On the Role of Static Analysis in Operating System Checking and Runtime Verification

A RESEARCH PROFICIENCY EXAM PRESENTED
BY

ABHISHEK RAI
STONY BROOK UNIVERSITY

Technical Report FSL-05-01
May 2005

Abstract of the RPE

On the Role of Static Analysis in Operating System Checking and Runtime Verification

by

Abhishek Rai
Stony Brook University

2005

Software inevitably contains bugs. For certain classes of software like operating systems, reliability is a critical requirement. Recent research has shown that several commodity operating systems, even after careful design and extensive testing, still contain a number of bugs. Methods for automated detection of bugs in software can be classified into (1) static methods, (2) formal verification, and (3) runtime checking. In this paper, our focus is on verifying critical properties of large and complex software like OSs, one such property being correctness of memory accesses. For this purpose, we evaluate both static checking techniques and runtime checking techniques, and present our observations.

Static checking is useful because the cost of repairing a bug increases along the software development lifetime. Bugs discovered and fixed early result in significant cost savings. We evaluate a representative set of existing static checking techniques on the basis of soundness, precision, and usefulness in checking large software like the Linux kernel. We also cover the related problem of deriving property rules for automated checking.

Static checking though useful, cannot in general catch all possible bugs in C programs, without producing false alarms. Consequently, it is mostly a best-effort exercise to find some, but not all bugs. However, memory access checking is too critical to be trusted to static techniques. We introduce a survey of existing bounds checking techniques. In particular, we show that it is possible to classify the wide range of existing bounds checking techniques into a single hierarchy. We evaluated existing techniques based on soundness, performance, and usability. A software bounds checking framework for the Linux kernel is introduced and evaluated.

We observe that existing static checking methods have limited coverage in detecting bugs. For verifying the correctness of critical program properties like memory accesses, runtime checking is necessary. However, runtime methods by themselves are not very useful due to performance reasons. Hence, we conclude advocating static analysis for use in runtime checking systems.

To my family

Contents

List of Figures viii

List of Tables viii

Acknowledgments x

1 Introduction 1

2 Overview 3
- 2.1 Why Is C Not Enough . . . 3
  - 2.1.1 Types of Bugs . . . 4
    - Memory Errors . . . 5
    - Inconsistent Interfaces . . . 5
    - Violation of Temporal Properties . . . 5
    - Bugs due to Software Development Processes . . . 5
  - 2.1.2 The Nature of Software Bugs . . . 5
- 2.2 Bug Detection Techniques . . . 6
  - 2.2.1 Formal Verification . . . 6
  - 2.2.2 Runtime Checking . . . 7
  - 2.2.3 Static Checking . . . 7
  - 2.2.4 Issues and Trade-Offs in Static Checking . . . 8

3 Static Analysis Survey 11
- 3.1 Annotation-Based Techniques . . . 11
  - 3.1.1 Lint . . . 11
  - 3.1.2 LCLint . . . 12
    - LCL and Larch . . . 12
    - LCLint . . . 12
  - 3.1.3 Splint . . . 12
    - User-defined Annotations . . . 12
  - 3.1.4 Extended Static Checking . . . 13
    - Unsoundness and Incompleteness in ESC/Java . . . 14
  - 3.1.5 Cqual . . . 16
    - Usability . . . 17
- 3.2 Tool-Based Techniques . . . 17
  - 3.2.1 PREfix . . . 17
  - 3.2.2 ESP . . . 18
  - 3.2.3 Abstract Interpretation: ASTREE . . . 19

3.2.4 SLAM . . . 20
3.2.5 Metacompilation . . . 21
Implementation . . . 22
False Positive Suppression . . . 22
Conclusion . . . 23
3.3 Language-Based Approaches . . . 23
3.3.1 Cyclone . . . 24
From C to Cyclone . . . 24
Discussion . . . 25
Scalability . . . 26
Character strings in Cyclone . . . 26
3.3.2 Vault . . . 26
Extending type system with system-specific rules . . . 26
Memory Management . . . 26
Casting and converting between types . . . 26
3.3.3 Comparison of Cyclone and Vault . . . 27
Annotation Effort . . . 27
Code Migration Efforts . . . 27
Performance . . . 27
3.3.4 MOPS . . . 27

4 Automated Deduction of Checking Rules **29**
4.1 The Need for Automated Deduction . . . 29
4.2 Invariant Inference using Predicate Abstraction . . . 30
4.2.1 Inferring Loop Invariants . . . 30
4.2.2 Predicate Abstraction . . . 30
SLAM . . . 31
4.3 Invariant Inference for Annotation Based Checkers: Houdini . . . 31
4.4 Runtime Methods for Inferring Invariants . . . 32
4.5 Bugs as Deviant Behavior: Combining Static Analysis and Invariant Inference . . . 33
4.6 Extracting Finite State Models from Code . . . 34
4.6.1 Metal Based System for Flash . . . 34

5 Case Study: Memory Bounds Checking **36**
5.1 The Nature of Memory Bugs . . . 36
5.2 Static Methods for Detecting Memory Bugs . . . 37
5.2.1 Archer . . . 37
Loop Handling . . . 38
Error Triggers . . . 38
Symbolic Execution of Statements . . . 39
Size Inference using Statistical Belief Analysis . . . 39
5.3 Runtime Checking for Detecting Memory Bugs . . . 39
BCC . . . 41
5.3.1 Purify . . . 43
5.3.2 Safe-C . . . 43
5.3.3 Decoupling Computation and Bounds Checking . . . 44
5.3.4 Backward-Compatible Extensions to Safe-C . . . 45
5.3.5 Cash: Hardware Assisted Bounds Checking . . . 45

5.4 Bounds Checking for the Linux kernel . . . 47
5.4.1 KBCC . . . 47
Soundness . . . 47
Incompleteness . . . 48
BCC Bugs: An Example . . . 48
Advanced C Constructs . . . 49
Simple C Constructs that BCC cannot handle . . . 50
Out-of-bounds pointers . . . 51
Removing redundant pointer checks. . . . 51
Summary of Our Contributions . . . 52
5.4.2 Mudflap . . . 52
5.4.3 Kernel Code Electric Fence . . . 52
5.5 Performance Evaluation . . . 53
5.5.1 Kefence . . . 53
5.5.2 KBCC . . . 54

6 Static Analysis as a Means for Facilitating Runtime Checking **56**
6.1 Static Analysis for Bounds Checking . . . 56
6.1.1 CCured . . . 57
6.1.2 Static Analysis in BCC . . . 58
6.2 Static Analysis for Instrumentation of Code . . . 58
6.2.1 Aspect-Oriented Programming . . . 58
6.2.2 Source Code Instrumentation in Aristotle . . . 59

7 Conclusions and Future Work **61**
7.1 Static Checking . . . 61
7.2 Runtime Bounds Checking . . . 62
7.3 Future Work . . . 63
Bounds Checking . . . 63
Static Analysis for Runtime Checking . . . 63

List of Figures

2.1 Example of an Application Level Protocol that should be satisfied in program code 4

3.1 Static checkers plotted along the two dimensions coverage and effort 13

5.1 Bounds checking overhead for ReiserFS with Am-utils and Postmark 54

6.1 Architectural overview of the Aristotle system 59

List of Tables

3.1 Table showing how Cyclone handles various types of errors 25
3.2 Feature comparison of static checkers 28

5.1 A two-level classification for runtime bounds-checking technique 40
5.2 Table showing the differences between capability-based and availability-based runtime bounds checking 41

Acknowledgments

I would like to thank my advisor, Professor Erez Zadok, for his constant support for this project. The vision of this project shaped during several insightful discussions with him. Besides, he taught me several interesting and useful research practices along the way. Sean Callanan, Radu Grosu, Scotta Smolka, Scott Stoller, and Annie Liu provided an excellent forum for discussion and development of ideas related to this project.

My committee members, Tzi-cker Chiueh and Scott Stoller, were generous of their time and provided helpful input and comments on the work. Amit Sasturkar and Rahul Agarwal provided valuable feedback on some ideas in this paper.

I thoroughly enjoyed the company of members of File Systems and Storage Laboratory, Sean Callanan, Gopalan Sivathanu, Charles Wright, Avishay Traeger, and Nikolai Joukov during the course of this project. Kapil Kumar, Mahadev Konar, Susanta Nanda, Sumit Jain, and Mohit Gupta have made my stay in Stony Brook entertaining and pleasant. And, friends from A1 are just as cherished today, on a daily basis, as they were years before.

Finally, I would like to thank my family, papa, mummy, montu, and didi for everything.

This work was partially made possible thanks to NSF CAREER award EIA-0133589, NSF award CCR-0310493, and HP/Intel gifts numbers 87128 and 88415.1.

Chapter 1

Introduction

Operating system reliability has become more important than ever. Users trust their computers with important data, and OS failures can have catastrophic effects, like data loss, resource unavailability, and at the very least, frustrated users. Moreover, commodity operating systems are increasingly being used in enterprise scenarios to create large scale computing and storage clusters. But commodity operating systems are still unreliable. Recent techniques for automated debugging of some commodity operating systems have discovered numerous bugs [47].

Manual checking of code is not enough. For one, it is both error-prone and time consuming, and secondly, humans easily get overwhelmed by complexity. This motivates the need for automation of the checking process. Moreover, the wide variety of bugs found in these systems call for innovative techniques for automated bug detection. Existing techniques can be classified into static methods, model checking, and runtime checking. In this report we study applications of static techniques for automated debugging and analysis of software. In particular, we consider two different ways to apply static analysis to software. First, we present a survey of static checking techniques for automated debugging of OS kernels. Second, we study the use of static analysis and instrumentation for runtime checking of OS kernels.

Static methods have significant benefits over other approaches to automated analysis of software. In comparison to runtime checking, static analysis has three main advantages. First, static analysis does not need to actually execute the software. Thus bugs can be discovered and fixed early during the development process. Second, software that is statically checked does not suffer from the performance overhead of runtime checking. Third, static analysis covers all code uniformly; unlike runtime checking techniques, bugs on rarely executed code paths also get discovered. Although model checking techniques share some of these merits over runtime checking, they are rendered effectively intractable for large software systems due to scalability problems and the lack of a formal specification or *model* of the system.

We believe that static analysis of code has great potential in helping improve software quality. Static checking of code is just one of its possible applications. Additionally, static analysis of code can be combined with other techniques like model checking and runtime checking to produce even more powerful checking systems. For example, one can first check software statically, and then leave the rest for runtime verification. Similarly, one can combine static analysis with model checking to automatically discover predicates using model checking heuristics and then check them using static analysis [16].

We demonstrate the amalgamation of static analysis and runtime checking through a memory bounds checking system we have developed for the Linux kernel. Memory bound overrun bugs often have serious consequences. They can result in security breaches, program misbehavior and worse still, they can silently damage the system. It is all the more acute for OS kernels where memory bugs can lead to security breaches and permanent data loss.

We have ported two open source memory bounds checking systems into the Linux kernel, and with the help of static techniques, have significantly cut down their performance overheads. Additionally, we have made significant modifications to reduce the number of false positives, and support concurrency.

The rest of this paper is organized as follows. In Section 2 we present a classification for bug detection techniques and synthesize the basic ingredients for static checking. In Section 3 we present a survey of a static checking techniques. Section 4 motivates the need for automated deduction of checking rules and presents a quick survey. Section 6 generalizes static checking and advocates using it as a means for enhancing the effectiveness of runtime checking. We present some of our own work in this section. Section 5 focuses on memory related bugs in software. We present a survey of existing techniques and present our own work in this field. We finally conclude in Section 7 and identify future avenues of research in this field.

Chapter 2

Overview

In this section we derive the basic thesis of automated bug finding in software. We first examine the nature of bugs in systems software. Based on these bugs, we then present the design space for automatic static checking techniques with particular emphasis on techniques for checking large complex software written in C, like OS kernels. Finally, we compare static checking with other methods of detecting bugs in software, like formal verification and runtime checking.

2.1 Why Is C Not Enough

Since its very inception, C has been the de-facto language for systems programming. In fact, systems programming and C have evolved side by side ever since C's introduction in the 1970s. Like other trends in computing industry, this was not necessarily designed to be that way, but this is how things turned out to be. So far, C has suited well for the requirements of a systems programming language. In general, any systems programming language is required to provide the following features.

1. Control over low-level data representation

2. Explicit memory management

3. Performance, and "transparency" of performance: performance characteristics of a C program should directly manifest in the source code (e.g., language features like automatic garbage collection make runtime overhead of a program unpredictable).

C satisfies all these requirements to the extent mandated by traditional systems. However, as the scale and complexity of systems increases, systems design as it is known today is getting increasingly overwhelmed by complexity and bugs. This can be attributed to two basic deficiencies in the design of the C language.

First, C is too permissive. As a result, compilers cannot catch all possible bugs. For example, C tries to give programmers much control over memory layout of data structures; and, it allows them to control the lifetime of program resources (like memory). This is partly by design: C does not enforce strong semantics at the language level (type system). Also, at the time C was conceived, program analysis was not advanced enough to develop a strongly typed but flexible language. In any case, the design of C served well in accommodating a wide range of programming styles and implementation requirements. But, it also opens up the possibility of programming bugs. The default type checking done by C compilers is weak and misses many bugs. It is ultimately left to the developer to exercise their prudence to produce "good" error-free code.

The second serious drawback of C is its lack of extensibility. There is a large gap between the semantics of a typical systems program and the simple rules enforced that the C compiler tries to enforce. As a result, the compiler is bound to miss bugs. They may not be “language” bugs, but they are bugs nevertheless. Consider the following example. Synchronization primitives are a common ingredient of systems programs. In the context of the Linux kernel, it is required that the `spin_lock` function is called, only when the lock is not already held. Similarly, the `spin_unlock` function should be called while the lock is held. This property can be conveniently depicted using a state diagram as shown in Figure 2.1.

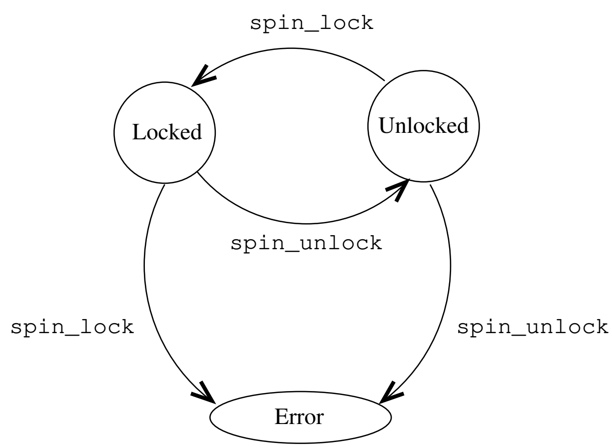

Figure 2.1: Example of an Application Level Protocol that should be satisfied in program code. During normal operation, the system can be in one of two states: `Locked` *or* `Unlocked`*. Calls to* `spin_lock` *and* `spin_unlock` *cause transitions between these states. Any other transition results in an error.*

However, the C language as such, provides no support for incorporating application level semantics into compile-time checking. The developer has to validate higher level properties of their code using alternative methods.

Code inspection is still the most widely exercised code checking technique (after compilation). But humans can easily get overwhelmed by complexity beyond a few hundred lines of code. As a result, manual checking is error-prone and time consuming. Moreover, manual checking has poor incremental returns: as different components of the system evolve and rules and invariants change, it is quite an art trying to maintain such code. Local changes applied to only a small part of the program often end up breaking other, seemingly unrelated parts of the code. Despite all these challenges, C has no built-in support for language-extensions for checking program conformance to higher level properties.

2.1.1 Types of Bugs

It is not possible to give a complete listing, or even a useful classification of all possible bugs. In this section we discuss some common bugs and their causes. This discussion will help us classify bug detection techniques in the next section.

Let us consider a simple bug. The following piece of C code tries to copy the string “hello world” into the array `arr`.

```
fn()
{
  char arr[4];
  strcpy(arr, "hello world");
}
```

One can make two observations regarding this code. First, the allocated space for `arr` is too small to accommodate the entire string in it. Secondly, `strcpy` does not detect and handle this situation. As is obvious, this operation will cause data to be written beyond the limits of the array `arr`. The result could be a program crash, or misbehavior due to memory corruption, or, no visible effect at all. One can take two points from this example:

1. simple static checking can detect some potentially dangerous bugs.

2. knowledge of procedure interfaces would be useful, e.g., knowing that `strcpy` will not check the operation for bounds overflow.

We now present some common types of bugs.

Memory Errors These are probably the most common bugs in C programs. Memory access bugs, especially memory buffer overruns have caused much grief to IT industry. We classify existing techniques for detecting memory bugs in Section 5. We also consider in detail the case of runtime bounds checking.

Inconsistent Interfaces In large complex systems, which have several smaller components making up the whole system, it is important that individual components have strictly enforced interfaces. When this is not the case, the software can behave unpredictably. This problem can be related to the issue of automatically enforcing higher-level system specifications. Traditionally, software written in C has not had access to checkable specifications. For example, Jass or JML for Java.

Violation of Temporal Properties Imperative programs are essentially temporal artifacts. There is often a prescribed temporal ordering of operations. For example, lock before unlocking, etc. Violations of temporal properties can result in synchronization errors (deadlocks, race conditions), memory leaks, use after free (dereferencing of pointers which have already been freed), etc.

Bugs due to Software Development Processes Software development processes introduce several bugs. One of the nastier ones is *regression bugs*. As code evolves over time, it is more difficult to maintain. First, developers leave and with them goes ad-hoc undocumented knowledge about the code. Second, as new code is added, components may get out of sync with each other, or existing code may be broken. This is very much the case when software specifications are not tightly coupled with the code, and the developer is responsible for ensuring compatibility. For example, checkable interface specifications, or annotation-based systems can help prevent such regressive digressions (Section 3).

This paper does not address problems arising from incorrect synchronization in concurrent programs, including problems like data races, atomicity violations, and deadlocks. These problems have been extensively studied in literature, and static, dynamic, as well as hybrid techniques addressing these problems have been developed [23, 30, 51, 52, 56, 112].

2.1.2 The Nature of Software Bugs

In recent times, a lot of research effort has been focused at detecting and studying bugs in operating system kernels. This is no surprise considering the possible implications of buggy operating systems. The two leading operating systems of today, Linux [47, 137] and Microsoft Windows [16] have received much scrutiny from researchers developing automated static checking systems. Also, these methods have been successful in finding numerous bugs.

Chou et al. studied of bugs in the Linux kernel [32]. The bugs they report were found using their static checking system which we discuss in Section 3.2.5. In particular, they made the following notable observations about bugs in the Linux kernel which justify some of the claims we have made above:

- Bug distribution is much higher in device drivers than the core kernel. This is because the core kernel undergoes much greater scrutiny and runtime testing than device drivers. However, a buggy device driver is just as likely to crash a kernel.

- Long and complex functions have a higher number of bugs per line of code. Thus, programming in C does not handle complexity very well.

- Bug distribution between files is logarithmic: few files have most bugs while most files have none or few bugs. This is probably due to the fact that thousands of developers are involved in kernel development, and they have very different backgrounds—they all not be trusted with producing completely bug-free code.

- Some bugs in the Linux kernel had lived for a couple of years before they were detected. This serves to demonstrate that traditional methods do not achieve good coverage.

Apart from the issue of programming errors, another grossly mishandled case is the absence of a unified mechanism for graceful recovery from errors (exceptions). Since error detection is often postponed to runtime (instead of being pointed out at compile time), the system is usually not well equipped to handle every possible error case. In the worst case, this could cause the whole system to crash, leading to loss of data or service. The cases in which the system misbehaves without explicitly crashing are equally disagreeable.

The approaches we discuss in this section try to address some or all of these issues in different ways. Existing methods for detecting or preventing memory bugs can be classified into language based approaches, static analysis, and runtime checking techniques

2.2 Bug Detection Techniques

Bug detection is a very broad field. The central thesis of this paper is that static detection of bugs can be useful; however, they cannot completely verify large complex systems. For such systems, static checking is best combined with runtime checking for catching all bugs. Static checking is useful because the cost of repairing a bug increases along the software development time-line. An early discovery can result in a substantial saving. This paper demonstrates that static analysis can be very useful in these regards. Of course, it would be a better idea to avoid bugs altogether (better programming practices); however, in the absence of completely sound bug avoidance, having redundant checking is better than producing buggy programs.

Apart from "discovering" bugs, it would be very useful if the bug is detected as close to its *origin* in the code as possible. This makes for easy fixing of the bug. For instance, run-time memory checking facilities can also detect `NULL` pointer dereferences. However, they usually do not provide detailed history of events leading to the `NULL` pointer, a good static checker can in many cases backtrack in its analysis state to provide more useful information.

We now compare static checking with alternative bug detection strategies for the case of checking OS kernels.

2.2.1 Formal Verification

Twenty years ago, Rushby outlined the difficulties in verifying secure kernels [108] and suggested a physical separation of OS components. Neumann et al. built comprehensive mechanisms for proving the security of the PSOS operating system [96], a hierarchically-structured OS written in a strongly-typed language. Later works focused on compartmentalization, boundary protection, and some verification of OSs such as Mach [9, 62], Pebble [59], EROS [117], DEOS [101], and Fiasco [68, 124]. In the Kit project, Bevier tried to prove the correctness of an OS using theoretical techniques [21]. As security concerns grew with the Internet, the

U.S. Department of Defense released *The Orange Book* [41], which serves as a "bible" for evaluating secure systems. Arbaugh showed the life-cycle of system vulnerabilities, emphasizing the importance of early discovery, disclosure and fixing of bugs, and then releasing patches [12]. They also showed how important it was for the security and integrity of computer systems, to be able to verify each component and layer independently [11].

Model checking of OSs has been used to detect specific problems within OS components. Pike et al. used the SPIN [20] verification system to detect a certain subtle race condition in Plan 9 [103]. More recent efforts seek to extract models from OS source code [14, 17, 48, 87], with an emphasis on checking data-independent properties of device drivers and network protocols, and to apply model checking directly to software [36, 61, 80, 93, 121].

However, model checking techniques do not trivially extend to operating systems that were not designed with verifiability as a goal (e.g., Linux). Model checking such a huge system is impractical for two reasons: (1) lack of a complete model or specification for the system, (2) state space explosion. Nevertheless, several static checking techniques use ideas from the model checking world [16, 36, 42].

2.2.2 Runtime Checking

Most runtime external checking of systems is incidental. For example, the Bonnie tool is a program that exercises the memory (VM) and cache subsystems of the OS by creating very large files and then randomly reading and writing parts of them [24, 35]. Similarly, file-system developers test a new or modified file system by compiling a large package (such as the kernel source itself) inside a mounted instance of the new file system [100, 134–136]. The POSIX standard specifies a common set of system calls for OSs [70] and also offers a standard description of how to test code for POSIX compliance [69].

In the fields of security and intrusion detection, wrappers are used to guard software interfaces by monitoring inputs, outputs, and sequences of operations [18, 19, 85, 114, 115, 126, 133]. Chiueh showed how to protect loadable kernel modules from each other on x86-based OSs such as Linux [29]. We also propose to wrap system calls and monitor them for compliance, but our wrappers are generated from formal specifications of the system-call interfaces. Wrapper mechanisms with an overhead of no more than 1–2% are possible [18, 19, 134–136]. Our monitoring is intended not only for security, but for any deviant behavior.

Runtime checking in tools such as the Bounds-Checking Gnu C (BCC) [25], Purify [106] and Insure++ [79], perform runtime monitoring for memory corruptions; e.g., running off of the end of a string in C. We discuss this in detail in Section 5. Run-time monitoring can also detect potential race conditions [113] and atomicity violations [52, 83]. Also, some methods have combined runtime checking with static checking to obtain robust property checking. Specifically, the static checking properties validates all that can be verified statically, the remaining checks are left to be checked at runtime.

2.2.3 Static Checking

The central tenet of static checking is that many abstract program restrictions map clearly to source code constructs. Static checking is based on the science of *program analysis*. Program analysis offers static techniques for predicting safe and computable approximations to the set of values or behaviors arising dynamically at run-time when executing a program on a computer [97]. However, program analysis—including finding possible run-time errors—is undecidable: there is no mechanical method that can always answer truthfully whether programs may not exhibit runtime errors. When faced with this mathematical impossibility, the alternative has been to apply certain techniques that simplify the overall program analysis problem into more tractable problems. There are four basic approaches to program analysis:

1. **Data Flow Analysis** treats programs as graphs: the nodes are the elementary blocks and the edges describe how control might pass from one elementary block to another. For example, for handling the

problem of *reaching definitions*, the standard approach is to coin mathematical abstractions for the role of edges in the graph and solve them using a set of equations. Symbolic execution as used in PREfix and SLAM (Section 3.2) are types of data flow analysis.

