Symbolic Object Code Analysis

Model Checking Pointer Safety in Compiled Programs

Jan Tobias Mühlberg
muehlber@cs.york.ac.uk

Ph.D. Thesis

This thesis is submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy.

The University of York
Programming Languages and Systems Group
Department of Computer Science
United Kingdom

30th September 2009

Supervisors: Gerald Lüttgen and Richard Paige

Examiners: Jim Woodcock and Daniel Kroening

Abstract

This thesis introduces a novel technique for the automated analysis of compiled programs, which is focused on, but not restricted to, pointer safety properties. Our approach, which we refer to as *Symbolic Object Code Analysis* (SOCA), employs bounded symbolic execution, and uses an SMT solver as execution and verification engine. Analysing the object code enables us to bypass limitations of other software model checkers with respect to the accepted input language, so that analysing code sections written in inline assembly does not represent a barrier for us. Our technique is especially designed for programs employing complex heap-allocated data structures and provides full counterexample paths for each error found. In difference to other verification techniques, our approach requires only a bare minimum of manual modelling efforts. While generating counterexamples is often impossible for static analysis techniques due to precision loss in join and widening operations, traditional model checking requires the manual construction of models or the use of techniques such as predicate abstraction which do not work well in the presence of heap-allocated data structures. Hence, symbolic execution is our method of choice over static analysis and model checking.

We also present the SOCA Verifier as a prototypical implementation of our technique. We show that the SOCA Verifier performs competitively with state-of-the-art software model checkers with respect to error detection and false positive rates. Despite only employing path-sensitive and heap-aware program slicing, the SOCA Verifier is further shown to scale well in an extensive evaluation using 250 Linux device drivers. An in-depth case study on the Linux Virtual File System illustrates that the SOCA technique can be applied to verify program properties beyond pointer safety. Our evaluation testifies SOCA's suitability as an effective and efficient bug-finding tool.

Extended Abstract

A major challenge in validating and verifying complex software systems lies in the proper analysis of pointer operations: if a program dereferences a pointer pointing to an invalid memory cell, the program may either crash or behave in an undefined way. Writing software that is free of such errors is difficult since many pointer safety problems result in program crashes at later points in program execution. Hence, the statement causing a memory corruption may not be easily identifiable using conventional testing techniques. On the other hand, automated means of program analysis and program verification either do not cover pointer safety or are often not applicable due to limitations regarding the programming language a program to be analysed may be written in.

A major disadvantage of today's software verification tools, regarding the ability to correctly handle pointer operations, results from the tools being restricted to the analysis of the source code of a given program. Source-code-based tools usually ignore powerful programming constructs such as pointer arithmetic, pointer aliasing, function pointers and computed jumps. Furthermore, they suffer from not being able to consider the effects of program components that are not available in the desired form of source code. Functions linked from libraries and the use of multiple programming languages including inlined assembly code are common examples to this. In addition, many pointer safety problems exist because of platform-specific and compiler-specific details such as memory layout, padding between structure fields and offsets.

In this thesis we introduce a novel technique for the automated analysis of compiled programs, which is focused on, but not restricted to, pointer safety properties. Our approach, which we refer to as *Symbolic Object Code Analysis* (SOCA), employs bounded symbolic execution, and uses an SMT solver as execution and verification engine. Analysing the object code enables us to bypass limitations of other software model checkers with respect to the accepted input language, so that analysing code sections written in inline assembly does not represent a barrier for us. Our technique is especially designed for programs employing complex heap-allocated data structures and provides full counterexample paths for each error found. In differ-

ence to other verification techniques, our approach requires only a bare minimum of manual modelling efforts, namely the abstract specification of a program's execution context that symbolically specifies input and initial heap content. While generating counterexamples is often impossible for static analysis techniques due to precision loss in join and widening operations, traditional model checking requires the manual construction of models or the use of techniques such as predicate abstraction which do not work well in the presence of heap-allocated data structures. Hence, symbolic execution is our method of choice over static analysis and model checking.

The thesis also introduces the SOCA Verifier as a prototypical implementation of our technique. Using the Verisec benchmark suite we show that the SOCA Verifier performs competitively with state-of-the-art software model checkers in respect to error detection and false positive rates. Despite only employing path-sensitive and heap-aware program slicing, the SOCA Verifier is further shown to scale well in an extensive evaluation using 250 Linux device drivers. In an in-depth case study on the Linux Virtual File System implementation we illustrate that the SOCA technique can be applied to verify program properties beyond pointer safety. Our evaluation testifies SOCA's suitability as an effective and efficient bug-finding tool during the development of operating system components.

Contents

Abstract **1**

Extended Abstract **2**

Contents **4**

List of Tables **6**

List of Figures **7**

Acknowledgements **8**

Author's Declaration **9**

1 Introduction **11**
1.1 Defects in Operating Systems 12
1.2 This Thesis 13

2 Background and Related Work **16**
2.1 Common Defects in Operating Systems 16
2.2 Finding Bugs in Device Drivers 18
2.2.1 Runtime analysis 18
2.3 Static Analysis and
Software Model Checking 19
2.3.1 Analysing Pointer Programs 19
2.3.2 Aliasing 21
2.3.3 Abstraction and Partial Order Techniques 22
2.3.4 Software Model Checking 24
2.3.5 Object Code Verification vs.
Source Code Verification 28

3 Evaluation of Existing Software Model Checkers **30**
3.1 Checking Memory Safety with Blast 32
3.1.1 The I2O Use-After-Free Error 33
3.1.2 Verification With a
Temporal Safety Specification 34
3.1.3 Verification Without a
Temporal Safety Specification 37
3.1.4 Blast and Pointers 38

3.1.5 Results . . . 39
3.2 Checking Locking Properties with Blast . . . 40
3.3 Summary of Results . . . 43

4 Symbolic Object Code Analysis 45
4.1 Background . . . 46
4.2 Valgrind's Intermediate Representation Language . . . 49
4.2.1 A semantics for Valgrind's IR . . . 53
4.3 Symbolic Execution . . . 61
4.4 Complications and Optimisations . . . 70
4.5 Experimental Results . . . 72
4.5.1 Tool Development . . . 73
4.5.2 Small Benchmarks: Verisec . . . 74
4.5.3 Large-Scale Benchmarks: Linux Device Drivers . . . 77

5 Beyond Pointer Safety: The Linux Virtual File System 81
5.1 The Linux Virtual File System . . . 82
5.2 VFS Execution Environment and Properties . . . 84
5.3 Applying the SOCA Verifier to the VFS . . . 86
5.4 Evaluating the Effectiveness of SOCA . . . 90
5.4.1 Choice of VFS versions . . . 90
5.4.2 Case study setup . . . 91
5.4.3 Results . . . 92
5.5 Related Work on File System Verification . . . 93

6 Summary and Conclusions 95
6.1 Conclusions . . . 97
6.2 Open Issues and Future Work . . . 98

Bibliography 100

Author Index 108

List of Tables

2.1 Results of an empirical study of operating system errors [Chou et al., 2001] 17

3.1 Result summary for memory safety properties 40
3.2 Result summary for locking properties properties 43

4.1 Detection rate $R(d)$, false positive rate $R(f)$ and discrimination rate $R(\neg f|d)$ for SatAbs, LoopFrog and SOCA 75
4.2 Performance statistics for the Verisec suite 77
4.3 Performance statistics for the Kernel modules 80

5.1 Experimental Results I: Code statistics by VFS function analysed . . 87
5.2 Experimental Results II: SOCA Verifier statistics 87
5.3 Experimental Results III: Yices statistics 88
5.4 Experimental Results IV: Evaluating the effectiveness of SOCA . . . 92

List of Figures

3.1 Extract of `drivers/message/i2o/pci.c`. 33
3.2 A temporal safety specification for `pci.c`. 35
3.3 Manual simplification of `pci.c`. 36
3.4 An example for pointer passing. 38
3.5 A temporal safety specification for spinlocks. 41
3.6 Extract of `drivers/scsi/libata-scsi.c`. 42

4.1 Overview of the Linux OS kernel 46
4.2 Example for pointer aliasing in C: `e_loop.c` 48
4.3 Output of the program given in Fig. 4.2 compiled with *(a)* gcc version 4.1.2 (left) and *(b,c)* gcc version 4.3.1 (right). 48
4.4 Excerpt of the disassembled code from Fig. 4.3.*b*. 48
4.5 Basic syntax of Valgrind's IR language 50
4.6 First two instructions of Fig. 4.4 and their respective IR instructions. 51
4.7 Valgrind IR: `EFLAGS` usage. 52
4.8 Illustration of the SOCA technique. 62
4.8 Illustration of the SOCA technique. 63
4.8 Illustration of the SOCA technique. 64
4.9 Software architecture of the SOCA Verifier 73
4.10 Performance results for the Verisec suite. Left: *(a)* numbers of test cases verified by time. Right: *(b)* numbers of constraint systems solved by time. 76
4.11 Performance results for the Kernel modules. Left: *(a)* numbers of test cases verified by time. Right: *(b)* numbers of constraint systems solved by time. 79

5.1 VFS environment and data structures, where arcs denote pointers. . . 82

Acknowledgements

This thesis would not have been possible without the support of my supervisors, many people at the University of York, and all my friends and family members around the world. I am grateful to Gerald Lüttgen who provided assistance in numerous ways; also to Jim Woodcock who was available for discussions during the last four years, and who finally examined this thesis together with Daniel Kroening. I also thank Andy Galloway and Radu I. Siminiceanu for their contributions to our previous work on the Linux VFS and for their comments on many other parts of the research presented here. Last, but not least, I like to thank the anonymous reviewers of FMICS 2006, SBMF 2009 and VMCAI 2009 for their valuable comments on those parts of this thesis which are already published.

It is a pleasure to thank Filo, Pauline, Mandy and Bob for their help with all sorts of organisational tasks in York. My thanks also go to the technical staff of the Computer Science department who were always very helpful and provided a lot of assistance for this thesis.

