{R}, \mathcal{S}, \mathcal{X} \rangle \\ \mathcal{S} = (\mathsf{false} \Rightarrow \sigma_0, \mathsf{true} \Rightarrow \sigma_1) \quad \sigma_* = \sigma_0 \cap \sigma_1 \\ \sigma'_0 = \mathsf{subst}_{\bar{v}}(\sigma_0 - \sigma_*) \quad \sigma'_1 = \mathsf{subst}_{\bar{v}}(\sigma_1 - \sigma_*)\end{array}}{\Gamma \vdash lv \leftarrow f(e_1, \ldots, e_n) \overset{\texttt{S}}{\Rightarrow} \Gamma[l \mapsto \mathsf{untaint}(\sigma'_0, \sigma'_1)]} \ \mathsf{fun\text{-}bool}$$

In the rule above, $\mathsf{subst}_{\bar{v}}(\sigma)$ substitutes actual parameters (v_i) for formal parameters in σ.

If the callee returns a string value, we use the return set component of the function summary $(\mathcal{R})$ to determine the set of input parameters and global variables that might become a substring of the return value:

$$\frac{\begin{array}{l}\Gamma \vdash lv \overset{\texttt{Lv}}{\Rightarrow} l \quad \Gamma \vdash e_1 \overset{\texttt{E}}{\Rightarrow} v_1 \ \ldots \ \Gamma \vdash e_n \overset{\texttt{E}}{\Rightarrow} v_n \\ \texttt{Summary}(f) = \langle \mathcal{E}, \mathcal{R}, \mathcal{S}, \mathcal{X} \rangle \quad \sigma' = \mathsf{subst}_{\bar{v}}(\mathcal{R})\end{array}}{\Gamma \vdash lv \leftarrow f(e_1, \ldots, e_n) \overset{\texttt{S}}{\Rightarrow} \Gamma[l \mapsto \mathsf{contains}(\sigma')]} \ \mathsf{fun\text{-}str}$$

Since we require the summary information of a function before we can analyze its callers, the order in which functions are analyzed is important. Due to the dynamic nature of PHP (e.g., include statements), we analyze functions on demand—a function f is analyzed and summarized when we first encounter a call to f. The summary is then memoized to avoid redundant analysis. Effectively, our algorithm analyzes the source codebase in topological order based on the static function call graph. If we encounter a cycle during the analysis, the current implementation uses a dummy "no-op" summary as a model for the second invocation (i.e., we do not compute fix points for recursive functions). In theory, this is a potential source of false negatives, which can be removed by adding an iterative algorithm that handles recursion. However, practically, such an algorithm may be unnecessary given the rare occurrence of recursive calls in PHP programs.

3.4 Experimental Results

The analysis described in Section 3.3 has been implemented as two separate parts: a frontend based on the open source PHP 5.0.5 distribution that parses the source files into abstract syntax trees and a backend written in O'Caml [52] that reads the ASTs into memory and carries out the analysis. This separation ensures maximum compatibility with existing PHP programs while minimizing dependence on the PHP implementation.

The decision to use different levels of abstraction in the intrablock, intraprocedural, and interprocedural levels enabled us to fine tune the amount of information we retain at one level independent of the algorithm used in another and allowed us to quickly build a usable tool. The checker is largely automatic and requires little human intervention for use. We seed the checker with a small set of query functions (e.g., mysql_query) and sanitization operations (e.g., is_numeric). The checker infers the rest automatically.

Regular expression matching presents a challenge to automation. Regular expressions are used for a variety of purposes including, but not limited to, input validation. Some regular expressions match well-formed input while others detect malformed input; assuming one way or the other results in either false positives or false negatives. Our solution is to maintain a database of previously seen regular expressions and their effects, if any. Previously unseen regular expressions are assumed by default to have no sanitization effects, so as not to miss any errors due to incorrect judgment. To make it easy for the user to specify the sanitization effects of regular expressions, the checker has an interactive mode where the user is prompted when the analysis encounters a previously unseen regular expression and the user's answers are recorded for future reference.[6] Having the user declare the role of regular expressions has the real potential to introduce errors into the analysis; however, practically, we found this approach to be very effective and it helped us find at least two vulnerabilities caused by overly lenient regular expressions being used for sanitization.[7] Our tool collected information for 49 regular expressions from the user over all our experiments (the user replies with one keystroke for each inquiry), so the burden on the user is minimal.

The checker detects errors by using information from the summary of the main function—it marks all variables that are required to be sanitized on entry as potential security vulnerabilities. From the checker's perspective, these variables are defined in the environment and used to construct SQL queries without being sanitized. In reality, however, these variables

[6]Here we assume that a regular expression used to sanitize input in one context will have the same effect in another, which, based on our experience, is the common case. Our implementation now provides paranoid users with a special switch that ignores recorded answers and repeatedly ask the user the same question over and over if so desired.

[7]For example, `Utopia News Pro` misused "`[0-9]+`" to validate some user input. This regular expression only checks that the string contains a number, instead of ensuring that the input *is* actually a number. The correct regular expression in this case is "`^[0-9]+$`".

Application (KLOC)	**Err Msgs**	**Bugs (FP)**		**Warn**
News Pro (6.5)	8	8	(0)	8
myBloggie (9.2)	16	16	(0)	23
PHP Webthings (38.3)	20	20	(0)	6
DCP Portal (121)	39	39	(0)	55
e107 (126)	16	16	(0)	23
Total	99	99	(0)	115

Table 3.1: Summary of experiments. **LOC** statistics include embedded HTML, and thus is a rough estimate of code complexity. **Err Msgs**: number of reported errors. **Bugs**: number of confirmed bugs from error reports. **FP**: number of false positives. **Warn**: number of unique warning messages for variables of unresolved origin (uninspected).

are either defined by the runtime environment or by some language constructs that the checker does not fully understand (e.g., the extract operation in PHP which we describe in a case study below). The tool emits an *error* message if the variable is known to be controlled by the user (e.g., $_GET['...'], $_POST['...'], $_COOKIE['...'], etc). For others, the checker emits a *warning*.

We conducted our experiments on the latest versions of six open source PHP code bases: `e107 0.7`, `Utopia News Pro 1.1.4`, `mybloggie 2.1.3beta`, `DCP Portal v6.1.1`, `PHP Webthings 1.4patched`, and `PHP fusion 6.00.204`. Table 3.1 summarizes our findings for the first five. The analysis terminates within seconds for each script examined (which may dynamically include other source files). Our checker emitted a total of 99 error messages for the first five applications, where unsanitized user input (from $_GET, $_POST, etc) may flow into SQL queries. We manually inspected the error reports and believe all 99 represent real vulnerabilities. We have notified the developers about these errors and five security advisories have been issued by Secunia based on the bug reports. We have not inspected warning messages—unsanitized variables of unresolved origin (e.g., from database queries, configuration files, etc) that are subsequently used in SQL queries due to the high likelihood of false positives.

PHP-fusion is different from the other five code bases because it does not directly access HTTP form data from input hash tables such as $_GET and $_POST. Instead it uses the extract operation to automatically import such information into the current variable scope. We describe our findings for PHP-fusion in the following subsection.

3.4.1 Case Study: Two SQL Injection Attacks in PHP-fusion

In this section, we show two case studies of exploitable SQL injection vulnerabilities in PHP-fusion detected by our tool. PHP-fusion is an open-source content management system (CMS) built on PHP and MySQL. Excluding locale specific customization modules, it consists of over 16,000 lines of PHP code and has a wide user-base because of its speed, customizability and rich features. Browsing through the code, it is obvious that the author programmed with security in mind and has taken extra care in sanitizing input before use in query strings.

Our experiments were conducted on the then latest 6.00.204 version of the software. Unlike other code bases we have examined, PHP-fusion uses the extract operation to import user input into the current scope. As an example, extract($_POST, EXTR_OVERWRITE) has the effect of introducing one variable for each key in the $_POST hash table into the current scope, and assigning the value of $_POST[key] to that variable. This feature reduces typing, but introduces confusion for the checker and security vulnerabilities into the software—both of the exploits we constructed involve use of uninitialized variables whose values can be manipulated by the user because of the extract operation.

Since PHP-fusion does not directly read user input from input hashes such as $_GETor $_POST, there are no direct error messages generated by our tool. Instead we inspect warnings (recall the discussion about errors and warnings above), which correspond to security sensitive variables whose definition is unresolved by the checker (e.g., introduced via the extract operation, or read from configuration files).

We ran our checker on all top level scripts in PHP-fusion. The tool generated 22 unique warnings, a majority of which relate to configuration variables that are used in the construction of a large number of queries.[8] After filtering those out, 7 warnings in 4 different files remain.

We believe all but one of the 7 warnings may result in exploitable security vulnerabilities. The lone false positive arises from an unanticipated sanitization:

```
/* php-files/lostpassword.php */
if (!preg_match("/^[0-9a-z]{32}$/", $account))
        $error = 1;
if (!$error) { /* database access using $account */ }
if ($error) redirect("index.php");
```

[8]Database configuration variables such as $db_prefix accounted for 3 false positives, and information derived from the database queries and configuration settings (e.g., locale settings) caused the remaining 12.

Instead of terminating the program immediately based on the result from preg_match, the program sets the $error flag to true and delays error handling. This idiom can be handled by adding more information in the block summary.

We investigated the first two of the remaining warnings for potential exploits and confirmed that both are indeed exploitable on a test installation. Unsurprisingly both errors are made possible because of the extract operation. We explain these two errors in detail below.

1) Vulnerability in script for recovering lost password. This is a remotely exploitable vulnerability that allows any registered user to elevate his privileges via a carefully constructed URL. We show the relevant code below:

```
1 /* php-files/lostpassword.php */
2 for ($i=0;$i<=7;$i++)
3    $new_pass .= chr(rand(97, 122));
4 ...
5 $result = dbquery("UPDATE ".$db_prefix."users
6   SET user_password=md5('$new_pass')
7   WHERE user_id='".$data['user_id']."'");
```

Our tool issued a warning for $new_pass, which is uninitialized on entry and thus defaults to the empty string during normal execution. The script proceeds to add seven randomly generated letters to $new_pass (lines 2-3), and uses that as the new password for the user (lines 5-7). The SQL request under normal execution takes the following form:

```
UPDATE users SET user_password=md5('???????')
    WHERE user_id='userid'
```

However, a malicious user can simply add a new_pass field to his HTTP request by appending, for example, the following string to the URL for the password reminder site:

```
&new_pass=abc%27%29%2cuser_level=%27103%27%2cuser_aim=%28%27
```

The extract operation described above will magically introduce $new_pass in the current variable scope with the following initial value:

$$\mathsf{abc}'), \mathsf{user\_level} =' 103', \mathsf{user\_aim} = ('$$

The SQL request is now constructed as:

```
UPDATE users SET user_password=md5('abc'),
      user_level='103', user_aim=('???????')
   WHERE user_id='userid'
```

Here the password is set to "abc", and the user privilege is elevated to 103, which means "Super Administrator." The newly promoted user is now free to manipulate any content

```
 1 if (isset($msg_view)) {
 2   if (!isNum($msg_view)) fallback("messages.php");
 3   $result_where_message_id="message_id=".$msg_view;
 4 } elseif (isset($msg_reply)) {
 5   if (!isNum($msg_reply)) fallback("messages.php");
 6   $result_where_message_id="message_id=".$msg_reply;
 7 }
 8 ... /* ~100 lines later */ ...
 9 } elseif (isset($_POST['btn_delete']) ||
10   isset($msg_delete)) { // delete message
11   $result = dbquery("DELETE FROM ".$db_prefix.
12    "messages WHERE ".$result_where_message_id. // BUG
13    " AND ".$result_where_message_to);
```

Figure 3.5: An exploitable vulnerability in PHP-fusion 6.00.204.

on the website.