2. **Constraint-Based Analysis** verifies global properties by solving a set of local constraints generated from the program text. Constraint generation from program text has the effect of separating specification from implementation. Constraints hide implementation details and pave the way for tractable, structural proofs. There are two basic constraint theories: data flow analysis and type-inference. In Section 3.1.5 we present Cqual which is a constraint-based type inference system.

3. **Abstract Interpretation** (AI) maps programs into more abstract domains [8, 74]. This makes analysis more tractable and potentially useful for checking. The theory of AI differs from other approaches, it is a general methodology for calculating analyses rather than specifying them and then relying on a posteriori validation. For AI to be effective, it imposes a constraint on the nature of the abstraction and the *concretization* functions. This is called the *galois connection* and serves to produce a safe over-approximation analysis. ASTREE (discussed in Section 3.2.3) is an example of an AI-based static checker. In Sections 3.2.4 and 4.2, we use predicate abstraction [110], which is a special form of abstract interpretation in which the abstract domain is constructed using a given set of predicates.

4. **Type and Effect Analysis** is an amalgamation of two ingredients: an *effect system* and an *annotated type system*. The effect system tells the effect a statement's execution has. The annotated type system provides the semantics for the effect system. Type analysis is the most commonly used technique enforced by compilers for static checking (e.g., the `gcc -Wall` option). Type inference (Section 3.1.5) is a useful approach. Most of the checkers discussed in Section 3 use some form of strong type enforcement.

Despite the apparent differences, these approaches to program analysis are equally powerful. There is some commonality among these approaches. The art of program analysis entails cultivating the ability to choose the right approach for the right task and in exploiting insights developed in one approach to enhance the power of the other. Moreover, one can argue that it is not possible to give a single clear-cut classification of static checking techniques. This is because (1) there is a significant overlap in different techniques, (2) seemingly radical approaches are often equivalent in terms of program analysis power, and (3) designing good static checkers is increasingly an *engineering* problem—innovative implementations often beat more ambitious but less useful ones. Nonetheless, for the purpose of discussion, it helps to divide our subjects into a few classes. In Section 3 we classify static checking techniques into annotation-based methods, and tool-based methods. Although this may not be the right choice, it certainly has the advantage of better capturing the programmer's imagination.

2.2.4 Issues and Trade-Offs in Static Checking

Before we discuss existing static analysis techniques, it would be useful to get a taste of common *engineering* issues and trade-offs in designing static checkers. We qualify these as "engineering" because they require a deep knowledge of available information and a creative intuition for effective handling. The choices made in handling these issues often have little effect on the "power" of the static analysis as such, but they often are the deciding factor in the acceptance of the system.

We present below a listing of the issues and trade-offs we identified. This is not an exhaustive list. Also, they are mostly orthogonal; in most cases one can realign one trade-off without disrupting others too much.

1. **Soundness and Completeness**

   It is well known that program analysis—including finding possible run-time errors—is undecidable. When faced with this mathematical impossibility, the alternative has been to apply certain techniques that simplify the overall program analysis problem into more tractable problems. This approximation invariably results in loss of soundness or precision. Soundness in the context of static checking is defined as catching *all* the bugs in the code. Completeness reflects that all bugs caught by the checker should be *true* bugs in the system and not "false positives." We also use the term precision to describe completeness in this paper. So, if a sound analysis is desired, precision has to be sacrificed, and vice versa. In other words, there is always a soundness-precision trade-off. For example, the PREfix system (Section 3.2.1) sacrifices soundness for precision. Imprecise checkers usually generate a lot of false error notices and separating them from real errors is a time-consuming process. PREfix takes the approach of analyzing only *some* program paths, but it analyzes them well, yielding precise results. This precision comes at the cost of soundness. The ASTREE checker (Section 3.2.3) which is based on abstract interpretation allows the soundness-completeness trade-off to be adjusted by simply controlling a parameter. This is an interesting and powerful use of abstract interpretation.

2. **Annotations vs. External Checking**

   Someone designing a static checker is invariably faced with this issue regarding what interface to provide to the programmers: code annotations or external checking. Whereas annotations are applied as a layer on top of the program code, external checking can directly apply to unmodified program code. Annotations provide a fine-grained specification intermixed with program code. Thus it is tightly bound to the code and can even double up as a protocol specifications. Consequently, the likelihood of annotations getting out of sync with the code (e.g., due to regression) is small. Also, we believe that due to tight coupling between program code and annotations, annotation-based systems are more capable of attacking soundness than external checking systems. External checking systems have some advantages over annotations. First, annotations have poor *reuse*: one has to add them to every place in the code where they apply. External checkers isolate checking specifications into modular specifications which can then be *uniformly* applied to the program code. Second, users may not favor a tight integration between checking and program code the way annotations do it, because tight coupling implies difficulty in migration to different checkers in the form of rewriting effort. Third, the annotation effort is usually difficult to predict and measure *in advance* at the checker-design stage. For example, consider the Cqual annotation-based checker (Section 3.1.5). Its designers initially chose annotations believing that the annotation overhead will be manageable and will give good returns on the cost of annotating. However, when Cqual was applied to increasingly complex programs, it was found to yield a large number of false error notices. As a result, *new annotations were added* to suppress false positives. Thus it is difficult to always predict the amount of annotation effort needed. However, recent research has been focused at automatic inference of annotations saving the programmer from the effort. We discuss one such system, Houdini, in Section 4.3.

3. **Scalability**

   Automated bug detection can potentially be most useful for large complex programs. However, traditional automated checkers can easily get overwhelmed with the complexity of these programs. The key is to design *scalable* checkers. Consider the case of interprocedural vs. Intraprocedural analysis. Interprocedural analysis can potentially discover more bugs, and can be more precise than intraprocedural analysis. For example, 90% of the errors caught by the PREfix interprocedural checker (Section 3.2.1) involve interactions of multiple functions. However, interprocedural analysis can be very expensive in terms of CPU and memory requirements. Also, interprocedural analysis may scale super-linearly with program size. Intraprocedural, possibly flow sensitive analysis is more tractable. One commonly followed approach is *memoization*: maintaining "summaries" of analyzed functions for possible reuse (Section 3.2.5). Summaries make interprocedural analysis much more tractable. In fact, systems like Archer (Section 5.2.1) implement highly scalable path-sensitive interprocedural analysis by using summaries for analyzed functions. Note that summaries are mostly used in conjunction with *bottom-up* analysis so that any function call can be replaced by a summary for that function.

4. **Aliasing**

   Aliasing is often regarded as the bane of program analysis. Combating aliasing is a recurrent theme of the checkers presented in this paper. Static analysis systems that aim to give soundness guarantees (Section 3.1.5) need to make conservative assumptions about the presence of aliases. This significantly limits the power of a static analyzer. A common solution is to allow the programmer to declare a variable as un-aliased in a certain scope, allowing the analyzer to aggressively infer properties in that scope without fearing any aliases.

Chapter 3

Static Analysis Survey

In this section we describe a representative set of existing static checking systems. We place particular emphasis on the scalability and the usefulness of the checker. Scalability is important because our ultimate goal is checking large complex software like OS kernels. Usefulness has many dimensions: the more bugs a checker finds the more useful it is, fewer false bug reports, and ease of use by the end-*programmer* to easily add custom checks. We classify static checking techniques into annotation based methods, and tool based methods. Although this may not be the right choice, it certainly has the advantage of better capturing the programmer's imagination. We conclude this section with a quick analysis of how individual systems compare against each other.

3.1 Annotation-Based Techniques

Program annotations is the term given to specifying behavioral aspects of code. For example, the `const` qualifier in C specifies a behavioral constraint. Program annotations serve as a useful tool for automated checking of behavioral aspects of programs. Annotations can also be viewed as a means for augmenting the semantic and behavioral content of a program, by specifying additional constraints that the code must satisfy. Thus, annotations help enforce additional semantic checking of the program over the default checking performed by the compiler.

In this section we consider a few annotation-based tools for static checking. Although this is not in any way a comprehensive listing, still to the best of our knowledge, it is a representative list of the state of annotation assisted checking in software.

3.1.1 Lint

Lint is probably the oldest static checker (other than the language compiler) that continues to be widely used today. Lint checks C programs for a small fixed set of syntax and semantic errors. It performs simple flow sensitive and type aware checking, but can only detect violations of basic program properties. In fact, Lint is now essentially superfluous since many of the checks performed by Lint have been in GCC for a long time now. However, Lint is of historical value given the set of checkers that were inspired by it, two of which are the subject of this section: LCLint and Splint.

Lint is a command-line driven tool with command line arguments for specifying which of the several checks provided by Lint should be applied to the program. The annotation support in Lint was introduced for suppressing false alarms. One additional feature of Lint is its "code portability" check: when invoked with the `-p` option, Lint checks the code for possible portability issues from its dictionary of known portability problems [122].

3.1.2 LCLint

LCLint [2, 49] grew out of the Larch Project at the MIT Lab for Computer Science and DEC Systems Research Center. It was originally named after LCL [123], the Larch C Interface Language and Lint.

LCL and Larch The Larch family of languages [65, 92] represent a two-tiered approach to formal specification. A specification is built using two language: *the Larch Shared Language* (LSL), which is independent of the implementation language, and a *Larch Interface Language* designed for the specific implementation language. An LSL specification defines *sorts*, analogous to abstract types in a programming language, and *operators*, analogous to procedures. It expresses the underlying semantics of an abstraction.

The interface language specifies an interface to an abstraction in a particular programming language. It captures the details of the interface needed by a client using the abstraction and places constraints on both correct implementations and uses of the module. The semantics of the interface are described using primitives and the sorts and the operators defined in LSL specifications. Interface languages have been designed for several programming languages. LCL [123] is a Larch interface language for Standard C. LCL uses a C-like syntax.

LCLint Much of the difference between Lint and LCLint is that Lint's annotations are related to minimizing the number of false positives, whereas LCLint's annotations have more features, only some of which are used for this purpose. LCLint makes extensive use of annotations to define behavioral aspects of code and to suppress false positives. LCLint's annotations were originally derived from a subset of LCL specifications. Although separate LCL specifications can provide more precise documentation on program interfaces than is possible with LCLint annotations, LCLint's designers chose the annotations approach over separately provided specifications for obvious reasons.

LCLint can detect problems like violations of information hiding; inconsistent modifications of caller-visible state or uses of global variables; memory management errors including uses of dead storage and memory leaks; and undefined program behavior. LCLint checking is done using simple dataflow analyses and is potentially as fast as a compiler.

LCLint was designed to run frequently on large programs and hence checking efficiency and scalability were important requirements. As a result, LCLint employs a number of simplifications for faster analysis and ease of use. For example, LCLint does not incorporate inter-procedural data flow analysis, instead relies on programmer annotations for adjudging the role of function calls, and loops are considered identical to `if` statements for the purpose of the analysis. Also, LCLint focuses specifically on errors at interface points. Hence, it only uses weak intra-procedural alias analysis that infers inter-procedural actions from the annotations.

3.1.3 Splint

Splint 3.0.1 [3] is the successor of LCLint version 2.5q. The main changes are support for secure programs (hence the name Splint) and extensible checks and annotations.

User-defined Annotations Currently, LCLint users are limited to a pre-defined set of annotations and associated checking. This works well as long as their programming style is consistent with the methodology supported by LCLint (e.g., abstract data types implemented by separate modules, pointers used in a limited systematic way), but is problematic if they need to check a program that does not adhere to this methodology. For example, LCLint provides annotations for checking storage that is managed using reference counting. An annotation is used to denote an integer field of a structure as the reference count, and LCLint will report inconsistencies if new pointers to the structure are created without increasing the reference count, or if the storage associated with the referenced object is not deallocated when the reference count reaches zero. If a program implements reference counting in some other way (for example, by keeping the reference counts in a separate lookup table), however, LCLint provides no relevant annotations or checking. More generally, applications have application-specific constraints that should be checkable statically. These could include value consistency requirements, event ordering requirements, and global constraints. Splint provides extensions to LCLint that address this problem by supporting user-defined annotations. Programmers have some freedom to invent new annotations, express syntactic constraints on their usage, and define checking associated with the user-defined annotation in different contexts.

3.1.4 Extended Static Checking

ESC/Java [54] descends from the intellectual tradition of program verification. It is an annotation-based static checker for Java programs that uses theorem solving of program predicates to check for correctness. ESC/Java is intermediate in both power and ease of use between type-checkers and theorem-provers, but it aims to be more like type-checkers and is lightweight in comparison with to theorem-provers. This way, the designers of ESC/Java hoped to make it easy to use by the programmers (like type-checkers) rather than extraordinarily powerful but so difficult to use that programmers shy away from it (like theorem-provers). Also, hence the name "extended static checker." Figure 3.1 shows the intuitive placement of "extended static checking" between conventional type-checking and program verification [54].

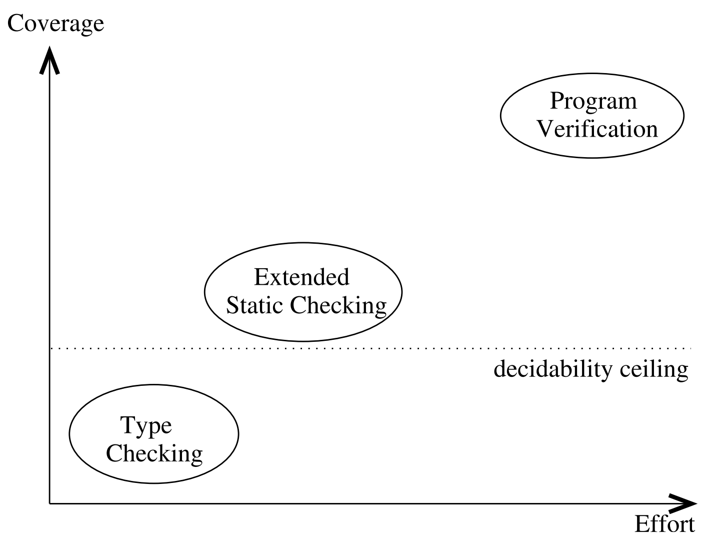

Figure 3.1: Static checkers plotted along the two dimensions coverage and effort (not to scale). Extended Static Checking as implemented in ESC/Java is intermediate both in coverage and effort needed between type-checkers and program verification systems.

At the time ESC was first proposed [42], powerful static checkers needed programmer assistance for checking. This is not in agreement with ESC's goal for ease of use and hence ESC does not require any programmer assistance while checking.

ESC/Java is designed to scale well with code size. It employs modular checking. Specifically, when checking the body of a function `f`, ESC/Java does not examine bodies of functions called by `f`. Rather, it relies on the specifications of those functions, as expressed by annotations. Hence, ESC/Java can check individual components of a system separately, leading to good scalability.

ESC/Java supports a powerful specification language. The specification language is the language used to write ESC/Java annotations. It is composed of one or more *specification expressions*, or *pragmas*. ESC/Java provides a rich specification language which revolves around these expressions. ESC/Java expressions can be arbitrary first order logic formulas. Also, they can be nested or combined like standard Java expressions, and are type-checked in ways similar to Java code. However, there are inevitably some differences between them and standard Java syntax. Also, ESC/Java specifications can be provided in separate files (`.spec` files). This is useful when the Java sources they annotate are unavailable for use by the checker.

ESC/Java checks the first-order logic expressions using the Simplify automatic theorem prover [38]. First-order is logic is undecidable, hence no theorem prover can prove it in polynomial time. Simplify is a best effort theorem prover and tries to prove or disprove its input till as long as it can before giving up.

The operation of ESC/Java consists of the following steps:

1. First, ESC/Java loads, parses, and type-checks the files named on the command line, as well as any other files needed because their contents are directly or indirectly used by files named on the command line.

2. Next, for each class whose function bodies are to be checked, ESC/Java generates a type-specific background predicate encoding such information as subtype relations, types of fields, etc. in the class to be checked and the classes and interfaces it uses.

3. Next, ESC/Java translates each function to be checked into a logical formula called a verification condition (VC). As an intermediate step in this translation, ESC/Java produces a command in an intermediate language [86] based on Dijkstra's guarded commands. The intermediate language includes commands of the form `assert E`, where `E` is a boolean expression of the language. An execution of a command is said to "go wrong" if control reached subcommand of the form `assert E` when `E` is false. Ideally, when a function `f` is translated into a command `C` and then to a verification condition `V`, the following three conditions should be equivalent:

   (a) There is no way that `f` can be invoked from a state satisfying its specified preconditions and then behave erroneously by, for example, dereferencing null, violating an assert pragma, terminating in a state that violates its specified postconditions, etc.

   (b) There is no execution of `C` that starts in a state satisfying the background predicate of `f`'s class and then goes wrong.

   (c) `V` is a logical consequence of the background predicate.

In practice, the translation is incomplete and unsound, so there may be semantic discrepancies between `f`, `C`, and `V`. Finally, ESC/Java invokes the Simplify [38] theorem prover, asking it to prove each body's verification given the appropriate background predicate. If an attempted proof succeeds (or if Simplify exceeds specified resource limits in attempting the proof, or if ESC/Java exceeds specified resource limits generating the verification condition), then ESC/Java reports no warnings for the body. If the proof fails (other than by exceeding resource limits), Simplify produces a potential counterexample context, from which ESC/Java derives a warning message.

Unsoundness and Incompleteness in ESC/Java ESC/Java is unsound and incomplete and can hence miss errors actually present in the program it is analyzing. This is inevitable from ESC/Java's design, since it is meant to be an extended static checker rather than a program verifier. However, ESC/Java provides pragmas for the programmer to control the soundness and incompleteness of checking. Thus users can balance annotation effort, completeness, and verification time. Here are some known causes of unsoundness in ESC/Java (taken from the ESC/Java Manual [76]).

- **Loops:** ESC/Java does not consider all possible execution paths through a loop. It considers only those that execute at most one complete iteration (plus the test for being finished before the second iteration). This is simple, and avoids the need for loop invariants, but it is unsound. The users can modify ESC/Java's treatment of loops by using the `-loop` command-line option to specify how many loop iterations should ESC/Java consider. We study methods for automatically inferring loop invariants in Section 4.2.1. Archer (Section 5.2.1) uses symbolic execution and heuristics to aggressively unroll loops whenever possible.

- **Trusting Pragmas:** The `assume`, `axiom`, and `nowarn` pragmas allow the user to introduce assumptions into the checking process. ESC/Java trusts them. If the assumptions are invalid, the checking can miss errors. For example, ESC/Java depends on programmers to supply `monitored` and `monitored_by` pragmas telling which locks protect which shared variables. In the absence of such annotations, ESC/Java will not produce a warning when a routine might access a variable without holding the appropriate lock.

- **Search Limits in Simplify:** If Simplify cannot find a proof or a (potential) counterexample for the verification condition for a routine within a set time limit, then ESC/Java issues no warnings for the method, even though it might have issued a warning if given a longer time limit. If Simplify reaches its time limit after reporting one or more (potential) counterexamples, then ESC/Java will issue one or more warnings, but perhaps not so many warnings as it would have issued if the time limit were larger. There is also a bound on the number of counterexamples that Simplify will report for any conjecture, and thus on the number of warnings that ESC/Java will issue for any routine. Again, these limits can be set by the programmer.

- **Concurrency:** Although we do not consider concurrency in this paper, it is nonetheless an important topic in program analysis research. Interestingly, ESC/Java assumes that the value of a shared variable stays unchanged if a routine releases and then reacquires the lock that protects it, ignoring the possibility that some other thread might have acquired the lock and modified the variable in the interim.

Here are some causes and consequences of incompleteness in ESC/Java:

- **Incompleteness of the Theorem-Prover:** The verification conditions that ESC/Java give to the Simplify theorem prover are in a language that includes first-order predicate calculus along with some (interpreted) function symbols of arithmetic. Since the true theory of arithmetic is undecidable, Simplify is necessarily incomplete. In fact, the incompleteness of Simplify's treatment of arithmetic goes well beyond that necessitated by Godel's Incompleteness Theorem. In particular, (1) Simplify has no built-in semantics for multiplication, except by constants, and (2) Simplify does not support mathematical induction. Also, first-order predicate calculus (FOPC) is only semidecidable—that is, all valid formulas of FOPC are provable, but any procedure that can prove all valid formulas must loop forever on some invalid ones. But it is not useful for Simplify to loop forever, since ESC/Java issues warnings only when Simplify reports (potential) counterexamples. Therefore Simplify will sometimes report a (potential) counterexample `C`, even when it is possible that more work could serve to refute `C`.

- **Modular Checking:** ESC/Java's use of modular checking causes it to miss some inferences that might be possible through whole program analysis. However, the designers chose to retain modular checking in lieu of its scalability benefits.

3.1.5 Cqual

Cqual (previously Carillon) [57] is an annotation-based extension to C that uses the power of *flow-sensitive* type-state analysis to catch bugs statically. Cqual is an amalgamation of ideas from type-checking and constraint-based analysis. The user extends the language type system with flow-sensitive type qualifiers, which are atomic properties that refine standard types similar to standard C type qualifiers such as `const`. Cqual checks constraint satisfaction on these type qualifiers inter-procedurally. Cqual's analysis is *sound*, meaning that it does not miss a bug, although it may report false errors (it is *incomplete*).

Programmers uses Cqual by specifying two things: they augment standard C code with type qualifiers, and, identify sub-typing relationship between the type qualifiers. For example, consider the following program constraint introduced to prevent format string attacks [116].

> *Data entered by a user should not directly be used as argument to* `printf` *since the user may be an adversary.*

This constraint can be specified in Cqual by introducing two type qualifiers for character strings: `tainted` for strings that could be controlled by an adversary, and `untainted` for strings that must not be controlled by an adversary. The sub-typing relationship between the two qualifiers is denoted as:

```
untainted < tainted
```

meaning that `untainted` is a subtype of `tainted`, any piece of code that accepts a `tainted` argument can be passed an `untainted` argument, but not vice versa.

The sub-typing relation is typically specified as a lattice of type qualifiers with partial-order relations between the qualifiers. CQUAL has a few built-in inference rules that extend the subtype relation to qualified types. Using the extended inference rules, CQUAL performs qualifier inference to detect violations against the type relations defined by the lattice. Intuitively, Cqual works by converting a program augmented by type qualifiers into constraints on type qualifier compatibility, which can be represented graphically. The problem of checking correctness is then reduced to graph reachability.

Cqual's analysis employs a unique combination of flow-insensitive and flow-sensitive analysis. It models program state as an *abstract store*, which is a mapping from variables to types (types contain qualifiers). The effect of a program statement on the state is emulated as a "Hoare logic" type effect with possible changes in variable *types* (`ALLOC` and `ASSIGN` operations). This flow-insensitive phase also includes alias analysis and effect inference. Following this, the flow-sensitive analysis phase handles flow-sensitive qualifiers and linearity issues for alias analysis. Finally, even though Cqual's analysis is not path-sensitive, it does not suffer from false-positives due to impossible paths. This is because Cqual's constraint satisfaction logic tries to find at least one way to satisfy the constraints, rather than finding all possible violations. However, since it does not do path-sensitive analysis, the error message may not be as informative (as in PREfix, for example).

One of the key innovations in Cqual is its practical approach to alias analysis. The general problem in alias analysis is making updates to one alias visible to others. Cqual handles this by adding a layer of indirection between pointers and their types by introducing an *abstract location* with each variable. With this change, a variable is associated with an abstract location, and abstract stores map abstract locations to qualified types. Thus a type update to an abstract location is automatically available to other references pointing to that location. An abstract location is defined as *linear* if the type system can prove that it corresponds to a single concrete memory location in every execution; otherwise, it is non-linear. Updates to linear locations can arbitrarily change their qualifiers (strong updates). Updates to a non-linear location result in a unification of qualifiers (weak update). Locations may be inferred as non-linear even if strong updates are desired on them (Cqual is sound). Evidently, strong updates give better opportunities to test qualifier constraints.

Linearity can be reclaimed by using Cqual's `restrict` keyword (or equivalently the `confine` keyword). The expression `restrict x = e1 in e2` introduces a new name bound to the value of `e1`. The name `x` is given to a new abstract location, and among all aliases of `e1` only `x` and values derived from `x` may be used within `e2`. Thus, the location of `x` may be linear, and hence may be strongly updated, even if the location of `e1` is non-linear.

Cqual's scalable, annotation-based analysis has been applied to large software including the Linux kernel. It discovered several locking bugs in Linux [57], and has been used to detect format-string vulnerabilities too [137]. Cqual is easy to use and provides strong soundness guarantees. However, there are two problems with Cqual. Firstly, checking pre-written code with Cqual needs substantial rewriting effort. Secondly, Cqual's conservative aliasing assumptions may lead to a high number of false-positives requiring additional annotation effort to suppress them (restrict operator). Recently, Cqual has introduced automatic *confine inference* by the static analyzer to reduce the burden of aliasing annotations. In fact, it has been observed to apply to roughly 95% of pre-written C code [10].