I owe my deepest gratitude to Katina, Marlene, Heidrun and Wolf-Diethard, just for being there, for enduring my changing moods and for still caring about me, even when I am sometimes not in touch for weeks. Maybe longer. For cooking for me and for complaining about me being messy, for making me wake up every couple of hours at night and for still loving me.

There is a long list of people who helped me a lot to survive and to go on. First of all, there are my ex-house-mates Tanja, Mariagrazia, Gitta and Adrianne with whom I had a great time in York. My very special thanks go to Juan Perna and Nada Seva who taught me to walk on my balls and into the floor. I practice hard but I think I still did not get it quite right. Most of the other people I got to know during the past four years, and who became good friends, are somehow divided into two groups: those who walk on their balls and those who don't. They are Alexia, Anja, Angie, Anna, Berna, Christian, Eleni, Katti, Leo, Malihe, Marek, Maressa, Markus, Marta, Michelle, Pierre, Radhika, Rick, Sandra, Silvana, 2×Silvia, Srdjan, Tatjana, 2×Tom and Uli. Honestly, I would not have submitted this thesis without you dragging me away from my desk, visiting me regularly, giving me reasons to travel and telling me that there's something besides work. I hope we stay in touch and see each other from time to time.

Author's Declaration

Parts of this thesis have been previously published in:

Mühlberg, J. T. and Lüttgen, G. (2010). Symbolic object code analysis. Technical report appeared in "Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik" 85/2010, Faculty of Information Systems and Applied Computer Science, The University of Bamberg. ISSN 0937-3349.

Mühlberg, J. T. and Lüttgen, G. (2009). Verifying compiled file system code. In *SBMF '09*, volume 5902 of *LNCS*, pages 306–320, Heidelberg. Springer.

Galloway, A., Lüttgen, G., Mühlberg, J. T., and Siminiceanu, R. (2009). Model-checking the Linux Virtual File System. In *VMCAI '09*, volume 5403 of *LNCS*, pages 74–88, Heidelberg. Springer.

Galloway, A., Mühlberg, J. T., Siminiceanu, R., and Lüttgen, G. (2007). Model-checking part of a Linux file system. Technical Report YCS-2007-423, Department of Computer Science, University of York, UK.

Mühlberg, J. T. and Lüttgen, G. (2007). BLASTing Linux Code. Technical Report YCS-2007-417, Department of Computer Science, University of York, UK.

Mühlberg, J. T. and Lüttgen, G. (2006). BLASTing Linux code. In *FMICS '06*, volume 4346 of *LNCS*, pages 211–226, Heidelberg. Springer.

This thesis has not previously been accepted in substance for any degree and is not being concurrently submitted in candidature for any degree other than Doctor of Philosophy of the University of York. This thesis is the result of my own investigations, except where otherwise stated. Other sources are acknowledged by explicit references.

I hereby give consent for my thesis, if accepted, to be made available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

Signed ....

Date ....

Chapter 1

Introduction

During the past decades, safety and security of computer programs has become an increasingly important issue. Indeed, more and more problems arise from the high complexity of modern software systems and the difficulties of locating subtle errors in them. Common software defects such as buffer overflows and deadlocks decrease system's reliability, rendering them unusable as components of dependable systems. Frequently, these defects also have security implications.

In recent years, automated approaches to discover errors in software components via runtime checks or source code analysis have been explored. However, a major challenge in validating and verifying complex software systems remains in the thorough handling of pointer operations. While programming errors related to *pointer safety* – e.g. dereferencing null-pointers, buffer overflows or accessing memory that has already been deallocated – are a common source of software failures, these errors frequently remain undiscovered until they are exploited in security attacks or are found "by accident". One reason for this is that accessing invalid pointers often results in memory corruption, which may lead to undefined behaviour or program crashes at later points in program execution. Hence, the statement causing such an error may not be easily identifiable using conventional testing techniques. However, automated means of program analysis and program verification either do not cover pointer safety or are often not applicable due to limitations regarding the programming language a program to be analysed may be written in.

Especially in the development of operating system components such as device drivers, memory safety problems have disastrous consequences. They render all application level programs relying on the operating system unsafe and give way to serious security problems. Recent literature shows that a majority of all errors found in device drivers are related to memory safety issues [Chou et al., 2001]. Despite being relatively small in size, device drivers represent an interesting challenge as they implement hardware access and are usually written in a mixture of C code and inlined assembly. This combination makes them particularly hard to test and to

analyse with currently available testing and verification tools.

A big disadvantage of today's verification tools regarding the ability to analyse operating system components for pointer safety issues results mainly from being restricted to the analysis of a program's source code. Source-code-based tools usually ignore powerful programming constructs such as pointer arithmetic, pointer aliasing, function pointers and computed jumps. Furthermore these tools suffer from not being able to consider the effects of program components that are not available in the desired form of source code. Functions linked from libraries and the use of multiple programming languages including inlined-assembly are a common examples for this. In addition, many memory safety problems exist because of platform-specific and compiler-specific details such as the memory-layout, padding between structure fields and offsets [Balakrishnan et al., 2008]. Thus, software model checking tools such as SLAM/SDV [Ball and Rajamani, 2001] and others assume either that the program under consideration "does not have wild pointers" [Ball et al., 2006] or, as we explain in Chapter 3 along the lines of a case study on BLAST [Henzinger et al., 2002a], perform poorly for memory safety issues. We also show that these tools usually require substantial manual simplification of the source code of a program to be analysed in order to work around unsupported language features. Hence, many available tools are hard to use by practitioners during software development.

1.1 Defects in Operating Systems

Due to their complicated task of managing a system's physical resources, operating systems are difficult to develop and even more difficult to debug. As recent publications show, most defects causing operating systems to crash are not in the system's kernel but in the large number of operating system extensions available [Chou et al., 2001; Swift et al., 2005]. In Windows XP, for example, 85% of reported failures are caused by errors in device drivers [Ball, 2005]. As [Chou et al., 2001] explains, the situation is similar for Linux and FreeBSD. Error rates reported for device drivers are up to seven times higher than error rates stated for the core components of these operating systems. However, errors in kernel extensions such as device drivers affect the whole operating system and hence have deep impact on the reliability of programs at application level.

There are several reasons for the high number of errors in device drivers. Firstly, a device driver is a nondeterministic reactive system. It continuously responds to different events, e.g., user requests and hardware interrupts. For these events, neither order nor time of occurrence are predictable in advance. Furthermore, operating systems are often required to provide timely responses to events. To do this, drivers

must be able to run in a preemptive operating system kernel where the driver's normal operation may be interrupted at any time [Corbet et al., 2005].

Secondly, and as pointed out in [Ball, 2005], drivers run in a highly concurrent environment provided by the operating system. This concurrency is exposed to the driver programmer, who needs to take reasonable means of resource locking in order to enable the driver to safely deal with concurrent calls of its functions. Concurrent operating systems are running in two or more simultaneous threads of control. While these threads perform sequential operations, they dynamically depend on each other and access the same physical resources, often resulting in race conditions.

Thirdly, device drivers are frequently written by developers who are less experienced in using the kernel's interface than those who built the operating system itself [Swift et al., 2005]. As a result, driver developers tend to be unaware of side-effects of the kernel's Application Programming Interface (API), and thereby introduce subtle errors that break the operating system's safety and security, and that are often difficult to locate. All this renders driver development rather difficult.

The current practice of finding memory safety related bugs in device driver development is debugging and testing. However, the state-of-the-art in research on software development lies in verification techniques such as static analysis and software model checking. By having the potential of being exhaustive and fully automatic, these methods allow errors to be detected early, with reduced effort, and provide a high level of confidence in the safety of a program with respect to a given property.

1.2 This Thesis

This thesis deals with the problem of finding software defects related to pointer safety in computer programs. By the term "pointer safety" we mean that a given program does not violate basic safety rules of the involved programming interfaces by de-referencing invalid pointers, exceeding boundaries of memory structures or calling de-allocation functions in an erroneous context.

In Chapter 3 we evaluate, via case studies and from a practitioner's point of view, the utility of the popular software model checker Blast for revealing errors in Linux kernel code. The emphasis is on memory safety in and locking behaviour of device drivers. Our case studies show that, while Blast's abstraction and refinement techniques are efficient and powerful, the tool has deficiencies regarding usability and support for analysing pointers. These limitations are likely to prevent kernel developers from using Blast.

Motivated by the case study on Blast, we present a novel approach to identifying violations of pointer safety properties in compiled programs in Chapter 4.

Our research hypothesis is that symbolic execution of object code programs is a feasible verification technique with respect to pointer safety properties. Our technique, *Symbolic Object Code Analysis* (SOCA), is based on bounded path-sensitive symbolic execution of compiled and linked programs. More precisely, we translate a given program path-wise into systems of bit-vector constraint that leave the input to the program largely unspecified. The analysis has to be bounded since the total number of paths as well as the number of instructions per path is potentially infinite. In order to deal with the vast amount of instructions available in today's CPUs, we decided to base our analysis on an intermediate representation borrowed from the Valgrind binary instrumentation framework [Nethercote and Fitzhardinge, 2004]. We employ the Yices SMT solver [Dutertre and de Moura, 2006] to check the satisfiability of the generated constraints systems. Our approach allows us to express a range of memory safety properties as simple assertions on constraint systems. In contrast to other methods for finding pointer safety violations, our technique does not employ program abstraction. The program's input and initial heap content is initially left unspecified in order to allow the SMT solver to search for inputs that will drive the program into an error state.

For evaluating our work, we present a prototypical implementation of the SOCA technique for programs in ELF format [Tool Interface Standards (TIS) Committee, 1995] compiled for the 32-bit Intel Architecture (IA32, [Intel Corporation, 2009]), which we apply to the Verisec benchmark suite [Ku et al., 2007]. As we explain in Section 4.5.2, Verisec consists of 298 test cases for buffer overflow vulnerabilities taken from various open source programs. Our results show that the SOCA Verifier performs competitively with state-of-the-art software model checkers with respect to error detection and false positive rates.