2) Vulnerability in the messaging sub-system. This vulnerability exploits another use of potentially uninitialized variable $result_where_message_id in the messaging sub system. We show the relevant code in Figure 3.5.

Our tool warns about unsanitized use of $result_where_message_id. On normal input, the program initializes $result_where_message_id using a cascading `if` statement. As shown in the code, the author is very careful about sanitizing values that are used to construct $result_where_message_id. However, the cascading sequence of `if` statements does not have a default branch. And therefore, $result_where_message_id might be uninitialized on malformed input. We exploit this fact, and append

&request_where_message_id=1=1/*

The query string submitted on line 11-13 thus becomes:

DELETE FROM messages **WHERE** 1=1 /* AND ...

Whatever follows "/*" is treated as comments in MySQL and thus ignored. The result is loss of all private messages in the system. Due to the complex control and data flow surrounding the variable, this error is unlikely to be discovered via code review or testing.

We reported both exploits to the author of PHP-fusion, who immediately fixed these vulnerabilities and released a new version of the software.

3.5 Related Work

3.5.1 Static techniques

WebSSARI is a type-based analyzer for PHP [42]. It uses a simple intraprocedural tainting analysis to find cases where user controlled values flow into functions that require trusted input (i.e., *sensitive functions*). The analysis relies on three user written "prelude" files to provide information regarding: 1) the set of all sensitive functions–those require sanitized input; 2) the set of all untainting operations; and 3) the set of untrusted input variables. Incomplete specification results in both substantial numbers of false positives and false negatives.

WebSSARI has several key limitations that restrict the precision and analysis power of the tool:

1. WebSSARI uses an intraprocedural algorithm and thus only models information flow that does not cross function boundaries.

   Large PHP codebases typically define a number of application specific subroutines handling common operations (e.g., query string construction, authentication, sanitization, etc) using a small number of system library functions (e.g., mysql_query). Our algorithm is able to automatically infer information flow and pre- and post-conditions for such user-defined functions whereas WebSSARI relies on the user to specify the constraints of each, a significant burden that needs to be repeated for each source codebase examined. Examples in Section 3.3.3 represent some common forms of user-defined functions that WebSSARI is not able to model without annotations.

   To show how much interprocedural analysis improves the accuracy of our analysis, we turned off function summaries and repeated our experiment on `Utopia News Pro`, the smallest of the five codebases. This time, the analysis generated 19 error messages (as opposed to 8 with interprocedural analysis). Upon inspection, all 11 extra reports are false positives due to user-defined sanitization operations.

2. WebSSARI does not seem to model conditional branches, which represent one of the most common forms of sanitization in the scripts we have analyzed. For example, we believe it will report a false warning on the following code:

```
if (!is_numeric($_GET['x']))
    exit;
mysql_query("... $_GET['x'] ...");
```

Furthermore, interprocedural conditional sanitization (see the example in Section 3.3.1.6) is also fairly common in codebases.

3. WebSSARI uses an algorithm based on static types that does not specifically model dynamic features in scripts. For example, dynamic typing may introduce subtle errors that WebSSARI misses. The include statement, used extensively in PHP scripts, dynamically inserts code to the program which may contain, induce, or prevent errors.

We are unable to directly compare the experimental results due to the fact that neither the bug reports nor the WebSSARI tool are available publicly. Nor are we able to compare false positive rates since WebSSARI reports per-file statistics which may underestimate the false positive ratio. A file with 100 false positives and 1 real bug is considered to be "vulnerable" and therefore does not contribute to the false positive rate computed in [42].

Livshits and Lam [55] develop a static detector for security vulnerabilities (e.g., SQL injection, cross site scripting, etc) in Java applications. The algorithm uses a BDD-based context-sensitive pointer analysis [77] to find potential flow from untrusted sources (e.g., user input) to trusting sinks (e.g., SQL queries). One limitation of this analysis is that it does not model control flow in the program and therefore may misflag sanitized input that subsequently flows into SQL queries. Sanitization with conditional branching is common in PHP programs, so techniques that ignore control flow may cause large numbers of false positives on such code bases. On the other hand, our technique does not model pointers, which are prevalent in Java programs.

Other tainting analysis that are proven effective on C code include CQual [30], MECA [88], and MC [35, 5]. Collectively they have found hundreds of previously unknown security errors in the Linux kernel.

Christensen *et al* [18] develop a string analysis that approximates string values in a Java program using a context free grammar. The result is widened into a regular language and checked against a specification of expected output to determine syntactic correctness. However, syntactic correctness does not entail safety, and therefore it is unclear how to adapt this work to the detection of SQL injection vulnerabilities. Minamide [58] extends the approach and constructs a string analyzer for PHP, citing SQL injection detection as a possible application. However, the analyzer models a small set of string operations in PHP (e.g., concatenation, string matching and replacement) and ignores more complex features such as dynamic typing, casting, and predicates. Furthermore, the framework only seems to model sanitization with string replacement, which represents a small subset of all sanitization in real code. Therefore, accurately pinpointing injection attacks remains challenging.

Gould *et al* [32] combine string analysis with type checking to ensure not only syntactic correctness but also type correctness for SQL queries constructed by Java programs. However, type correctness does not imply safety, which is the focus of our analysis.

3.5.2 Dynamic Techniques

Scott and Sharp [69] propose an application-level firewall to centralize sanitization of client input. Firewall products are also commercially available from companies such as NetContinuum, Imperva, Watchfire, etc. Some of these firewalls detect and guard against previously known attack patterns, while others maintain a white list of valid inputs. The main limitation here is that the former is susceptible to both false positives and false negatives, and the latter is reliant on correct specifications, which are difficult to come by.

Martin et al [57] propose a query language called PQL that allows users to write queries about runtime event patterns. They show how to use the language to find a variety of program errors in Java, including SQL injections. The Perl taint mode [64] enables a set of special security checks during execution in an unsafe environment. It prevents the use of untrusted data (e.g., all command line arguments, environment variables, data read from files, etc) in operations that require trusted input (e.g., any command that invokes a subshell). Nguyen-Tuong *et al* [62] propose a taint mode for PHP, which, unlike the Perl taint mode, does not define sanitizing operations. Instead, it tracks each character in the user input individually, and employs a set of heuristics to determine whether a query is safe when it contains fragments of user input. For example, among others, it detects an injection if an operator symbol (e.g., "(", ")", "%", etc) is marked as tainted. This approach is susceptible to both false positives and false negatives. Note that static analyses are also susceptible to both false positives and false negatives. The key distinction is that in static analyses, inaccuracies are resolved at compile time instead of at runtime, which is much less forgiving.

3.6 Conclusion

In this chapter, we have presented a static analysis algorithm for detecting security vulnerabilities in PHP. Our analysis employs a novel three-tier architecture that enables us to handle dynamic features unique to scripting languages such as dynamic typing and code inclusion. We have demonstrated the effectiveness of our approach by running our tool on six popular open source PHP code bases and finding 105 previously unknown security vulnerabilities, most of which we believe are remotely exploitable.

Chapter 4

Using Redundancy to Find Errors

4.1 Introduction

Programming language tools have long exploited the fact that many high-level conceptual errors map to low-level type errors. In this chapter, we demonstrate a twist on this idea: many high-level conceptual errors also map to low-level redundant operations. With the exception of a few stylized cases, programmers are generally attempting to perform useful work. If they perform an action, it was because they believe it serves some purpose. Spurious operations violate this belief and are likely errors. However, in the past redundant operations have been typically regarded as merely cosmetic problems, rather than serious mistakes. Evidence for this perception is that to the best of our knowledge the many recent error checking projects focus solely on hard errors such as NULL pointer dereferences or failed lock releases, rather than redundancy checking [2, 7, 9, 22, 29, 68, 76].

We experimentally demonstrate that in fact many redundancies signal mistakes as serious as traditional hard errors. For example, impossible Boolean conditions can signal erroneous expressions; critical sections without shared states can signal the use of the wrong variable; variables written but not read can signal an unintentionally lost result. Even when harmless, these redundancies signal conceptual confusion, which we would also expect to correlate with hard errors such as deadlocks, NULL pointer dereferences, etc.

In this chapter we use redundancies to find errors in three ways: (1) by writing checkers that automatically flag redundancies, (2) using these errors to predict non-redundant errors (such as NULL pointer dereferences), and (3) using redundancies to find incomplete program specifications. We discuss each below.

We wrote five checkers that flag potentially dangerous redundancies: (1) idempotent operations, (2) assignments that were never read, (3) dead code, (4) conditional branches that were never taken, and (5) redundant NULL-checks. The errors found would largely be missed by traditional type systems and checkers. For example, as Section 4.2 shows, assignments of variables to themselves can signal mistakes, yet such assignments type check in all common programming languages.

Of course, some legitimate actions cause redundancies. Defensive programming may introduce "unnecessary" operations for robustness; debugging code, such as assertions, can check for "impossible" conditions; and abstraction boundaries may force duplicate calculations. Thus, to effectively find errors, our checkers must separate such redundancies from those induced by high-level confusion.

The technology behind the checkers is not new. Optimizing compilers use redundancy detection and elimination algorithms extensively to improve code performance. One contribution of our work is the realization that these analyses have been silently finding errors since their invention.

We wrote our redundancy checkers in the *MC* extensible compiler system [35], which makes it easy to build system-specific static analyses. Our analyses do not depend on an extensible compiler, but MC does make it easier to prototype and perform focused suppression of false positive classes.

We evaluated how effective flagging redundant operations is at finding dangerous errors by applying the above five checkers to three open source software projects: Linux, OpenBSD and PostgreSQL. These are large, widely-used source code bases (we check 3.3 million lines of code) that serve as a known experimental base. Because they have been written by many people, they are representative of many different coding styles and abilities.

We expect that redundancies, even when harmless, strongly correlate with hard errors. Our relatively uncontroversial hypothesis is that confused or incompetent programmers tend to make mistakes. We experimentally test this hypothesis by taking a large database of hard Linux errors that were found in prior work [17] and measure how well redundancies predict these errors. In our experiments, files that contain redundancies are roughly 45% to 100% more likely to have traditional hard errors compared to those drawn by chance. This difference holds across the different types of redundancies.