Usability Cqual's design places strong emphasis on usability. Firstly, due to its unique combination of intra-procedural and inter-procedural analysis, Cqual is quite scalable. This is further demonstrated by the number of bugs it has caught in the Linux kernel (4M+ LOC and counting). Secondly, Cqual is easier to use than some other annotation-based systems: it needs less annotation-effort, thanks to automatic inference of type qualifiers. Also, confine-inference has been shown to work 95% of the time on pre-written code. Thus, one does not need to add heavy annotations to enforce linearity. Finally, Cqual is getting ready for mainstream usage: it currently presents the analysis results using *Program Analysis Mode*, an Emacs-based GUI. However, there are two concerns with Cqual: (1) how much will the rewriting effort be for porting legacy code to Cqual (a Houdini [53] like annotation assistant would be useful), and (2) it is not clear what is the false alarm rate of Cqual's conservative alias analysis.

3.2 Tool-Based Techniques

Today, commodity-class static-checking technologies are still coming up and are in a constant state of flux. While it is widely recognized that static-checking should be integrated into the regular development process, emerging static-checking techniques have two major drawbacks. First, they suffer from highly noisy outputs, and second, existing technologies are still not very mature and continue to evolve. Consequently, developers tend to shy away from intrusive techniques like annotations. Tool-based checkers find it easier to be adopted, even if it is just out of curiosity. Indeed, many tool-based checkers do not aim for soundness and just try to find as many bugs as possible which help them achieve mass acceptance.

3.2.1 PREfix

PREfix [26] is a static-checking tool that was designed to achieve two main goals. First, PREfix should catch maximum number of bugs while keeping the number of false positives. This kind of requirement almost always translates to an unsound system. Second, and most importantly, PREfix should be *developer-friendly*, that is, developers should easily be able to integrate PREfix into their existing development practices. Although many static-checking tools cite this as their primary goal, none of them do it as good as PREfix does. In fact, Microsoft acquired the company that owned PREfix, and PREfix is now used in-house in Microsoft check code as it is written on a daily basis. It is also going to be available as a feature in upcoming releases of Visual Studio. PREfix makes three key choices to ensure developer friendliness:

- Design a tool-based system rather than an annotation based system.
- Minimum false positives: False positives are a serious practical drawback to the mainstream acceptance of any static-checking system. PREfix designers made minimizing false positives a first class design goal. Also, it is known that less than 10% of the code of PREfix is said to concern with analysis per se, most applies to the filtering and presentation of output, to reduce the number of false positives [125].
- Provide useful error flow information.

PREfix mainly targets memory errors like uninitialized memory, buffer overflows, NULL-pointer dereferences, and memory leaks. One main weakness of PREfix is that in the general case, there is not a method for programmers to define their own custom checks.

PREfix is a path-sensitive static analyzer that employs symbolic evaluation of execution paths. Path-sensitivity ensures that program paths analyzed are only the paths that can be taken during execution, thus keeping the number of false positives to a minimum. Also, path-sensitivity helps PREfix provide useful information about the reported errors. However, path-sensitive analysis can incur a heavy cost; exponential path blowup due to control constructs and possibly infinite paths due to loops make it impractical. The solution adopted by PREfix is to explore only a representative set of paths, whose number is configurable by the user. This is archetypal of PREfix's design goals: sacrificing soundness to get maximum errors with minimal false positives. It is possible to make the system sound, by for example, applying less heavy-weight analysis on the remaining paths (e.g., fixed-point based data flow analysis). However, this would be in direct contradiction with PREfix's goal of keeping the false-positive count extremely low.

PREfix employs a summary-based *bottom-up* inter-procedural analysis:

- Functions are analyzed in bottom-up order in the call graph.
- As functions are analyzed, a summary of constraints is generated for them.
- Since the analysis is bottom-up, every function call can be replaced by the callee's summary. Thus a function gets analyzed for each possible call site that calls it. In other words, the analysis minimizes the impact of missing summaries for functions yet to be analyzed.
- Recursion is broken arbitrarily.

PREfix running time scales linearly with program size due to the fixed cutoff on number of paths. Some other interesting lessons learned from PREfix are:

- A bulk of errors come for inter-procedural interactions. Hence, static analyzers should be inter-procedural.
- Good user-interfaces with minimal false positives and visual traces of error paths are very handy.

3.2.2 ESP

ESP is a highly scalable, property verification system for large C/C++ programs. It takes as input a set of source files and a specification of a high level protocol that the code in the source files is expected to satisfy. Its output is either a guarantee that the code satisfies the protocol along all execution paths, or a browsable list of execution traces that lead to violations of the protocol. ESP is targeted towards large code bases where it is nearly impossible to verify system-wide properties manually . Also, in existing large scale systems, it is infeasible to annotate the code for potentially checking all possible properties of interest. In addition, protocol specifications can typically be isolated into separate aspects rather than be spread apart through out the entire code base. ESP uses these protocol specifications to check conformance in the code.

The analysis engine of ESP was designed to achieve four goals: it must not validate any program that contains errors, it must scale to very large programs, it must report few false errors, and it must produce useful feedback that allows the programmer to investigate and fix errors.

ESP achieves these goals by combining two separate phases of program analysis: Phase one is a global context-sensitive control-flow-insensitive analysis that produces a call graph and information about the flow of values in the program. Phase two is an inter-procedural context-sensitive data-flow analysis that incorporates a restricted form of inter-procedural path simulation. The first phase produces the program abstractions used by the second phase, and also serves as a safety net for the second phase: it is used to conservatively answer questions about the corner cases that the more precise (and expensive) analysis in the second phase cannot handle.

ESP's design reflects the fundamental trade-off between precision and scalability. Whereas type-inference or data-flow analysis are highly scalable (due to their unification semantics: program state is unified at each join point), path-sensitive analysis is not. However, path-sensitive analysis is more precise since it considers only feasible paths. Just like PREfix, it is ESP's primary goal to implement a precise analysis thus keeping the number of false positives very low while finding a lot of errors. Hence, ESP chose to retain path-sensitive analysis. Scalability concerns are addressed by restricting the property space to finite state temporal safety properties. This was assumed to be an intuitive model. The property is specified using a finite state automaton (FSA). As the program executes, a monitor maintains the current state of the FSA and signals an error when the FSA transitions into an error state. So the goal of verification is: is there some execution path that would cause the monitor to signal an error ?

Let us compare path-sensitive analysis with data-flow analysis. Assume that each of the analyses attempts to verify a certain property. The fundamental difference between the two is that data-flow property analysis tracks only the state of the FSA and ignores non-state–changing code. On the other hand, path-sensitive analysis symbolically executes the program and can thus track execution state in addition. As a result, at branch points, path-sensitive analysis can avoid infeasible paths: it queries the current execution state, if the current execution state can decide a branch direction, it is taken. Otherwise, the current state is split and both branches are processed individually. In comparison, data-flow analysis would simulate the state along both branches combining the end result at the join point.

ESP has been successfully used for checking some large software [91]. ESP has been used to verify certain file I/O properties of a version of the gnu C compiler (roughly 150,000 lines of C code). ESP has also been used for validating the Windows OS kernel against a category of security properties.

3.2.3 Abstract Interpretation: ASTREE

Abstract interpretation [8, 74](AI) is a form of program analysis that maps programs into more abstract domains. This makes analysis more tractable and potentially useful for checking. ASTREE is a static program analyzer that uses abstract interpretation to discover bugs. The bugs targeted by ASTREE include out-of-bound array accesses, integer division by zero, and similar errors. Additionally, ASTREE checking can also include user-defined *asserts*. One restriction imposed by ASTREE on the class of C programs it checks is that they should not contain any dynamic memory allocation, string manipulation, and restricts pointer use. This allows for a fast and precise memory analysis which would not be possible otherwise. Also, this restriction has not proved to be a problem for verifying the class of programs ASTREE was originally designed for: embedded programs as found in automobiles and aerospace.

ASTREE makes certain design choices which can be explained in the context of safety-critical embedded programs as follows:

- ASTREE's static-checking is sound considering the need to give correctness guarantees on safety-critical applications. Hence, in the case of ASTREE, the soundness-completeness trade-off is more in favor of soundness than completeness.

- ASTREE is fully automatic and does not need any user help. In particular, ASTREE does not use annotations, since a huge body of code already exists for the target systems and annotating it would be too much effort.

- ASTREE is designed to be efficient and scalable to handle industry class program sizes.

A consequence of soundness can be low precision or high rates of false alarms. ASTREE is carefully designed to not suffer from this problem. First, ASTREE is parametric, that is, the trade-off between cost of the analysis and precision of analysis can be fully adapted depending on need. Second, ASTREE is modular. It is made of pieces (called *abstract domains*) that can be assembled and parameterized to build application-specific analyzers, adapted to a domain of application or to end-user needs. Third, ASTREE has been extended with domain-specific knowledge about target applications to better test sophisticated behavior. ASTREE counts on its modular, parametric, and domain-aware features to provide precise analysis with minimal false alarms.

One of the key benefits of using abstract interpretation is a parametrizable precision-efficiency trade-off. There is usually an easily-tunable balance between precision and efficiency in abstract interpretation that is duely exploited by ASTREE. This feature arises from the theory of abstract interpretation, especially, the notion of abstract domains [90].

Overall, ASTREE is sound, automatic, efficient, domain-aware, parametric, modular and precise. However, it still does not offer the same easy extensibility present in ESP or Metal (the official Website of ASTREE recognizes this as a focus of current work).

ASTREE has been successfully used in static analysis of industry-class synchronous, time-triggered, real-time, safety-critical, embedded software written or automatically generated in the C programming language. Specifically, it has been used for verifying parts of the Airbus A340 and A380 subsystems. Notably, one of the applications involved the verification of 132,000 lines of C code and was completed in 1 hour 20 minutes on a modestly equipped machine.

One serious criticism of ASTREE is that in the presence of sound analysis, it gets increasingly more difficult to keep false alarms low as the program size increases. The current methods employed in ASTREE seem to require developer participation. Introducing SLAM like predicate discovery and contradiction would potentially allow the system to reduce the number of false positives.

3.2.4 SLAM

SLAM combines and extends techniques from predicate abstraction, model checking, and predicate refinement to soundly and precisely check C programs. SLAM output is a set of error traces. SLAM is sound in that it does not miss errors, but it is nearly precise in that it raises very few false alarms. The key thesis behind SLAM is that it is possible to *soundly* and *precisely* check a program against a specification of API rules by:

- creating a program abstraction,

- exploring the abstraction's state space, and

- refining the abstraction.

SLAM accepts as input the client C code as is, that is, without any annotations, and a specification of API usage rules in the SLIC language. The program abstraction component of SLAM is called `c2bp` (for C to Boolean Program). It accepts as input a `C` program `P` and a set of predicates `E` specified in SLIC. SLIC allows predicates to be specified as pure C boolean expressions. It outputs a boolean program `bp(P,E)` that is a sound and (boolean-) precise abstraction of `P`. In particular, the skeletal boolean program so produced has all its if conditionals removed thus exposing every possible path in the underlying program. This boolean program is then analyzed by Bebop. Bebop is the SLAM component that checks a boolean program for reachability of predicate violations. It uses symbolic inter-procedural data flow analysis and tests CFL reachability. However, not all error traces produced by Bebop may be real errors. This is because the boolean program does not have any conditionals and all paths are analyzed invariably. SLAM's answer to this imprecise analysis is a third stage of analysis called Newton. Newton performs counterexample-driven refinement. It symbolically executes the error paths in the original C program, checking for path infeasibility at each conditional. This is done with the help of the Simplify theorem prover. If the path is found infeasible, Newton generates new predicates to rule out the infeasible path.

An important problem in SLAM's design is proving termination. SLAM draws ideas from abstract interpretation: *widening* and abstract interpretation with infinite lattices (WAIL) [37]. The authors have shown that their strategy of finite abstractions combined with iterative refinement (FAIR) is more powerful than widening in abstract interpretation over infinite lattices [15]. Like widening, their algorithm loses precision as it tries to achieve a fixed-point. However, FAIR's termination condition is stronger than WAIL's termination condition and hence always terminates if WAIL does.

SLAM is considered a promising new technology for seamlessly combining code checking with development. The key features of SLAM that make this possible are no need for user intervention (API specifications can double up as SLIC input), low noise rate, soundness, and scalability.

3.2.5 Metacompilation

Metacompilation (MC) [47, 66] is the process of augmenting a traditional compiler with system rules for checking at compile-time. It represents a clear division of labor: the compiler provides a generic infrastructure for static analysis, and the programmer provides the rules to be checked. The focus of MC is on pragmatism. Checking rules are specified using a language called *Metal*. Metal is designed for (1) ease of use and (2) flexibility to support a wide range of rules within a unified framework. Ease of use is important since MC by itself cannot detect bugs: it needs the developers to specify checking rules governing their system in Metal. MC can be just as successful in catching bugs as the developer is in specifying them effectively. Flexibility is important so that developers can specify a wide range of properties for checking.

Checking-rules in Metal are specified using a state machine (SM) abstraction. SMs were chosen since they are a familiar concept to programmers. However, the authors claim that SMs only provide sugar for common operations and they do not limit extensions to checking finite-state properties. When needed, extensions can be augmented with general-purpose code. The SMs transition from one state to the other based on the source code pattern observed. For example, the following transition tells Metal to transition to the state `v.stop` whenever the pointer `v` is dereferenced in the `v.freed` state:

```
v.freed: { *v } ==> v.stop
```

Thus the alphabet of each SM is defined by the Metal patterns used within the extension. Patterns serve the purpose of identifying source code actions that are relevant to a particular rule. Intuitively, SM transitions map directly to *path-specific* actions in the source code.

Each individual SM's current state consists of one *global* state value and one or more *variable-specific* state values. Global state values capture a program-wide property (e.g., "interrupts are disabled"). Variable-specific state values capture program properties associated with specific source objects (e.g., "pointer p is freed"). Besides, state transitions in Metal can be augmented with general-purpose code which has the effect of a "dynamic" state space for the SM.

Implementation The inter-procedural analysis engine of the MC system is called *xgcc*. It performs bottom-up summary-based traversal of the call graph like PREfix (Section 3.2.1). xgcc's role can be described as mapping of the SM abstraction provided by Metal to lower level program analysis. xgcc follows the approach of splitting the analysis into an intra-procedural pass followed by an inter-procedural pass. The intra-procedural analysis is implemented as a DFS traversal over the abstract syntax tree (AST) of the function. Block-level state caching is used to speed things up. Since the analysis is path sensitive, a program point may have different states associated with different paths. Storage requirement is further reduced by separating variable states from program point states. In this way, variable states may be shared, and if the states are the same, applying the same transition will have the same effect.

The main contribution of the inter-procedural analysis phase is *refinement* and *restore* of the SM state. State refinement occurs when a function call is encountered and followed. Any object that passes from the caller's scope to the callee's scope should retain its state which has the effect of refining the incoming state at the callee. The state is restored when the analysis returns from the callee and resumes analyzing the caller. The restore operation may need to move the state back from an object in the callee's scope to the corresponding object in the caller's scope. The extension's global state passes across function call boundaries unchanged.

The inter-procedural path-sensitive analysis employed by the MC system is memory and CPU intensive. In fact, for analyzing systems of the size of the Linux kernel (more than 4 million lines of code and counting), analysis scalability can play a determining role in the overall feasibility of the system. The MC system uses composable summaries of function behaviors for reducing running time. Once functions are analyzed, they are condensed into summaries which state how the function transforms the possible incoming transition states. Function summaries are used to reproduced the effect of analyzing the function once there is a hit in the summary-cache at a function call boundary.

False Positive Suppression The MC system adopts several methods to suppress false positives. *False path pruning* is used to identify infeasible paths. For example, in the following code, there are only two possible paths, not four. The two impossible paths can drive an SM into an error state: freeing unallocated pointer, and allocating a pointer but not using it.

```
if(x)
{
    v = kmalloc(10);

    // use v
}

if(x)
{
    kfree(v);
}
```

MC performs limited symbolic execution of the program to statically deduce which conditionals can be taken and which ones cannot. Based on this information, it prunes impossible paths. The actual implementation uses a congruence closure algorithm [45]. Variables that must have the same value are placed into a single equivalence class. The congruence closure algorithm is then used to derive as many equalities and non-equalities as possible. This information, combined with the symbolic tracking of inequalities, helps deduce the truth value of most conditionals. The developer may also be able to rewrite Metal extensions to weed out observed false paths in further runs. Finally, Metal saves false path pruning effort by storing descriptions of identified false paths for use in later runs of the extension, thus avoiding redundant effort.

Some false positives may be left behind despite the above measures. MC handles them by *ranking* error reports based on many heuristics so that errors most likely to be genuine bugs are reported at the top.

Conclusion The MC system represents a significant point in the design space of static checkers. First, it follows a pragmatic approach to tackle the soundness-usefulness trade-off. MC is not intended to be sound: it uses incomplete symbolic execution, does not unroll loops, etc. However, it aims for ease-of-use by developers thus maximizing its impact on software development. Second, MC adds another dimension to "usefulness": finding as many bugs and as many *types* of bugs as possible. The authors justify unsoundness by claiming that soundness directly conflicts with this goal of maximum coverage. Indeed, the MC system has discovered several thousand bugs in the Linux and BSD kernels.

However, the scalability issues of the MC system are not very clear. Inter-procedural path-sensitive analysis can easily explode in CPU and memory requirement. It seems that the MC system avoids this by (1) having small or moderately sized SMs that do not generate a lot of state throughout the entire Linux kernel, and (2) running only a few extensions at a time.

3.3 Language-Based Approaches

Language-based methods address the problem of programming bugs at a fundamental level: by defining a new programming language that ensures *safety*. Particular emphasis is placed on memory related errors since they pose the most immediate threat to systems programs

It is tempting to explore the application of existing type safe languages like `Java`, `C#`, and `OCaml` for systems programming. However, these languages are not suitable for this purpose due to the following reasons:

1. Performance overhead from dynamic array-bound checks, garbage collection, dynamic typing, etc.

2. Memory overhead due to dynamic typing information.

3. Lack of control on low level layout of data structures and resource management.

4. Effort needed to port legacy C code to the new language.

5. Lack of language extensibility to add system-specific compile-time checking.

Due to these problems with existing type-safe languages, researchers are exploring the option of defining new safe languages that combine C's flexibility and performance with type-safety and extensibility. Cyclone [73] is a C-like language that enforces stronger semantics without sacrificing performance. Vault [104] is another C-like language that allows the programmer to extend the type system with system-specific rules.

The key language facility leveraged by these languages is the language type system. A language type system has two important roles. First, it serves to provide the semantics for various operations so that the compiler can generate appropriate machine code. For example, generating different instructions for adding integers and floating points. Second, an object's type defines the kind of operations and their semantics that can apply on that object. Equivalently, types define an equivalence class for all objects of that type with regard to the kind of allowed operations. For example, given a value `x` of `struct` type with a field `f`, it is valid to form the index `x.f` but the expression `f.x` is invalid.

3.3.1 Cyclone

Cyclone [73] is a programming language-based on C that has the following goals:

- safety,
- flexibility to give nearly the same control over data representations and memory management as C,
- no worse performance than C, and
- as close to C as possible in power, syntax, and semantics.

Cyclone achieves these goals without being drastically different from C. In fact, the language syntax and semantics of Cyclone are the same as C, with some differences. In particular, the language is enhanced with annotation support and additional keywords and operators. This would probably give the impression that Cyclone is a superset of C. But, there are C programs that a Cyclone compiler will not accept either because they are unsafe or because the compiler cannot prove that they are safe. Hence Cyclone is best described as a *dialect* of C.

Cyclone uses a combination of program annotations, static analysis, and runtime checking to ensure safety. The Cyclone compiler performs static analysis on the source code, and inserts run-time checks into the compiled output at places where the analysis cannot determine whether an operation is safe. The compiler can also refuse to compile a program, which can happen if the program is truly unsafe, or because Cyclone cannot guarantee that the program is safe even after inserting runtime checks. A downside is that the compiler can reject some truly safe programs as well since it is impossible to implement an analysis that perfectly separates the safe programs from the unsafe programs.

Cyclone can be best understood by taking C, removing unsafe constructs, and replacing them with new, safe operations.

From C to Cyclone

The goals of Cyclone seem to be conflicting and this makes its design interesting. The main goal is to give absolute safety guarantees in C-like code. For this purpose, it tries to address all possible causes of programming bugs in C code individually. The engineering approach taken is to strip off all those constructs in C which are absolutely not possible to be accommodated in a safe language without sacrificing any of the goals identified above.

The first thing to go under the axe is C's pointer arithmetic. Pointer arithmetic is unsafe and difficult to verify for safety. Note that it is not impossible to ensure pointer safety; this can be done with runtime checking, but such an approach would have serious effects on performance. Hence, Cyclone simply rejects all potentially unsafe and unverifiable code. Instead, Cyclone provides language mechanisms to safely implement the corresponding programming idioms. Let us consider buffer overflow bugs. Buffer overflows are often the result of incorrect pointer arithmetic. Cyclone simply disallows pointer arithmetic on regular C pointers. Instead, if programmers want to use pointer arithmetic, they are required to use Cyclone provided pointers for this purpose. Cyclone provides *fat pointers*, which are standard C pointers augmented with the length of the target buffer. Thus, it can check arithmetic on fat pointers at runtime by checking for overflow against the buffer length. Cyclone is careful to handle all possible corner cases possible with using fat pointers (pointer assignments, pointer parameter passing, etc.). For example, when a fat pointer is cast to a regular C pointer its bounds are checked, and when a regular C pointer is cast to a fat pointer, its length is initialized to one.

Dereferencing NULL pointers is a common programming error, Cyclone avoids it by inserting runtime checks for NULL values. However, the programmer has control over the placement of checks, in particular turning off unnecessary checks. Cyclone provides the *never-NULL* pointer type for this purpose. A never-NULL pointer is not checked for NULL value since it is known to be not null. When a pointer is first cast to a never-NULL pointer type it is checked for NULL. Thereafter, the never-NULL pointer is not checked. Note that the fat pointer and never-NULL attributes of pointers are orthogonal and hence can be combined freely. These are examples of Cyclone's extended pointer type system for ensuring safety.

Let us now look at Cyclone's mechanisms for implementing safe memory management [63, 67]. This can be helpful in preventing common problems like dangling pointers. Cyclone provides a range of flexible options for implementing safe memory management:

- Region-based memory management using lexical or dynamic regions.
- Linear objects.
- Reference-counted objects.
- Garbage collection.

A *region* is a segment of memory that is deallocated all at once. For example, Cyclone considers all of the local variables of a block to be in the same region, which is deallocated on exit from the block. For code that is spread in different blocks but wants to use the same region, Cyclone provides *growable regions*. Cyclone's static region analysis keeps track of what region each pointer points into, and what regions are live at any point in the program. Any dereference of a pointer into a non-live region is reported as a compile-time error. However, Cyclone's region analysis is intra-procedural, it relies on programmer annotations to track regions across function calls.

In all, Cyclone ensures safety by outright disallowing a small subset of operations, and by checking the code using a combination of static analysis and runtime checking. Table 3.1 summarizes how Cyclone handles some of the common memory bugs found in C programs.

Problem	**Cyclone's Solution**
Null pointer	insert checks, cut down checks using static analysis and `@never-null` attribute
Buffer overflows	use fat pointers, cut down checks using: static analysis and `@nozeroterm` attribute
Uninitialized pointers	reject using static analysis at compile time
Dangling pointers	detect dangling pointer dereferences using static analysis (region analysis), convert all calls to `free` to no-ops
Type-varying arguments (e.g., arguments to `printf`)	tagged union, injections, varargs
goto,switch,setjmp/longjmp	reject at compile-time

Table 3.1: Table showing how Cyclone handles various types of errors

Discussion

Cyclone is an attempt at enforcing a stronger type system using annotations. For example, programmers can help Cyclone in its analysis, and in the process help themselves (by avoiding unnecessary checks to be inserted by Cyclone). One relevant question is what is the extent and power of cyclone's static analysis ?

Scalability Cyclone only implements intra-procedural analysis. Consequently, it requires that function interfaces be clearly qualified with the appropriate types expected by the function. The designers argue that the ability to compile different components of the system separately is critical for scalability in systems design. Although interfaces may be used for unavailable code, a complete system-wide inter-procedural analysis is not acceptable as a standard programming language feature.

Character strings in Cyclone One notable aspect of Cyclone's checking is its approach to inferring the end of character strings. The end of a character string is either the first occurrence of a zero byte in it or, the allocated length for that string pointer. Cyclone's type analysis automatically checks for the end of string using both these clauses in this order. Although this may be useful, it seems that this could result in false positives. A possible alternative to have the best of both worlds is to add another annotation for `char` that identifies the string as null-terminable.

3.3.2 Vault

Vault [104] is intended to be a safe version of C, although many of its features are closer to Java. In this section we present some interesting features of Vault.