We chose Linux device drivers as our application domain for large-scale evaluation of the SOCA Verifier. The Linux operating system kernel consists of a freely available, large and complex code base implementing key tasks such as process management, memory management, file system access, device control and networking for about 20 different computer architectures. It features a relatively small monolithic core of components such as the scheduler and the memory management subsystem. However, the majority of its functionality is implemented in terms of *kernel modules* or *device drivers* that can be built separately from the kernel and loaded at runtime. These modular components of the Linux kernel are responsible, for example, for making a particular physical device attached to the computer respond to a well-defined internal programming interface. Hence, user access to this device can be performed by means of standardised functions, the System Call Interface, which are independent of the particular driver or device. Kernel modules amount to roughly

two-thirds (≈200 MBytes of code) of the entire Linux kernel distribution.

We present the results of an extensive case study on applying the SOCA Verifier to 9296 functions taken from 250 device drivers compiled for IA32, which is the hardware platform for which the majority of drivers of recent Linux kernel distributions can be compiled. By being able to successfully analyse 95% of this sample of functions and revealing a total of 887 program locations at which a null-pointer may be dereferenced, our experimental results show that the SOCA Verifier scales well for that particular application domain. Our bounded symbolic analysis approach is even able to achieve exhaustiveness for 27.8% of the sample, while for the remaining functions it was possible to perform the analysis until bounds were exhausted.

In Chapter 5 we present a case study on retrospective verification of the Linux Virtual File System (VFS). Since VFS maintains dynamic data structures and is written in a mixture of C and inlined assembly, modern software model checkers cannot be applied. We demonstrate that the SOCA technique can be utilised to check for violations of API usage rules regarding commonly used locking mechanisms of the Linux kernel. Despite not considering concurrent executions of the VFS functions, we demonstrate that our technique can be applied to check program properties clearly beyond pointer safety issues. Our results show that the SOCA Verifier is capable of reliably and efficiently analysing complex operating system components such as the Linux VFS, thereby going beyond traditional testing tools and into semantic niches that current software model checkers do not reach. This testifies the SOCA Verifier's suitability as an effective and efficient bug-finding tool during the development of operating system components.

The thesis is summarised and concluded in Chapter 6. We also outline open issues and future work in this chapter.

Chapter 2

Background and Related Work

In this chapter we outline open issues, ongoing research, techniques and tools for the analysis of computer programs. We centre on the verification and testing of pointer programs, especially operating system components, alias analysis, software model checking and abstraction techniques.

2.1 Common Defects in Operating Systems

There are a large number of commonly found operating system errors. An insightful study on this topic has been published in [Chou et al., 2001]; see Table 2.1 for a summary of its results. The authors of [Chou et al., 2001] highlight that most errors are related to problems causing either deadlock conditions or driving the system into undefined states by de-referencing invalid pointers. While problems resulting in deadlock conditions are well covered by several formal software engineering tools such as SLAM/SDV [Ball and Rajamani, 2001], an industry strength software model checker for Microsoft Windows device drivers, memory safety remains a major issue. Likewise our case study on the Blast software verification tool (Blast, cf. Chapter 3) comes to the result that this state-of-the-art verification toolkit does not cover pointer and memory safety in full.

Although memory safety problems have a direct impact on an operating system's reliability, API safety rules for operating system kernels are usually described in an informal way. For example, it is stated in the Linux device driver handbook [Corbet et al., 2005, p. 61] that one "should never pass anything to *kfree* that was not obtained from *kmalloc*" since, otherwise, the system may behave in an undefined way. As a result of this, an exhaustive set of safety rules that can be used as

% of Bugs	Rule checked
63.1%	**Bugs related to memory safety**
38.1%	Check potentially `NULL` pointers returned from routines.
9.9%	Do not allocate large stack variables (> 1K) on the fixed-size kernel stack.
6.7%	Do not make inconsistent assumptions about whether a pointer is `NULL`.
5.3%	Always check bounds of array indices and loop bounds derived from user data.
1.7%	Do not use freed memory.
1.1%	Do not leak memory by updating pointers with potentially `NULL` realloc return values.
0.3%	Allocate enough memory to hold the type for which you are allocating.
33.7%	**Bugs related to locking behaviour**
28.6%	To avoid deadlock, do not call blocking functions with interrupts disabled or a spinlock held.
2.6%	Restore disabled interrupts.
2.5%	Release acquired locks; do not double-acquire locks.
3.1%	**Miscellaneous bugs**
2.4%	Do not use floating point in the kernel.
0.7%	Do not de-reference user pointers.

Table 2.1: Results of an empirical study of operating system errors [Chou et al., 2001]

properties in static program verification is hard to identify. Nevertheless, analysis techniques such as [Engler et al., 2000] have been applied to open-source operating systems, identifying hundreds of bugs related to memory safety based on restricted sets of such rules.

Correct locking of resources is another major issue causing problems in operating system code. As shown in [Chou et al., 2001], deficiencies resulting in deadlocks in the Linux and BSD kernels make up a large amount of the overall number of errors found. In the documentation explaining the API of the Linux kernel, quite strict rules about the proper use of functions to lock various resources are stated. For example, in [Corbet et al., 2005, p.121], one of the most basic rules is given as follows: "Neither semaphores nor spinlocks allow a lock holder to acquire the lock a second time; should you attempt to do so, things simply hang." The rational for this lies in the functionality provided by spinlocks: a kernel thread holding a lock is spinning on one CPU and cannot be preempted until the lock is released. Another important rule is that any code holding a spinlock cannot relinquish the processor for anything except for serving interrupts; especially, the thread must never sleep because the lock might never be released in this case [Corbet et al., 2005, p.118].

2.2 Finding Bugs in Device Drivers

The problem of locating programming errors in device drivers is mainly addressed in testing. However, there are two main difficulties that limit a driver's testability. As Ball et al. state in [Ball et al., 2006], these are related to the restricted observability inside operating system kernels and to the limited chances of achieving a high test coverage using traditional testing techniques. Ball et al. point out that, for example, the Windows operating system provides several different kernel-level programming interfaces, "which gives rise to many ways in which a driver can misuse these APIs". Most of the Application Programming Interface violations do rarely result in immediate failures but leave the operating system in an inconsistent state. This may be a crash or improper behaviour at a later time, mostly without revealing the source of the error.

2.2.1 Runtime analysis

While popular runtime analysis tools that target memory safety problems are mainly available for the development of software at the application level, the large domain of operating system kernels and device drivers is rarely covered. Toolkits such as Purify [Purify, 2009] and Valgrind [Valgrind, 2009] provide debugger-like runtime environments that observe the memory access of an application program under consideration. While these tools can deal with concurrency issues and unbounded allocations, they are not meant for automatic and exhaustive code inspection: In order to find problems, the program needs to be run with a set of test cases or tested manually. This results in the fact that erroneous program behaviour may not be revealed due to a lack of coverage. Furthermore, the use of the extensive profiling support slows program execution down by a factor between 20 and 100. Also the Electric Fence [Perens, 2009] library provides an additional runtime environment by linking a program against it. Electric Fence replaces standard functions for allocation and de-allocation with customised versions that perform additional runtime checks.

On the kernel level, tools such as "kmdb" [KMDB, 2009] for Solaris or the Novell Linux Kernel Debugger [NLKD, 2008] provide an extensive analysis and testing framework for software development. As for the above tools, they are neither automatic nor exhaustive.

The major advantages of debugging tools lie in their relative efficiency and the fact that they are not operating on a program's source code but directly on the compiled object code. Therefore, testing tools perform better for detecting faults that are closely related to the actual architecture. However, due to the lack of

exhaustiveness, results obtained from software testing are not as strong as those gained from more formal verification approaches such as software model checking [Clarke et al., 2000]. Formal verification can establish a much higher confidence in a program under consideration by assuring that a certain property holds for all possible executions.

2.3 Static Analysis and Software Model Checking

Static analysis is a powerful technique for inspecting source code for bugs. Indeed, hundreds of bugs related to memory safety and erroneous locking behaviour had been detected in Linux device drivers via an approach based on system-specific compiler extensions, known as meta-level compilation [Engler et al., 2000]. This method is implemented in the tool Coverity [Coverity, Inc.] and was used in an extensive study on operating system errors [Chou et al., 2001]. Also most of the examples for memory safety bugs in the Linux kernel analysed in our case study on Blast (cf. Chapter 3) have previously been detected using this technique.

A further recent attempt to find bugs in operating system code is based on abstract interpretation [Cousot and Cousot, 2002] and presented in [Breuer and Pickin, 2006]. The authors checked about 700k lines of code taken from recent versions of the Linux kernel for correct locking behaviour. The paper focuses on the kernel's spinlock interface and problems related to sleep under a spinlock. Several new bugs in the Linux kernel were found during the experiments. However, the authors suggest that their approach could be improved by adopting model checking techniques in order to guide the analysis in situations where the current method has to consider all, even unreachable paths within the control flow.

An extensive survey on automated techniques for the formal verification of software, focusing on abstract static analysis, software model checking and bounded software model checking has recently been published in [D'Silva et al., 2008]. On the following pages we focus on approaches and techniques for the analysis and verification of pointer programs.

2.3.1 Analysing Pointer Programs

The automated, static analysis of pointer programs has been an important but still unsolved issue in computer science for more than thirty years. In [Wilhelm et al., 2000], Wilhelm et al. give a summary of questions that should be answered by automatic reasoning about memory structures used by pointer programs:

Null-Pointers. Does a pointer contain the value `NULL` at a certain point in program execution?

Aliasing and Sharing. May two pointer variables point to the same heap cell? Do they always point to the same heap cell? Is more than one pointer component pointing to a certain heap cell?

Reachability. Is a heap cell reachable from any pointer variable or pointer component?

Disjointness. Do allocated data structures have common elements?

Cyclicity. Are heap cells parts of cyclic data structures?

Shape. What do data structures on the heap look like? Can we derive safety properties from regularities in their structure?