Finally, we discuss how traditional checking approaches based on annotations or specifications can use redundancy checks as a safety net to find missing annotations or incomplete specifications. Such specification mistakes commonly map to redundant operations. For example, assume we have a specification that binds shared variables to locks. A missed binding will likely lead to redundancies: a critical section with no shared state and locks that protect no variables. We can flag such omissions because we know that every lock should protect some shared variable and that every critical section should contain some shared state.

The techniques described in this chapter make four contributions:

1. The idea that redundant operations, like type errors, commonly flag serious correctness errors.
2. Experimentally validating this idea by writing and applying five redundancy checkers to real code. The errors found often surprised us.
3. Demonstrating that redundancies, even when harmless, strongly correlate with the presence of traditional hard errors.
4. Showing how redundancies provide a way to detect dangerous specification omissions.

The main caveat with our approach is that the errors we count might not be errors since we were examining code we did not write. As in Chapter 2, we only diagnosed errors we were reasonably sure about.

In addition, some of the errors we diagnose are not traditional "hard errors"– they by themselves would probably not cause system crashes or security breaches. Rather, they are nonsensical redundancies that in our opinion result in unnecessary complexity and confusion. So the diagnosis of these errors involve personal judgments that may not be shared by all readers. Although they are not as serious as hard errors, we think they should nevertheless be fixed in order to improve program clarity and readability.

Sections 4.2 through 4.6 present the five checkers. Section 4.7 measures how well these redundant errors correlate with and predict traditional hard errors. Section 4.8 discusses how to check for completeness using redundancies. Section 4.9 discusses related work. Finally, Section 4.10 concludes.

4.2 Idempotent Operations

The checker in this section flags idempotent operations where a variable is: (1) assigned to itself (`x = x`), (2) divided by itself (`x / x`), (3) bitwise or'd with itself (`x | x`) or (4) bitwise

System	Bugs	Minor	False
Linux 2.4.5-ac8	7	6	3
OpenBSD 3.2	2	6	8
PostgreSQL	0	0	0

Table 4.1: Bugs found by the idempotent checker in Linux version 2.4.5-ac8, OpenBSD 3.2, and PostgreSQL 7.2.

and'd with itself (`x & x`). This checker is the simplest in the chapter (it requires about 10 lines of code in our system). Even so, it found several interesting cases where redundancies signal high-level errors. Four of these were apparent typos in variable assignments. The clearest example was the following Linux code, where the programmer makes a mistake while copying structure `sa` to `da`:

```
/* linux2.4.1/net/appletalk/aarp.c:aarp_rcv */
else { /* We need to make a copy of the entry. */
    da.s_node = sa.s_node;
    da.s_net = da.s_net;
```

This is a good example of how redundancies catch cases that type systems miss. This code—an assignment of an integer variable to itself—type checks in all common languages we know of, yet clearly contains an error. Two of the other errors in Linux were caused by integer overflow (or'ing an 8-bit variable by a constant that only had bits set in the upper 16 bits) which was optimized away by the `gcc` frontend. The final error in Linux was caused by an apparently missing conversion routine. The code seemed to have been tested only on a machine where the conversion was unnecessary, which prevented the tester from noticing the missing functionality.

The two errors we found in OpenBSD are violations of the ANSI C standard. They both lie in the same function in the same source file. We show one of them below:

```
1 /* openbsd3.2/sys/kern/subr_userconf.c:userconf_add */
2 for (i = 0; i < pv_size; i++) {
3     if (pv[i] != -1 && pv[i] >= val)
4             pv[i] = pv[i]++; /* error */
5 }
```

The error occurs at line 4, and is detected with the help of the code canonicalization algorithm in the *xgcc* front end that translates this statement into:

```
pv[i] = pv[i]; /* redundant */
pv[i]++;
```

The ANSI C standard (Section 6.3) stipulates that "between the previous and next sequence point an object shall have its stored value modified at most once by the evaluation of an expression." It is mere coincidence that `gcc` chooses to implement the side-effects of `pv[i]++` after that of the assignment itself. In fact, the Compaq C compiler[1] evaluates `pv[i]++` first and stores the *old* value back to `pv[i]`, causing the piece of code to fail in a non-obvious way. We tested four C compilers by different vendors on different architectures.[2] None of them issued a warning on this illegal statement.

The minor errors were operations that seemed to follow a nonsensical but consistent coding pattern, such as adding 0 to a variable for typographical symmetry with other non-zero additions such as the following

```
/* linux2.4.5-ac8/drivers/video/riva/riva_hw.c:nv4CalcArbitration */
nvclks += 1;
nvclks += 1;
nvclks += 1;
if (mp_enable)
    mclks+=4;
nvclks += 0; /* suspicious, is it a typo or should it really be "+=1"? */
```

Curiously, each of the eleven false positives we found was annotated with a comment explaining why the redundant operation was being done. This gives evidence for our belief that programmers regard redundancy as somewhat unusual.

Macros are the main source of false positives. They represent logical operations that may not map to concrete actions. For example, networking code contains many calls of the form "`x = ntohs(x)`" used to reorder the bytes in variable `x` in a canonical "network order" so that a machine receiving the data can unpack it appropriately. However, on machines on which the data is already in network order, the macro expands to nothing, resulting in code that will simply assign `x` to itself. To suppress these false positives, we modified the preprocessor to note which lines contain macros — we simply ignore warnings on these lines.

4.3 Redundant Assignments

The checker in this section flags cases where a value assigned to a variable is not subsequently used. The checker tracks the lifetime of variables using a simple intraprocedural analysis.

[1]Compaq C V6.3-129 (dtk) on Compaq Tru64 UNIX V5.0A (Rev. 1094).

[2]Sun Workshop 6 update 2, GNU GCC 3.2.1, Compaq C V6.3-129, and MS Visual Studio .NET.

System	**Bugs**	**False**	**Uninspected**
Linux 2.4.5-ac8	129	26	1840
OpenBSD 3.2	63	36	117
PostgreSQL 7.2	37	10	0

Table 4.2: Bugs found by the redundant assignment checker in Linux version 2.4.5-ac8, OpenBSD 3.2, and PostgreSQL 7.2.

At each assignment it follows the variable forward on all paths. It emits an error message if the variable is not read on any path before either exiting scope or being assigned another value. As we show, in many cases such lost values signal real errors, such as control flow following unexpected paths, results computed but not returned, etc.

The checker found thousands of redundant assignments in the three systems we checked. Since it was so effective, we minimized the chance of false positives by restricting the variables it would follow to non-global and non-volatile ones that were not aliased in any way (i.e., local variables that never had their addresses taken).

Most of the checker code deals with differentiating the errors into three classes, which it ranks in the following order:

1. Variables assigned values that are not read, and which are never subsequently reassigned. Empirically, these errors tend to be the most serious, since they flag unintentionally lost results.

2. Variables assigned a non-constant value that are not read, and which are subsequently reassigned. These are also commonly errors, but tend to be less severe. False positives in this class tend to come from assigning a return value from a function call to a dummy variable that is ignored, which is prevalent in PostgreSQL. Fortunately, such variables tend to share a consistent naming pattern (e.g., they are commonly prefixed with double underscores (__) in PostgreSQL) and therefore can be easily suppressed with `grep`. In presenting bug counts in Table 4.2, we do not count warnings that are so suppressed.

3. Variables assigned a constant and then reassigned other values without being read. These are frequently due to defensive programming, where the programmer always initializes a variable to some safe value (most commonly: `NULL`, 0, `0xffffffff`, and -1) but does not read it before redefinition. We track the initial value and emit it

```
/* linux2.4.5-ac8/net/decnet/af_decnet.c:dn_wait_run */
do {
  ...
  if (signal_pending(current)) {
     err = −ERESTARTSYS; /* BUG: lost value */
     break;
  }
  ...
} while(scp−>state != DN_RUN);
return 0;    /* should be "return err" here */
```

Figure 4.1: Lost return value caught by flagging the redundant assignment to `err`.

when reporting the error so that messages with a common defensive value can be easily filtered out. Again, we do not count filtered messages in Table 4.2.

Suppressing false positives. As the other redundancy checkers, macros and defensive programming cause most false positives. To minimize false positives, the checker does not track variables killed or generated within macros. Its main remaining vulnerability are values assigned and then passed to debugging macros that are turned off:

```
x = foo−>bar;
DEBUG("bar = %d", x);
```

Typically there are a small number of such debugging macros, which we manually turn back on by modifying the header file in which they are defined.

We use ranking to minimize the impact of defensive programming. Redundant operations that can be errors when done within the span of a few lines can be robust programming practice when separated by 20 lines. Thus we rank reported errors based on (1) the line distance between the assignment and reassignment and (2) the number of conditional branches on the path. Close errors are most likely; farther ones arguably defensive programming.

The errors. This checker found more errors than all the others we have written combined. There were two interesting error patterns that showed up as redundant assignments: (1) variables whose values were (unintentionally) discarded and (2) variables whose values were not used because of surprising control flow (e.g., an unexpected return).

The majority of the errors (126 of the 129 diagnosed ones) fall into the first category. Figure 4.1 shows a representative example from Linux. Here, if the function `signal_pending` returns true (a signal is pending to the current process), an error code is set (`err = -ERESTARTSYS`) and the code breaks out of the enclosing loop. The value in `err` must

```
/* linux2.4.1/net/atm/lec.c:lec_addr_delete: */
for(entry=priv->lec_arp_tables[i];
      entry != NULL;
      entry=next) { /* BUG: never reached */
   next = entry->next;
   if (...) {
         lec_arp_remove(priv->lec_arp_tables, entry);
         kfree(entry);
   }
   lec_arp_unlock(priv);
   return 0;
}
```

Figure 4.2: A single-iteration loop caught by flagging the redundant assignment `next = entry->next`. The assignment appears to be read in the loop iteration statement (`entry = next`) but it is dead code, since the loop always exits after a single iteration. If the entry the loop is trying to delete is not the first one in the list, it is not deleted.

be passed back to the caller so that it will retry the system call. However, the code always returns 0 to the caller no matter what happens inside the loop, which leads to an insidious error: the code usually works but, occasionally, it aborts and returns a success code, causing the client to assume that the operation succeeded. There were numerous similar errors on the caller side where the result of a function was assigned to a variable but then ignored rather than being checked.

The second class of errors contains three diagnosed errors stemming from calculations aborted by unexpected control flow. Figure 4.2 gives one example from Linux: here all paths through a loop end in a `return`, wrongly aborting the loop after a single iteration. This error is caught by the fact that an assignment used to walk down a linked list is never read because the loop iterator that would do so is dead code. Figure 4.3 shows a variation on the theme of unexpected control flow. Here an `if` statement has an extraneous statement terminator at its end, causing the subsequent `return` to be always taken. In these cases, a coding mistake caused "dangling assignments" that were not used. This fact allows us to flag such unexpected structures even when we do not know how control flows in the code. It is the the presence of these errors that has led us to write the dead-code checker in the next section.