Extending type system with system-specific rules The key novelty in Vault as a programming language is that the programmer can easily extend the language type system with system-specific rules. Type system extensions allow the programmer to express behavioral aspects of their code in Vault, and have the compiler check for conformance automatically at compile time. In this way, Vault adds an important new dimension to the type system: resource management and access control.

Memory Management A novel feature of Vault is that its analysis can support aliasing, at least to some extent. Aliasing is by far the bane of static analysis. Specifically, to be sound, static analyzers have to make conservative assumptions about aliasing. This hurts the precision of checking. Vault provides linear data types to handle aliasing. Linear data types cannot be aliased, and hence the static analysis can make strong predictions about their state. This helps provide powerful safety guarantees. However, past attempts at using linear types imposed strict non-aliasing restrictions limiting the use of linear types in mainstream programming. Vault introduces *adoption* and *focus* operations to have the best of both worlds [50]. The programmer uses these keywords to declare to the compiler one of two things: (1) no alias exists for this variable in this scope, and (2) this variable may be aliased in this scope. The compiler can use the information about the absence of any aliases in a particular scope to implement more aggressive analysis.

Vault uses region analysis to implement safe memory management. By combining region analysis with its advanced linear type system, Vault is able to effectively track memory usage at compile time. Also, Vault allows programmers to attach states to linear types. Programmers can use this feature to enforce higher-level protocols on linear objects.

Casting and converting between types Experience suggests that C's permissiveness in casting incompatible types into each other is the cause of some of the most notorious bugs in C programs. Vault designers address this problem by disallowing such arbitrary casts. The underlying thesis is that there are two main reasons for programmers to use casting between different C data types: to create generic data structures and algorithms, and to create a string of raw bits. Allowing arbitrary casts is not the right way to implement these operations since it is bug prone. Instead, Vault handles the first case using parametric polymorphism, and provides specific mechanisms to support the second.

3.3.3 Comparison of Cyclone and Vault

Cyclone and Vault are different systems with different compatibility goals and hence different trade-offs. Cyclone places the emphasis on preserving the programming style of C even if that means runtime checking of some invariants. For example, Cyclone gives the programmer complete control over memory layout and memory management, and as a result, cannot avoid deferring some tests to runtime since they cannot be validated at compile time. Vault, on the other hand, aims for detecting all violations at compile-time. As a result, its memory usage model is designed for completeness rather than resemblance to C.

Annotation Effort Cyclone needs less annotation effort than Vault. However, Vault allows the programmer to check system-specific properties using annotations, which can significantly increase the total amount of annotations required. Still, Cyclone can infer most region annotations automatically, which is not true for Vault. Finally, even though Vault's annotations for system-specific properties can get quite verbose, these annotations can double up as formal specification for the properties.

Code Migration Efforts Cyclone presumably will need less time for code migration from C. This will mostly be for making syntactic changes to C programs. Certain advanced memory management idioms could need more work though. On the other hand, porting C code to Vault will need significantly more effort.

Performance Cyclone programs have been shown to have a performance overhead of 0% to 150% over their original C implementations. This can largely be attributed to fat pointers and runtime bounds checking. We wrote some simple programs using Cyclone and analyzed the runtime checks Cyclone compiler inserted in the output. Our experience indicates that Cyclone in its pursuit of soundness, inserts checks for catching all possible violations in code. Although programmers can insert annotations to reduce the inserted checks to minimum, they often cannot cover all possible cases. This is similar to the problem static checking set out to solve in the first place, only that it is simpler. It seems that Cyclone can possibly benefit from automated annotation inference based on usage, or, an automated "Cyclone editor" that provides annotation hints to programmers as they type code.

Even though these proposed replacements to C are designed to allow easy migration for programmers from C, language-based solutions are not likely to be adopted in mainstream systems because they are not completely backward compatible with all C code. Rewriting the huge existing C code base would be a monumental exercise. However, software projects that create a new code base from scratch may be more willing to embrace a new language which gives soundness guarantees, though compatibility issues with existing libraries and interfaces is not very well understood. Alas, most software development is incremental.

3.3.4 MOPS

MOPS is not a new language like Cyclone and Vault, but is similar in scope; it extends the core language by defining modular extensions called *meta-object protocols* (MOPS). MOPS was pioneered by Xerox PARC [130] and is part of their broader vision in which traditional software abstraction mechanisms are considered insufficient to divide complex properties into modules. They propose meta-level specifications which control or augment existing software. This idea further led to aspect-oriented programming.

Metaobject protocols were originally defined as supplemental interfaces to programming languages that give users the ability to incrementally modify the language's behavior and implementation. Languages that incorporate metaobject protocols blur the distinction between a language designer and a language user. Traditionally, designers were expected to produce languages with well-defined, fixed behaviors (or semantics). Users were expected to treat these languages as immutable black-box abstractions, and to derive any needed flexibility or power from constructs built on top of them. This sharp division was thought to constitute an appropriate division of labor. Programming language design was viewed as a difficult, highly-specialized art, inappropriate for average users to engage in. It was also often assumed that a language design must be rigid in order to support portable implementations, efficient compilers, and the like.

The metaobject protocol approach, in contrast, is based on the idea that one can and should "open" languages up, allowing users to adjust the design and implementation to suit their particular needs. In other words, users are encouraged to participate in the language design process. If handled properly, opening up the language design need not compromise program portability or implementation efficiency.

In a language-based metaobject protocols, the language implementation itself is structured as an object-oriented program. This allows the power of object-oriented programming techniques to be exploited to make the language implementation adjustable and flexible. In effect, the resulting implementation does not represent a single point in the overall space of language designs, but rather an entire region within that space.

More recently, it has become clear that systems other than programming languages can have a metaobject protocol. In fact, many existing systems can be seen as having ad-hoc metaobject protocols. This was one of the observations leading to the concept of Open Implementations. Open Implementation is a software design technique that helps write modules that are both reusable and very efficient for a wide range of clients. In the Open Implementation approach, modules allow their clients individual control over the module's own implementation strategy. This allows the client to tailor the module's implementation strategy to better suit their needs, effectively making the module more reusable, and the client code more simple. This control is provided to clients through a well-designed auxiliary interface.

Kiczales et al. [58] and Lamping et al. [84] are two of the earlier works on the use of MOPS that take the approach of pushing domain-specific information into compilation. However, such protocols are typically dynamic and have fairly limited analysis abilities. Open C++ [28] introduces static MOP that allow users to extend the compilation process. All these extensions are limited to syntax-based tree traversal or transformation. In particular, they do not perform elaborate data flow analysis for applying the MOP specification to the code.

	Feature	**Splint**	**ESC/Java**	**Cqual**	**PREfix**	**ESP**	**ASTREE**	**SLAM**	**MC**	**Cyclone**	**Vault**
1	Uses Annotations	✔	✔	✔						✔	✔
2	Flow-Sensitive Analysis	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
3	Path-Sensitive Analysis				✔	✔		✔	✔		
4	Extensibility		✔	✔		✔		✔	✔		✔
5	Sound			✔		✔	✔	✔		✔	✔
6	Precision		✔	✔	✔	✔	✔	✔	✔	✔	✔
7	Inter-Procedural	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
8	Scalability	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
9	Predicate Discovery							✔	✔		

Table 3.2: Feature comparison of static checkers. A check mark indicates that the feature is supported, otherwise it is not.

Chapter 4

Automated Deduction of Checking Rules

The checking methodologies we have seen so far serve the purpose of automating the checking process, freeing humans of the arduous and boring task of checking software manually. These automated checking techniques have been successful at catching bugs that have missed the human eye. This represents a significant advancement in software checking: checking software automatically without human intervention, and doing it better. However, human beings are not entirely out of the picture yet, due to the huge semantic gap between the artifacts produced by developers and those accepted by automated checking tools. Most development is done with general-purpose programming languages (e.g., C, C++, Java), but most checking and verification tools accept specification languages designed for the simplicity of their semantics (e.g., state machines, process algebras). In order to use a verification tool on a real program, the developer must extract an abstract model of the program's salient properties and specify this model as a "checking rule" in the input language of the checking tool. This process is both error-prone and time-consuming. In this section we motivate the need for automated deduction of checking rules, and present a representative set of existing techniques for doing the same.

4.1 The Need for Automated Deduction

Most of the software produced today lacks comprehensive documentation. Documentation is needed for specifying checks or writing annotations for use by the static checkers. The current state of poor documentation in software design can be attributed to many factors. First, documentation is perceived by programmers as a completely unrewarding exercise because it is often boring, too verbose, and annoyingly difficult to get completely right. Secondly, there is sometimes a huge semantic gap between the code of a program and the higher-level system specification it represents. As a result, bridging the gap needs significant documentation effort in the form of detailed specifications of how the code relates to the higher level specification. Thirdly, as software evolves, developers join and leave and with them goes the ad-hoc implementation details about the system. Consequently, as code grows older, less is known about its internals to write checkers for it.

Good documentation habits, in our opinion, can be cultivated. However, the real issue is the cost-to-benefit ratio in documentation. If the documentation is really worth its effort, programmers can be convinced to do it. Unfortunately, current state of systems design does not make for such documentation. The end result of having an incomplete documentation is that the code has much more semantic content than the documentation, and hence documentation is not enough for writing checks for the code. Manually understanding the code for writing checkers is virtually intractable.

There are other reasons that call for automation in writing checks. Systems development is incremental by nature. Programmers often have to interface with or use legacy code that has little or no documentation in order to write checkers for them. Finally, modern systems may have just too many "checkable" properties than human beings can handle. Automation can achieve significant cost savings while giving much better results.

One might argue that a better way to address this problem is at a more fundamental level by generating code from the documentation, and getting rid of the whole problem once and for all. For instance, systems should be designed from scratch using a high-level, modular specification language that is easily amenable to software verification. In fact, this is indeed how things are being done with some application-level software. However, low-level system design under such a framework is not very well understood yet. Incremental refinement of existing code has been the holy grail of systems design, and asking systems designers to abandon all the existing code in favor of a new approach is asking too much. Hence, instead of adopting a revolutionary new approach to designing systems, one has to find clever ways to make current systems more amenable to automated checking and verification.

The central connecting thread of the techniques we present in this section is *reverse engineering*. Reverse engineering is the process of analyzing a subject system to identify the system's components and their interrelationships, and to create representations of the system in another form or at a higher level of abstraction. For example, producing control-flow graphs from source code, or generating interface specifications, would qualify as reverse engineering. The main goal of reverse engineering is to help with program comprehension. Techniques discussed in this section analyze program code or its runtime behavior to reverse-engineer *invariants* about the code. Code invariants are properties which should always remain true on the code and hence are a useful first step in automated generation of checking rules.

4.2 Invariant Inference using Predicate Abstraction

We introduced *abstract interpretation* in Section 2.2.3. Predicate abstraction [110] is a special form of abstract interpretation in which the abstract domain is constructed using a given set of predicates.

4.2.1 Inferring Loop Invariants

A long standing independent problem encountered in static checking of un-annotated or legacy code is automatic deduction of loop invariants. A loop invariant for a loop is a proposition composed of variables from the program that is true before the loop, during each iteration of the loop, and after the loop completes (if it completes). Knowledge of loop invariants considerably simplifies static checking of un-annotated or legacy programs. Loop invariants by definition do not depend on the number of loop iterations actually taken, and thus provide useful information for static checking of the program.

Loop invariant discovery is a non-trivial problem considering the enormity of possible invariants to consider. Blass and Gurevich proved [7] that as a consequence of Cook's completeness theorem, there exists a program whose loop invariants are undecidable. However, it seems that the case presented by them would be uncommon in practice.

Earlier attempts at automatic inference of loop invariants for sequential programs were based on heuristics [78, 129]. These were followed by algorithms based on iterative forward and backward traversal, with heuristics for terminating the iteration [111]. The *induction-iteration* method for example [94], works by finding the weakest liberal precondition for the loop. Although the algorithm may not always terminate, it works well in most cases. ESC/Modula-3 [43] experimented with generating loop invariants using the *widening* operator from abstract interpretation and a second technique called loop modification inference.

4.2.2 Predicate Abstraction

Flanagan et al. [55] have used predicate abstraction (Section 2.2.3) for automatically inferring loop invariants in C code. Their technique uses predicate abstraction over predicates generated in a heuristic manner from the program text. The programmer is free to provide additional predicates. Given a suitable set of predicates for a given loop, their algorithm infers loop invariants that are boolean combinations of these predicates. In effect, the problem of guessing loop invariants is reduced to the easier problem of guessing a relevant set of simple predicates.

One key novelty in their algorithm is in connection to dealing with the universal quantifier over loop iterators. For example, in the following code, `i` is universally quantified over all possible values from `0` to `n`.

```
for (int i = 0; i < n; i++) {
    array[i] = i;
}
```

Systems like SLAM have faced similar problems but do not handle this issue. Due to the universal quantification over `i`, the analyzer will have to guess all the possible quantifications of the expression as predicates, which is no easier than guessing the invariant itself. They avoid this limitation by resorting to *skolemization*: predicates can refer to skolem constants instead of using the universal quantification. Skolem constants are arbitrary unknown constants whose values may not be known during analysis, but they are known to be constant.

SLAM The SLAM project (Section 3.2.4) has also explored predicate abstraction for sequential programs. The boolean program generated from the C input by c2bp has boolean predicates over the values of system variables at different points in the program. SLAM's use of a counter-example–driven technique to refine predicates has the effect of inferring stronger invariants for the program.

4.3 Invariant Inference for Annotation Based Checkers: Houdini

Houdini [53] was motivated by the high burden of annotating static checking systems. It targets ESC/Java (Section 3.1.4), which has been reported as needing approximately one programmer hour of annotation effort for every 300 lines of code. In fact, this drawback has severally impeded mainstream adoption of ESC/Java despite its powerful bug finding capabilities.

Houdini adopts an iterative refutation refinement algorithm. To begin with, it guesses invariants for the program code using heuristics, and inserts appropriate annotations into the code. For example, for `int i,j` it adds the following annotations:

```
//@ invariant i cmp j;
//@ invariant i cmp 0;
where cmp is any of <, <=, =, >=, >, !=
```

Similarly, for the field `Object[] a`, it guesses:

```
//@ invariant a != null;
//@ a.length cmp i;
//@ invariant (forall int k; 0 <= k && k < a.length ==> a[k] != null);
```

The program thus annotated is run through the ESC/Java checker which identifies annotations that are not satisfied in the program code. Houdini automatically removes those annotations and re-runs the program through ESC/Java until there are no more warnings about annotation violations. If ESC/Java does not give any program correctness warnings, it means that Houdini has correctly figured out a minimal set of true annotations and it terminates. However, if the program still produces warnings, Houdini provides valuable information to the user to further debug.

Overall, Houdini's analysis is two-level. The inter-procedural analysis is derived from ESC/Java and is used for refuting and testing annotations. The intra-procedural analysis is similar to abstract interpretation based on powerset lattice.

4.4 Runtime Methods for Inferring Invariants

Invariants are valuable in many aspects of program development, including design, coding, verification, testing, optimization, and maintenance. They also enhance programmers' understanding of data structures, algorithms, and program operation. Unfortunately, explicit invariants are usually absent from programs, depriving programmers and automated tools of their benefits.

The Daikon tool by Ernst et al. [89] tries to infer invariants on variable values by observing program execution. All invariants are with this respect to program points, thus Daikon infers *possible invariants on every possible program point*. It works by instrumenting the program with tracing code and exercising it using a suite of test inputs. The trace is analyzed offline through various analyses to yield possible invariants on the values of variables at different program points. However, these "invariants" are not invariant in the true sense since they are guaranteed to hold only on that suite of test inputs and may not hold in general, hence this technique is not sound. However, knowledge of such invariants can be useful in the following scenarios:

- Understanding program execution for the purpose of designing static checks or test cases.
- Documenting legacy code by reverse-engineering them.
- Supporting program evolution: invariants discovered this way can be used for regression testing as the program evolves.
- Runtime checking of programs.

There are two issues in dynamic invariant inference. First, one has to choose which invariants to infer, and second, how to go about inferring them. Daikon's choice of which invariants to infer is guided by a library of inference patterns, it can only infer invariants which match this pattern. Invariants may only use scalar variables or sequences of scalar variables. Variables that do not fit this specification are converted into one of these forms (split an array of structs into individual arrays of struct fields). Daikon supports a wide but limited range of invariant patterns: for example, simple invariants over variable values, linear relationships over two variables including their sum, difference, etc., three variables and simple properties on sequence of variables (minimum, maximum, and membership).

The invariant inference algorithm employs an *early discard* scheme: an invariant is discarded once it is clear that it is not satisfiable. This allows Daikon to efficiently handle the large space of possible invariants. For example, in the invariant `ax + by + cz = 1`, constants `a`, `b`, and `c` can be determined from the first three trace observations. All later observations are checked against these constant values for satisfiability. If a violation is found, the invariant is discarded. In most cases it takes just a few program point observations to prune an unsatisfiable invariant. However, in the worst case, the cost of invariant inference is:

- linear in the number of samples,
- linear in the number of program points,
- cubic in the number of variables at a program point (because some invariants involve three variables), and
- linear in the number of invariants.

Invariant inference over 70 variables in a program with 10,000 calls whose entry and exit points were instrumented with monitoring code took only a few minutes per procedure.

Some program invariants may involve *meta*-values for variables or summary values, for example, `j < size(A) where A is an array`, or `min(A) < i` where `min(A)` is the minimum element of array `A`. Daikon includes these variables by monitoring values for "derived variables" alongside regular program variables. Thus derived variables are simply computable expressions on existing program variables.

Daikon performs several optimizations to speed up the inference process, early discard being just one of them. Some of the other optimizations are:

- Multi-stage inference. For example, `A[j]` is useful only if `j < size(A)`, so first infer `j < size(A)` and only if it is true, infer `A[j]`,

- Do not consider every program point, the default behavior is to consider only function entry and exit points, and loop headers.

- Exploit relationships between invariants to automatically infer one invariant from another.

Finally, it is reasonable to assume that the kind of analysis Daikon performs can result in a huge deluge of possible invariants, all of which may not be true. While Daikon does not give any guarantees regarding the truthness of any invariants, it is conceivable that the majority of discovered invariants may simply reflect properties of the test set and may not hold true in general. Spurious invariants in the output can significantly reduce the readability of the output, dwarfing the benefits from the tool. Daikon performs statistical analysis on variable values available from the traces in order to make intelligent predictions about the likelihood of the invariants being true for the general case. Specifically, it calculates the chances that a discovered invariant is true only by coincidence. This is derived by assuming a uniform distribution on the range of values observed in the traces and calculating the probability for a chance validity of the invariant. If the probability is less than a user-specified threshold, the invariant is passed as true. Note that the precision of this method can be improved by increasing the size and rigor of the test suite.

Daikon's running times and scalability are not very well understood, and, there are conflicting results of Daikon performance and scalability issues (few minutes to 220 minutes per procedure). Another fundamental question is: "how useful are the invariants generated by Daikon ?" It is not clear how useful the restricted forms of invariants inferred by Daikon can be. However, in Section 4.3, we have seen that even Houdini starts out with seemingly simple invariants and ends up generating useful bug-finding annotations for the program. This suggests that even basic invariants can serve for detecting symptoms of common program bugs. Moreover, one way to *automatically* utilize the large number of invariants generated by Daikon can be as input annotations for an annotation-based checker. Nimmer et al. [72] explore this approach in the context of Daikon and ESC/Java. They also suggest using Daikon output as Houdini input so that Houdini can identify and remove invariants that do not hold true in the general case.

However, the weakest aspect of Daikon remains the test suite; if the test suite does not exercise a sufficiently large number of code paths, ESC/Java cannot do much about it. Finally, the general impression is that invariant inference is probably useful only for static checking (and regression testing). So, to be useful, Daikon has to do a good job at that.

4.5 Bugs as Deviant Behavior: Combining Static Analysis and Invariant Inference

This is largely the work of Engler et al. [48]. They use *belief analysis* to automatically deduce erroneous behavior in source code. The key idea is that the programmer's intention is to produce correct code. However, they might make errors in a *few* places, and these errors show up as deviant behavior from the rest of the code. Thus the problem of finding bugs is reduced to the problem of identifying anomalous behavior in code. An interesting consequence of this approach is that even bugs that have not been formally specified by the programmer to the checker can be found. The end result is a fully automatic system that can potentially discover bugs in programs without any human intervention at all.

The system collects sets of programmer beliefs, which are then checked for contradictions. Beliefs are facts about the system, implied by the code. Two kinds of beliefs are considered: MUST beliefs and MAY beliefs. MUST beliefs are directly implied by the code, and there is no doubt that the programmer has that belief. A pointer dereference implies that a programmer must believe that the pointer is non-null. MAY beliefs are cases where the code has certain patterns that suggest a belief but may instead be just a coincidence. For example, a call to lock followed by a call to unlock implies that the programmer may believe they must be paired, but it could be a coincidence.

Once the set of beliefs are identified, the algorithm handles MUST and MAY beliefs differently. For MUST beliefs, it looks for contradictions. Any contradiction implies the existence of an error in the code. For MAY beliefs, the algorithm tries to separate true beliefs from coincidences. This is done using statistical analysis over the number of program locations where the belief is satisfied and the locations where it is violated. MAY beliefs with high satisfaction ratio are regarded as true beliefs and any violations are reported as errors. The errors are ranked on the basis of the confidence in the corresponding MAY belief, and lower errors may be discarded. Since this method is imprecise and may contain false alarms, statistical ranking of errors provides a trade-off between the effort expended in analyzing the error traces and the number of bugs discovered.

The static analysis engine used in this system is based on Metal and xgcc discussed in Section 3.2.5. The search for MAY beliefs is guided by specification of generic rule *templates*: for example, a rule present by default is: <a> must be paired with <b>, where <a> and <b> will be dynamically matched by the analyzer. Some useful may belief patterns are: checking that the return value of a function is always checked to ensure success or handle a failure condition, lock <l> is always held when accessing variable <v>, etc.

One of the key novelties of this work is the successful use of statistical analysis for ranking errors. Errors that are more likely to be real errors are ranked higher. However, statistical analysis is still fuzzy, and as a result coincidences can cause false positives, or imperfect analysis can cause low-ranked real bugs.

Intuitively, this method exploits the "redundancy of correctness" and "rarity of bugs" feature of software to first define buggy behavior and then to report it. Correctness rules of the system do not have to be known before hand; logical inconsistencies and programmer behavior inconsistencies usually indicate errors. Once again, this method draws from the paradigm that program errors directly map to detectable code patterns.

4.6 Extracting Finite State Models from Code

This section discusses a related field, that of automatic extraction of models from code for the purpose of formal verification. We discuss the work by Engler et al. [88]. A related system that we do not discuss in this paper is Bandera [36].

4.6.1 Metal Based System for Flash

Engler et al. [88] used their Metal-based system to automatically extract model descriptions from low level software implementations. Their system has found previously unknown bugs in the implementation of cache coherence protocols for the FLASH multiprocessor [81].

Their method is implemented using an extensible compiler, *xg*++[31, 47], and its accompanying extension language *metal*. The goal is to statically analyze program code and translate it into a specification that can serve as input for the Murphy model checker. Metal, with its extensible static analysis, offers an excellent system to use for this purpose. The user performs the following steps for extracting models from code:

1. Write a *metal slicer extension* for *xg*++. The slicer extension identifies variables and functions of interest in connection with the properties being checked. The Metal slicer strips of code that does not involve these elements. The *xg*++based slicer computes a *backward slice* at the level of statements with a PDG like algorithm [71].

2. Write a *metal printer extension* for *xg*++. This extension controls the translation phase. It instructs to *xg*++'s translator how the remaining constructs in the sliced program should be translated into Murphy input.

3. Manually create a model for the environment, correctness properties, and initial state.

4. Automatically combine the extracted model with the manually specified components.

The model is now ready for checking with Murphy.

By using the Metal slicer which is based on well-known powerful algorithms, their system can utilize a rich set of properties for finding and extracting the desired models. However, it seems that detailed implementation knowledge is needed in writing the metal printer extension. Also, the effort will scale super-linearly with code size (the reported FLASH code was only 10k lines). Also, their system does not entirely rule out manual effort. However, most of the manual effort is only during the first time the model is extracted or during major rewrites.

This system is a progressive step because it has modeled and found bugs in real world software. It is more of a model extraction tool than an invariant extraction.

Chapter 5

Case Study: Memory Bounds Checking

In this section we study existing techniques for checking programs for memory errors. We first present a discussion on the nature of memory bugs followed by a survey of existing static and run-time techniques for detecting memory bugs. We then present our system for run-time checking of memory errors in the Linux kernel.