The above list is not exhaustive. For example from [Balakrishnan et al., 2008] we can obtain questions that are more related to the security of software systems:

Confidentiality. Does the program leak any sensitive information like keying material or passwords?

Early work on analysing pointer programs goes back to Burstall, who published on "techniques for proving correctness of programs which alter data structures" in 1972 [Burstall, 1972]. In this paper, Burstall introduces a novel kind of assertion called "distinct nonrepeating tree system". This approach utilises a sequence of such assertions where each element of the sequence describes a distinct region of storage [Burstall, 1972][1]. The basic idea of Burstall's work provides a "store-based" operational semantics [Kirchner, 2005] for heap usage by modelling the heap used by a program under consideration as a collection of variables providing a mapping from memory addresses to values. Analysis and Verification are then done by reasoning about this model using Hoare logic [Hoare, 1969]. The approach has been applied in recent research on verifying pointer programs using separation logic with spatial conjunction [Kuncak and Rinard, 2004; Reynolds, 2000, 2002] and on proof automation by providing integration in existing theorem proving infrastructures [Mehta and Nipkow, 2005]. Techniques based on store-based semantics have several advantages. Firstly, they are very natural because they correspond closely to the architecture of current computer hardware, operating systems, as well as imperative programming languages that allow the direct manipulation of pointers. Furthermore, store-based techniques can be assumed to scale well to large programs because it is possible to

[1] As cited by Reynolds in [Reynolds, 2000].

compute the effect of procedures on the global heap from their effect on sub-heaps [Rinetzky et al., 2005].

However, with the emergence of programming languages such as Java, store-less semantics for heap access have been developed [Bozga et al., 2003]. By abstracting away from specific memory addresses, these heap representations provide a conceptual and compact view on the memory usage of a program.

2.3.2 Aliasing

Identifying sharing relationships between memory cells and variables in computer programs is the central problem to be solved in order to answer most of the above questions. Thus, alias analysis is a wide research area. Several generic shape graph-based approaches for performing shape analysis for imperative programs have been published [Sagiv et al., 1998; Wilhelm et al., 2000]. However, most practical work on this topic has been conducted by the compiler construction and optimisation community. In order to give a simple systematics for these approaches, we distinguish between algorithms based on source code analysis and those working on executable object code.

Analysing source code. In [Deutsch, 1992, 1994], Deutsch provides a very exact alias analysis for high-level programs based on a store-less semantics and abstract interpretation (cf. Section 2.3.3). The algorithm can deal with dynamic allocation and de-allocation of heap objects as well as recursive program structures. However, the analysis makes heavy use of explicit data type declarations defining the shape of allocated structures. Therefore, the algorithm is not usable for untyped programming languages or languages that allow pointer arithmetic and unchecked type conversion such as type casts in `C`. Deutsch's work has been extended in several recent publications. In [Venet, 1999], Venet proposes an algorithm based on Deutsch's research that does not rely on correct type information but works for untyped programs. The core idea behind this algorithm is to represent access paths within data structures as finite-state automata. Alias pairs are then described using numerical constraints on the number of times each transition of an automaton may be used. However, pointer arithmetic remains an unsolved issue in all approaches on alias analysis for high-level programs. The problem is partially covered by algorithms such as the one proposed by Wilson and Lam in [Wilson and Lam, 1995], but makes conservative assumptions about aliasing for several cases in which the analysis will fail.

Recently, compositional approaches [Calcagno et al., 2009; Yang et al., 2007] to shape analysis [Wilhelm et al., 2000] for proving pointer safety have been proposed.

However, all available work in this area is based on analysing the source code of a program under consideration. Hence, calls to library functions or switches to another programming language as well as programming constructs such as function pointers are treated as non-deterministic assignments.

A major restriction for pointer analysis techniques based on abstractions of high-level programming languages lies in the control flow of many programs. Since analysis techniques need to follow the program execution in order to trace memory access, program constructs like function pointers, computed jumps and calls of external library functions constrain the practicability of these algorithms.

Analysing object code. The limitations of source code-based algorithms lead to the development of alias analysis techniques that operate on object code. This group of algorithms is of interest for the optimisation of systems that manipulate executable code directly – runtime linkers are an interesting examples for this. In [Debray et al., 1998], Debray et al. introduce a simple and efficient flow-sensitive alias analysis for executable code which has been used link-time optimisation. Despite the fact that this algorithm explicitly sacrifices precision for efficiency in several cases, it can handle complex program flows and pointer arithmetic. In a modified version, Debray's algorithm has also been considered for the use on the intermediate language of the `gcc` compiler family [Gupta and Sharma, 2003]. Another recent approach for a memory analysis algorithm based on the inspection of object code is given by Balakrishnan and Reps in [Balakrishnan and Reps, 2004]. Their algorithm "value-set analysis" uses "an abstract domain for representing over-approximation of the set of values that each data object can hold at each program point". Therefore, the algorithm tracks addresses and integer values simultaneously.

2.3.3 Abstraction and Partial Order Techniques

One of the major limitations of exhaustive verification techniques such as model checking lies in the complexity of modern software systems. While early approaches in model checking aimed on the verification of the alternating bit protocol with 20 states [Clarke et al., 1983], current software systems, especially in the domain of operating system verification, are infinite-state systems. Constructing their state space leads to the state explosion problem as explained by Godefroid in [Godefroid, 1994]. Therefore, model checking such systems requires the use of efficient data structures for storing and manipulating large sets of states, as well as automatic techniques that reduce a systems state space by abstracting away from unneeded

details [Clarke et al., 1992].

Predicate abstraction. Most abstraction techniques currently used in software model checking are based on the work of Graf and Saïdi in [Graf and Hassen Saïdi, 1997]. The author's approach employs abstract interpretation [Cousot, 1996] to compute program invariants in order to map the concrete states of a system to abstract states according to their evaluation under a finite set of predicates. This results in reducing an infinite-state model under consideration to a finite-state one, in which, for example, boolean variables correspond to assertions over the concrete model.

Recently, algorithms performing predicate abstraction directly on the source code of a program under consideration have been developed [Ball et al., 2001; Henzinger et al., 2002b] and implemented in tools such as SLAM [Ball and Rajamani, 2001] and Blast [Henzinger et al., 2002a]. Despite the fact that these algorithms and tools provide a valuable contribution to the field of static source code analysis, their capabilities are limited by not covering the problem of memory safety in full. This is mainly because of unspecified constructs in high-level programming languages and the use of function pointers and computed jumps, which are decided at compile-time or runtime. Furthermore, the aliasing problem has a deep impact on such analysis techniques. As we show in a case study on the BLAST toolkit provided in Chapter 3, this exemplarily but state-of-the-art tool does not provide sufficient facilities for tracking values that are passed in a call-by-reference manner to functions without manually instrumenting the program or providing additional alias information. The techniques also turned out to be inapplicable for keeping track of unbounded numbers of allocations and concurrent program flow.

Partial order techniques. Verification techniques based on state space exploration are limited by the excessive size of the state space. Especially for modelling concurrency the state explosion problem has a high impact because one has to consider interleaving program executions. However, one can assume that many interleavings of concurrent events corresponding to the same execution contain related information. Therefore, model checking or simulating all interleavings possible in a program under consideration may not be required. This has been discussed by the model checking community under the term "partial-order methods" as a technique that reduces the impact of the state-explosion problem [Godefroid, 1994]. The intuition behind these techniques is that instead of exploring all interleaving executions only a part of the state space is explored. This part is chosen in a way that makes it provably sufficient to check a given property. Partial order techniques have been

implemented in model checking frameworks such as Spin [Holzmann, 2003] for communication protocols as well as VeriSoft for verifying software systems [Chandra et al., 2002; Godefroid, 1997]. The VeriSoft approach is particularly interesting. As summarised in [Chandra et al., 2002] its focus lies on verifying communication related properties in concurrent software systems. VeriSoft involves model checking by stateless guided program execution where program runs are chosen nondeterministically.

Furthermore, partial order techniques have also been used in program testing [Gälli et al., 2006, 2004; Memon et al., 2001]. These approaches aim on the reduction of the total amount of test cases by identifying and removing cases, which are already covered by others.

Program slicing. Another important abstraction technique and SOCA ingredient is *path-sensitive* slicing. *Program slicing* was introduced by Weiser [Weiser, 1981] as a technique for automatically selecting only those parts of a program that may affect the values of interest computed at some point of interest. Different to conventional slicing, our slices are computed over a single path instead of an entire program, similar to what has been introduced as *dynamic slicing* in [Korel and Laski, 1990] and *path slicing* in [Jhala and Majumdar, 2005]. In contrast to those approaches, we use conventional slicing criteria and leave a program's input initially unspecified. In addition, while collecting program dependencies is relatively easy at source code level, it becomes difficult at object code level when dependencies to the heap and stack are involved. The technique employed by SOCA for dealing with the program's heap and stack is an adaptation of the *recency abstraction* described in [Balakrishnan and Reps, 2006].

2.3.4 Software Model Checking

By having the potential of being exhaustive and fully automatic, model checking, in combination with abstraction and refinement, is a successful technique used in software verification [Clarke et al., 2000]. Intensive research in this area has resulted in software model checkers like Bandera [Corbett et al., 2000] for Java programs or SLAM/SDV [Ball and Rajamani, 2001], Blast [Henzinger et al., 2002a], SatAbs [Clarke et al., 2005] and CBMC [Clarke et al., 2004] for analysing C source code. The major advantage of these tools over model-based model checkers such as Spin [Holzmann, 2003] is their ability to automatically abstract a model from the source code of a given program. User interaction should then only be necessary in order to provide the model checker with a specification against which the program can be checked. Since complete formal specifications are not available for most programs,

verification will usually be relative to a partial specification that covers the usage rules of the Application Programming Interface used by the program. However, up to now all releases of SLAM are restricted to verifying properties for Microsoft Windows device drivers and do not cover memory safety problems [Microsoft Corporation, 2004], while BLAST is able to verify a program against a user defined temporal safety specification [Henzinger et al., 2002a] and thus allows checking of arbitrary C source code. Such a temporal safety specification in BLAST is a monitor automaton with error locations. It can reflect detailed behavioural properties of the program under consideration. As we will explain in Chapter 3, the BLAST toolkit has several shortcomings related to the detection of memory safety problems and concurrency issues. Recent work [Sery, 2009] shows further that, again in contrast to our SOCA Verifier, BLAST cannot analyse programs with multiplicities of locks since its specification language does not permit the specification of observer automatons for API safety rules with respect to function parameters.