Reassigning values is typically harmless, but it does signal fairly confused programmers. For example:

```
/* linux2.4.5-ac8/fs/ntfs/unistr.c:ntfs_collate_names */
for (cnt = 0; cnt < min(name1_len, name2_len); ++cnt) {
   c1 = le16_to_cpu(*name1++);
   c2 = le16_to_cpu(*name2++);
   if (ic) {
       if (c1 < upcase_len)
           c1 = le16_to_cpu(upcase[c1]);
       if (c2 < upcase_len)
           c2 = le16_to_cpu(upcase[c2]);
   }
   /* [META] stray terminator! */
   if (c1 < 64 && legal_ansi_char_array[c1] & 8);
       return err_val;
   if (c1 < c2)
       return −1;
   ...
```

Figure 4.3: Catastrophic return caught by the redundant assignment to `c2`. The last conditional is accidentally terminated because of a stray statement terminator ("`;`") at the end of the line, causing the routine to always return `err_val`.

```
/* linux2.4.1/net/ipv6/raw.c:rawv6_getsockopt */
switch (optname) {
  case IPV6_CHECKSUM:
     if (opt−>checksum == 0)
         val = −1;
     else
         val = opt−>offset;
     /* BUG: always falls through */
  default:
     return −ENOPROTOOPT;
  }
  len=min(sizeof(int),len);
  ...
```

Figure 4.4: Unintentional switch "fall through" causing the code to always return an error. This maps to the low-level redundancy that the value assigned to `val` is never used.

```
/* linux2.4.5-ac8/drivers/net/wan/sdla_x25.c:
              alloc_and_init_skb_buf */
struct sk_buff *new_skb = *skb;
new_skb = dev_alloc_skb(len + X25_HRDHDR_SZ);
```

where `new_skb` is assigned the value `*skb` but then immediately reassigned another allocated value. A different case shows a potential confusion about how C's iteration statement works:

```
/* linux2.4.5-ac8/drivers/scsi/scsi.c:
        scsi_bottom_half_handler */
SCnext = SCpnt->bh_next;
for (; SCpnt; SCpnt = SCnext) {
    SCnext = SCpnt->bh_next;
```

Note that the variable `SCnext` is assigned and then immediately reassigned in the loop. The logic behind this decision remains unclear.

The most devious error. A few of the values reassigned before being used were suspicious lost values. One of the worst (and most interesting) was from a commercial system which had the equivalent of the following code:

```
        c = p->buf[0][3];
        c = p->buf[0][3];
```

At first glance this seems like a harmless obvious copy-and-paste error. It turned out that the redundancy flags a much more devious bug. The array `buf` actually pointed to a *memory mapped* region of kernel memory. Unlike normal memory, reads and writes to this region cause the CPU to issue I/O commands to a hardware device. Thus, the reads are not idempotent, and the two of them in a row rather than just one can cause very different results to happen. However, the above code does have a real (but silent) error — in the variant of C that this code was written, pointers to device memory must be declared as `volatile`. Otherwise the compiler is free to optimize duplicate reads away, especially since in this case there were no pointer stores that could change their values. Dangerously, in the above case `buf` was declared as a normal pointer rather than a volatile one, allowing the compiler to optimize as it wished. Fortunately the error had not been triggered because the GNU C compiler that was being used had a weak optimizer that conservatively did not optimize expressions that had many levels of indirection. However, the use of a more aggressive compiler or a later version of `gcc` could have caused this extremely difficult to track down bug to surface.

System	Bugs	False
Linux 2.4.5-ac8	66	26
OpenBSD 3.2	11	4
PostgreSQL 7.2	0	0

Table 4.3: Bugs found by the dead code checker on Linux version 2.4.5-ac8, OpenBSD 3.2, and PostgreSQL 7.2.

```
/* linux2.4.1/drivers/char/rio/rioparam.c:RIOParam */
if (retval == RIO_FAIL) {
  rio_spin_unlock_irqrestore(&PortP->portSem, flags);
  pseterr(EINTR); /* BUG: returns */
  func_exit();
  return RIO_FAIL;
}
```

Figure 4.5: Unexpected return: The call `pseterr` is a macro that returns its argument value as an error. Unfortunately, the programmer does not realize this and inserts subsequent operations, which are flagged by our dead code checker. There were many other similar misuses of the same macro.

4.4 Dead Code

The checker in this section flags dead code. Since programmers generally write code to run it, dead code catches logical errors signaled by false beliefs that unreachable code can execute.

The core of the dead code checker is a straightforward mark-and-sweep algorithm. For each routine it (1) marks all blocks reachable from the routine's entry node and (2) traverses all blocks in the routine, flagging any that are not marked. The checker has three modifications to this basic algorithm. First, it truncates all paths that reach functions that would not return. Examples include "`panic`," "`abort`" and "`BUG`" which are used by Linux to signal a terminal kernel error and reboot the system — code dominated by such calls cannot run. Second, we suppress error messages for dead code caused by constant conditions involving macros or the `sizeof` operator, such as

```
/* linux2.4.1/drivers/scsi/53c7,8xx.c:
    return_outstanding_commands */
for (c = hostdata->running_list; c;
     c = (struct NCR53c7x0_cmd *) c->next) {
    if (c->cmd->SCp.buffer) {
        printk ("...");
        break;
    } else {
        printk ("Duh? ...");
        break;
    }
    /* BUG: cannot be reached */
    c->cmd->SCp.buffer =
        (struct scatterlist *) list;
    ...
```

Figure 4.6: Broken loop: the first `if-else` statement of the loop contains a `break` on both paths, causing the loop to always abort, without ever executing the subsequent code it contains.

```
/* linux2.4.5-ac8/net/decnet/dn_table.c:
        dn_fib_table_lookup */
for(f = dz_chain(k, dz); f; f = f->fn_next) {
    if (!dn_key_leq(k, f->fn_key))
        break;
    else
        continue;

   /* BUG: cannot be reached */
   f->fn_state |= DN_S_ACCESSED;

   if (f->fn_state&DN_S_ZOMBIE)
      continue;
   if (f->fn_scope < key->scope)
      continue;
```

Figure 4.7: Useless loop body: similarly to Figure 4.6 this loop has a broken `if-else` statement. One branch aborts the loop, the other uses C's `continue` statement to skip the body and begin another iteration.

```
/* openbsd3.2/sys/dev/pci/bktr/bktr_core.c:tuner_ioctl */
unsigned temp;
...
temp = tv_channel( bktr, (int)*(unsigned long *)arg );
if ( temp < 0 ) { /* gcc frontend turns this into 0 */
    /* BUG: cannot be reached */
    temp_mute( bktr, FALSE );
    return( EINVAL );
}
*(unsigned long *)arg = temp;
```

Figure 4.8: Unsigned variable tested for negativity.

```
/* case 1: debugging statement that is turned 'off' */
#define DEBUG 0
if (DEBUG) printf("in foo");

/* case 2: gcc coverts the condition to 0
   on 32-bit architectures */
if (sizeof(int) == 64)
    printf("64 bit architecture\n");
```

The former frequently signals code "commented out" by using a false condition, while the latter is used to carry out architecture dependent operations. We also annotate error messages when the flagged dead code is only a `break` or `return`. These are commonly a result of defensive programming. Finally, we suppress dead code caused by macros.

Despite its simplicity, dead code analysis found a high number of clearly serious errors. Three of the errors caught by the redundant assignment checker are also caught by the dead code detector: (1) the single iteration loop in Figure 4.2, (2) the mistaken statement terminator in Figure 4.3, and (3) the unintentional fall through in Figure 4.4.

Figure 4.5 gives the most frequent copy-and-paste error. Here the macro "`pseterr`" has a `return` statement in its definition, but the programmer does not realize it. Thus, at all seven call sites that use the macro, there is dead code after the macro that the programmer intended to have executed.

Figure 4.6 gives another common error — a single-iteration loop that always terminates because it contains an `if-else` statement that breaks out of the loop on both branches. It is hard to believe that this code was ever tested. Figure 4.7 gives a variation on this, where one branch of the `if` statement breaks out of the loop but the other uses C's `continue`

System	Bugs	False	Uninspected
Linux 2.4.5-ac8	49	52	169
OpenBSD 3.2	64	33	316
PostgreSQL 7.2	0	0	0

Table 4.4: Bugs found by the redundant conditionals checker in Linux 2.4.5-ac8, OpenBSD 3.2, PostgreSQL 7.2.

statement, which skips the rest of the loop body. Thus, none of the code thereafter can be executed.

Type errors can also result in dead code. Figure 4.8 shows an example from OpenBSD. Here the function `tv_channel` returns `-1` on error, but since `temp` is an unsigned variable, the error handling code in the true branch of the `if` statement is never executed. An obvious fix is to declare `temp` as `int`. As before, none of the four compilers we tested warned about this suspicious typing mistake.

4.5 Redundant Conditionals

The checker in this section uses path-sensitive analysis to detect redundant (always true or always false) conditionals in branch statements such as `if`, `while`, `switch`, and etc. Such conditionals cannot affect the program state or control flow. Thus, their presence is a likely indicator of errors. To avoid double reporting bugs found in the previous section, we only flag non-constant conditional expressions that are not evaluated by the dead code checker.

The implementation of the checker uses the false path pruning (FPP) feature in the *xgcc* system [35]. FPP was originally designed to eliminate false positives from infeasible paths. It prunes a subset of logically inconsistent paths by keeping track of assignments and conditionals along the way. The implementation of FPP consists of three separate modules, each keeping track of one class of program properties as described below:

1. The first module maintains a mapping from variables to integer constants. It tracks assignments (e.g., `x = 1; x = x * 2;`) and conditional branch statements (e.g., `if (x == 5) {...}`) along each execution path to derive variable-constant bindings.

2. The second module keeps track of a known set of predicates that must hold true along the current program path. These predicates are collected from conditional branches (e.g., in the true branch of `if (x != NULL) {...}`, we collect `x != NULL` into the

```
1 /* linux2.4.5-ac8/fs/fat/inode.c */
2 error = 0;
3 if (!error) { /* causes unnecessary code complexity */
4     sbi->fat_bits = fat32 ? 32 :
5         (fat ? fat :
6         (sbi->clusters > MSDOS_FAT12 ? 16 : 12));
```

Figure 4.9: Nonsensical programming style: the check at line 3 is clearly redundant.

known predicate set) and are used later to test the validity of subsequent control predicates (e.g., if we encounter `if (x != NULL)` later in the program, we prune the `else` branch). Note if the value of variable `x` is killed (either directly by assignments or indirectly through pointers), we remove any conditional in the known predicate set that references `x`.

3. The third module keeps track of constant bounds of variables. We derive these bounds from conditional statements. For example, in the true branch of `if (x <= 5)`, we enter 5 as `x`'s upper bound. We prune paths that contradict the recorded bound information (e.g., the true branch of `if (x > 7)`).

These three simple modules capture enough information to find interesting errors in Linux and OpenBSD.

With FPP, the checker works as follows: for each function, the checker traverses the control flow graph (CFG) and marks all reachable CFG edges. At the end of the analysis, the checker emits conditionals with untraversed edges as errors.

Macros and concurrency are the two major sources of false positives for this checker. To suppress these false positives, we discard warnings that take place within macros, and ignore conditionals that involve global, static or volatile variables whose values might be changed by another thread.