5.1 The Nature of Memory Bugs

Memory bugs are probably the most common programming bugs that are encountered by programmers on a daily basis. Still, they are also the most insidious and difficult to detect bugs. Incentives for detecting and fixing memory bugs are many. Memory bugs are found in every scale of computing, be it a desktop computer or high end server. The end result is equally severe: data loss, program crash, security breach, etc. Well-tested systems have often been shown to contain serious vulnerabilities involving a single buffer-overrun that allows an attacker to execute arbitrary code on a machine, thus gaining control over it. Such bugs have already caused much grief in the IT industry.

Memory bugs in C programs can be divided into *safety* violations and *liveness* violations. Intuitively, a safety violation is defined as a condition in which a pointer expression accesses a different object from the one intended by the programmer. The intended object is called the *referent* of the pointer. Memory errors are classified into *spatial errors* and *temporal errors*. A spatial error occurs when dereferencing a pointer that is outside the bounds of its referent. Examples include array bounds errors, dereferencing un-initialized or `NULL` pointers, and even pointers derived from illegal pointer arithmetic. A temporal error occurs when dereferencing a pointer whose referent no longer exists (i.e., it has been freed previously), or its referent has not been created yet (e.g., uninitialized pointers). Liveness violations as the name implies relate to memory leak or garbage collection bugs.

Combating memory bugs is a multi-faceted research problem. First, the class of "memory bugs" as we discuss in this section is very broad: it includes `NULL` pointer dereferences, bound overflows, dangling pointer dereferences, memory leaks, etc. Secondly, memory bugs can be the result of widely different source code actions, hence a single unified framework is not easy to conceive. For these reasons, researchers have actively embraced a wide range of static and run-time checking techniques for detecting memory bugs. In fact, memory errors are one class of errors for which we advocate using run-time techniques in this paper.

The notoriety of memory bugs in software is further demonstrated using concepts from software testing. Sometimes, a memory error will not cause the program to crash, but will silently corrupt its internal state. Since the program has not crashed and seems to be working fine, this may be taken as normal behavior. This kind of behavior has been studied before using *sensitivity analysis* introduced by Voas et al. [128]. Sensitivity analysis estimates the probability that a program location may be hiding a failure-causing fault.

Memory bugs can potentially display a wide range of sensitivity. In fact, due to non-determinism, the same bugs can display different sensitivity on different systems, or even between different runs on the same system. Particularly insidious are low sensitivity cases in which a failure-causing fault may go undetected (an event known as *infection*). Here, failure is defined as any deviant behavior from the intended behavior. Thereafter, the fault may *propagate* to other parts of the program, corrupting the entire system state.

5.2 Static Methods for Detecting Memory Bugs

In this section we present a brief overview of static techniques for detecting memory bugs. We have already discussed static checking for software in Section 3. In this section we delve deeper into implementation aspects of static checkers for detecting memory errors.

Most static bounds checkers assume that the input data can assume any possible value and symbolic execution is used to check if array accesses are within the proper array bounds. Sizes of memory buffers are aggressively inferred statically: for example, using array size or arguments to `malloc`. We have seen PREfix in Section 3.2.1. The Archer system discussed below also employs the same methodology. Coen-Porisini et al. [34] give a good overview of the approach for a subset of the C-programming language. Evans and Larochelle [39], and Rugina and Regard [107] have used similar techniques for finding buffer overflow bugs statically. BOON [40] represents arrays or strings as ranges and the problem of detecting buffer overflows is transformed into a system of integer constraints, which can then be solved for detecting overflows. Ganapathy et al. [60] add flow and context sensitive analysis to a BOON like framework producing a high coverage static bounds checker.

Sound techniques for bounds checking either modify the language, or take help from runtime checking for operations that cannot be verified statically. Cyclone (Section 3.3.1) is an example of a new language. Cyclone infers buffer lengths statically using fat pointers that store the pointer size. Cyclone adds runtime checking to detect bound overflows. CCured is an example of the second approach. We discuss CCured in detail in Section 6.1.1.

CSSV [44], like Splint is an annotation driven bounds checker. CSSV aims to catch all buffer overflow vulnerabilities with minimum false alarms. The basic idea is to convert a C program into an integer program with correctness assertions included, and use a conservative static analysis algorithm to detect faulty integer manipulations, which directly translate to bugs in the C source code. The analysis is intra-procedural, however, by using procedure annotations ("contracts"), it becomes inter-procedural. The number of false alarms generated by the tool depends on the accuracy of the contracts. The analysis used by CSSV to check the correctness of integer manipulations was heavyweight, and may scale poorly to large programs. For instance, CSSV took > 200 seconds to analyze a string manipulation program with a total of about 400 lines of code. Splint on the other hand, sacrifices soundness and completeness, and uses a light-weight static analysis to detect bugs in source code.

5.2.1 Archer

Archer [131] is a memory error developed by Dawson Engler's group at Stanford University. Archer does not target the broad range of memory errors caught by PREfix (Section 3.2.1), like `NULL` pointer dereference, access to uninitialized or freed memory, etc. Instead, it focuses only on bound overflow errors. This is probably because the authors had already developed metacompilation-based tools to detect other kinds of memory errors [47, 66]. By concentrating on bound overflow errors, Archer is able to provide a more powerful and more precise bounds overflow detector than PREfix.

Archer's analysis has two key components: a *traversal module* and a *solver module*. The traversal module performs a randomized, bottom-up traversal of the system wide call graph to exhaustively explore all possible legal program execution paths (in practice, some paths will not be taken, and some impossible paths will be taken). The solver module keeps track of the set of constraints generated by the traversal module and solves them to check for satisfiability. The goal is to build up a knowledge base (stored in the *solver state*) that tracks values of and relations between as many scalars, arrays, and pointers as possible, which is then used to detect errors and prune infeasible paths. Initially this state is empty. As each statement is handled in order, Archer (1) *symbolically* evaluates the statement using the current state and (2) records any side-effects of the statement, producing a new state. The solver's role can be seen as keeping track of *assumptions* and check safety properties that the traversal module specifies in the form of *assertions* to the solver.

Loop Handling Archer tries to evaluate the termination condition of loops using the solver module. If the termination condition is known and the solver can evaluate it (symbolically), the loop body is iteratively evaluated until the termination condition is satisfied. If the termination condition is unknown or cannot be evaluated, Archer applies two heuristics to handle these cases. For "iterator loops" used to iterate through buffer accesses, even if the termination condition cannot be evaluated, the (symbolic) bounds of the iteration variable can often be used for analysis. For example in the following piece of code,

```
for (i = start; i < end; i++) {
  ...  /* loop body does not modify i */
}
```

the iteration variable `i` is replaced by the range `[start, end)`. For all other termination conditions that cannot be evaluated by the solver, Archer evaluates the loop exactly once checking for any errors, and after exiting from the loop, it undoes all assignments performed in the loop.

Error Triggers Bound overflow errors can either involve local memory objects whose sizes can be deduced intra-procedurally, or pointers passed as arguments to the function (in which case, the size cannot be ascertained unless the caller is analyzed). In the former case, the analysis may be able to detect an error and flag it. However, in the latter case, the presence of an error will depend on the actual arguments passed to this function. For these cases, Archer summarizes any possible error-causing arguments to a function in the form of *triggers*. Later on when analyzing calls to this function, if the prevailing condition at a call-site reaches any of the triggers for this function, an error is flagged. Consider the following contrived example:

```
char arr[10];

void fn() {
  gn(20);
}

void gn(int n) {
  arr[n] = 'x';
}
```

Archer will analyze the function `gn` first (bottom-up order). The solver infers that the array dereference `arr[n]` can result in a bounds overflow depending on the value of `n`. However, it will not know for sure until all possible callers of `gn` are analyzed. Hence it creates the error trigger: $n >= 10$ where 10 is the size of the array `arr`. During the analysis of function `fn`, the predicates prevailing at the call to `gn` satisfy this trigger $(20 >= 10)$ and hence Archer will flag an error. Note that by using summaries in the form of error-triggers, Archer is able to implement highly scalable inter-procedural path-sensitive analysis.

Symbolic Execution of Statements Assignment statements (`(x = y)`) are handled by first evaluating `y` against the current solver state and binding the result to `x`. For control flow edges, the traversal module uses the solver to evaluate the condition; if the condition can be evaluated, the appropriate path is taken. If not, then all possible paths are taken, and the condition's truth value is added to the solver state along each path. For function calls, first the triggers are evaluated, if no triggers are violated, any global state passed by reference to the callee is evaluated and the analysis resumes after the function call. Finally, for memory accesses, the solver is queried to see if the memory access can potentially go out of bounds. If so, an error is emitted, otherwise, Archer produces no errors.

Size Inference using Statistical Belief Analysis There are two cases which traditional inter-procedural analysis cannot handle: (1) when the callee's code is not available, and (2) when the callee's code is too complicated to make inferences about its effect on the caller's state. Archer uses idea from statistical belief analysis [48] (Section 4.5) to solve this problem. Rather than analyzing a routine's *implementation* to determine how it acts, we instead analyze the routine's *calling contexts* to do so.

Archer's analysis is quite powerful due to its path-sensitive and inter-procedural nature. It seems to be more powerful than PREfix for the case of detecting bound overflows, since it tracks properties such as buffer lengths and pointer offsets with a symbolic solver, and can potentially infer interfaces of unavailable functions. Archer has discovered previously unknown bugs in several open source software packages [131].

5.3 Runtime Checking for Detecting Memory Bugs

Operating systems that run in protected mode already provide a crude form of checking for memory errors. Such operating systems, for security and reliability reasons, separate processes from each other. So if one process accidentally or deliberately tries to read or write the memory of another process, the first process is prevented from doing this. When this happens, Unix-based systems give the message.

```
Segmentation fault
```

and the process is killed. Other operating systems like Windows NT for instance, give more information. In a sense, then, there is a primitive form of checking going on. However, this default support is insufficient in two ways:

1. *error detection:* it provides very coarse grained checking. For example, memory is not protected at the granularity of memory objects.

2. *error recovery:* it does not provide a graceful method to recover from errors at runtime, nor does it provide sufficient information to identify the origin of a memory bug that causes the program to crash.

These factors motivate the need for more fine-grained runtime checking for memory errors. The general problem of runtime bounds checking requires comparing the target address for each memory reference against the bounds of its referent object. Accordingly, bounds checking comprises of two subproblems: (1) identifying a given memory reference's referent object and thus its bounds (*bound lookup problem*), and (2) comparing the reference's address with the referent's bounds and raising an exception if the bound is violated (*bound comparison problem*). For the purpose of this discussion, we classify existing bounds checking techniques into two categories based on their approach to the bound lookup problem: (1) *capability*-based techniques, and (2) *availability*-based techniques. In capability checking, each pointer object is augmented with a capability. The capability specifies among other things, the exact memory bounds of the object referenced by the pointer. Before any pointer dereference, a check is performed on the pointer's capability to verify that the pointer is within the bounds, or in other words, is "capable" of dereferencing the address.

Thus, illegal pointer dereferences are detected. Also, language semantics are modified to conservatively preserve capability information across pointer operations like assignments, arithmetic, etc. For example, in each of the following examples of pointer usage, pointer `p` always inherits the capabilities of `q` *unchanged*:

```
char *p, *q;
int i;

p = q;

p = q + i;
```

Similarly, in the case of `p++`, the capabilities of `p` remain unchanged even though the address value of `p` changes. This *immutability* of capability is the key idea behind capability-based checking: as pointers and addresses are assigned and changed during the execution of a program, the set of valid capabilities remain unchanged. Capabilities change only as a result of allocation and deallocation, and all addresses must have capabilities associated with them (arbitrary integer to pointer casts are not allowed).

Since capability checking entails augmenting pointer variables with extra capability information, it is not backward compatible with standard C code. Availability checking is a backward compatible form of run-time bounds checking. The idea is not to store any extra information along with the pointer, but to maintain a running map of currently allocated memory for the program (availability map). All pointer dereferences and pointer arithmetic can then be checked against this map: if it falls within the availability map, it is allowed; otherwise it is a bug. Table 5.1 shows a classification of runtime bounds checking techniques discussed in this section.

Capability based		**Availability based**	
Fat Pointers	*Shadow Variables*	*Track Arithmetic*	*Do Not Track Arithmetic*
Safe-C (Section 5.3.2), Cyclone (Section 3.3.1)	Patil and Fischer (Section 5.3.3), Xu et al. (Section 5.3.4, Lam et al. (Section 5.3.5)	Jones and Kelly's BCC (Section 5.3), KBCC (Section 5.4.1)	Mudflap (Section 5.4.2), Purify (Section 5.3.1)

Table 5.1: A two-level classification for runtime bounds-checking technique

There is an interesting soundness and completeness trade-off between capability checking and availability checking that we present below.

We are interested in two operations on pointers: dereferences and arithmetic. Any run-time bounds checking scheme must check pointer dereferences, since dereferencing an incorrect address can have severe consequences. However, it is not always necessary to track pointer arithmetic. A pointer arithmetic operation typically takes the address stored in a pointer, recomputes a new address, and assigns it back to the pointer. The vulnerability in this operation is that the newly assigned address could be out of bounds with respect to the object referred by the pointer before the operation. This is a bounds *overflow* in action. However, arithmetic that causes a bounds overflow does not crash the program yet. For the program to crash, the new address should be an illegal address, and it should get dereferenced. A pointer update can have two results: (1) the new pointer points to an unallocated address, and (2) the new pointer now points within another (possibly adjacent) object. With capability checking, whatever the new value, any dereferences will have to be verified against the pointer's capability (which does not change from arithmetic). With availability checking, if the new pointer points within another object, the checking system cannot detect an overflow since the pointer falls within the availability map. However, if pointer arithmetic operations were to be tracked and verified to catch the case when pointers *crossed* object bounds, this scenario could be avoided. Hence, this is a soundness-performance issue: checking pointer arithmetic is important for soundness, but it may also have a lot of overhead. Jones and Kelly's bounds checking patch to GCC [75] is an example of an availability-based checker that tracks pointer arithmetic, and Purify [106] is an example of an availability-based checker that does not track pointer arithmetic. However, this soundness guarantee holds only under the assumption that the program does not write to any arbitrary numerical address.

In Section 5.4.1 we will see that tracking pointer arithmetic in an availability-based checker can lead to loss of precision. In particular, the C definition does not restrict pointers from crossing object boundaries as long as they are not dereferenced outside. For a variety of reasons (mostly having to do with convenience), programmers often have C pointers crossing object boundaries. An availability checker that checks pointer arithmetic flags an error in such cases even though it may be legitimate usage. Hence, among availability-based checkers, whereas pointer arithmetic checking schemes are more sound than other schemes, they are also less precise.

Table 5.2 summarizes the difference between capability-based checking and availability-based checking.

Capability based	**Availability based**
May not be backward compatible with C code (e.g., Safe-C), unless the capabilities are stored separately (Section 5.3.3). Even then, they are incompatible with pre-compiled C programs (e.g., libraries).	Backward compatible since data representation or layout is not changed.
Faster since no bound lookups: bound information is available within in the variable or shadow variable.	Slower: bound lookup entails searching the availability map for the address.
Can detect memory-recycling errors.	Cannot detect memory-recycling errors.
Sound	Cannot be completely sound due to memory recycling errors. However, if pointer arithmetic is not tracked, it will be even less sound.
Memory overhead due to shadow variables or fat pointers.	Memory overhead due to object padding and availability map. Note that padding is only required if the checker does not track pointer arithmetic.
Can catch overflows between `struct` members.	Cannot catch

Table 5.2: Table showing the differences between capability-based and availability-based runtime bounds checking

BCC

Jones and Kelly's patch to GCC [25] (BCC) is an availability based backward-compatible runtime bounds checker for C programs. BCC has two components: (1) a compiler component based on GCC that compiles C programs and instruments them with special function calls which call (2) an external runtime library called `libcheck`. The executable thus compiled runs in the context of `libcheck`. At runtime `libcheck` maintains an availability map containing all memory allocated by the program, with help from these inserted calls. Some of the calls inform `libcheck` of memory allocation and deallocation events so that the availability map can always be kept up to date. Others are placed immediately before pointer operations, and call `libcheck` to verify the correctness of the operation. The operation can execute only if `libcheck` validates the operation. A more detailed description is presented below:

1. If runtime bounds-checking is desired for a C program, it should be compiled with the `-fbounds-checking` flag to BCC.

2. During compilation, BCC identifies three categories of operations in the C program and instruments them with calls to `libcheck` as follows:

   - All memory allocation operations, whether explicit (`malloc`, `free`, etc.) or implicit (function local variables), are followed by calls to `libcheck` detailing the location and the size of the memory thus allocated or deallocated.
   - All pointer dereferences are preceded by calls to `libcheck`, passing details of the pointer: location, size of the target, etc.
   - All pointer arithmetic expressions are preceded by calls `libcheck`, passing the pointer location and the details of the arithmetic.

3. The program thus instrumented is completely backward compatible with C code and can be linked with C code not compiled with runtime bounds checking support.

4. When the program is executed, it is checked for bounds errors by the inserted function calls with help from `libcheck`:

   - For memory allocation and de-allocation operations, `libcheck` updates its availability map.
   - For pointer dereferences, `libcheck` verifies if the pointer points to an address within the availability map. If it does, it returns and the program continues, if not, an error message is generated and the program terminates.
   - For pointer arithmetic operations, `libcheck` verifies if (1) the pointer points to a valid location in the availability map, and (2) if before and after the operation, the pointer references the *same* referent object.

5. In the event of an error being discovered by BCC, a verbose trace of the error is displayed, providing the exact location and nature of the error (`NULL` pointer dereference or invalid arithmetic). After displaying the error the program can either be terminated, or allowed to execute. The behavior is specified at program compile time. Also, if the program continues after an error, its behavior may be undefined.

BCC maintains per-object allocation information in an availability map which is searched during the above operations to locate the referent object corresponding to the address passed to `libcheck`. The availability map is organized as a *splay-tree*. Splay-trees are self-adjusting binary search trees first introduced by Sleator and Tarjan [118]. The search key in this case is the address of the referent object. Splay-trees have the useful property that recently accessed elements are quick to access again. All normal operations on a splay tree are combined with one basic operation, called *splaying*. Splaying the tree for a certain element rearranges the tree so that the element is placed at the root of the tree. It performs basic operations such as insertion, look-up and removal in `O(log(n))` amortized time. For many non-uniform sequences of operations, splay trees perform better than other search trees, even when the specific pattern of the sequence is unknown. The choice of a splay-tree for storing an availability map is justified by the temporal and spatial locality inherent in variable accesses in C programs.

BCC potentially has complete control on the compilation process and can insert runtime checks to check other operations too. In fact, BCC replaces calls to standard string library functions, like `strlen`, `strcpy`, etc. with calls to their bounds-safe counterparts in `libcheck`. This is because the standard library functions do not (and in most cases, cannot) perform safety checks on their arguments. `libcheck` ensures their safety by redirecting them to their safe counterparts which check the operation against the availability map.

BCC pads all memory allocations so that some unallocated memory is left before and after each allocated chunk. It seems that this padding is superfluous since its use is to detect overflows, which BCC can detect anyway by tracking pointer arithmetic. In fact, this padding can be harmful at times. For example, BCC pads all variables on the stack whose address is taken in the program. This frequently results in highly bloated, sparsely filled stack frames. This is clearly an undesirable feature for in-kernel code, since the kernel stack is typically very small (4–8K). Also, the padding can sometimes have undesirable side effects. For example, in our adaptation of BCC for the Linux kernel (Section 5.4.1) (KBCC), KBCC compiles a kernel module and loads it into the kernel using the `insmod` module loader tool in Linux. A key operation during the `insmod` process is passing the global data values from user space to kernel space. KBCC generates global variables padded on both sides, but the kernel is not aware of this and crashes when trying to parse the module.

We discuss other BCC in more detail in Section 5.4.1, when we discuss our adaptation of BCC for the Linux kernel.

5.3.1 Purify

Purify [106] is a commercial availability-based run-time bounds checker. It can check bound overflows, bad pointer dereferences, use of uninitialized data, and memory leaks. Purify instruments programs at object-code level. Hence, Purify is language independent, and it can instrument binary-only third-party software. Purify associates one of three possible states with every byte in the program's address space: *unallocated* (unwritable and unreadable), *allocated and initialized* (writable but unreadable), and *allocated and initialized* (writable and readable). Memory accesses are checked at the granularity of bytes and operations (read or write).

Unlike BCC, Purify stores availability information on a per-byte basis (possibly compressed into integer-ranges), instead of on a per-object basis. Hence, Purify cannot catch errors due to overlapping memory allocations whereas BCC can. Also unlike BCC, Purify does not check pointer arithmetic. However, it is still able to detect bounds overflows in some cases. Purify pads arrays and heap allocated memory with a small "red-zone." The memory in the red-zone is recorded as unallocated (unwritable and unreadable). If a program accesses these bytes, Purify signals an array bounds error. Note that this is not a fool-proof method, a bounds overflow that jumps over the red-zone to access outside memory can still go undetected. This is an example of a performance-soundness trade-off in availability-based checking: tracking pointer arithmetic makes the checking sound, but at the cost of increased performance overhead. Checkers like Purify and Mudflap (Section 5.4.2) trade-off soundness and memory (for the red-zone) for performance. This is no surprise since both these checkers are targeted for mainstream use: Purify is a commercial tool for runtime bounds checking, and Mudflap is available by default in GCC 4.0 (no longer a patch like BCC).

5.3.2 Safe-C

Safe-C [13] is an example of a capability-based runtime bounds checker. It modifies the internal representation of pointers at compile-time to support runtime checking. Safe-C uses *fat pointers* to store additional information about the referent object. Such a fat pointer is represented by a four-tuple comprising of the following attributes:

- the value of the safe pointer,
- base address and size of the referent object,
- storage class of the referent object (stack/heap/global),
- a *capability* to the referent (explained below).

Thus the fat pointer logically resembles the following template specification:

```
typedef {
        <type> *value;
        <type> *base;
        unsigned size;
        enum {Heap=0, Local, Global} storageClass;
        int capability;
} SafePointer<type>
```

The value attribute is the only attribute that can be manipulated by the program; the rest are used by the Safe-C runtime. A capability is a unique identifier attached to a referent. As objects are allocated and deallocated at runtime (due to function calls, heap allocation, etc.), a new capability value is assigned to each new object from a monotonically increasing program-wide counter. This capability value is maintained in a *global capability store* for later verification. Safe-C ensures that pointer attributes are preserved across pointer arithmetic and assignments, thus all pointer dereferences in the program can be checked by checking the corresponding attributes. Note that Safe-C does not change array references. Array references by nature are static and hence their bounds are known statically.

Assuming the safe pointer attributes are correct, complete safety for all pointer and array accesses is provided by inserting an access check before each pointer or array dereference. The dereference check first verifies that the referent is alive by performing an associative search for the referent's capability. If the referent has been freed, the capability would no longer exist in the capability store and the check would fail. Because capabilities are never re-used, the temporal check fails even if the storage has been reallocated. Once the storage is known to be alive, a bounds check is applied to verify that the entire extent of the access fits into the referent.

Pointer operations must interact properly with the composite safe pointer structure. When applied, they must reach into the safe pointer to access the pointer value and pass the rest of the attributes along as appropriate. For example, the assignment statement `q = p + 6` causes the pointer attributes of `p` to be passed along to `q`. In fact, operations which manipulate pointer values never modify the copied object attributes because changing the value of the pointer does not change the attributes of the storage it references. However, certain cases need special handling. For example, if the right hand side of an assignment is `NULL`, the pointer's capability is set to a special value `NEVER` indicating an invalid pointer that should not be dereferenced. Similarly, for assignments from string constants the pointer attributes can be generated at compile time itself.

The technique proposed in Safe-C can be applied both at compile time, or using a source-to-source translator. Safe-C can detect all memory access errors provided (1) storage management is always apparent to the translator, and (2) the program must not manipulate any object attributes. However, a major drawback of Safe-C is that due to its use of fat pointers, it is incompatible with a majority of legacy C code. As a result, it is either incompatible with existing libraries, or breaks existing programs due to its use of fat pointers.

5.3.3 Decoupling Computation and Bounds Checking

Patil and Fischer [98] propose to get rid of fat pointers as used in Safe-C by separating pointer attributes from the actual pointers. Corresponding to each pointer `p`, a *shadow variable* `G_p` (they call it the "guarded pointer") stores its attributes. Their approach suggests correlating accesses to `p` with checks on `G_p`. In one respect, the scheme proposed by Patil and Fischer represents a significant departure from conventional runtime bounds checking. They make the following observations about the state of runtime bounds checking:

- Performance overhead of traditional runtime bounds checking systems is due to in-line checks for memory access errors. If checking could somehow be decoupled from the main computation, performance overhead can be reduced.

- Performance overhead is a necessary evil of bounds-checking. Given these overheads, the possibility of using runtime bounds checking in production systems seems unlikely. Hence, when bounds checking a system, its other computations do not matter.