In [Beyer et al., 2005], the use of CCURED [Necula et al., 2005] in combination with software model checking as implemented in BLAST for verifying memory safety of C source code is explained. This is done by inserting additional runtime checks at all places in the code where pointers are de-referenced. BLAST is then employed to check whether the introduced code is reachable or can be removed again. The approach focuses on ensuring that only valid pointers are de-referenced along the execution of a program, which is taken to mean that pointers must not equal `NULL` at any point at which they are de-referenced. However, invalid pointers in C do not necessarily equal `NULL` in practice.

Model checking bytecode and assembly languages. In recent years, several approaches to model checking *compiled programs* by analysing bytecode and assembly code have been presented. In [Visser et al., 2003], *Java PathFinder* (*JPF*) for model checking Java bytecode is introduced. *JPF* generates the state space of a program by monitoring a virtual machine. Model checking is then conducted on the states explored by the virtual machine, employing collapsing techniques and symmetry reduction for efficiently storing states and reducing the size of the state space. These techniques are effective because of the high complexity of *JPF* states and the specific characteristics of the Java memory model. In contrast, the SOCA technique to verifying object code involves relatively simple states and, in difference to Java, the order of data within memory is important in IA32 object code. Similar to *JPF*, *StEAM* [Leven et al., 2004] model checks bytecode compiled for the Internet C Virtual Machine, while *BTOR* [Brummayer et al., 2008] and *[mc]square* [Noll and Schlich, 2008; Schlich and Kowalewski, 2006] are tools for model checking assembly

code for micro-controllers.

All the above tools are explicit model checkers that require the program's entire control flow to be known in advance of the analysis. As we have explained above, this is not feasible in the presence of computed jumps. The SOCA technique has been especially designed to deal with OS components that make extensive use of jump computations.

Furthermore, *BTOR* and *[mc]square* accept assembly code as their input, which may either be obtained during compilation of a program or, as suggested in [Schlich and Kowalewski, 2006], by disassembling a binary program. As shown in [Horspool and Marovac, 1980], the problem of disassembling a binary program is undecidable in general. The SOCA technique focuses on the verification of binary programs without the requirement of disassembling a program at once.

Symbolic Execution and Bounded Model Checking Symbolic execution has been introduced in [King, 1976] as a means of improving program testing by covering a large class of normal executions with one execution in which symbols representing arbitrary values are used as input to the program. A recent approach based on manually instrumenting the source code of a program and then using symbolic execution to derive inputs that make the program crash, has been proposed in [Cadar et al., 2006]. In contrast to our work, [Cadar et al., 2006] relies on manual annotations, is not focused on memory safety, and works at source code level.

Several frameworks for integrating symbolic execution with model checking have recently been presented, including *Symbolic JPF* [Păsăreanu et al., 2008] and *DART* [Godefroid et al., 2005]. *Symbolic JPF* is a successor of the previously mentioned *JPF*. *DART* implements directed and automated random testing to generate test drivers and harness code to simulate a program's environment. The tool accepts C programs and automatically extracts function interfaces from source code. Such an interface is used to seed the analysis with a well-formed random input, which is then mutated by collecting and negating path constraints while symbolically executing the program. Unlike the SOCA Verifier, *DART* handles constraints on integer types only and does not support pointers and data structures.

A bounded model checker for C source code based on symbolic execution and SAT solving is SATURN [Xie and Aiken, 2007]. This tool is specialised on checking locking properties and null-pointer de-references. The authors show that their tool scales for analysing the entire Linux kernel. Unlike the SOCA Verifier, the approach in [Xie and Aiken, 2007] computes function summaries instead of adding the respective code to the control flow, unwinds loops a fixed number of times and does not handle recursion. Hence, it can be expected to produce more unsound results but

scale better than our SOCA technique.

A language agnostic tool in the spirit of *DART* is *SAGE* [Godefroid et al., 2008], which is used internally at Microsoft. *SAGE* works at IA32 instruction level, tracks integer constraints as bit-vectors, and employs machine-code instrumentation in a similar fashion as we do in [Mühlberg and Lüttgen, 2009]. *SAGE* is seeded with a well-formed program input and explores the program space with respect to that input. Branches in the control flow are explored by negating path constraints collected during the initial execution. This differs from our approach since SOCA does not require seeding but explores the program space automatically from a given starting point. The SOCA technique effectively computes program inputs for all paths explored during symbolic execution.

Concolic testing. An area of research closely related to ours is that of *concolic testing* [Kim and Kim, 2009; Sen et al., 2005]. This technique relies on performing concrete execution on random inputs while collecting path constraints along executed paths. The constraints are then used to compute new inputs driving the program along alternative paths. In difference to this approach, SOCA uses symbolic execution to explore all paths and concretises only in order to resolve computed jumps. Concrete execution in SOCA may also be employed to set up the environment for symbolic execution [Mühlberg and Lüttgen, 2009].

Alternative approaches to object code verification. Alternative approaches to proving memory safety, other than the kinds of software model checking discussed in previous sections, are *shape analysis* [Wilhelm et al., 2000] and *separation logic* [Reynolds, 2002]. All recent work in this area [Calcagno et al., 2009; Josh Berdine and O'Hearn, 2005] is based on analysing the source code of a program, and calls to library functions and programming constructs such as function pointers are simply abstracted using non-deterministic assignments.

Techniques applying theorem proving to verify object code and assembly code are presented in [Boyer and Yu, 1996] and [Yu and Shao, 2004]. In [Boyer and Yu, 1996] the Nqthm prover is employed for reasoning about the functional correctness of implementations of well-known algorithms. [Yu and Shao, 2004] proposes a logic-based type system for concurrent assembly code and uses the Coq proof assistant to verify programs. In contrast to our work, both techniques do not support "higher-order code pointers" including return pointers in procedure calls.

2.3.5 Object Code Verification vs. Source Code Verification

While research in programming languages, computer security and software engineering has led to several tools for analysing source code for programming errors, program testing has still one major advantage: It is based on the execution of machine code and not source code.

Shortcomings of Source Code Verification. As Balakrishnan et al. explain in [Balakrishnan et al., 2008], the analysis of source code has several drawbacks. It is pointed out that severe defects in software may be introduced during compilation and optimisation. As an example for this, compiler optimisations may remove write operations to a memory area that occur directly before the area is freed. While this behaviour appears to be reasonable at first glance – the values are never read afterwards and therefore cannot have any impact on the further program execution – it gives rise to confidentiality issues if the memory contained sensitive information. Furthermore, platform-specific details, such as memory-layout details, the positions and offsets of variables, as well as the padding between structure fields or the register usage of a program, are only visible after compilation [Balakrishnan et al., 2008].

More advantages of the use of object code lie in the fact that software components may make use of modules such as libraries that are not available in source code and hence, can only be analysed in object code representation. Also, quite a lot of software is written in more than one programming language, e.g., device drivers often contain inlined assembly, and language switches are rarely supported by verification tools operating on source code. However, for executing a program, all its fragments are transformed into object code, either via compilation or interpretation. Hence, the object code should be considered as a common representation for programs that are written in multiple languages. [Balakrishnan et al., 2008]. Another serious shortcoming of source code verification is that high-level programming languages are often described informally and do not have a formally defined semantics. Therefore, assumptions about undefined programming constructs must be made. However, those assumptions concerning the intended semantics of a high-level language are not necessarily correct [Wahab, 1998].

Related to the verification of memory safety properties is the un-decidability of the aliasing problem. It is impossible to determine syntactically whether a pointer identifies a given variable and to distinguish syntactically between executable and un-executable commands [Wahab, 1998]. The aliasing problem renders many approaches to verify memory safety futile since all source code-based analysis techniques operate on program variables. Today, even industry strength verification

tools such as SDV which is specialised on device drivers, provide sound results only based on the assumption "that the device driver does not have wild pointers" [Ball et al., 2006]. This means that the tool does not check memory safety but assumes it.

Advantages of Object Code-Based Verification. For verifying properties related to memory safety, the object code representation has several advantages. While the analysis of source code is limited by the aliasing problem, object code uses explicit addresses. Since there is no syntactical way of distinguishing between integer values and addresses and the use of indirect addressing in object code, it is considered to be hard to analyse. However, due to explicit addresses and compiler optimisations such as advanced register allocation algorithms, reasoning about memory safety becomes easier. As an example, tracking the contents of registers is a less complex task than tracking arbitrary heap cells or variables [Balakrishnan and Reps, 2005; Xu et al., 2000]. Furthermore, the use of function pointers and computed jumps (`setjmp()` and `longjmp()` in C), which breaks many source code based tools such as Blast and SDV during the analysis of a program's control flow, does not need to be handled in a different way than any other piece of object code.

Object code programs are in the native language of a specific CPU. Since they are executed directly, no further errors may be introduced by a compiler or a runtime environment. However, a processor language consists of a large number of highly specialised instructions [Wahab, 1998], carrying out rather simple actions. This results in the fact that object code programs consist of many more statements than the original high-level program. Hence, in the step of abstracting a model from a program under consideration, i.e. for software model checking, all instructions need to be taken into account. This is because the program in in high-level source code.

In order to analyse an object code program, it needs to be disassembled first. As explained in [van Emmerik, 2003], this step requires the separation of data from code, which is not given in machine code programs. [Horspool and Marovac, 1980] show this problem to be undecidable in general, thus requiring approximation. However, recent work such as [Balakrishnan and Reps, 2005; Xu et al., 2000] demonstrates that acceptable results can be achieved as long as self-modifying programs are not considered.