The checker finds hundreds of redundant conditionals in Linux and OpenBSD. We group them into three major categories, described below.

The first class of errors, which are the least serious of the three, are labeled "nonsensical programming style." Figure 4.9 shows a representative example from Linux 2.4.5-ac8. The `if` statement at line 3 is clearly redundant and leaves one to wonder about the purpose of such a check. These errors, although harmless, signal a confused programmer, which is supported by the statistical analysis described in section 4.7.

```
1  /* linux2.4.5-ac8/drivers/net/wan/sbni.c:sbni_ioctl */
2  slave = dev_get_by_name(tmpstr);
3  if(!(slave && slave->flags & IFF_UP &&
4       dev->flags & IFF_UP))
5  {
6    ... /* print some error message, back out */
7    return -EINVAL;
8  }
9  if (slave) {   ... }
10 /* BUG: !slave is impossible */
11 else {
12   ... /* print some error message */
13   return -ENOENT;
14 }
```

Figure 4.10: Nonsensical programming style: the check of `slave` at line 9 is guaranteed to be true; also notice the difference in return value.

```
/* linux2.4.5-ac8/drivers/net/tokenring/smctr.c:
                              smctr_rx_frame */
while((status = tp->rx_fcb_curr[queue]
                    ->frame_status) != SUCCESS)
{
    err = HARDWARE_FAILED;
    ... /* large chunk of apparent recovery code,
           with no updates to err */
    if (err != SUCCESS)
        break;
}
```

Figure 4.11: Redundant conditional that suggests a serious program error.

```
/* linux2.4.1/drivers/fc/iph5526.c:rscn_handler */
if ((login_state == NODE_LOGGED_IN) ||
    (login_state == NODE_PROCESS_LOGGED_IN)) {
    ...
}
else
if (login_state == NODE_LOGGED_OUT)
    tx_adisc(fi, ELS_ADISC, node_id,
      OX_ID_FIRST_SEQUENCE);
else
/* BUG: redundant conditional */
if (login_state == NODE_LOGGED_OUT)
    tx_logi(fi, ELS_PLOGI, node_id);
```

Figure 4.12: Redundant conditionals that signal errors: a conditional expression being placed in the `else` branch of another identical test.

```
/* linux2.4.5-ac8/drivers/scsi/qla1280.c:qla1280_putq_t */
srb_p = q->q_first;
while (srb_p )
  srb_p = srb_p->s_next;

if (srb_p) { /* BUG: this branch is never taken*/
  sp->s_prev = srb_p->s_prev;
  if (srb_p->s_prev)
    srb_p->s_prev->s_next = sp;
  else
    q->q_first = sp;
  srb_p->s_prev = sp;
  sp->s_next = srb_p;
} else {
  sp->s_prev = q->q_last;
  q->q_last->s_next = sp;
  q->q_last = sp;
}
```

Figure 4.13: A serious error in a linked list insertion implementation: `srb_p` is always NULL after the `while` loop (which appears to be checking the wrong Boolean condition).

Figure 4.10 shows a more problematic case. The second `if` statement is redundant because the first one has already taken care of the case where `slave` is false. The fact that a different error code is returned signals a possible bug in this code.

The second class of errors are again seemingly harmless on the surface, but when we give a more careful look at the surrounding code, we often find serious problems. The `while` loop in Figure 4.11 is obviously trying to recover from hardware errors encountered while reading network packets. But since the variable `err` is not updated in the loop body, it would never become `SUCCESS`, and thus the loop body never executes more than once, which is suspicious. This signals a possible error where the programmer forgets to update `err` in the large chunk of recovery code in the loop. This bug would be difficult to detect by testing, because it is in an error handling branch that is only executed when the hardware fails in a certain way.

The third class of errors are clearly serious bugs. Figure 4.12 shows an example from Linux 2.4.5-ac8. As we can see, the second and third `if` statements carry out entirely different actions on identical conditions. Apparently, the programmer has cut-and-pasted the conditional without changing one of the two `NODE_LOGGED_OUT`s into a more likely fourth possibility: `NODE_NOT_PRESENT`.

Figure 4.13 shows another serious error. The author obviously wanted to insert `sp` into a doubly-linked list that starts from `q->q_first`, but the `while` loop does nothing other than setting `srb_p` to `NULL`, which is nonsensical. The checker detects this error by inferring that the ensuing `if` statement is redundant. An apparent fix is to replace the `while` condition (`srb_p`) with (`srb_p && srb_p->next`). This bug can be dangerous and hard to detect, because it quietly discards everything that was in the original list and constructs a new one with `sp` as its sole element. In fact, the same error is still present in the stable 2.4.20 release of the Linux kernel source.

4.6 Redundant NULL-checks

The checker described in this section uses redundancies to flag misunderstandings of function interfaces in Linux. Certain functions, such as `kmalloc`, return a NULL pointer on failure. Callers of these functions need to check the validity of their return values before they can safely dereference them. Prior work [26] describes an algorithm that automatically derives the set of potential NULL-returning functions in Linux. Here we flag functions whose return

```
/* linux2.5.53/fs/ntfs/attrib.c:ntfs_merge_run_lists */
if (!slots) {
    /* FIXME/TODO: We need to have the extra
     * memory already! (AIA) */
    drl = ntfs_rl_realloc(drl, ds, ds + 1);
    if (!drl) /* BUG: drl is never NULL */
        goto critical_error;
}
```

Figure 4.14: Redundant NULL-check of `drl` signals a more serious problem: return values of `ntfs_rl_realloc` should in fact be checked with IS_ERR. A NULL-check will never catch the error case.

values should never be checked against NULL. A naive view is that at worst such redundant checks are minor performance errors. In practice, we found they can flag two dangerous situations:

1. The programmer believes that a function can fail when it cannot. If programmers misunderstand the function's interface at this basic level, they likely misunderstand other aspects.
2. The programmer correctly believes that a function can fail, but misunderstands the correct way to check for failure. Linux and other systems have functions that indicate failure in many other ways besides returning a NULL pointer.

For each pointer-returning function `f` the checker tracks two counts:

1. The number of call sites where the pointer returned by `f` was checked against NULL before use.
2. The number of call sites where the returned pointer was not checked against NULL before use.

A function `f` whose result is often checked against NULL implies the belief that `f` could potentially return NULL. Conversely, many uses with few NULL checks implies the belief that the function should not be checked against NULL. We use the z-statistic [31] to rank functions from most to least likely to return NULL based on these counts. Return values from functions with the highest z-values should probably be checked before use, whereas NULL-checks on those from the lowest ranked functions are most likely redundant.

Figure 4.14 shows one of the two bugs we found in a recent release of Linux. Here, the redundant NULL-check at line 6 signals the problem: the programmer has obviously misunderstood the interface to the function `ntfs_rl_realloc`, which, as shown below,

```
/* 2.5.53/fs/ntfs/attrib.c:ntfs_rl_realloc */
...
new_rl = ntfs_malloc_nofs(new_size);
if (unlikely(!new_rl))
    return ERR_PTR(−ENOMEM);
...
```

never returns NULL. Instead, on memory exhaustion, it returns what is essentially `((void*) -ENOMEM)`, which should be checked using the special `IS_ERR` macro. When `ntfs_rl_realloc` fails, the NULL check fails too, and the code dereferences the returned value, which likely corresponds to a valid physical address, causing a very difficult-to-diagnose memory corruption bug. Unsurprisingly, the same bug appeared again at another location in this author's code.

4.7 Predicting Hard Errors with Redundancies

In this section we show correlation between redundant errors and hard bugs that can crash a system. The redundant errors are collected from the redundant assignment checker, the dead code checker, and the redundant conditional checker.[3] The hard bugs were collected from Linux 2.4.1 with checkers described in [17]. These bugs include use of freed memory, dereferences of NULL pointers, potential deadlocks, unreleased locks, and security violations (e.g., the use of an untrusted value as an array index). These bugs have been reported to and largely confirmed by Linux developers. We show that there is a strong correlation between these two error populations using a statistical technique called the *contingency table method* [10]. Further, we show that a file containing a redundant error is roughly 45% to 100% more likely to have a hard error than one selected at random. These results indicate that (1) files with redundant errors are good audit candidates and (2) redundancy correlates with confused programmers who will probably make a series of mistakes.

[3]We exclude the idempotent operation and redundant NULL-check results because the total number of bugs is too small to be statistically significant.

4.7.1 Methodology

This subsection describes the statistical methods used to measure the association between program redundancies and hard errors. Our analysis is based on the 2×2 contingency table method [10]. It is a standard statistical tool for studying the association between two different attributes of a population. In our case, the population is the set of files we have checked in Linux, and the two attributes are: (a) whether a file contains redundancies, and (b) whether it contains hard errors.

In the contingency table approach, the population is cross-classified into four categories based on two attributes, say **A** and **B**, of the population. We obtain counts (o_{ij}) in each category, and tabulate the results as follows:

	B		
A	True	False	Totals
True	o_{11}	o_{12}	$n_{1\cdot}$
False	o_{21}	o_{22}	$n_{2\cdot}$
Totals	$n_{\cdot 1}$	$n_{\cdot 2}$	$n_{\cdot\cdot}$

The values in the margin ($n_{1\cdot}, n_{2\cdot}, n_{\cdot 1}, n_{\cdot 2}$) are row and column totals, and $n_{\cdot\cdot}$ is the grand total. The null hypothesis H_0 of this test is that **A** and **B** are mutually independent, i.e., knowing **A** gives no additional information about **B**. More precisely, if H_0 holds, we expect that:[4]

$$\frac{o_{11}}{o_{11}+o_{12}} \approx \frac{o_{21}}{o_{21}+o_{22}} \approx \frac{n_{\cdot 1}}{n_{\cdot 1}+n_{\cdot 2}}$$

We can then compute expected values (e_{ij}) for the four cells in the table as follows:

$$e_{ij} = \frac{n_{i\cdot} n_{\cdot j}}{n_{\cdot\cdot}}$$

We use a χ^2 test statistic [31]:

$$T = \sum_{i,j \in \{1,2\}} \frac{(o_{ij} - e_{ij})^2}{e_{ij}}$$

[4]To see this is true, consider 100 white balls in an urn. We first randomly draw 40 of them and put a red mark on them. We put them back in the urn. Then we randomly draw 80 of them and put a blue mark on them. Obviously, we should expect roughly 80% of the 40 balls with red marks to have blue marks, as should we expect roughly 80% of the remaining 60 balls without the red mark to have a blue mark.

to measure how far the observed values (o_{ij}) deviates from the expected values (e_{ij}). Using the T value, we can derive the the probability of observing o_{ij} if the null hypothesis H_0 is true. This probability is called the p-value.[5] The smaller the p-value, the stronger the evidence against H_0, thus the stronger the correlation between attributes **A** and **B**.

4.7.2 Data acquisition and test results

Previous work [17] uses the *xgcc* system to check 2055 files in Linux 2.4.1. It focused on serious system crashing hard bugs and collected more than 1800 serious such bugs in 551 files. The types of bugs include NULL pointer dereference, deadlocks, and missed security checks. We use these bugs to represent the class of serious hard errors, and derive correlation with program redundancies.