Based on these premises, they argue for decoupling a program's computation from bounds checking using one of two methods:

- **Alternative 1:** Strip down a program into a simplified version (akin to slicing) that has only the memory operations and any other operations that can affect these memory operations. This simplified program is called the *guarded program*. If no memory access errors are detected in the guarded program, there should be none in the original program, and vice versa.

- **Alternative 2:** Execute the original program and the guarded program *in parallel*. The original program executes normally, however, the guarded program executes on the same input as the original program, but executes only memory access code and checking. This scheme is called *shadow-checking*. Any memory access errors in the original program will be detected and reported by the shadow program which simulates an identical memory state.

It is not very clear how generally applicable the shadow-checking scheme is. For example, for complete equivalence between the main program and the shadow program, it is required that they execute in exactly the same environments. However, sometimes it may not possible to reproduce *exactly* the same environments for the main and the shadow process (e.g., system call return status). Although their method does employ heuristics for handling such situations, it does not seem to be general enough in its current form.

Note that the assumptions made by Patil and Fischer regarding the unavoidable overhead in runtime bounds checking are not entirely true today. However, this was a plausible assumption at the time this tool was developed (1997). Recent techniques have achieved success in reducing bounds checking overheads to acceptable limits: for example, CCured (Section 6.1.1) addresses the bound comparison problem by removing redundant bound comparisons using static analysis, and Cash (Section 5.3.5) addresses the bound comparison problem by using hardware support to perform fast comparisons.

5.3.4 Backward-Compatible Extensions to Safe-C

Xu et al. [132] use ideas from the above technique to derive a fast, backward-compatible source-to-source translation based on Safe-C. For each pointer `p`, a new shadow variable `p_info` (they call it the "metadata variable") is defined in the output C source. `p_info` stores the attributes associated with the pointer `p`. The shadow variable in this case is a C `struct` that stores the base and size of the referent object (like Safe-C), and a capability. However, Xu's notion of capability is different from Safe-C, and this is one of the key novelties in this approach. Whereas Safe-C associates a capability with each pointer variable, this method associates capabilities with memory objects. A field in the shadow variable points to the capability of the referent object. In this way, their method introduces a level of indirection between referent objects and pointers. Multiple pointers that refer to the same referent object now share the same capability. This is a very effective way to handling *aliasing*: any changes to the state of the memory object through one alias (e.g., a deallocation operation) is immediately available to other aliases through the shared capability. However, their method is still not completely compatible with existing C code, especially when they pass pointers to external library interfaces.

5.3.5 Cash: Hardware Assisted Bounds Checking

Segmentation hardware has been used to implement protection domains starting from the Multics operating system [6].

Electric Fence [102] uses a system's virtual memory hardware to place an inaccessible memory page immediately after (or, at the user's option, before) each memory area allocated using `malloc`. It also aligns the allocated memory area to the end of page boundaries. When software reads to or writes from this page, the hardware issues a segmentation fault, stopping the program at the offending instruction. It is then easy to find the statement in the C source using a debugger. Memory that has been released by `free` is made similarly inaccessible. Although this method has zero overhead to perform array bound checking, depending on the allocation pattern, it can waste a lot of virtual address space available to the process. Also, its checking ability is limited to only checking dynamically allocated buffers.

In recent times, the X86 segmentation hardware has been used in novel ways to support software evolution. For example, the Palladium system uses segmentation [29] for implementing safe and efficient software extensions, and the Cash bounds checker [82] uses segmentation to solve the bound comparison problem thus reducing bounds checking overhead. Pearson has also used segmentation hardware feature to get bounds comparison for free [99], although only for the case of dynamically allocated memory buffers.

The Cash bounds checker is a capability-based checker that uses shadow pointers for storing bounds information associated with pointer variables. Each pointer variable P is augmented with another pointer variable P_A which stores:

- Lower bound of P
- Upper bound of P
- the LDT index (described below) associated with the segment allocated to the object.

The pointer variable P_A is stored immediately after P, and is thus trivially available from the knowledge of P's location.

In the Intel X86 architecture, a virtual address consists of a 16-bit *segment selector* value, which resides in one of the six on-chip segment registers, and a 32-bit *offset* which is given by general purpose registers (GPR), EIP, or ESP depending on usage. The segment selector contains a 13-bit index into the *Global Descriptor Table* (GDT) or the current process's *Local Descriptor Table* (LDT). The choice between GDT and LDT is determined by one of the remaining three bits in the segment selector. The number of entries in each LDT and the GDT is 8,192 (only 8,191 are available for use). The GDT or LDT entry indexed by the segment selector contains a *segment descriptor*, which includes the base and limit addresses of the segment, and read and write protection bits (and some other fields). Combining the 32-bit offset with the segment's base address produces the requisite linear address.

The basic idea behind Cash is that every time an array is used, Cash allocates a segment register for the array. The segment register is loaded with the appropriate LDT entry indexed by the array's segment selector (stored in the shadow variable). Thus all accesses to the array are automatically checked. Since the number of segment registers is limited (three for general use, but authors have successfully used four in some cases), the available segments are mapped to buffers on-demand. Note that whenever an array or buffer is allocated in Cash, either due to a function call, or `malloc`, a segment is allocated for it and entered into the LDT. Similarly, the segment is freed on de-allocation. In case there are more live buffers at a program point than segments in an LDT (8,191), Cash checks only 8,191 buffers and performs software bounds checking on the excess buffers. One alternative to this approach is to use multiple LDTs and dynamically switch between them. The authors argue that such an implementation could have resulted in two many LDT switches (thrashing).

The Cash framework tackles some serious hurdles that otherwise prevent it from achieving good performance. A useful optimization is with respect to inserting new LDT entries into the LDT table when a new array is allocated. Linux provides a system call for doing this (`modify_ldt`). However, a system call imposes unacceptable overhead for such a frequent operation. Cash's solution is to provide an alternative call gate to the user process, allowing the user to execute a Cash-specific kernel function which modifies the LDT. The advantage of this approach is that it has less overhead than a system call (it backs up and restores only the `EDX` and `DS` registers).

Cash has two major limitations. First, it checks only code within loops. This was done given the high overhead associated with setting up segments for function local arrays. In the case of loop variables, this cost pays itself off over many iterations. Furthermore, the authors argue that most buffer overflow attacks are loop-based. A second weakness of Cash is its incompatibility with pre-compiled C code, like binary libraries. This is due to the shadow pointer variable that needs to be passed along in function calls.

Cash has impressive performance compared to other bounds checkers. Whereas other schemes like Purify and Jones and Kelly's approach have been shown to be 5 to 20 times slower, the overhead of Cash is within 50% for most benchmarks, and only marginally larger for others.

5.4 Bounds Checking for the Linux kernel

Programmers make mistakes, and C semantics leave ample opportunity for introducing memory errors. This problem becomes all the more acute when programming inside the kernel as a small memory-access bug could crash the entire system. Runtime bounds checking through hardware is an efficient method of detecting program bugs. We have developed such a system called *Kefence* (Section 5.4.3. KBCC, which is a software based approach provides more comprehensive checking.

5.4.1 KBCC

KBCC is an availability-based runtime bounds checker for the Linux kernel. It can detect many kinds of memory access errors like bound overflows, `NULL` pointer dereferences, bad arithmetic, use after free, double free, etc. The goal of KBCC is to perform runtime bounds checking on the entire Linux kernel. KBCC is an extension to BCC (Section 5.3 that is specifically targeted towards the Linux kernel. It is based on the GNU C compiler (GCC). KBCC was derived from Jones and Kelly's bounds checking patch to GCC [75] (BCC). However, KBCC extends BCC in several ways. Because KBCC is based on GCC, it gets GCC's excellent support for a wide range software for free. Also, KBCC leverages GCC's optimization and analysis features for help with bounds checking. KBCC's development has coincided with GCC development. We have kept KBCC uptodate with the latest GCC version until 4.0. The implementation we describe here was derived from GCC-3.4.2.

The development of KBCC can be described as a porting effort to port BCC for using with kernel code. There are two components in BCC: the compiler that instruments compiled code, and the runtime bounds-checking library `libcheck`; both needed different kinds of attention. For example, we taught the KBCC compiler to recognize `kmalloc` as the memory allocator, instead of `malloc`. Similarly, `libcheck` was packaged into a kernel module. Sending `libcheck` to kernel space was necessary since the checking library should run in the same address space as the checked program to minimize communication overhead. During this whole exercise, several interesting issues came up which offer useful insight into runtime bounds-checking systems, and are the subject of this section.

Soundness KBCC is an availability based checker and is not completely sound. In Section 5.3 we saw that BCC inserts calls to the BCC runtime library (`libcheck`) before each memory access. The call serves the purpose of verifying the address value for validity with help from `libcheck`. `libcheck` maintains an availability map of all currently allocated memory regions (the "availability"), against which the address passed by the checking call is verified. If the address is found, the memory access must be valid and `libcheck` returns control to the program, else it signals an error. Now consider the following piece of code:

```
1.  char *p, *q;
2.  p = malloc(...);
3.  free(p);
4.  q = malloc(...);
5.  *p = 'x';
```

A chunk of memory is allocated using `malloc` and assigned to pointer `p`. In line 3, the memory is deallocated. Line 4 calls `malloc` again to allocate another chunk of memory. The dereference of pointer `p` in line 5 is illegal since the address corresponding to `p` has been de-allocated. However, it is possible that the same address was allocated to `q` in line 4. In this case, statement 5 is still illegal since `p` is a stale pointer from a previous incarnation of the memory location. But `libcheck` will not flag an error. `libcheck` is an availability-based checker that simply searches for the address value of `p` in its address map. In this case it will be found and `libcheck` incorrectly assumes that there is no error. Hence, KBCC is unsound and cannot catch *memory recycling* errors. Note that this is a problem with all availability based checkers.

Incompleteness KBCC is incomplete, because it can report bugs that are actually not real bugs in the checked program. KBCC's incompleteness comes from its strict checking of pointer arithmetic operations. Languages with mutable pointer operations like C allow the program to legally create invalid pointers; for example, iterating a pointer across all the elements of an array exits the loop with the pointer pointing to the memory location following the last object. If the invalid pointer is never dereferenced, the program would not be in error. However, KBCC recognizes this operation as an illegal pointer arithmetic, since the pointer has gone beyond its legal bounds. In such scenarios it can generate false alarms. This might suggest that KBCC not flag an error during possibly incorrect pointer arithmetic, since it does not have enough information about whether the produced pointer will be dereferenced later on in the code. However, such behavior can result in more unsoundness for the following reason. Imagine that two pointers `p` and `q` point to memory areas next to each other. A bad arithmetic on `p` can cause it to *overflow* out of its bounds and point to within `q`'s bounds. Since KBCC's bounds checking is availability based, it will not be able to detect such a scenario *if* it does not track pointer arithmetic operations. Thus, there is a trade-off in KBCC's design involving precision of checking, performance, and soundness. Systems code usually has a plethora of pointer arithmetic operations; tracking them all means poor performance. Thus tracking pointer arithmetic gives soundness at the cost of performance and precision of checking. In KBCC, we decided to track pointer arithmetic operations, since there is a large loss in soundness otherwise. Kernel objects are usually tightly packed next to each other, hence overflows are more likely to fall within adjacent object's territory than in unallocated space. An availability checker cannot detect this and will thus silently miss errors. One might argue that KBCC cannot be sound anyway from the memory recycling example above, and hence it makes little sense to sacrifice performance trying to be sound. However, the loss of soundness in the case of undiscovered overflows is much more serious than the rare case of memory recycling errors.

BCC was designed to be completely backward compatible with standard C code. In fact, one should be able to freely mix code compiled with BCC and code compiled with any other GCC-like compilers. However, in reality BCC fails to compile a majority of open-source C programs [109]. We have extensively experimented with one particular case, the Linux kernel and found it to fail frequently. There were two main reasons for this: (1) several previously unknown bugs in BCC which we fixed, and (2) BCC could not handle several "advanced" C programming constructs. We give examples of each of these below.

BCC Bugs: An Example BCC 3.2 and 3.3 could not inline function calls declared as `inline`. Although most C programs would still work, the Linux kernel refused to compile without inlining. The reason was traced to the `extern inline` directive of GCC. This directive provides easy support for users to write plugins. If a function `f` is declared as `extern inline` in a header file `header.h` included from the C source file `source.c`, then GCC will try to locate another function definition for `f` in `source.c`. If found, the new definition replaces the definition in `header.h`. Otherwise, the compiler inlines all calls to `f` using the function body provided in `header.h`. Since BCC could not inline functions, this step could not be performed, resulting in a compile time error. We tracked this problem to a bug in the BCC source code, which resulted in patches to BCC.

Advanced C Constructs Consider the following piece of code from the Linux kernel.

```
/* from linux-2.6.7/include/linux/kernel.h */
#define container_of(ptr, type, member) ({                      \
        const typeof( ((type *)0)->member ) *__mptr  = (ptr);   \
        (type *)( (char *)__mptr - offsetof(type,member) );})

/* from linux-2.6.7/include/linux/list.h */
#define list_entry(ptr, type, member) \
        container_of(ptr, type, member)

#define list_for_each_entry(pos, head, member)                          \
        for (pos = list_entry((head)->next, typeof(*pos), member);      \
             &pos->member != (head);                                    \
             pos = list_entry(pos->member.next, typeof(*pos), member))

/* from linux-2.6.7/include/linux/list.h */
struct list_head {
        struct list_head *next, *prev;
};

/* from linux-2.6.7/include/linux/fs.h */
struct super_block {
        struct list_head       s_files;   /* List of open files */
      ...

/* from linux-2.6.7/fs/super.c */
static void mark_files_ro( struct super_block *sb)
{
        struct file *f;

list_for_each_entry(f, &sb->s_files, f_list) {
  ...
}
```

`list_head` is a frequently used data structure in the Linux kernel to implement doubly linked lists of `inode` or `dentry` objects. Each member of a list has a `list_head` object embedded within it. The `list_head` has pointers to the `list_heads` of the predecessor and successor objects in the list. Note that the `list_head` pointers make a list of `list_heads` and not of the containing object in which they are embedded. However, the containing object can be derived by simply subtracting from the address of the embedded `list_head` its offset from the start of the containing object.

Consider the function `mark_files_ro` shown above. It calls the macro `list_for_each_entry` with `sb->s_files` as the `head`. This is a common usage pattern of `list_head` structs: one special `list_head` serves as the "head" of the list. The head is special because unlike other `list_heads` that make up the list of `files`, the head does not lie within a `file` object. The purpose of having this special head is to be a placeholder through which the list can be traversed. In this case, the kernel can traverse all `files` associated with a `super_block`.

Thus, the function `mark_files_ro` iterates over the entire list using the macro `list_for_each_entry` which is implemented using a `for` loop. Now consider the termination condition for the `for` loop: $\&pos- > member! = head$. It will be satisfied when `pos` has made one full circle of the list and has returned to the head. Note that when the termination condition is satisfied, `pos->member` is pointing to `head`. `pos` is of type `file` and is generated using the `list_entry` macro from a `list_head` in the list. `list_entry` as we have seen above obtains the `file` containing a `list_head` by simply subtracting from the `list_head` its offset from the start of `file`. Thus, when the termination condition is satisfied, the `list_head` in question is the special "head" `list_head` that does not have a containing `file` object. However, the comparison in the condition still works fine for C because $\&pos- > member$ is computed as `pos + OFFSET_OF(TYPE_OF(pos), member)`, where `OFFSET_OF` and `TYPE_OF` have obvious meanings. So even though `pos` is a non-existent object, its address is never dereferenced; instead, it is simply added back to the appropriate offset to obtain the address of the head. However, BCC is not aware of this future usage of `pos` at the time it is generated. Consequently, this is flagged as an error.

Simple C Constructs that BCC cannot handle The above example is a case of pointer arithmetic *overflowing* an object (in this case, the `head`) but not being dereferenced. It only serves to demonstrate the possible complexity of underlying operations in the kernel. The same effect can also be demonstrated using a simple example. Consider the following contrived example:

```
1.  void fn (unsigned i, unsigned j)
2.  {
3.      char arr[10], char *p;

4.      p = arr + i - j;
5.      *p = 'x';
```

On line 4, expression `p = arr + i - j` will be evaluated as `p = ( (arr + i) - j)`. We can rewrite the code using a temporary variable as:

```
1.  void fn (unsigned i, unsigned j)
2.  {
3.      char arr[10], char *p;
3.1     char *t;
3.2     t = arr + i;
4.      p = t - j;
5.      *p = 'x';
```

This is the exact manner in which this function will be evaluated by KBCC. In line 3.2, the expression `arr + i` can be out of the bounds of array `arr` depending on the value of `i`. However, in line 4, upon evaluation of `t - j`, the result `p` may be back within the bounds of `arr` (if $0 \leq (i - j) < 10$). Thus on line 5, the dereference will be legitimate. However, `libcheck` flags an error on line 3.2 if it detects that `t = arr + i` has gone out of `arr`'s bounds. This is undesirable since this could be perfectly legal usage. We have fixed all these problems with BCC in our KBCC compiler. We describe the nature of our fix below.

Out-of-bounds pointers We use the term *out-of-bound* pointers (OOB) to describe the above phenomenon. These are (often temporary) pointer values generated during computation that may fall out of bounds, but are perfectly safe since they are not dereferenced. BCC flags an error for such temporary out-of-bounds addresses, which is undesirable. Ruwase and Lam proposed a scheme [109] to fix this problem. In their method, whenever an OOB pointer is generated, `libcheck` enters a record for that OOB into a global table. The record has just one element: the out-of-bounds address of the OOB object. Thereafter, `libcheck` replaces the OOB pointer in the execution stream by the address of its record in the table. During all later operations on the OOB, the program passes the address of the record to `libcheck` and `libcheck` transparently uses the actual out-of-bounds address from the corresponding record to check an access or perform an arithmetic. Thus by replacing an OOB address with a special address in the table, `libcheck` gracefully handles the case when the OOB pointer undergoes some further operation.

This method does not work when there is a mix of KBCC-compiled code and other code. The problem arises due to the use of alternative addresses (into the table) to replace real OOB addresses. If one such address is passed to a part of code that is not compiled with KBCC, it is bound to fail. In our case, we want to have the flexibility of compiling only parts of our system with bounds checking. Hence, checked and unchecked code should mix without any problems.

We suggest two alternatives in this paper. In the first approach, `libcheck` makes note of every OOB pointer value in a table. However, we do not replace the OOB address in the execution stream. Instead, when the OOB pointer is later passed again to `libcheck`, it is first searched in the table, and if not found, it is searched in the splay tree.

This approach though sound is likely to be slow, since even in the common case of no OOBs, the OOB table will be searched before the splay tree is searched. In our other approach, we define a special relation among objects called *peering*. Whenever an out-of-bounds address is created by an arithmetic operation on an object O, we insert a special *peer* object into the splay tree at the new address, making it a peer of O. Object O becomes the *peer-leader* of the set of peers that originate from it. The only operations permitted on a peer pointer are those that either generate another peer or return to the bounds of the peer-leader. The peering relation between objects helps us track OOB pointers and unlike the method of Ruwase and Lam, we do not need to replace its value in the execution stream.

Removing redundant pointer checks. Completely safe bounds checking of C code is difficult in concurrent environments. For sound bounds checking, pointer accesses must be validated and executed atomically, failing which, bounds checking is no longer sound. A concurrently executing thread could modify or free the pointer between validation and execution, causing the bounds checker to miss errors or access freed memory. The probability of such concurrent accesses is increased when redundant pointer checks are removed, since a check now potentially covers more than one access. However, redundant pointer checks do not necessarily imply freedom from such race conditions.

Our benchmarking results given in Section 5.5 indicate that redundancy elimination can significantly boost bounds-checking performance. Atomicity can be ensured by detecting and removing race conditions from bounds checking code by using static race-detection techniques [46, 120]. However, KBCC provides redundancy elimination as an optional feature.

Several algorithms exist to remove redundant pointer checks from bounds checking code [40, 64, 95] and most of these can be incorporated into KBCC's bounds checking framework. We discuss some of these techniques in detail in Section 6.1. For the purpose of demonstrating the benefits of removing redundant checks from bounds checking code, we used a simple strategy based on common subexpression elimination. Redundant pointer checks are removed by marking checking calls to the monitor with the GCC attribute `__attribute__ (("const"))`, which makes them candidates for removal by GCC's common subexpression elimination and loop-unrolling optimizations. GCC provides this attribute to mark functions whose only effect is their return value, and the return value depends only on the parameters. If parameter values have not changed since the last checking call, the new call can simply be replaced by the last return value. This optimization weeds out a bulk of redundant memory checks. Although the bounds checking monitor makes use of external state in the form of the splay tree, this does not violate the `"const"` attribute. The global state is maintained on a per-memory region basis and is only accessed by the monitor.

Summary of Our Contributions

- We produced a robust and efficient runtime bounds checking framework for the Linux kernel.
- Our system has high coverage: we have been able to compile most of the Linux kernel. We believe that using our techniques, BCC can be able to support a wide range of existing programs. We can support several version of the GCC compiler and several versions of the Linux kernel (both 2.4 and 2.6) including the latest releases.
- We modified KBCC to be reentrant. The library `libcheck` is completely reentrant inside the kernel.
- We optimized KBCC for size and efficiency in several ways. For example, by reducing stack usage using informed compile–time analysis, removing redundant tests by applying common–subexpression elimination on the abstract syntax tree, by removing redundant code with the help of compile–time analysis, function call argument packing, etc.
- We added support to KBCC to effectively manage out-of-bounds pointers.
- While working towards a kernel-ready BCC, we also created a more robust version of BCC, fixing several serious bugs (e.g., several problems with inline function calls and out-of-bounds pointers).

5.4.2 Mudflap

GCC Version 4.0 comes with a built-in bounds checker called Mudflap. It validates pointer operations against a fast cache to avoid querying the splay tree. The code for searching the fast cache is inlined in the checked code instead of a function call, resulting in inflated binary. In our port of Mudflap, we provide the option for optimizing for size rather than performance, by allowing sharing of the code for searching the cache.

However, the main difference between Mudflap and KBCC is that Mudflap does not check pointer arithmetic. Hence, it is less sound than KBCC and can miss some bound overflow bugs. Nevertheless, Mudflap does not suffer from the problem of false positives due to out-of-bounds pointers like BCC.

5.4.3 Kernel Code Electric Fence

Kernel Code Electric Fence (Kefence) is designed to detect memory buffer overflows at the hardware level. Kefence aligns memory buffers allocated in the kernel virtual address space (using `vmalloc`) to page boundaries. The kernel's `vmalloc` function allocates one or several pages for each request, facilitating this alignment. A guardian page table entry (PTE) is added adjacent to each buffer so that whenever a buffer overflow occurs, the guardian PTE is accessed. The guardian PTE has read and write permissions disabled; hence, accessing it causes a page fault. The page fault handler of the Linux kernel is modified so that whenever there is an access to a guardian PTE, it reports a buffer overflow.

Exact details about the context and location of buffer overflows are logged through syslog. The modified page fault handler can be configured to perform various additional tasks. When security is critical, Kefence can be configured to crash the module upon a memory overflow, thereby preventing further malicious operations. The system administrator can look at the logs to determine the location of the overflow. If debugging is more important, Kefence can be configured to just log the buffer overflow without terminating the module. We implemented this by auto-mapping a read-only or read-write page to the guardian PTE whenever there is an overflow. This way the code which caused the overflow can either be allowed to write or to just read the out-of-bounds memory locations. Since the logs contain full information about the location and the code which caused the overflow, buffer overflows in kernel code can be diagnosed easily. Kefence can do this in real time, making it suitable for security critical applications.

Since Kefence can only protect virtually-mapped buffers, those allocated using `kmalloc` are not protected. Therefore, to add bounds checking to a kernel module, one must use `vmalloc` instead of `kmalloc` for memory allocations. We have modified the Linux header files in such a way that this replacement is done automatically if a special compiler flag is set.

Using `vmalloc` consumes more virtual memory, since it allocates at least a page for each memory allocation. This is partly mitigated by the fact that modern 64-bit architectures make the address space a virtually inexhaustible resource. However, replacement of `kmalloc`s with `vmalloc`s results in extra consumption of physical memory because the memory is allocated in units of pages. To speed up the default `vfree` function we have added a hash table to store the information about virtual memory buffers. Since the alignment of buffers to page boundaries can be done either at the beginning or at the end, Kefence cannot detect buffer overflows and underflows simultaneously, unless the allocation is in multiples of the page size. Although this is a common case inside the kernel, we have found overflow protection to be sufficient in most other cases.

5.5 Performance Evaluation

In this section we present our benchmarking results for KBCC, Kefence, and Mudflap. We conducted our benchmarks on a 1.7GHz Pentium 4 machine with 1GB of RAM running Red Hat 9 with a vanilla 2.6.10 kernel. The machine was equipped with 10GB Seagate U5 5,400 RPM IDE hard drives.