Despite this we consider analysing the object code representation of programs as a valuable technique for verifying memory safety properties of software systems.

Chapter 3

Evaluation of Existing Software Model Checkers

In this chapter we investigate to which extent software model checking as implemented in Blast (*Berkeley Lazy Abstraction Software verification Tool*, [Henzinger et al., 2002a]) can aid a practitioner during operating system software development. To do so, we analyse whether these tools are able to detect errors that have been reported for recent releases of the Linux kernel. We consider programming errors related to *memory safety* (cf. Section 3.1) and *locking behaviour* (cf. Section 3.2). As pointed out in [Chou et al., 2001] memory safety and incorrect handling of locks are the main reasons for defects found in operating system components. Here, "memory safety" is interpreted as the property that an operating system component never de-references an invalid pointer, since this would cause the program to end up in an undefined state. "Correct locking behaviour" means that functions that ensure mutual exclusion on the physical resources of a system are called in a way that is free of deadlocks and starvation. Both classes of problems are traceable by checking whether an operating system component complies with basic usage rules of the program interface provided by the kernel.

The code examples utilised in this chapter are taken from releases 2.6.13 and 2.6.14 of the Linux kernel. They have been carefully chosen by searching the kernel's change log for fixed memory problems and fixed deadlock conditions, in a way that the underlying problems are representative for memory safety and locking behaviour as well as easily explainable without referring to long source code listings.[1] Our studies use version 2.0 of Blast, which was released in October 2005.

The focus of our work is on showing at what scale a give problem statement and a program's source code need to be adapted in order to detect an error. We

[1] All source code used is either included or referenced by a *commit key* as provided by the source code management system *git* which is used in the Linux kernel community; see `www.kernel.org` for further information on *git* and Linux.

discuss how much work is required to find a certain usage rule violation in a given snippet of a Linux driver, and how difficult this work is to perform in BLAST. Due to space constraints, we cannot present all of our case studies in full here; however, all files necessary to reproduce our results can be downloaded from `http://www.beetzsee.de/blast/`. The majority of this chapter has been previously published in [Mühlberg and Lüttgen, 2007a]. There is also a technical report version [Mühlberg and Lüttgen, 2007b] with additional details available.

The BLAST toolkit The popular BLAST toolkit implements an advanced abstraction algorithm, called "lazy abstraction" [Henzinger et al., 2002b], for building a model of some C source code, and model-checking algorithm for checking whether some specified label placed in the source code is reachable. This label can either be automatically introduced by instrumenting the source with an explicit temporal safety specification, be added via `assert()` statements, or be manually introduced into the source. In any case, the input source file needs to be preprocessed using a standard C preprocessor like `gcc`. In this step, all header and source files included by the input file under consideration are merged into one file. It is this preprocessed source code that is passed to BLAST to construct and verify a model using *predicate abstraction*.

Related Case Studies with BLAST BLAST has been applied for the verification of memory safety as well as locking properties before [Beyer et al., 2005; Henzinger et al., 2004, 2002a, 2003]. In [Beyer et al., 2005], the use of CCURED [Necula et al., 2005] in combination with BLAST for verifying memory safety of C source code is explained. This is done by inserting additional runtime checks at all places in the code where pointers are de-referenced. BLAST is then employed to check whether the introduced code is reachable or can be removed again. The approach focuses on ensuring that only valid pointers are de-referenced along the execution of a program, which is taken to mean that pointers must not equal `NULL` at any point at which they are de-referenced. However, invalid pointers in C do not necessarily equal `NULL` in practice. In contrast to [Beyer et al., 2005], we will interpret pointer invalidity in a more general way and conduct our studies on real-world examples rather than constructed examples.

A methodology for verifying and certifying systems code on a simple locking problem is explained in [Henzinger et al., 2002a], which deals with the spinlock interface provided by the Linux kernel. *Spinlocks* ensure that a kernel process can spin on a CPU without being preempted by another process. The framework studied in [Henzinger et al., 2002a] is used to prove that calls of `spin_lock()` and

spin_unlock() in Linux device drivers always alternate. In contrast to this work, our case studies will be more detailed and thereby will be providing further insights into the usability of BLAST.

Two project reports of graduate students give further details on BLAST's practical use. In [Mong, 2004], Mong applies BLAST to a doubly linked list implementation with dynamic allocation of its elements and verifies correct allocation and de-allocation. The paper explains that BLAST was not powerful enough to keep track of the state of the list, i.e., the number of its elements. Jie and Shivkumar report in [Jie and Shivaji, 2004] on their experience in applying BLAST to a user level implementation of a virtual file system. They focus on verifying correct locking behaviour for data structures of the implementation and were able to successfully verify several test cases and to find one new error. However, in the majority of test cases BLAST failed due to documented limitations, e.g., by not being able to deal with function pointers, or terminated with obscure error messages. Both studies were conducted in 2004 and thus based on version 1.0 of BLAST. As shown in this chapter, BLAST's current version has similar limitations.

A further case study on applying BLAST to a protocol stack is presented in [Kolb et al., 2009], focusing on verifying the correct implementation of three API usage rules in that stack. The authors agree with the limitations of the BLAST toolkit we are pointing out in [Mühlberg and Lüttgen, 2007a] and in this Chapter.

3.1 Checking Memory Safety with BLAST

This section focuses on using BLAST for checking usage rules related to memory safety, for which we have analysed several errors in different device drivers. The examples studied by us include use-after-free errors in the kernel's SCSI[2] and InfiniBand[3] subsystems. The former is the *small computer system interface* standard for attaching peripheral devices to computers, while the latter is an industry standard designed to connect processor nodes and I/O nodes to form a system area network. In each of these examples, an invalid pointer that is not NULL is de-referenced, which causes the system to behave in an undefined way. This type of bug is not covered by the work on memory safety of Beyer et al. in [Beyer et al., 2005] and cannot easily be detected by runtime checks.

The example we will study here in detail is a use-after-free error spotted by the Coverity source code analyser (www.coverity.com) in the I2O subsystem of the Linux kernel (cf. Section 3.1.1). To check for this bug in BLAST we first specify

[2] Commit 2d6eac6c4fdaa69656d66c80754d267be233cc3f.

[3] Commit d0743a5b7b837334cb414b773529d51de3de0471.

```
drivers/message/i2o/pci.c:
 300  static int __devinit
       i2o_pci_probe(
       struct pci_dev *pdev,
 301   const struct pci_device_id
       *id)
 302  {
 303   struct i2o_controller *c;

 330   c = i2o_iop_alloc();

 423   free_controller:
 424    i2o_iop_free(c);
 425    put_device(
          c->device.parent);

 432  }
```

Figure 3.1: Extract of `drivers/message/i2o/pci.c`.

a temporal safety specification in the Blast specification language. Taking this specification, Blast is supposed to automatically generate an instrumented version of the C source code for analysis (cf. Section 3.1.2). However, due to an apparent bug in Blast, this step fails for our example, and we are therefore forced to manually instrument our code by inserting `ERROR` labels at appropriate positions (cf. Section 3.1.3). However, it will turn out that Blast does not track important operations on pointers, which is not mentioned in Blast's user manual and without which our example cannot be checked (cf. Section 3.1.4).

3.1.1 The I2O Use-After-Free Error

The I2O subsystem bug of interest to us resided in lines 423–425 of the source code file `drivers/message/i2o/pci.c`. The listing in Fig. 3.1 is an abbreviated version of the file `pci.c` before the bug was fixed. One can see that function `i2o_iop_alloc()` is called at line 330 of the code extract. This function is defined in `drivers/message/i2o/iop.c` and basically allocates memory for an `i2o_controller` structure using `kmalloc()`. At the end of the listing, this memory is freed by `i2o_iop_free(c)`. The bug in this piece of code arises from the call of `put_device()` in line 425, since its parameter `c->device.parent` causes an already freed pointer to be de-referenced. The bug has been fixed in commit `d2b0e84d195a341c1cc5b45ec2 098ee23bc1fe9d`, by simply swapping lines 424 and 425 in the source file.

This bug offers various different ways to utilise Blast. A generic temporal safety property for identifying bugs like this would state that *any pointer that has been an argument to `kfree()` is never used again* unless it has been re-allocated. A probably easier way would be to check whether *the pointer `c` in `i2o_pci_probe()` is never used again after `i2o_iop_free()` has been called* with `c` as its argument. Checking the first, more generic property would require us to put function definitions from other source files into `pci.c`, since Blast considers only functions that are available

in its input file. Therefore, we focus on verifying the latter property.

Checking for violations even of the latter, more restricted property will lead to a serious problem. A close look at the struct `i2o_controller` and its initialisation in the function `i2o_iop_alloc()` reveals that `i2o_controller` contains a function pointer which can be used as a "destructor". As is explained in Blast's user manual, the "current release does not support function pointers"; they are ignored completely. Further, the manual states that "correctness of the analysis is then modulo the assumption that function pointer calls are irrelevant to the property being checked." This assumption is however not always satisfied in practice, as we will see later in our example.

3.1.2 Verification With a Temporal Safety Specification

Ignoring the function pointer limitation, we developed the temporal safety specification presented in Fig. 3.2. The specification language used by Blast is easy to understand and allows the assignment of status variables and events. In our specification we use a global status variable `allocstatus_c` to cover the possible states of the struct `c` of our example, which can be set to 0 meaning "not allocated" and 1 meaning "allocated". Furthermore, we define three events, one for each of the functions `i2o_iop_alloc()`, `i2o_iop_free()` and `put_device()`. All functions have special preconditions and calling them may modify the status of `c`. The special token `$?` matches anything. Intuitively, the specification given in Fig. 3.2 states that `i2o_iop_alloc()` and `i2o_iop_free()` must be called alternately, and `put_device()` must only be called when `c` has not yet been freed. Note that this temporal safety specification does not cover the usage rule for `i2o_iop_free()` and `put_device()` in general. We are using one status variable to guard calls of `i2o_iop_free()` and `put_device()` regardless of its arguments. Hence, the specification will work only as long as there is only one pointer to an `i2o_controller` structure involved.