We cross-classify the program files in the Linux kernel into the following four categories and obtain counts in each:

1. o_{11}: number of files with both redundancies and hard errors.
2. o_{12}: number of files with redundancies but not hard errors.
3. o_{21}: number of files with hard errors but not redundancies.
4. o_{22}: number of files with neither redundancies nor hard errors.

We can then carry out the test described in section 4.7.1 for the redundant assignment checker, dead code checker, and redundant conditional checker.

The result of the tests are given in Tables 4.5 through 4.8. Note that in the aggregate case, the total number of redundancies is less than the sum of the number of redundancies from each of the four checkers. That is because we avoid double counting files that are flagged by two or more checkers. As we can see, the correlation between redundancies and hard errors are high, with p-values being approximately 0 in all four cases. The results strongly suggest that redundancies often signal confused programmers, and therefore are a good predictor for hard, serious errors.

[5]Technically, under H_0, T has a χ^2 distribution with one degree of freedom. p-value can be looked up in the cumulative distribution table of the χ^2_1 distribution. For example, if T is larger than 4, the p-value will go below 5%.

Redundant Assignments	Hard Bugs Yes	Hard Bugs No	Totals
Yes	345	435	780
No	206	1069	1275
Totals	551	1504	2055

$T = 194.37$, p-value $= 0.00$

Table 4.5: Contingency table: Redundant Assignments vs. Hard Bugs. There are 345 files with both error types, 435 files with a redundant assignments and no hard bugs, 206 files with a hard bug and no redundant assignments, and 1069 files with no bugs of either type. A T-statistic value above four gives a p-value of less than 0.05, which strongly suggests the two events are not independent. The observed T value of 194.37 gives a p-value of essentially 0, noticeably better than the standard threshold. Intuitively, the correlation between error types can be seen in that the ratio of 345/435 is considerably larger than the ratio 206/1069 — if the events were independent, we expect these two ratios to be approximately equal.

Dead Code	Hard Bugs Yes	Hard Bugs No	Totals
Yes	133	135	268
No	418	1369	1787
Totals	551	1504	2055

$T = 81.74$, p-value $= 0.00$

Table 4.6: Contingency table: Dead code vs. Hard Bugs

Redundant Conditionals	Hard Bugs Yes	Hard Bugs No	Totals
Yes	75	79	154
No	476	1425	1901
Totals	551	1504	2055

$T = 40.65$, p-value $= 0.00$

Table 4.7: Contingency table: Redundant Conditionals vs. Hard Bugs

	Hard Bugs		
Aggregate	Yes	No	Totals
Yes	372	573	945
No	179	931	1110
Totals	551	1504	2055

$T = 140.48$, p-value $= 0.00$

Table 4.8: Contingency table: Program Redundancies (Aggregate) vs. Hard Bugs

R	**R $\wedge$ E**	**R**	**P(E|R)**	**P(E|R) – P(E)**	**Standard Error**	**95% Confidence Interval for T'**
Assign	353	889	0.3971	0.1289	0.0191	48.11% ± 13.95%
Dead Code	30	56	0.5357	0.2676	0.0674	99.82% ± 49.23%
Conditionals	75	154	0.4870	0.2189	0.0414	81.65% ± 30.28%
Aggregate	372	945	0.3937	0.1255	0.0187	46.83% ± 13.65%

Table 4.9: Program files with redundancies are on average roughly 50% more likely to contain hard errors

4.7.3 Predicting hard errors

In addition to the qualitative measure of correlation, we want to know quantitatively how much more likely it is that we will find hard errors in a file that contains one or more redundant operations. More precisely, let E be the event that a given source file contains one or more hard errors, and let R be the event that it has one or more forms of redundant operations. We can compute a confidence interval for $T' = (P(E|R) - P(E))/P(E)$, which measures how much more likely we are to find hard errors in a file given the presence of redundancies.

The prior probability of hard errors is computed as follows:

$$P(E) = \frac{\text{Number of files with hard errors}}{\text{Total number of files checked}} = \frac{551}{2055} = 0.2681$$

We tabulate the conditional probabilities and T' values in Table 4.9. (Again, we exclude the idempotent operations and redundant NULL-checks because of the small sample size.) As shown in the table, given presence of any form of redundant operation, it is roughly $45\% - 100\%$ more likely we will find an error in that file than in a randomly selected file.

4.8 Detecting Specification Mistakes

This section describes how to use redundant code actions to find several types of specification errors and omissions. Often program specifications give extra information that allow code to be checked: whether return values of routines must be checked against `NULL`, which shared variables are protected by which locks, which permission checks guard which sensitive operations, etc. A vulnerability of this approach is that if a code feature is not annotated or included in the specification, it will not be checked. We can catch such omissions by flagging redundant operations. In the above cases, and in many others, at least one of the specified actions makes little sense in isolation — critical sections without shared states are pointless as are permission checks that do not guard known sensitive actions. Thus, if code does not intend to do useless operations, then such redundancies happen exactly when checkable actions have been missed. (At the very least we will have caught something pointless that should be deleted.) We sketch four examples below, and close with two case studies that use redundancies to find missing checkable actions.

Detecting omitted NULL annotations. Tools such as Splint [28] let programmers annotate functions that can return a NULL pointer with a "`null`" annotation. The tool emits an error for any unchecked use of a pointer returned from a NULL routine. In a real system, many functions can return NULL, making it easy to forget to annotate them all. We can catch such omissions using redundancies. We know only the return value of NULL functions should be checked. Thus, a check on a non-annotated function means that either the function: (1) should be annotated with NULL or (2) the function cannot return NULL and the programmer has misunderstood the interface. A variant of this technique has been applied with success in [26] and also in the checker described in Section 4.6.

Finding missed lock-variable bindings. Data race detection tools such as Warlock [72] let users explicitly bind locks to the variables they protect. The tool warns when annotated variables are accessed without their lock held. However, lock-variable bindings can easily be forgotten, causing the variable to be (silently) unchecked. We can use redundancies to catch such mistakes. Critical sections must protect *some* shared state: flagging those that do not finds either (1) useless locking (which should be deleted for better performance) or (2) places where a shared variable was not annotated.

Missed "volatile" annotations. As described in Section 4.4, in C, variables with unusual read/write semantics must be annotated with the `volatile` type qualifier to prevent the compiler from doing optimizations that are safe on normal variables, but incorrect on volatile ones, such as eliminating duplicate reads or writes. A missing `volatile` annotation is a silent error, in that the software usually works and only occasionally gives incorrect results for certain hardware-compiler combinations. Such omissions can be detected by flagging redundant operations (reads or writes) that do not make sense for non-volatile variables.

Missed permission checks. A secure system must guard sensitive operations (such as modifying a file or killing a process) with permission checks. A tool can automatically catch such mistakes given a specification of which checks protect which operations. The large number of sensitive operations makes it easy to forget a binding. As before, we can use redundancies to find such omissions: assuming programmers do not do redundant permission checks, finding a permission check that does not guard a known sensitive operation signals an incomplete specification.

4.8.1 Case study: Finding missed security holes

Ashcraft and Engler [5] describe a checker that found operating system security vulnerabilities caused when an integer read from untrusted sources (network packets, system call parameters) was passed to a trusting sink (array indices, length parameters in memory copy operations) without being checked against a safe upper and lower bound. A single violation can let a malicious attacker take control of the entire system. However, the checker is vulnerable to omissions. An omitted source means the checker will not track the data produced. An omitted sink means the checker will not warn when unsanitized data reaches it.

When implementing the checker we used the ideas in this section to detect such omissions. Given a list of known sources and sinks, the normal checking sequence is: (1) the code reads data from an unsafe source, (2) checks it, and (3) passes it to a trusting sink. Assuming programmers do not do gratuitous sanitization, a missed sink can be detected by flagging when code does steps (1) and (2), but not (3). Reading a value from a known source and sanitizing it implies the code believes the value reaches a dangerous operation. If the value does not reach a known sink, we have likely missed one in our specification. Similarly, we could (but did not) infer missed sources by doing the converse of this analysis: flagging when the OS sanitizes data we do not think is tainted and then passing it to a trusting sink.

The analysis found roughly 10 common uses of sanitized inputs in Linux 2.4.6 [5]. Nine of these uses were harmless; however one was a security hole. Unexpectedly, this was not from a specification omission. Rather, the sink was known, but our inter-procedural analysis had been overly simplistic, causing us to miss the path to it. The fact that redundancies flag errors both in the specification and in the tool itself was a nice surprise.

4.8.2 Case study: Helping static race detection

A static race detection tool, RacerX [24], has been dramatically improved by explicitly using the fact that programmers do not perform redundant operations.

At a high level, the tool is based on a static lockset algorithm similar to the dynamic version used in Eraser [68]. It works roughly as follows: (1) the user supplies a list of locking functions and a source base to check; (2) the tool compiles the source base and does a context-sensitive interprocedural analysis to compute the set of locks held at all program points; (3) RacerX warns when shared variables are used without a consistent lock held.

This simple approach needs several modifications to be practical. We describe two problems below that can be countered in part by using the ideas in this chapter.

First, it is extremely difficult to determine if an unprotected access is actually a bug. Many unprotected accesses are perfectly acceptable. For example, programmers intentionally do unprotected modifications of statistics variables for speed, or they may orchestrate reads and writes of shared variables to be non-interfering (e.g., a variable that has a single reader and writer may not need locking). An unprotected access is only an error if it allows an application-specific invariant to be violated. Thus, reasoning about such accesses requires understanding complex code invariants and actual interleavings (rather than potential ones), both of which are often undocumented. In our experience, a single race condition report can easily take tens of minutes to diagnose. Even at the end it may not be possible to determine if the report is actually an error. In contrast, other types of errors found with static analysis often take seconds to diagnose (e.g., uses of freed variables, not releasing acquired locks).

We can simplify this problem using a form of redundancy analysis. The assumption that programmers do not write redundant critical sections, implies that the first, last, and only shared data accesses in a critical section are special:

- If a variable or function call is the only statement within the critical section, we have

```
/* ERROR: linux-2.5.62/drivers/char/esp.c:
 *    2313:block_til_ready: calling <serial_out-info>
 *    without "cli()"!
 */
restore_flags(flags); /* re-enable interrupts */
set_current_state(TASK_INTERRUPTIBLE);
if (tty_hung_up_p(filp)
|| !(info->flags & ASYNC_INITIALIZED)) {
     ...
 }

/* non-disabled access to serial_out-info! */
serial_out(info, UART_ESI_CMD1, ESI_GET_UART_STAT);
if (serial_in(info, UART_ESI_STAT2) & UART_MSR_DCD)
         do_clocal = 1;
...

/* Example 1 drivers/char/esp.c:1206 */
save_flags(flags); cli();
/* set baud */
serial_out(info, UART_ESI_CMD1, ESI_SET_BAUD);
serial_out(info, UART_ESI_CMD2, quot >> 8);
serial_out(info, UART_ESI_CMD2, quot & 0xff);
restore_flags(flags);

...

/* Example 2: drivers/char/esp.c:1426 */
cli();
info->IER &= ~UART_IER_RDI;
serial_out(info, UART_ESI_CMD1, ESI_SET_SRV_MASK);
serial_out(info, UART_ESI_CMD2, info->IER);
serial_out(info, UART_ESI_CMD1, ESI_SET_RX_TIMEOUT);
serial_out(info, UART_ESI_CMD2, 0x00);
sti();
```

Figure 4.15: Error ranked high because of redundancy analysis: there were 28 places where the routine `serial_out` was used as the first or last statement in a critical section.

very strong evidence that the programmer thinks (1) the state should be protected in general and (2) that the acquired lock enforces this protection.