5.5.1 Kefence

To evaluate the performance of Kefence, we instrumented a stackable file system [136] called Wrapfs. Wrapfs is a wrapper file system that just redirects file system calls to a lower-level file system. The vanilla Wrapfs uses `kmalloc` for allocation. Each Wrapfs object (inode, file, etc.) contains a private data field which gets dynamically allocated. In addition to this, temporary page buffers and strings containing file names are also allocated dynamically. We mounted Wrapfs on top of a regular Ext2 file system with a 7,200 RPM IDE disk to conduct our tests. In the instrumented version of Wrapfs, we used `vmalloc` for all memory allocations so that we could exercise Kefence for all dynamically allocated buffers.

We compiled the Am-utils [100] package inside Wrapfs and compared the time overhead of the instrumented version of Wrapfs with vanilla Wrapfs. The instrumented version of Wrapfs had an overhead of 1.4% elapsed time over normal Wrapfs. This overhead is due to two main causes. First, `vmalloc` and `vfree` functions are slower than `kmalloc` and `kfree` functions. Second, allocating an entire page for each memory buffer increases TLB contention. We found that the maximum number of outstanding allocated pages during the compilation of Am-utils over the instrumented version of Wrapfs was 2,085 and the average size of each memory allocation was 80 bytes. We conclude that Kefence performs well for normal user workloads. However, memory-intensive code may exhaust virtual or physical memory.

5.5.2 KBCC

For software bounds checking, we compared the performance of kernel code compiled with KBCC with the performance of kernel code compiled with plain GCC. We used the ReiserFS file system in Linux 2.6.10 as our code base, and exercised it using standard file-system benchmarks. The choice of ReiserFS was motivated by our experience with Linux file systems, which indicates that the source code of ReiserFS in general, uses more complicated C constructs and pointer operations than the code of most other file systems. We remounted ReiserFS before every benchmark run to purge the page cache. We ran each test at least ten times and used the Student-t distribution to compute 95% confidence intervals for the mean elapsed, system, user, and wait times. Wait time is the elapsed time less CPU time used and consists mostly of I/O, but process scheduling can also affect it. In each case, the half-widths of the confidence intervals were less than 5% of the mean.

We benchmarked both KBCC and Mudflap.

The first benchmark we ran was an Am-utils build. Am-utils is a CPU-intensive benchmark. We used Am-utils 6.1b3 [100]: it contains over 60,000 lines of C code in 430 files. Though the Am-utils compile is CPU intensive, it contains a fair mix of file system operations, which are compiled with bounds checking enabled. Figure 5.1 compares the results for building Am-utils inside ReiserFS compiled with our port of BCC and ReiserFS compiled with plain GCC. Previously, code compiled with software bounds checking has been reported to run 2–6x times slower than the original version [40, 64, 75]. ReiserFS compiled with our port of BCC exhibits only 19% overhead in elapsed time, and around 35% overhead in system time. This demonstrates that our optimizations have saved significant overhead over what was previously reported.

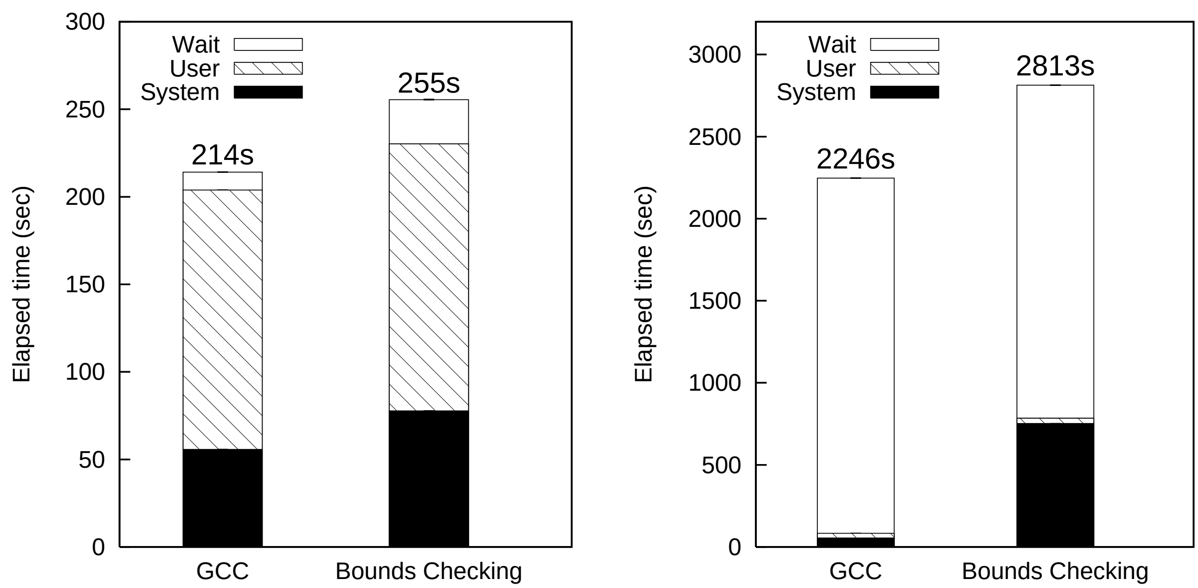

Figure 5.1: Bounds checking overhead for ReiserFS with Am-utils (left) and Postmark (right).

The second benchmark we ran was Postmark [77], which simulates the operation of electronic mail servers. It performs a series of file system operations such as appends, file reads, creations, and deletions. This benchmark uses little CPU, but is I/O intensive. We configured Postmark to create 20,000 files, between 512–10K bytes, and perform 200,000 transactions. Create and delete, and, read and write operations were selected with equal probability. We chose these parameters as they are typically used and recommended for file system benchmarks [127]. Figure 5.1 compares Postmark results for ReiserFS compiled with our port of BCC and ReiserFS compiled with plain GCC. In this case, the system time overhead is around 14 times.

However, since there is a significant wait time spent waiting for I/O, the overall elapsed time has only 25% overhead.

To determine the residual overhead from using instrumented code, we measured system performance once all bounds-checking tests have been turned off. We observed that the performance in this case was within 3–4% of the original uninstrumented performance.

Our results indicate that we have significantly cut down on the overhead from bounds checking. Moreover, once bounds checking is turned off, performance is close to the unchecked case.

Chapter 6

Static Analysis as a Means for Facilitating Runtime Checking

In this section we discuss possible uses of static analysis to facilitate runtime checking. In particular, we focus on two categories of uses: static analysis for helping with runtime bounds checking (Section 6.1), and the general case of using static analysis for helping with code instrumentation (Section 6.2).

6.1 Static Analysis for Bounds Checking

Static analysis has been used for improving bounds checking system in several ways. The general problem of bounds checking can be divided into two subproblems: (1) identifying a given memory reference's referent object and thus its bounds (*bound lookup problem*), and (2) comparing the reference's address with the referent's bounds and raising an exception if the bound is violated (*bound comparison problem*). One of the common uses of static analysis has been used in solving the bound comparison subproblem. The general approach is to eliminate unnecessary checks, so that the number of checks is reduced. However, as seen in Section 5.4.1, these techniques may not be valid in the presence of concurrency.

Gupta [64] used flow analysis techniques to avoid redundant bound checks. Although this technique guarantees to identify any array bound violation, it does not necessarily detect them at the time of the error. This was done to reduce the overhead associated with bounds checking within a loop. Such bound-checks are moved *outside* the loop so that checking them guarantees prevention of bound overflow errors within the loop body.

ABCD [22] is a light-weight algorithm for eliminating Array Bounds Checks on Demand. Despite its simplicity, ABCD has proved to be quite effective. BOON [40] checks memory errors in using standard string library functions. It formulates memory buffer overruns as integer range analysis problem. Its approach consists of two logical steps: (1) infer possible ranges for array indices, and (2) for each possible input, check using range analysis if any memory access falls beyond an array's limits. Larson and Austin present an interesting extension to this idea. Their method is identical to BOON's approach in step (1). However, for detecting possible inputs in step (2), they argue against using static analysis. In general, any static analysis cannot always determine exactly which paths will be taken at runtime. Hence, BOON's method (which relies on symbolic execution) may produce false positives. Larson and Austin's approach is to dynamically monitor variable values at runtime possibly inferring all possible ranges in which they can fall. Using this information in the range analysis, one can potentially perform well-informed bounds checks.

6.1.1 CCured

CCured [95] is the project most similar to our work in spirit. CCured uses a combination of static analysis and runtime checking to detect all memory errors in C programs. CCured is a source-to-source converter that accepts as input a C program, and outputs another C program instrumented with bounds-checks. It uses type-based static analysis to eliminate redundant memory checks statically. The idea is to assign type qualifiers (through user annotations) to individual pointers. Thereafter, the static analysis *infers* type qualifiers throughout the program using this initial set of qualifiers. If the inferred type of a pointer is safe, it is not bounds-checked at runtime; for all other memory accesses, runtime checks are inserted.

In CCured, pointers and the memory objects they point to can be *typed* or *untyped*. A typed pointer is either `SAFE`, or `SEQ` (sequential), and points only to typed memory. Memory typing information is dynamically maintained in a bitmap at runtime. For operations on `SAFE` pointers, CCured does not insert any bounds checking code; safety is ensured by disallowing any arithmetic on a `SAFE` pointer, and checking the pointer for only `NULL` value at runtime. Pointer arithmetic is allowed on `SEQ` pointers: in fact, `SEQ` pointers were introduced for memory traversal operations (e.g., array references). In contrast, an untyped pointer always points to an untyped memory region. All operations on an untyped pointer are checked for bound overflows every time it is dereferenced.

With this type system in place, CCured's working can be understood as a sequence of three steps: (1) disallow incompatible operations, like pointer arithmetic on `SAFE` types, (2) statically validate using type inference which operations need not be checked, and (3) for all other operations, insert runtime checks. CCured enforces the invariant that untyped pointers cannot access typed memory areas. Consider the following example:

```
01    struct elem {
02        int * UNTYPED unsafe_pointer;
03        int * SAFE safe_pointer;
04    };
05    int func (struct elem * SEQ p) {
06        struct elem * SAFE e = NULL;
07        int s = 0;
08        for (int i = 0; i < 100; i++) {
09            e = (struct elem * SAFE) (p+i); //out-of-bounds check
10            s += *(e->unsafe_pointer; //No out-of-bounds check on e,
//out-of-bounds check on e->unsafe_pointer
11            s += *(e->safe_pointer; //No out-of-bounds check on e,
//Only NULL check on e->safe_pointer
12        }
13        return s;
14    }
```

This example demonstrates the use of typed pointers and untyped pointers. In this case the programmer has annotated the code with `SAFE` and `UNTYPED` annotations. However, in this case CCured could have inferred these attributes. Line 09 casts a `SEQ` pointer `p` to `SAFE` pointer `e`. A bounds-check is performed at this step. However, `e` is not checked for bounds overflow after this step.

A most conservative translation is to consider all pointers untyped. Thereafter, CCured infers `SAFE` and `SEQ` pointers using a flow-sensitive analysis. This involves constraint generation and solving for type inference. The authors report that in general even without programmer annotations CCured can infer many cases of pointer-typing information. However, the programmers may use annotations for optimizations.

CCured is sound: it detects all spatial memory errors using bounds checks and `NULL`-pointer checks. For temporal errors it provides its own garbage collection mechanism. Note that a language-based garbage collector makes it a poor candidate for use in systems programming, since the performance characteristic of the code is not obvious from the source code itself—there is an unpredictable and varying garbage collection overhead.

6.1.2 Static Analysis in BCC

BCC employs various static analyses during the course of compilation. Consider the issue of padding variables for the purpose of detecting bound overflows. Jones and Kelly's implementation [75] does not pad stack variables if both of the following conditions hold for them: (1) the variable is not a pointer, and (2) the variable's address is not assigned to any pointer in the scope of the function. Note that BCC derives this information from GCC's analysis framework. By not padding these variables, BCC reduces stack usage of the checked program.

As seen in Section 5.4.1, KBCC leverages GCC's optimization framework to remove unnecessary checks from the program. This is an example of conservative flow-sensitive analysis for mitigating the bound conversion problem.

6.2 Static Analysis for Instrumentation of Code

Static analysis of code can also be useful in the more general case of instrumenting code for tracing support, checking, etc. In this section we first discuss *aspect oriented programming*, a generic approach to instrumenting arbitrary code. We then present the compile-time instrumentation framework of **Aristotle**, a runtime verification system we developed.

6.2.1 Aspect-Oriented Programming

Aspect-oriented programming (AOP) is an upcoming programming paradigm which allows crosscutting concerns to be modularized as aspects that are separated from objects. Aspects can specify synchronization policies, resource sharing and performance optimizations over objects. A compiler called *weaver*, weaves aspects and objects together into a program.

The AOP paradigm was inspired by the difficulty programmers often face in understanding how the code works because it is often poorly localized, and its relation to the execution flow of the main code is hard to follow. Often, implementation for aspects of the program, like the scheduling policy is spread throughout the entire source code. As a result, understanding the code and maintaining it becomes difficult. Often, changing only a particular aspect of the code requires changing the source code in many different places. Aspect-oriented approaches attempt to address these problems by modularizing the scattered pieces of source code that implement a single concept.

Aspects are described as specifications in a *spec language*. The language comprises of a series of *pattern-action* pairs. The weaver searches for each occurrence of the pattern in the source code, and performs the corresponding action on it. For example, the action could be to insert a function call, or signal an error. **AspectJ** is an aspect-oriented framework for Java. It was an offshoot of the MOPS project (Section 3.3.4) for separating higher level protocol specifications from source code. The system comprises of an aspect-oriented language (AspectJ) for Java and a weaver. Some other frameworks currently under development are AspectC++ [1] and AspectC [33].

All these approaches use simple pattern matching when searching for the pattern in the target code (C or Java). The only form of flow-sensitive pattern searching supported is using the *cflow* keyword. However, the weaver as such does not perform any flow-sensitive analysis. Any flow-sensitive conditions in the pattern code are evaluated at *runtime*. This can significantly slow down runtime performance for frequently executed code.

The Bossa system [4] is an example of an AOP system that allows the programmer to use flow-sensitive static analysis for specifying patterns. The Bossa spec language supports temporal logic based flow-sensitive pattern specifications. Bossa has been used with the Linux kernel for separating the scheduling framework from the mainstream kernel.

Several other tools exist for code instrumentation. CTool [5] is a simple source transformation tool. ASTlog [105] is a tool for simple pattern matching in parse trees. Tools such as ATOM [119] provide support for binary level translation. Purify [106] performs binary level translation of generic programs for inserting bound checking.

6.2.2 Source Code Instrumentation in Aristotle

Aristotle is a runtime verification system for the Linux kernel currently under development in our research group. Aristotle continually monitors kernel execution at runtime, checking it for conformance to pre-defined properties. Kernel monitoring is facilitated by tracing events generated by an instrumented kernel. The tracing events are typically function calls made to Aristotle's runtime checking engine as shown in Figure 6.1. For the purpose of our discussion, we will focus on **KGCC**.

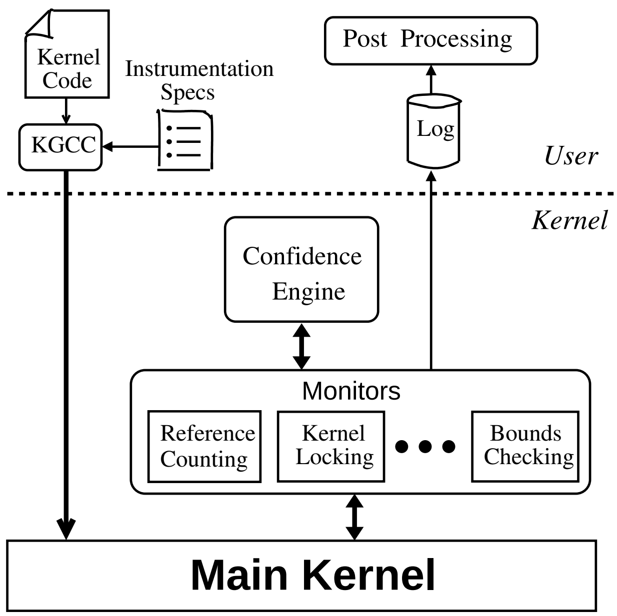

Figure 6.1: Architectural overview of the Aristotle system. KGCC instruments kernel code with monitoring calls. During the execution of the instrumented kernel, the monitoring calls invoke individual monitors to dynamically verify system properties.

KGCC is a special compiler used by Aristotle for instrumenting the Linux kernel for runtime monitoring. KGCC is an extension to GCC. In fact, the KBCC compiler introduced in Section 5.4.1 is a subset of KGCC. KGCC is a general source code instrumentation framework based on GCC.

Aristotle inserts custom monitoring code into the Linux kernel using KGCC. KGCC scans the parse tree generated during compilation and instruments it with the monitoring code. Using a compiler for statically instrumenting kernel code offers several advantages. Compile-time instrumentation can benefit from higher-level semantic information about the code, e.g. user-defined data types that are typically not available to binary rewriters. Moreover, static analysis can be used to guide the placement of the monitoring code. In this context, Aristotle uses the compiler both as a provider of parse trees, type information, etc., and as a useful tool for instrumenting programs at the parse-tree level, code optimization, etc. KGCC also allows us to separate the monitoring code from the source code, making monitoring code easily portable to different kernel versions in the spirit of aspect-oriented programming.

GCC is a heavy-weight C compiler with little documentation of its internals. Static checking and meta-compilation [66] projects in the past have preferred light-weight compilers with well-documented internals over GCC [27], since they are much easier to understand and modify. The Linux kernel, however, is designed to compile exclusively with GCC, forcing Aristotle to execute code compiled with KGCC rather than just use the compiler for its parsing capabilities. These considerations obviate our choice of a GCC-based approach. KGCC was originally based on GCC-3.2 and later ported to GCC-4.0 (beta). GCC-4.0 offers several advantages: a cleaner interface to manipulate parse trees, better documentation which significantly reduced development times, and advanced optimizations.

Like AOP, KGCC's instrumentations are also guided by *pattern–action* pairs. Patterns identify source code constructs of interest, and actions identify the corresponding actions to be performed on the source code. For each such pair, GCC scans the parse tree for every occurrence of *pattern*, and inserts the corresponding *action* in the appropriate parse-tree node. Patterns can be specified using any information typically available in the parse tree: types, identifiers, containing functions, file names, etc. Note that in our current implementation, we do not yet support a generic language for specifying patterns and actions. KGCC currently hard-codes these into the compiler itself. However, our design is easily amenable to an AOP "weaver" like setup.

KGCC actions can be logically divided into two categories: *pre-checks* and *post-ops*. Pre-checks are to be performed before executing the operation identified in the pattern. For example, in the case of memory-bounds checking (see Section 5.4.1), a pre-check would verify pointer sanity before each pointer dereference. A post-op is inserted after the specified operation and can be an arbitrary piece of C code (as is the case for bounds checking) or simply a call to the event dispatcher. In the latter case, the operation (event) will be dispatched to the confidence engine and may be logged for post-processing purposes. For example, when reference-counting monitoring is enabled, an operation that manipulates an object's reference count would be dispatched to the reference-counting monitor to check that the operation does not violate a system-correctness property.

Chapter 7

Conclusions and Future Work

The contributions of this work are that we have presented a survey of available techniques for static checking of software. We have performed a comprehensive survey of existing runtime bounds checking techniques. The main conclusion of this paper is that, where correctness guarantees are desired, static analysis should be combined with runtime checking.

We have seen that static checking of C programs is limited in its scope due to computational limits. Although static checkers have been designed for detecting bugs in software, these checkers are often unsound and can miss bugs. Attempts to achieve complete soundness result in loss of precision in the form of a large number of false alarms. These false alarms are a significant practical drawback.

Existing static checking systems have favored one of soundness and precision in their design trade-offs. Soundness, as incorporated in systems like Cqual, requires user participation in the form of program annotations to be useful. Hence, although they may be adopted in fresh projects, it is unlikely that they will be adopted in existing systems.

Other systems favor bug finding power and precision over soundness, and can be best described as best-effort checkers. Even though they are unsound, they are nevertheless useful. Completely sound and precise systems are not compatible with existing C code. Since software checking for legacy C code like the Linux kernel cannot give correctness guarantees, best-effort bug detection is a useful practical alternative for increasing software quality.

It is evident that static checking alone is not in the state to give correctness guarantees about large complex software like the Linux kernel. Faced with this impossibility, the alternative has been to use runtime techniques for providing correctness guarantees. However, runtime checking techniques themselves are not acceptable in production systems due to performance reasons. The central thesis of this paper is that, where correctness guarantees are desired, static analysis should be combined with runtime checking.

The rest of this chapter is organized as follows. Section 7.1 discusses our conclusions about static checking techniques. Section 7.2 discusses our conclusions about runtime bounds checking. We present possible future directions of research in Section 7.3.

7.1 Static Checking

The cost of repairing a bug increases along the software development time-line. An early discovery can result in substantial savings, making static checking a useful tool for software developers.

From our survey of annotation-based checkers and tool-based checkers, we believe that both have some unique benefits. Annotation-based systems can be more sound and precise, but require additional effort on the part of the programmer. Tool-based checkers are easier to use, especially if they are extensible, but they are often unsound and need innovative design to be powerful in catching bugs.

Annotations can be viewed as a mechanism by which programmers *help* a static checker to check their code. Static checking systems use annotations for some or all of the following purposes:

- **Assumptions and Invariants:** Programmers use annotations to specify assumptions and invariants. Thus the static analyzer is freed from the responsibility of inferring invariants. Completely sound and precise static invariant inference is often intractable. Thus with programmer help, the static analyzer can still usefully check program properties without being any more unsound and imprecise.

- **Interface Specifications:** Programmers use annotations to specify procedure interfaces. This often saves the static analyzer from (a possibly unsound and imprecise) inter-procedural analysis. More importantly, this leads to highly scalable static analyzers since procedures can be easily checked in isolation. Interface level annotations are arguably a useful means to automatically generate code documentation as well.

- **Extensibility:** Some static analyzers like ESC/Java (see Section 3.1.4) give programmers much freedom in specifying annotations (ESC/Java allows programmers to specify arbitrary first order logic annotations). Hence programmers can *extend* the default checking done by the checker with their own custom checks.

Tool-based checkers are trying to solve a much harder problem: checking possibly arbitrary code where the code *speaks for itself* (no annotations). In fact, tool-based checkers represent a much advanced point in the automated checking design space, especially with regard to engineering novelty. Since this problem is hard, tool based checkers trade-off soundness for tractability. From our survey we infer the following key features that increase the power and usefulness of tool-based checkers:

- **Extensibility:** Most of the bugs in software are due to violations of some system-specific properties. For a checker to be useful in general, it should be able to detect such bugs. Tool-based checkers like ESP and Metacompilation allow the programmer to specify system-specific properties separately from code. The checker then uniformly checks the entire code base with this property specification. Thus, both tool-based and annotation-based methods use the programmer—only that tool-based methods can use them even with unmodified code.

- **Statistical Inference:** Best-effort checkers can be more bold in their bug detection strategies than their sound and complete counterparts. One example of this ideology is the use of statistical inference for deriving system invariants, and then checking the code for violations of these invariants. Note that such an analysis can be very imprecise, and heuristics are needed to properly sort and rank such errors.

- **Rank Bugs:** To win programmers' confidence, the checker must be useful. Just finding bugs is not enough. There should be minimum false alarms. Again, this is a difficult problem especially if the checker is aggressive in finding as many bugs as possible. One solution is to rank bugs: bugs are ranked in the order of decreasing likelihood of being a genuine bug and not just a false alarm.

7.2 Runtime Bounds Checking

Memory access errors cannot always be verified statically. However, they also represent a serious problem in systems today. Runtime bounds checking, possibly with help from static analysis, is a viable approach to ensuring memory safety at runtime.

Runtime memory access checking schemes can be evaluated based on four criteria [132]:

1. The ability to detect all memory errors,

2. low performance overhead,
3. the ability to handle all C programs without modification, and
4. preserving the existing C memory-allocation model.

We have seen classification of bounds checking methods based on their approach for handling the bounds lookup subproblem. We proposed a formal classification for existing techniques based on capability checking and availability checking. Capability checking are much better than availability-based techniques in most respects except one. Capability-based techniques are sound in that they can potentially detect all memory errors, they are much faster, they can protect existing C memory allocation model, and they can be backward-compatible with most existing C source code. However, capability-based methods are often incompatible with existing pre-compiled C code (e.g., libraries).

The independent subproblem of bounds comparison has also received much attention. Static analysis has been successfully used for eliminating unnecessary checks on bound accesses. A radically different approach is the use of hardware-based techniques.