Using the specification of Fig. 3.2, Blast should instrument a given C input file by adding a global status variable and error labels for all violations of the preconditions. The instrumentation is done by the program `spec.opt` which is part of the Blast distribution. For our example taken from the Linux kernel, we first obtained the command used by the kernel's build system to compile `pci.c` with `gcc`. We appended the option `-E` to force the compilation to stop after preprocessing, resulting in a C source file containing all required parts of the kernel headers. This step is necessary since Blast cannot know of all the additional definitions and include paths

```
global int allocstatus_c = 0;

event
 {
  pattern { $? = i2o_iop_alloc(); }
  guard   { allocstatus_c == 0 }
  action  { allocstatus_c = 1; }
 }

event
 {
  pattern { i2o_iop_free($?); }
  guard   { allocstatus_c == 1 }
  action  { allocstatus_c = 0; }
 }

event
 {
  pattern { put_device($?); }
  guard   { allocstatus_c == 1 }
 }
```

Figure 3.2: A temporal safety specification for `pci.c`.

used to compile the file. Unfortunately, it expands `pci.c` from 484 lines of code to approximately 16k lines, making it difficult to find syntactical problems which BLAST cannot deal with. Despite spending a lot of effort in trying to use `spec.opt`, we never managed to get this work. The program mostly failed with unspecific errors such as `Fatal error: exception Failure(“Function declaration not found”)`. Finding such an error in a huge source without having a line number or other hint is almost impossible, especially since `gcc` compiles the file without any warning. We constructed several simplifications of the preprocessed file in order to trace the limitations of `spec.opt`, but did not get a clear indication of what the source is. We suspect it might be a problem with parsing complex data structures and inline assembly imported from the Linux headers.

Given the bug in BLAST and in order to demonstrate that our specification indeed covers the programming error in `pci.c`, we developed a rather abstract version of `pci.c` which is shown in Fig. 3.3. Using this version and the specification of Fig. 3.2, we were able to obtain an instrumented version of our source code without encountering the bug in `spec.opt`. Running BLAST on the instrumented version then produced the following output:

```
$ spec.opt test2.spc test2.c
[...]
$ pblast.opt instrumented.c
[...]
Error found! The system is unsafe :-(
```

In summary, the example studied here shows that the specification used in this section is sufficient to find the bug. However, the approach required by BLAST has

```
test2.h:
#include <stdio.h>
#include <stdlib.h>

typedef struct device
{
 int parent;
} device;

typedef struct i2o_controller
{
 struct device device;
} i2o_controller;

i2o_controller *i2o_iop_alloc
 (void);
void i2o_iop_free
 (i2o_controller *c);
void put_device (int i);
```

```
test2.c:
#include "test2.h"

i2o_controller *i2o_iop_alloc
 (void)
{ i2o_controller *c;
  c = malloc(
   sizeof(struct i2o_controller));
  return (c); }

void i2o_iop_free
 (i2o_controller *c)
{ free (c); }

void put_device (int i) { }

int main (void)
{ i2o_controller *c;
  c = i2o_iop_alloc ();
  i2o_iop_free (c);
  put_device (c->device.parent);
  return (0); }
```

Figure 3.3: Manual simplification of `pci.c`.

several disadvantages. Firstly, it is not automatic at all. Although we ended up with only a few lines of code, it took quite a lot of time to produce this code by hand and to figure out what parts of the original `pci.c` are accepted by Blast. Secondly, the methodology works only if the bug is known beforehand; hence we did not learn anything new about unwanted behaviour of this driver's code. We needed to simplify the code to an extent where the relation to the original source code may be considered as questionable. The third problem lies in the specification used. Since it treats the allocation and de-allocation as something similar to a locking problem, we would not be able to use it in a piece of code that refers to more than one dynamically allocated object. A more generic specification must be able to deal with multiple pointers. According to [Beyer et al., 2004], such a generic specification should be possible to write by applying a few minor modifications such as defining a "shadow" control state and replacing `$?` with `$1`. However, in practice the program generating the instrumented C source file failed with obscure error messages.

3.1.3 Verification Without a Temporal Safety Specification

Since BLAST could not deal with verifying the original `pci.c` using an explicit specification of the use-after-free property, we will now try and manually instrument the source file so that our bug can be detected whenever an `ERROR` label is reachable.

When conducting our instrumentation, the following modifications were applied by hand to `pci.c` and related files:

1. A variable `unsigned int alloc_status` was added to the definition of `struct i2o_controller` in `include/linux/i2o.h`.

2. The prototypes of `i2o_iop_alloc()` and `i2o_iop_free()` were removed from `drivers/message/i2o/core.h`.

3. The prototype of `put_device()` was deleted from `include/ linux/device.h`.

4. C source code for the functions `put_device()`, `i2o_iop_free()`, `i2o_iop_ release()` and `i2o_iop_alloc()` was copied from `iop.c` and `drivers/base /core.c` into `pci.c`. The functions were modified such that the new field `alloc_status` of a freshly allocated `struct i2o_controller` is set to 1 by `i2o_iop_alloc()`. `i2o_iop_free()` no longer de-allocates the structure but checks whether `alloc_status` equals 1 and sets it to 0; otherwise, it jumps to the `ERROR` label. `put_device()` was modified to operate on the whole `struct i2o_controller` and jumps to `ERROR` if `alloc_status` equals 0.

By feeding these changes into the model checker it is possible to detect duplicate calls of `i2o_iop_free()` on a pointer to a `struct i2o_controller`, as well as calls of `put_device()` on a pointer that has already been freed. Even calls of `i2o_iop_free()` and `put_device()` on a pointer that has not been allocated with `i2o_ iop_alloc()`, should result in an error report since nothing can be said about the status of `alloc_status` in such a case.

After preprocessing the modified source files and running BLAST, we get the output "`Error found! The system is unsafe :-(`". Even after we reduced the content of `i2o_pci_probe()` to something quite similar to the `main()` function shown in Fig. 3.3 and after putting the erroneous calls of `put_device()` and `i2o_iop_free()` in the right order, the system was still unsafe from BLAST's point of view. It took us some time to figure out that BLAST does not appear to consider the content of pointers at all.

```
test5.c:
   1  #include <stdlib.h>
   2
   3  typedef struct example_struct
   4   {
   5    void   *data;
   6    size_t size;
   7   } example_struct;
   8
   9
  10  void init (example_struct *p)
  11   {
  12    p->data = NULL;
  13    p->size = 0;
  14
  15    return;
  16   }
  17
  18  int main (void)
  19   {
  20    example_struct p1;
  21
  22    init (&p1);
  23    if (p1.data != NULL ||
          p1.size != 0)
  24     { goto ERROR; }
  25     else
  26     { goto END; };
  27
  28  ERROR:
  29    return (1);
  30
  31  END:
  32    return (0);
  33   }
```

Figure 3.4: An example for pointer passing.

3.1.4 BLAST and Pointers

We demonstrate this apparent shortcoming of BLAST regarding handling pointers by means of another simple example, for which BLAST fails in tracing values behind pointers over function calls.

As can be seen in the code listing of Fig 3.4, label `ERROR` can never be reached in this program since the values of the components of our struct are explicitly set by function `init()`. However, BLAST produces the following output:

```
$ gcc -E -o test5.i test5.c
$ pblast.opt test5.i
[...]
Error found! The system is unsafe :-(
Error trace:
23 :: 23: Pred((p1@main).data!=0) :: 29
-1 :: -1: Skip :: 23
10 :: 10: Block(Return(0);) :: -1
12 :: 12: Block(* (p@init ).data = 0;* (p@init ).size = 0;) :: 10
22 :: 22: FunctionCall(init(&(p1@main))) :: -1
-1 :: -1: Skip :: 22
 0 ::  0: Block(Return(0);) :: -1
 0 ::  0: FunctionCall (__BLAST_initialize_test5.i()) :: -1
```

This counterexample shows that BLAST does not correlate the pointer `p` used in `init()` and the struct `p1` used in `main()`, and assumes that the `if` statement in line 23 evaluates to true. After adding a line "`p1.data = NULL; p1.size = 0;`" before the call of `init()`, BLAST claims the system to be safe, even if we modify `init()` to reset the values so that they differ from `NULL` (and `0`).

We were able to reproduce this behaviour in similar examples with pointers to integer values and arrays. Switching on the BDD-based alias analysis implemented in BLAST also did not solve the problem. The example shows that BLAST does not only ignore function pointer calls as stated in its user manual, but appears to assume that all pointer operations have no effect. This limitation is not documented in the BLAST manual and renders BLAST almost unusable for the verification of properties related to our understanding of memory safety.

3.1.5 Results

Our experiments on memory safety show that BLAST is able to find the programming error discovered by the Coverity checker. Out of eight examples, we were able to detect two problems after minor modifications to the source code, and three after applying manual abstraction. Three further programming errors could not be traced by using BLAST. Indeed, BLAST has some major restrictions. The main problem is that BLAST ignores variables addressed by a pointer. As stated in its user manual, BLAST assumes that only variables of the same type are aliased. Since this is the case in our examples, we initially assumed that our examples could be verified with BLAST, which is not the case. Moreover, we encountered bugs and deficiencies in `spec.opt` which forced us to apply substantial and time consuming modifications to source code. Most of these modifications and simplifications would require a developer to know about the error in advance. Thus, from a practitioner's point of view, BLAST is not of much help in finding unknown errors related to memory safety. However, it needs to be mentioned that BLAST was designed for verifying API usage rules of a different type than those required for memory safety. More precisely, BLAST is intended for proving the adherence of pre- and post-conditions denoted by integer values and for ensuring API usage rules concerning the order in which certain functions are called, regardless of pointer arguments, return values and the effects of aliasing. A summary of our experience with BLAST and memory safety is given in Table 3.1.