- Similarly, the first and last accesses of shared states in a critical section also receive special treatment (although to a lesser degree), since programmers often acquire a lock on the first shared state access that must be protected and release it immediately after the last one–i.e., they do not gratuitously make critical sections large.

A crucial result of these observations is that displaying such examples of where a variable or function was explicitly protected makes it very clear to a user of RacerX what exactly is being protected. Figure 4.15 gives a simple example from Linux. There were 37 accesses to `serial_out` with the argument `info` with some sort of lock held and in contrast there was only one unlocked use. This function-argument pair was the first statement of a critical section 11 times and the last one 17 times. Looking at the examples it is obvious that the programmer is explicitly disabling interrupts[6] before invoking this routine. In particular we do not have to look at the implementation of `serial_out` and try to reason about whether it or the device it interacts with needs to be protected with disabled interrupts. In practice we almost always look at errors that have such features over those that do not.

A second problem with static race detection is that many seemingly multi-threaded code paths are actually single-threaded. Examples include operating system interrupt handlers and initialization code that runs before other threads have been activated [68]. Warnings for accesses to shared variables on single-threaded code paths can swamp the user with false positives, at the very least hiding real errors and in the worse case causing the user to discard the tool.

This problem can also be countered using redundancy analysis. We assume that in general programmers do not do spurious locking. We can thus infer that any synchronization operation implies that the calling code is multithreaded. These operations include locking, as well as calls to library routines that provide atomic operations such as `atomic_add` or `test_and_set`. (From this perspective, synchronization calls can be viewed as carefully inserted annotations specifying that code is multithreaded.)

RacerX considers any function to be multithreaded if synchronization operations occur

[6]`cli()` clears the Interrupt Enable Flag on x86 architectures, thus it prevents preemption, and has the effect of acquiring a global kernel lock on single processor systems. `sti()` and `restore_flags(flags)` restores the Interrupt Enable Flag. They can be thought of as releasing the kernel lock previously acquired by `cli()`.

(1) anywhere within the function, or (2) anywhere above it in the call chain. Note that such operations in callees of the function may not indicate the function itself is multithreaded. For example, it could be calling library code that always conservatively acquires locks. RacerX computes the information for (1) and (2) in two passes. First, it walks over all functions, marking them as multithreaded if they do explicit concurrency operations within the function body. Second, when doing the normal lockset computation it also tracks if it has hit a known multithreaded function. If so, it adds this annotation to any error emitted.

Finally, static checkers have the invidious problem that errors in their analysis often cause silent false negatives. Redundancy analysis can help find these: critical sections that contain no shared states imply that either the programmer made a mistake or the analysis did. Eight errors were found in the RacerX implementation when the tool was modified to flag empty critical sections. There were six minor errors, one error where we mishandled arrays (and so ignored all array uses indexed by pointer variables) and a particularly nasty silent lockset caching error that caused us to miss over 20% of all code paths.

4.9 Related Work

In writing the five redundancy checkers, we make extensive use of existing research on program redundancy detection and elimination. Techniques such as partial redundancy elimination [47, 59, 60] and dead code elimination algorithms [1, 45, 48] have long been used in optimizing compilers to reduce code size and improve program performance. While our analyses closely mirror these ideas at their core, there are two key differences that distinguish our approach to that used in optimizing compilers:

1. Optimizers typically operate on a low-level intermediate representation (IR) with a limited set of primitive operations on typeless pseudo-registers. In contrast, our analyses operate at the source level and work with (or around) the full-blown semantic complexity of C. The reason is three-fold: a) many redundant operations are introduced in the translation from source constructs to intermediate representations, making it hard to distinguish ones that pre-exist in the source program, b) useful diagnostic information such as types (e.g., `unsigned` vs. `signed`) and variable or macro names (e.g., `DEBUG`) are typically discarded during the translation to IR—we find such information essential in focused suppression of false positives, and c) the translation to IR usually changes the source constructs so much that reporting sensible warning

messages at the IR level is extremely difficult, if not impossible.

2. While being sound and conservative is vital in optimizing compilers, error detection tools like ours can afford (and are often required) to be flexible for the sake of usefulness and efficiency. The redundant NULL checker in Section 4.6 harvests function interface specifications from a statistical analysis of their usage patterns. Although the analysis is effective, it is neither sound nor conservative, and therefore would not be admissible in optimizers.

3. Compilers are typically invoked far more frequently than checking tools during the development cycle. Therefore, speed is essential for the analyses being used in the optimizer. Expensive path-sensitive algorithms are carried out sparingly, if at all, on small portions of performance critical code simply because their time complexity usually outweighs their benefit. In contrast, checking tools like ours are less frequently run and can therefore afford to use more expensive analyses. The redundant conditional checker is one such example. The less stringent speed requirement does give us a substantial edge in detecting more classes of errors with more accuracy.

Redundancy analyses have also been used in existing checking tools. Fosdick and Osterweil first applied data flow *anomaly detection* techniques in the context of software reliability. In their DAVE system [63], they used a depth first search algorithm to detect a set of variable def-use type of anomalies such as uninitialized read, double definition, etc. However, according to our experiments, path insensitive analysis like theirs produces an overwhelming number of false positives, especially for the uninitialized read checker. We were unable to find experimental validations of their approach to make a comparison.

Recent releases of the GNU C compiler (version 3 and up) provides users with the "`-Wunreachable-code`" option to detect dead code in the source program. However, their analysis provides no means of controlled suppression of false positives, which we find essential in limiting the number of false warnings. Also, because of its recent inception, the dead code detection algorithm is not yet fully functional.[7]

Dynamic techniques [14, 41] instrument the program and detect anomalies that arise during execution. However, they are weaker in that they can only find errors on executed paths. Furthermore, the run-time overhead and difficulty in instrumenting low-level operating system code limits the applicability of dynamic approaches. The effectiveness of these

[7]One sample error in the GCC analysis is reported at `http://gcc.gnu.org/ml/gcc-prs/2003-03/msg01359.html`.

dynamic approaches is unclear because of the lack of experimental results.

Finally, some of the errors we found overlap with ones detected by other non-redundant checkers. For example, Splint [28] warns about fall-through `case` branches in `switch` statements even when they do not cause redundancies. It could have (but did not) issue a warning for the error shown in Figure 4.4.[8] However, many if not most of the errors we find (particularly those in Section 4.5 and Section 4.6) are not found by other tools and thus seem worth investigating. Unfortunately we cannot do a more direct comparison to Splint because it lacks experimental data and we were unable to use it to analyze the programs we check.

4.10 Conclusion

In this chapter, we explored the hypothesis that redundancies, like type errors, flag higher-level correctness mistakes. We evaluated the approach using five checkers which we applied to Linux, OpenBSD, and PostgreSQL. These simple analyses found many surprising (to us) error types. Further, they correlated well with known hard errors: redundancies seemed to flag confused or poor programmers who were prone to other errors. These indicators could be used to identify low-quality code in otherwise high-quality systems and help managers choose audit candidates in large code bases.

[8]Apparently the parser was having trouble understanding the code and did not get far enough to check the problematic part of the program.

Chapter 5

Conclusion

In this thesis we have described three novel techniques for static detection of program errors. We have described the SATURN analysis framework, which leverages recent advances in solving boolean satisfiability (SAT) constraints and serves as a general framework for building precise and scalable static analyses for multimillion-line software systems. We have described an algorithm for static detection of security vulnerabilities in scripting languages, which pose several unique challenges for static analysis due to their dynamic nature. Finally, we have presented several lightweight checkers for redundant operations in system software, which are based on the intuition that many high-level conceptual errors map to low-level redundant operations.

5.1 Future Work

Encouraged by the results from work described in this thesis, we intend to pursue the following research directions in the future.

More checkers for deeper properties. We have demonstrated in Chapter 2 that SATURN can serve as a general analysis framework for a number of program properties thanks to the expressiveness of Boolean constraints. A natural extension is to use SATURN to analyze deeper properties with higher precision. Hackett and Aiken [33] have constructed a sound pointer and shape analysis for C using SATURN. A number of other checkers (e.g., information flow, integer overflow, array bounds checker) are planned for the SATURN framework.

Other decision procedures for Saturn. The source of SATURN's precision and flexibility is its encoding of program constructs as Boolean formulas. However, there are a number of program properties that are not easily inferred and checked with SAT solvers. One example is array bounds analysis. In order to infer the constant or symbolic bounds of a variable in a program, a decision procedure that has better knowledge about linear constraints is needed [85]. We are planning to integrate decision procedures other than SAT-solvers into the SATURN framework.

Verification. Verification of program properties has been an elusive goal for static analysis. In practice, it has proven to be very difficult for all but a handful of small programs and simple properties. One obstacle to verification is the limited precision of traditional static analysis frameworks. Analyses that fail to take into account the full complexity of real world software (e.g., path sensitivity, pointers and heap) are often too imprecise to be useful in practice. SATURN provides a framework for precise and scalable static analysis, and we plan to explore whether it is a good basis for verification.

Symbolic (and interactive) exploration. Current development environments provide little direct support that help programmers reason about code modules in isolation from the rest of the program. Developers rely on methods such as unit testing which provide limited coverage and insight into how their code works in a larger codebase. Symbolic methods provide a natural way to generalize the input and dependencies from the external environment and can thus help programmers systematically examine various execution scenarios in their code, thereby improving the correctness and reliability of the software system as a whole. SATURN provides a solid basis for conducting symbolic analysis of programs and we are planning to use it for future work in this direction.

Bibliography

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, Massachusetts, 1986.

[2] A. Aiken, M. Fahndrich, and Z. Su. Detecting races in relay ladder logic programs. In *Proceedings of the 1st International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 184–200, Apr. 1998.

[3] A. Aiken, J. S. Foster, J. Kodumal, and T. Terauchi. Checking and inferring local non-aliasing. In *Proceedings of the 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 129–140. ACM Press, June 2003.

[4] A. Aiken, E. Wimmers, and T. Lakshman. Soft typing with conditional types. In *Proceedings of the 21st Annual Symposium on Principles of Programming Languages*, pages 163–173, Jan. 1994.

[5] K. Ashcraft and D. Engler. Using programmer-written compiler extensions to catch security holes. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 143–159, May 2002.