The Linux kernel bounds checking compiler is an instructive experiment in runtime bounds checking of large safety-critical software. In particular, we have successfully applied a known bounds checking technique to check the Linux kernel. Arguably, this is the only existing software approach that could be applied to bounds checking of the Linux kernel in a completely backward-compatible way. During the process of this port, we identified and solved several issues like invalid pointers, redundant checks, etc.

7.3 Future Work

We plan to pursue future work in two directions: (1) bounds checking, and (2) static analysis for runtime checking.

Bounds Checking Performance overhead is the bane of runtime bounds checking. Whereas removal of redundant pointer checks did improve KBCC's performance, it is still too slow for deployment in production environments. Hardware based techniques, though promising, fall short in comprehensiveness and speed in some cases. We are exploring an alternative novel approach to address this problem. The idea is that after bounds checks have been exercised a certain number of times, they can be turned off, giving back the original performance.

Static Analysis for Runtime Checking We are actively developing the Aristotle system introduced in Section 6.2.2. Aristotle will be used for verifying properties of the Linux kernel at runtime. We plan to leverage static analysis in two ways to help achieve this goal:

- Minimize performance overhead due to monitoring: statically deduce and verify all that is possible, leaving only the rest for runtime checking and verification.
- Provide fine-grained control to Aristotle developers for arbitrarily instrumenting kernel code at compile time.

In the context of bounds checking, we have achieved limited success in fulfilling the first requirement. For providing fine-grained instrumentation control, we plan to develop upon ideas from aspect-oriented programming research to implement a useful extension to GCC for arbitrarily instrumenting source code at compile time. In particular, our GCC support is nearly ready, and we are working on an advanced specification language for describing the "aspects." Recent deliberations have suggested the possibility that overloading the already heavy-weight GCC compiler with this facility may not be a good idea, and the analysis engine and instrumentation engine should be implemented as separate tools.

Bibliography

[1] AspectC++ Homepage. `www.aspectc.org`.

[2] Splint Guide. `www.splint.org/guide/guide.html`.

[3] Splint Manual. `www.splint.org/manual/manual.html`.

[4] The Bossa Project. `www.emn.fr/x-info/bossa/`.

[5] The GNU CTool Project. `http://ctool.sourceforge.net`.

[6] A. Bensoussan and C. T. Clingen and R. C. Daley. The Multics Virtual Memory: Concepts and Design. *Communcations of the ACM*, 15(5):308–318, 1972.

[7] A. Blass and Y. Gurevich. Inadequacy of Computable Loop Invariants. *ACM Transactions on Computational Logic*, 2(1):1–11, 2001.

[8] S. Abramsky and C. Hankin, editors. *Abstract interpretation of declarative languages*. Halsted Press, New York, NY, USA, 1987.

[9] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of the Summer USENIX Technical Conference*, pages 93–112, Summer 1986.

[10] A. Aiken, J. S. Foster, J. Kodumal, and T. Terauchi. Checking and inferring local non-aliasing. *SIGPLAN Not.*, 38(5):129–140, 2003.

[11] W. Arbaugh, D. Farber, and J. Smith. A Secure and Reliable Bootstrap Architecture. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 1997.

[12] W. A. Arbaugh, W. L. Fithen, and J. McHugh. Windows of Vulnerability: A Case Study Analysis. *IEEE Computer*, 33(12):52–59, December 2000.

[13] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient detection of all pointer and array access errors. *SIGPLAN Not.*, 29(6):290–301, 1994.

[14] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of c programs. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM Press, 2001. Appeared in *ACM SIGPLAN Notices* 36(5), pp. 203-213.

[15] T. Ball, A. Podelski, and S. K. Rajamani. Relative completeness of abstraction refinement for software model checking. In Joost-Pieter Kaoen and Perdita Stevens, editors, *Proceedings of TACAS02: Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *LNCS*, pages 158–172, Grenoble, France, April 2002. Springer-Verlag.

[16] T. Ball and S. K. Rajamani. The SLAM toolkit. In *CAV '01: Proceedings of the 13th International Conference on Computer Aided Verification*, pages 260–264. Springer-Verlag, 2001.

[17] T. Ball and S. K. Rajamani. The slam project: Debugging system software via static analysis. In *Conference Record of the 29th Annual ACM Symposium on Principles of Programming Languages*, pages 1–3. ACM Press, 2002.

[18] R. Balzer. Mediating connectors. Technical report, USC/ISI, 1998. `www.isi.edu/software-sciences/instrumented-connectors.html`.

[19] R. Balzer and N. Goldman. Mediating connectors. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems Workshop*, pages 72–7, Austin, TX, June 1999.

[20] B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, D. Becker, M. Fiuczynski, C. Chambers, and S. Eggers. Extensibility, safety, and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP '95)*, pages 267–284, Copper Mountain Resort, CO, December 1995. ACM SIGOPS.

[21] W. R. Bevier. Kit: A Study in Operating System Verification. *IEEE Transactions on Software Engineering*, 15(11):1382–1396, 1989.

[22] R. Bodik, R. Gupta, and V. Sarkar. Abcd: Eliminating array bounds checks on demand. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 321–333, New York, NY, USA, 2000. ACM Press.

[23] C. Boyapati and M. C. Rinard. A parameterized type system for race-free Java programs. volume 36(11) of *SIGPLAN Notices*, pages 56–69, November 2001.

[24] T. Bray. The Bonnie home page. `www.textuality.com/bonnie`, 1996.

[25] H. T. Brugge. The BCC home page. `http://web.inter.NL.net/hcc/Haj.Ten.Brugge`, 2001.

[26] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Softw. Pract. Exper.*, 30(7):775–802, 2000.

[27] C. W. Fraser and D. R. Hanson. *A Retargetable C Compiler: Design and Implementation*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[28] S. Chiba. A metaobject protocol for c++. In *OOPSLA '95: Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications*, pages 285–299, New York, NY, USA, 1995. ACM Press.

[29] T. Chiueh, G. Venkitachalam, and P. Pradhan. Integrating segmentation and paging protection for safe, efficient and transparent software extensions. In *Proceedings of 17th ACM Symposium on Operating Systems Principles (SOSP '99)*, pages 140–153, Kiawah Island Resort, near Charleston, SC, December 1999. ACM SIGOPS.

[30] J. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. pages 258–269, 2002.

[31] A. Chou, B. Chelf, D. R. Engler, and M. Heinrich. Using meta-level compilation to check FLASH protocol code. In *Architectural Support for Programming Languages and Operating Systems*, pages 59–70, 2000.

[32] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An Empirical Study of Operating Systems Errors. In *Proceedings of the 18th ACM Symposium on Operating System Principles (SOSP 2001)*, Chateau Lake Louise, Banff, Canada, October 2001. ACM SIGOPS.

[33] Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn. Using AspectC to Improve the Modularity of Path-Specific Customization in Operating System Code. In *Joint 8th European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, pages 88–98, 2001.

[34] A. Coen-Porisini, G. Denaro, C. Ghezzi, and M. Pezze. Using symbolic execution for verifying safety-critical systems. In *ESEC/FSE-9: Proceedings of the 8th European Software Engineering Conference held jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 142–151, New York, NY, USA, 2001. ACM Press.

[35] R. Coker. The Bonnie++ home page. `www.coker.com.au/bonnie++`, 2001.

[36] J.C. Corbett, M.B. Dwyer, J. Hatcliff, S. Laubach, C.S. Pasareanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from java source code. In *Proceedings of 22nd International Conference on Software Engineering*, pages 439–448, 2000.

[37] P. Cousot and R. Cousot. Comparing the galois connection and widening/narrowing approaches to abstract interpretation. In *PLILP '92: Proceedings of the 4th International Symposium on Programming Language Implementation and Logic Programming*, pages 269–295, London, UK, 1992. Springer-Verlag.

[38] D. Detlefs and G. Nelson and J. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Labs, Palo Alto, CA, 2003.

[39] D. Evans and D. Larochelle. Improving Security Using Extensible Lightweight Static Analysis. *IEEE Software*, 19(1):42–51, 2002.

[40] D. Wagner and J. S. Foster and E. A. Brewer and A. Aiken. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. In *Networking and Distributed System Security Symposium 2000*, San Diego, CA, February 2000.

[41] Department of Defense. Trusted computer system evaluation criteria (The Orange Book). Technical Report DoD 5200.28-STD, National Computer Security Center, Alexandria, VA, December 1985. `www.dynamoo.com/orange`.

[42] D. L. Detlefs. An overview of the extended static checking (esc) system. In *Proceedings of the First Workshop on Formal Methods in Software Practice*, pages 1–9, January 1996.

[43] D. L. Detlefs, R. M. Leino, G. Nelson, and J. B. Saxe. Extended Static Checking. Technical Report SRC-159, Compaq SRC, 130 Lytton Ave., Palo Alto, December 1998.

[44] N. Dor, M. Rodeh, and M. Sagiv. Cssv: Towards a realistic tool for statically detecting all buffer overflows in c. *SIGPLAN Not.*, 38(5):155–167, 2003.

[45] P. J. Downey, R. Sethi, and R. E. Tarjan. Variations on the common subexpression problem. *J. ACM*, 27(4):758–771, 1980.

[46] D. Engler and K. Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 237–252. ACM Press, 2003.

[47] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking System Rules Using System-Specific Programmer-Written Compiler Extensions. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI 2000)*, pages 1–16, San Diego, CA, October 2000.

[48] D. Engler, D. Y. 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 18th ACM Symposium on Operating System Principles (SOSP 2001)*, Chateau Lake Louise, Banff, Canada, October 2001. ACM SIGOPS.

[49] D. Evans, J. Guttag, J. Horning, and Y. M. Tan. LCLint: A Tool for Using Specifications to Check Code. In *Proceedings of the ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering*, pages 87–96, 1994.

[50] M. Fahndrich and R. DeLine. Adoption and focus: practical linear types for imperative programming. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 13–24, New York, NY, USA, 2002. ACM Press.

[51] C. Flanagan and S. N. Freund. Type-based race detection for Java. pages 219–232, 2000.

[52] C. Flanagan and S. N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In *Proc. 31st ACM SIGPLAN Symposium on Principles of Programming LANGUAGES (POPL)*, January 2004.

[53] C. Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for esc/java. In *FME '01: Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*, pages 500–517, London, UK, 2001. Springer-Verlag.

[54] C. Flanagan, K. Rustan M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *PLDI*, pages 234–245, 2002.

[55] C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 191–202, New York, NY, USA, 2002. ACM Press.

[56] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Proc. ACM SIGPLAN Conference on Programming Language Design and IMPLEMENTATION (PLDI)*, pages 338–349. ACM Press, 2003.

[57] J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 1–12, New York, NY, USA, 2002. ACM Press.

[58] G. Kiczales and J. D. Rivieres. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, USA, 1991.

[59] E. Gabber, C. Small, J. Bruno, J. Brustoloni, and A. Silberschatz. The Pebble component-based operating system. In *Proceedings of the Annual USENIX Technical Conference*, pages 267–282, Monterey, CA, June 1999.

[60] V. Ganapathy, S. Jha, D. Chandler, D. Melski, and D. Vitek. Buffer overrun detection using linear programming and static analysis. In *CCS '03: Proceedings of the 10th ACM Conference on Computer and Communications Security*, pages 345–354, New York, NY, USA, 2003. ACM Press.

[61] P. Godefroid. Model checking for programming languages using verisoft. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, pages 174–186, 1997.

[62] D. B. Golub and R. P. Draves. Moving the Default Memory Manager out of the Mach Kernel. In *Proceeding of the Second USENIX Mach Symposium Conference*, pages 177–88, Monterey, CA, November 1991.

[63] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in cyclone. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 282–293, New York, NY, USA, 2002. ACM Press.

[64] R. Gupta. Optimizing array bound checks using flow analysis. *ACM Letters on Programming Languages and Systems*, 2(1-4):135–150, 1993.

[65] J. V. Guttag and J. J. Horning. *Larch: languages and tools for formal specification*. Springer-Verlag New York, Inc., New York, NY, USA, 1993.

[66] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A System and Language for Building System-Specific, Static Analyses. In *ACM Conference on Programming Language Design and Implementation*, pages 69–82, Berlin, Germany, June 2002.

[67] M. Hicks, G. Morrisett, D. Grossman, and T. Jim. Safe and flexible memory management in Cyclone. Technical Report CS-TR-4514, University of Maryland Department of Computer Science, July 2003. Also published as University of Maryland Institute for Advanced Computer Studies (UMIACS) Technical report UMIACS-TR-2003-82.

[68] M. Hohmuth. The Fiasco kernel: Requirements definition. Technical Report TUD–FI98–12, TU Dresden, December 1998. `http://os.inf.tu-dresden.de/fiasco/doc.html`.

[69] IEEE/ANSI. Information Technology–Test Methods for Measuring Conformance to POSIX–Part 1: System Interfaces. Technical Report STD-2003.1, ISO/IEC, 1992.

[70] IEEE/ANSI. Information Technology–Portable Operating System Interface (POSIX)–Part 1: System Application: Program Interface (API) [C Language]. Technical Report STD-1003.1, ISO/IEC, 1996.

[71] J. Ferrante and K. J. Ottenstein and J. D. Warren. The Program Dependence Graph and its use in Optimization. *ACM Transactions on Programming Language Systems*, 9(3):319–349, 1987.

[72] J. W. Nimmer and M. D. Ernst. Static Verification of Dynamically Detected Program Invariants: Integrating Daikon and ESC/Java. In *Proceedings of RV'01, First Workshop on Runtime Verification*, Paris, France, July 23, 2001.

[73] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A Safe Dialect of C. In *Proceedings of the Annual USENIX Technical Conference*, pages 275–288, Monterey, CA, June 2002.

[74] N. D. Jones and F. Nielson. Abstract interpretation: a semantics-based tool for program analysis. pages 527–636, 1995.

[75] R. Jones and P. Kelly. Bounds Checking for C. Technical report. `www-ala.doc.ic.ac.uk/phjk/BoundsChecking.html`.

[76] K. Rustan M. Leino and G. Nelson and J. B. Saxe. ESC/Java User's Manual. Technical Report 2000-002, Compaq Systems Research Center, October 2000.

[77] J. Katcher. PostMark: A New Filesystem Benchmark. Technical Report TR3022, Network Appliance, 1997. `www.netapp.com/tech_library/3022.html`.

[78] S. Katz and Z. Manna. Logical analysis of programs. *Commun. ACM*, 19(4):188–206, 1976.

[79] A. Kolawa and A. Hicken. Insure++: A Tool to Support Total Quality Software. `www.parasoft.com/insure/papers/tech.htm`, March 2001.

[80] S. Kumar and K. Li. Using Model Checking to Debug Device Firmware. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI 2002)*, pages 61–74, Boston, MA, December 2002.

[81] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The stanford flash multiprocessor. In *ISCA '98: 25 years of the international symposia on Computer architecture (selected papers)*, pages 485–496, New York, NY, USA, 1998. ACM Press.

[82] L. Lam and T. Chiueh. Checking Array Bound Violation Using Segmentation Hardware. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN 2005)*, June 2005.

[83] L. Wang and S. D. Stoller. Run-Time Analysis for Atomicity. In *Proceedings of the Third Workshop on Runtime Verification (RV)*, volume 89(2) of *Electronic Notes in Theoretical Computer Science*. Elsevier, July 2003.

[84] J. Lamping, G. Kiczales, L. H. Rodriguez Jr., and E. Ruf. An architecture for an open compiler. In *Proceedings of the IMSA'92 Workshop on Reflection and Meta-level Architectures (A. Yonezawa and B. C. Smith, eds.)*, pages 95–106, 1992.

[85] W. Lee, W. Fan, M. Miller, S. Stolfo, and E. Zadok. Toward cost-sensitive modeling for intrusion detection and response. In *Workshop on Intrusion Detection and Prevention, Seventh ACM Conference on Computer Security*, Athens, Greece, November 2000.

[86] K. Rustan M. Leino, J. B. Saxe, and R. Stata. Checking java programs via guarded commands. In *Proceedings of the Workshop on Object-Oriented Technology*, pages 110–111, London, UK, 1999. Springer-Verlag.

[87] D. Lie, A. Chou, D. Engler, and D. Dill. A Simple Method for Extracting Models from Protocol Code. In *The 28th Annual International Symposium on Computer Architecture (ISCA 2001)*, Göteborg, Sweden, June/July 2001.

[88] D. Lie, A. Chou, D. Engler, and D. L. Dill. A simple method for extracting models for protocol code. In *ISCA '01: Proceedings of the 28th annual international symposium on Computer architecture*, pages 192–203, New York, NY, USA, 2001. ACM Press.

[89] M. D. Ernst and J. Cockrell and W. G. Griswold and D. Notkin. Dynamically Discovering Likely Program Invariants to Support Program Evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, 2001.

[90] L. Mauborgne. ASTRÉE: Verification of Absence of Run-Time Error. In René Jacquart, editor, *Building the information Society (18th IFIP World Computer Congress)*, pages 384–392. The International Federation for Information Processing, Kluwer Academic Publishers, August 2004.

[91] Microsoft Corporation. ESP. www.microsoft.com/windows/cse/esp.mspx.

[92] MIT LCS. Larch Project Home. `www.sds.lcs.mit.edu/spd/larch/`.

[93] M. Musuvathi, D. Y.W. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: A Pragmatic Approach to Model Checking Real Code. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI 2002)*, pages 75–88, Boston, MA, December 2002.

[94] N. Suzuki and K. Ishihata. Implementation of an Array Bound Checker. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 132–143, New York, NY, USA, 1977. ACM Press.

[95] G. C. Necula, S. McPeak, and W. Weimer. Type-Safe Retrofitting of Legacy Code. In *Proceedings of the 29th Annual ACM Symposium on Principles of Programming Languages*, pages 128–139, Portland, OR, January 2002.

[96] P. Neumann, R. Boyer, R. Feiertag, K. Levitt, and L. Robinson. A Provably Secure Operating System: The System, its Applications, and Proofs. Technical Report CSL-116, SRI International, Menlo Park, CA, May 1980.

[97] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[98] H. Patil and C. Fischer. Low-cost, concurrent checking of pointer and array accesses in c programs. *Softw. Pract. Exper.*, 27(1):87–110, 1997.

[99] G. Pearson. Array Bounds Checking with Turbo C. *Dr. Dobb's Journal of Software Tools*, 16(5):72, 74, 78–79, 81–82, 104–107, May 1991.

[100] J. S. Pendry, N. Williams, and E. Zadok. *Am-utils User Manual*, 6.1b3 edition, July 2003. `www.am-utils.org`.

[101] John Penix, Willem Visser, Eric Engstrom, Aaron Larson, and Nicholas Weininger. Verification of Time Partitioning in the DEOS Scheduler Kernel. In *International Conference on Software Engineering*, pages 488–497, 2000.

[102] B. Perens. *efence(3)*, April 1993.

[103] R. Pike, D. Presotto, K. Thompson, and H. Trickey. Plan 9 from Bell Labs. In *Proceedings of the Summer UKUUG Conference*, pages 1–9, July 1990.

[104] R. DeLine and M. Fähndrich. Enforcing High-Level Protocols in Low-Level Software. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 59–69, 2001.

[105] R. F. Crew. ASTLOG: A Language for Examining Abstract Syntax Trees. In *Proceedings of the USENIX Conference on Domain-Specific Languages*, October 1997.

[106] Rational Software. Rational PurifyPlus. `www.rational.com/products/pqc/index.jsp`, May 2001.

[107] R. Rugina and M. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 182–195, New York, NY, USA, 2000. ACM Press.

[108] J. Rushby. Design and Verification of Secure Systems. In *Proceedings of the Eigth ACM Symposium on Operating Systems Principles (SOSP)*, volume 15(5), pages 12–21, 1981. Appearing also as *ACM Operating Systems Review, SIGOPS*.

[109] O. Ruwase and M. Lam. A practical dynamic buffer overflow detector. In *Proceedings of the Network and Distributed System Security (NDSS) Symposium*, pages 159–169, February 2004.

[110] S. Graf and H. Saidi. Construction of Abstract State Graphs with PVS. In O. Grumberg, editor, *Proc. 9th INternational Conference on Computer Aided Verification (CAV'97)*, volume 1254, pages 72–83. Springer Verlag, 1997.

[111] S. M. German and B. Wegbreit. A Synthesizer of Inductive Assertions. *IEEE Transactions on Software Engineering*, 1(1):68–75, 1975.

[112] A. Sasturkar, R. Agarwal, L. Wang, and S. D. Stoller. Automated type-based analaysis of data races and atomicity. In *Proceedings of the Tenth ACM/SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2005.

[113] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. ERASER: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[114] M. G. Schultz, E. Eskin, E. Zadok, M. Bhattacharyya, and S. J. Stolfo. MEF: Malicious Email Filter — A UNIX Mail Filter that Detects Malicious Windows Executables. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*, pages 245–252, Boston, MA, June 2001. (**Won best student paper award**).

[115] M. G. Schultz, E. Eskin, E. Zadok, and S. J. Stolfo. Data mining methods for detection of new malicious executables. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 2001.

[116] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting Format String Vulnerabilities with Type Qualifiers. In *Proceedings of the 10th Usenix Security Symposium*, Washington, D.C., August 2001.

[117] J. S. Shapiro and S. Weber. Verifying the EROS Confinement Mechanism. In *RSP: 21th IEEE Computer Society Symposium on Research in Security and Privacy*, 2000.

[118] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, 1985.

[119] A. Srivastava and A. Eustace. Atom: A system for building customized program analysis tools. *SIGPLAN Not.*, 39(4):528–539, 2004.

[120] N. Sterling. Warlock: A static data race analysis tool. In *Proceedings of the USENIX Winter Technical Conference*, pages 97–106, San Diego, CA, January 1993.

[121] S. D. Stoller. Model-checking multi-threaded distributed Java programs. *International Journal on Software Tools for Technology Transfer*, 4(1):71–91, October 2002.

[122] Sun Microsystems. LCLint Online Resource. `http://docs.sun.com/source/806-3567/lint.html`.

[123] T. Y. Meng. *Formal Specification Techniques for Engineering Modular C*, volume 1. Kluwer International Series in Software Engineering, Boston, 1995.

[124] H. Tews, H. Härtig, and M. Hohmuth. VFiasco — Towards a Provably Correct μ-Kernel. Technical Report TUD–FI01–1, TU Dresden, January 2001. http://os.inf.tu-dresden.de/fiasco/doc.html.

[125] The Spin Project Homepage. Static Analysis Tools for C code. www.spinroot.com/static/.

[126] P. Uppuluri. Pattern Matching Based Intrusion Detection Systems. Technical Report RPE report, State University of New York at Stony Brook, Computer Science Department, August 2001. http://seclab.cs.sunysb.edu/~prem/rpe.ps.

[127] VERITAS Software. VERITAS File Server Edition Performance Brief: A PostMark 1.11 Benchmark Comparison. Technical report, Veritas Software Corporation, June 1999. http://eval.veritas.com/webfiles/docs/fsedition-postmark.pdf.

[128] J. Voas, L. Morrel, and K. Miller. Predicting where faults can hide from testing. *IEEE Software*, 8(2):41–48, 1991.

[129] B. Wegbreit. The synthesis of loop predicates. *Commun. ACM*, 17(2):102–113, 1974.

[130] Xerox Parc. Metaobject Protocols. http://www2.parc.com/csl/groups/sda/projects/mops/.

[131] Y. Xie, A. Chou, and D. Engler. Archer: Using symbolic, path-sensitive analysis to detect memory access errors. *SIGSOFT Softw. Eng. Notes*, 28(5):327–336, 2003.

[132] W. Xu, D. C. DuVarney, and R. Sekar. An efficient and backwards-compatible transformation to ensure memory safety of c programs. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering*, pages 117–126, New York, NY, USA, 2004. ACM Press.

[133] E. Zadok. Stackable file systems as a security tool. Technical Report CUCS-036-99, Computer Science Department, Columbia University, December 1999. www.cs.columbia.edu/~library.

[134] E. Zadok, J. M. Anderson, I. Bădulescu, and J. Nieh. Fast Indexing: Support for size-changing algorithms in stackable file systems. In *Proceedings of the Annual USENIX Technical Conference*, pages 289–304, Boston, MA, June 2001.

[135] E. Zadok, I. Bădulescu, and A. Shender. Extending file systems using stackable templates. In *Proceedings of the Annual USENIX Technical Conference*, pages 57–70, Monterey, CA, June 1999.

[136] E. Zadok and J. Nieh. FiST: A Language for Stackable File Systems. In *Proceedings of the Annual USENIX Technical Conference*, pages 55–70, San Diego, CA, June 2000.

[137] X. Zhang, A. Edwards, and T. Jaeger. Using cqual for static analysis of authorization hook placement. In *Proceedings of the 11th USENIX Security Symposium*, pages 33–48, Berkeley, CA, USA, 2002. USENIX Association.