[6] T. Ball, B. Cook, V. Levin, and S. Rajamani. SLAM and Static Driver Verifier: Technology transfer of formal methods inside Microsoft. In *Proceedings of Fourth International Conference on Integrated Formal Methods*, pages 1–20. Springer, Jan. 2004.

[7] T. Ball and S. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN 2001 Workshop on Model Checking of Software*, pages 103–122, May 2001. LNCS 2057.

[8] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, Aug. 1986.

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

[10] G. Casella and R. L. Berger. *Statistical Inference.* Wadsworth Group, Pacific Grove, California, 2002.

[11] CERT advisory CA-2001-19: Code Red worm exploiting buffer overflow in IIS indexing service DLL. `http://www.cert.org/advisories/CA-2001-19.html`.

[12] CERT advisory CA-2003-04: MS-SQL server worm. `http://www.cert.org/advisories/CA-2003-04.html`.

[13] CERT advisory CA-2003-20: W32/Blaster worm. `http://www.cert.org/advisories/CA-2003-20.html`.

[14] F. T. Chan and T. Y. Chen. AIDA–a dynamic data flow anomaly detection system for Pascal programs. *Software: Practice and Experience*, 17(3):227–239, Mar. 1987.

[15] T. Chilimbi and M. Hauswirth. Low-overhead memory leak detection using adaptive statistical profiling. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 156–164, Oct. 2004.

[16] A. Chou. *Static Analysis for Bug Finding in Systems Software.* PhD thesis, Stanford University, 2003.

[17] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, pages 73–88, Oct. 2001.

[18] A. Christensen, A. Moller, and M. Schwartzbach. Precise analysis of string expressions. In *Proceedings of the 10th Static Analysis Symposium*, pages 1–18, June 2003.

[19] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In K. Jensen and A. Podelski, editors, *Tools and Algorithms for the Construction and*

Analysis of Systems (TACAS), volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, Mar. 2004.

[20] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Predicate abstraction of ANSI-C programs using SAT. *Formal Methods in Systems Design*, 25(2-3):105–127, Sept. 2004.

[21] M. Das, S. Lerner, and M. Seigle. Path-sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 57–68, Berlin, Germany, June 2002.

[22] R. DeLine and M. Fahndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 59–69, June 2001.

[23] M. Emami, R. Ghiya, and L. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, pages 242–256, June 1994.

[24] D. Engler and K. Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pages 237–252, Oct. 2003.

[25] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of Operating Systems Design and Implementation (OSDI)*, pages 1–16, Sept. 2000.

[26] D. Engler, D. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, pages 57–72. ACM Press, Oct. 2001.

[27] D. Evans. Static detection of dynamic memory errors. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, pages 44–53, May 1996.

[28] D. Evans, J. Guttag, J. Horning, and Y. Tan. LCLint: a tool for using specifications to check code. In *Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 87–96. ACM Press, Dec. 1994.

[29] C. Flanagan, M. Leino, M. Lillibridge, C. Nelson, J. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 234–245, June 2002.

[30] J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, June 2002.

[31] D. Freedman, R. Pisani, and R. Purves. *Statistics.* W W Norton & Co., New York, New York, third edition, Sept. 1997.

[32] C. Gould, Z. Su, and P. Devanbu. Static checking of dynamically generated queries in database applications. In *Proceedings of the 26th International Conference on Software Engineering*, pages 645–654, May 2004.

[33] B. Hackett and A. Aiken. How is aliasing used in systems software? Technical report, Stanford University, 2005.

[34] B. Hackett and R. Rugina. Region-based shape analysis with tracked locations. In *Proceedings of the 32nd Annual Symposium on Principles of Programming Languages*, pages 310–323, Jan. 2005.

[35] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 69–82, Berlin, Germany, June 2002.

[36] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter USENIX Conference*, pages 125–136, Dec. 1992.

[37] D. Heine and M. Lam. Static detection of leaks in polymorphic containers. In *Proceedings of the 28th International Conference on Software Engineering*, May 2006.

[38] D. L. Heine and M. S. Lam. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 168–181, June 2003.

[39] T. A. Henzinger, R. Jhala, and R. Majumdar. Lazy abstraction. In *Proceedings of the 29th Annual Symposium on Principles of Programming Languages*, pages 58–70, Jan. 2002.

[40] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with Blast. In *Proceedings of the SPIN 2003 Workshop on Model Checking Software*, pages 235–239, May 2003. LNCS 2648.

[41] J. C. Huang. Detection of data flow anomaly through program instrumentation. *IEEE Transactions on Software Engineering*, 5(3):226–236, May 1979.

[42] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D. Lee, and S.-Y. Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th International World Wide Web Conference*, pages 40–52, May 2004.

[43] W. S. Humphrey. The quality attitude. news@sei 2004, Number 3, `http://www.sei.cmu.edu/news-at-sei/columns/watts_new/2004/3/watts-new-2004-3.htm`.

[44] D. Jackson and M. Vaziri. Finding bugs with a constraint solver. In *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 14–25, Aug. 2000.

[45] K. Kennedy. *In S. Muchnick and N. Jones, editors, Program Flow Analysis: Theory and Applications*, chapter A Survey of Data Flow Analysis Techniques, pages 5–54. Prentice-Hall, 1981.

[46] S. Khurshid, C. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 553–568. Springer, Apr. 2003.

[47] J. Knoop, O. Rüthing, and B. Steffen. Lazy code motion. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, pages 224–234, June 1992.

[48] J. Knoop, O. Rüthing, and B. Steffen. Partial dead code elimination. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, pages 147–158, June 1994.

[49] D. Kroening, E. Clarke, and K. Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *Proceedings of the 40th Design Automation Conference*, pages 368–371. ACM Press, June 2003.

[50] W. Landi and B. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, pages 235–248, July 1992.

[51] J. R. Larus, T. Ball, M. Das, R. DeLine, M. Fahndrich, J. Pincus, S. K. Rajamani, and R. Venkatapathy. Righting software. *IEEE Software*, 21(3):92–100, May/June 2004.

[52] X. Leroy, D. Doligez, J. Garrigue, and J. Vouillon. The Objective Caml system. Software and documentation available on the web, `http://caml.inria.fr`.

[53] D. Liang and M. Harrold. Efficient computation of parameterized pointer information for interprocedural analysis. In *Proceedings of the 8th Static Analysis Symposium*, pages 279–298, July 2001.

[54] V. Livshits and M. Lam. Tracking pointers with path and context sensitivity for bug detection in C programs. In *Proceedings of the 4th joint European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundation of Software Engineering*, pages 317–326, Sept. 2003.

[55] V. Livshits and M. Lam. Finding security vulnerabilities in Java applications with static analysis. In *Proceedings of the 14th Usenix Security Symposium*, pages 271–286, Aug. 2005.

[56] M. Lucovsky. Microsoft Corp. internal memo, 2000.

[57] M. Martin, B. Livshits, and M. Lam. Finding application errors and security flaws using PQL: a program query language. In *Proceedings of the 20th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 365–383, Oct. 2005.

[58] Y. Minamide. Static approximation of dynamically generated web pages. In *Proceedings of the 14th International World Wide Web Conference*, pages 432–441, May 2005.

[59] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Commun. ACM*, 22(2):96–103, Feb. 1979.

[60] E. Morel and C. Renvoise. *In S. Muchnick and N. Jones, editors, Program Flow Analysis: Theory and Applications*, chapter Interprocedural Elimination of Partial Redundancies, pages 160–188. Prentice-Hall, 1981.

[61] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 39th Conference on Design Automation Conference*, pages 530–535, June 2001.

[62] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically hardening web applications using precise tainting. In *Proceedings of the 20th International Information Security Conference*, pages 1–12, May 2005.

[63] L. J. Osterweil and L. D. Fosdick. DAVE—a validation error detection and documentation system for Fortran programs. *Software: Practice and Experience*, 6(4):473–486, Dec. 1976.

[64] Perl documentation: Perlsec. `http://search.cpan.org/dist/perl/pod/perlsec.pod`.

[65] PHP: Hypertext Preprocessor. `http://www.php.net`.

[66] PHP usage statistics. `http://www.php.net/usage.php`.

[67] E. Ruf. Effective synchronization removal for Java. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 208–218, June 2000.

[68] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A dynamic data race detector for multithreaded programming. *ACM Trans. Comput. Syst.*, 15(4):391–411, Nov. 1997.

[69] D. Scott and R. Sharp. Abstracting application-level web security. In *Proceedings of the 11th International World Wide Web Conference*, pages 396–407, May 2002.

[70] Security space Apache module survey (Oct 2005). `http://www.securityspace.com/s_survey/data/man.200510/apachemods.html`.

[71] J. Seward. Valgrind. `http://developer.kde.org/sewardj/`.

[72] N. Sterling. WARLOCK - a static data race analysis tool. In *USENIX Winter Technical Conference*, pages 97–106, Jan. 1993.

[73] Symantec Internet security threat report: Vol. VII. Technical report, Symantec Inc., Mar. 2005.

[74] G. Tassey. The economic impacts of inadequate infrastructure for software testing. Technical report, National Institute for Standards and Technology, May 2002.

[75] TIOBE programming community index for November 2005. `http://www.tiobe.com/tpci.htm`.

[76] D. Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of the 2000 Network and Distributed Systems Security Conference*, pages 3–17, Feb. 2000.

[77] J. Whaley and M. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 131–144, June 2004.

[78] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 187–206, Nov. 1999.

[79] R. Wilson and M. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, pages 1–12, June 1995.

[80] A. Wright and R. Cartwright. A practical soft type system for Scheme. *ACM Trans. Prog. Lang. Syst.*, 19(1):87–152, Jan. 1997.

[81] Y. Xie and A. Aiken. Context- and path-sensitive memory leak detection. In *Proceedings of the 5th joint European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundation of Software Engineering*, pages 115–125, Sept. 2005.

[82] Y. Xie and A. Aiken. Scalable error detection using boolean satisfiability. In *Proceedings of the 32nd Annual Symposium on Principles of Programming Languages*, pages 351–363, Jan. 2005.

[83] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *Proceedings of the 15th Usenix Security Symposium*, 2006.

[84] Y. Xie and A. Chou. Path sensitive analysis using boolean satisfiability. Technical report, Stanford University, Nov. 2002.

[85] Y. Xie, A. Chou, and D. Engler. ARCHER—an automated tool for detecting buffer access errors. In *Proceedings of the 4th joint European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundation of Software Engineering*, pages 327–336, Sept. 2003.

[86] Y. Xie and D. Engler. Using redundancies to find errors. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 51–60, Nov. 2002.

[87] Y. Xie and D. Engler. Using redundancies to find errors. *IEEE Transactions on Software Engineering*, 29(10):915–928, Oct. 2003.

[88] J. Yang, T. Kremenek, Y. Xie, and D. Engler. MECA: an extensible, expressive system and language for statically checking security properties. In *Proceedings of the 10th Conference on Computer and Communications Security*, pages 321–334, Oct. 2003.

[89] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings of International Conference on Computer-Aided Design*, pages 279–285, San Jose, CA, Nov. 2001.