Table 3.1: Result summary for memory safety properties

Memory Safety Error Type	Example 1	2	3	4	5	6	7	8
`NULL` de-reference				x				
use-after-free	x	x	x		x			
double free								x
overrun error						x		
pointer arithmetic							x	
involves concurrency					x			
Error found by BLAST	M	M	f	M	f	m	f	m

Key: d = directly; m = minor modifications; M = manual abstraction; f = failed

3.2 Checking Locking Properties with BLAST

Verifying correct locking behaviour is something used in almost all examples provided by the developers of BLAST [Beyer et al., 2004; Henzinger et al., 2002a]. In [Henzinger et al., 2002a], the authors checked parts of the Linux kernel for correct locking behaviour while using the *spinlock* API and stated that BLAST showed a decent level of performance during these tests. Spinlocks provide a very simple but quite efficient locking mechanism to ensure, e.g., that a kernel thread may not be preempted while serving interrupts. The kernel thread acquires a certain lock by calling `spin_lock(l)`, where `l` is a previously initialised pointer to a struct `spinlock_t` identifying the lock. A lock is released by calling `spin_unlock()` with the same parameter. The kernel provides a few additional functions that control the interrupt behaviour while the lock is held. By their nature, spinlocks are intended for use on multiprocessor systems where each resource may be associated with a special spinlock, and where several kernel threads need to operate independently on these resources. However, as far as concurrency is concerned, uniprocessor systems running a preemptive kernel behave like multiprocessor systems.

Finding examples for the use of spinlocks is not difficult since they are widely deployed. While experimenting with BLAST and the spinlock functions on several small components of the Linux kernel we experienced that it performs well with functions using only one lock. We focused on functions taken from the USB subsystem in *drivers/usb/core*. Due to further unspecific parse errors with the program `spec.opt` we could not use a temporal safety specification directly on the kernel source. However, in this case we were able to generate the instrumented source file and to verify properties by separating the functions under consideration from the

```
global int lockstatus = 2;

event
 {
  pattern { spin_lock_init($?); }
  guard   { lockstatus == 2 }
  action  { lockstatus = 0; }
 }

event
 {
  pattern { spin_lock($?); }
  guard   { lockstatus == 0 }
  action  { lockstatus = 1; }
 }

event
 {
  pattern { spin_unlock($?); }
  guard   { lockstatus == 1 }
  action  { lockstatus = 0; }
 }

event
 {
  pattern { $? = sleep($?); }
  guard   { lockstatus == 0 }
 }
```

Figure 3.5: A temporal safety specification for spinlocks.

remaining driver source and by providing simplified header files.

In Fig. 3.5 we provide our basic temporal safety specification for verifying locking behaviour. Variable `lockstatus` encodes the possible states of a spinlock; the initial value 2 represents the state in which the lock has not been initialised, while 1 and 0 denote that the lock is held or has been released, respectively. The pattern within the specification varies for the different spinlock functions used within the driver source under consideration, and the specification can easily be extended to cover forbidden functions that may sleep. An example for a function `sleep()` is provided in the specification of Fig. 3.5.

Difficulties arise with functions that acquire more than one lock. Since all spinlock functions use a pointer to a struct `spinlock_t` in order to identify a certain lock, and since the values behind pointers are not sufficiently tracked in BLAST, we were forced to rewrite parts of the driver's source and the kernel's spinlock interface. Instead of the pointers to `spinlock_t` structs we utilise global integer variables representing the state of a certain lock. We have used this methodology to verify an example of a recently fixed deadlock[4] in the Linux kernel's SCSI subsystem. In Fig. 3.6 we provide an extract of one of the functions modified in the fix. We see that the spinlocks in this example are integrated in more complex data structures referenced via pointers. Even worse, this function calls a function pointer passed in the argument `done` in line 1581, which was the source of the deadlock before the bug was fixed. To verify this special case, removing the function pointer and

[4]Commit `d7283d61302798c0c57118e53d7732bec94f8d42`.

```
1564 int ata_scsi_queuecmd(struct
      scsi_cmnd *cmd, void
      (*done)(struct scsi_cmnd *))
1565 {
1566 struct ata_port *ap;
1567 struct ata_device *dev;
1568 struct scsi_device
      *scsidev = cmd->device;
1569 struct Scsi_Host
      *shost = scsidev->host;
1571 ap = (struct ata_port *)
      &shost->hostdata[0];
1573 spin_unlock(shost->host_lock);
1574 spin_lock(&ap->host_set->lock);

1581 done(cmd);

1597 spin_unlock(&ap->host_set->lock);
1598 spin_lock(shost->host_lock);
1600 }
```

Figure 3.6: Extract of `drivers/scsi/libata-scsi.c`.

providing a dummy function `done()` with a precondition assuring that the lock on `shost->host_lock` is not held is needed. However, we were able to verify both the deadlock condition before the fix had been applied, as well as deadlock freedom for the fixed version of the source.

During our experiments we analysed several other examples of deadlock conditions. The more interesting examples are the spinlock problem explained above, and another one in the SCSI subsystem,[5] as well as a bug in a IEEE1394 driver[6]. We were able to detect the locking problems in all of these examples and proved the fixed source files to be free of these bugs.

Results. Out of eight examples for locking problems we were able to detect only five. However, when comparing our results with the conclusions of the previous section, BLAST worked much better for the locking properties because it required fewer modifications to the source code. From a practitioner's point of view, BLAST performed acceptable as long as only one lock was involved. After considerable efforts in simplifying the spinlock API — mainly removing the use of pointers and manually adding error labels to the spinlock functions — we also managed to deal with multiple locks. However, we consider it as fairly difficult to preserve the behaviour of functions that may sleep and therefore must not be called under a spinlock. Even for large portions of source code, BLAST returned its results within a few seconds or minutes, on a PC equipped with an AMD Athlon 64 processor running at 2200 MHz and 1 GB of RAM. Hence, BLAST's internal slicing and abstraction techniques work very well.

We have to point out that the code listing in Fig. 3.6 represents one of the easily

[5] Commit `fe2e17a405a58ec8a7138fee4ebe101858b636e0`.
[6] Commit `910573c7c4aced8fd5f45c334cc67862e3424d92`.

Table 3.2: Result summary for locking properties properties

Locking Properties	Example							
Error Type	1	2	3	4	5	6	7	8
deadlock condition	x	x	x	x		x	x	
other API violation					x			x
involves concurrency					x			x
Error found by BLAST	M	M	f	f	f	m	m	f
Key: d = directly; m = minor modifications; M = manual abstraction; f = failed								

understandable programming errors. Many problems in kernel source code are more subtle. For example, calling functions that may sleep is something that needs to be avoided. However, if a driver calls a function not available in source code in the same file as the driver under consideration, BLAST will only be able to detect the problem if there is an event explicitly defined for this function. A summary of our results including all 8 examples for locking issues is given in Table 3.2.

3.3 Summary of Results

This section highlights various shortcomings of the BLAST toolkit which we experienced during our studies. We also present ideas on how BLAST could be improved in order to be more useful for operating system software verification.

Lack of documentation. Many problems while experimenting with BLAST were caused by the lack of consistent documentation. For example, a significant amount of time could have been saved in our experiments with memory safety, if the BLAST manual would state that almost all pointer operations are ignored. An in-depth discussion of the features and limitations of the alias analysis implemented in BLAST would also be very helpful to have.

Non-support of pointers. The fact that BLAST does not properly support the use of pointers, in the sense of Section 3.1.4, must be considered as a major restriction, and made our experiments with the spinlock API rather difficult. The restriction forces one to carry out substantial and time consuming modifications to source code. Furthermore, it raises the question whether all important predicates of a given program can be preserved in a manual step of simplification. In some of our experiments we simply replaced the pointers used by the spinlock functions with integers representing the state of the lock. This is obviously a pragmatic ap-

proach which does not reflect all possible behaviour of pointer programs. However, it turned out that it is expressive enough to cover the usage rules of the spinlock API. As such modifications could be introduced into the source code automatically, we consider them as an interesting extension for BLAST.

The missing support of function pointers has already been mentioned in Section 3.1. It is true that function pointers are often used in both application space and operating system development. In most cases their effect on the program execution can only be determined at run-time, not statically at compile-time. Therefore, we assume that simply skipping all calls of function pointers is acceptable for now.

Usability. There are several issues regarding BLAST's usability which are probably easy to fix, but right now they complicate the work with this tool. Basically, if a piece of C source is accepted by an ANSI C compiler, it should be accepted by BLAST rather than raising uninformative error messages.

A nice improvement would be to provide wrapper scripts that automate preprocessing and verification in a way that BLAST can be used with the same arguments as the compiler. It could be even more useful if functions that are of interest but from other parts of a given source tree, would be copied in automatically. Since we obviously do not want to analyse the whole kernel source in a single file, this should be integrated into BLAST's abstraction/model checking/refinement loop.

Chapter 4

Symbolic Object Code Analysis

In this chapter we present *Symbolic Object Code Analysis* (SOCA), a novel approach to identifying violations of memory safety properties based on bounded path-sensitive symbolic execution of compiled and linked programs. More precisely, we translate a given program path-wise into systems of bit-vector constraint using an *intermediate representation* (IR) borrowed from the Valgrind dynamic binary instrumentation framework [Nethercote and Seward, 2007]. In Section 4.2 we outline the features of this IR language, sketch a simple operational semantics and explain how IR instructions can be translated into constraints for the Yices SMT solver [Dutertre and de Moura, 2006].

As we describe in Section 4.3, the SOCA technique employs Yices as execution and verification engine, checking the satisfiability of the generated constraints systems. The analysis has to be bounded since the total number of paths as well as the number of instructions per path in a program is potentially infinite. Our approach allows us to express a range of memory safety properties as simple assertions over those constraint systems. In contrast to other methods for finding memory safety violations, our technique does not employ program abstraction other than leaving the program's input initially unspecified in order to allow the SMT solver to search for inputs that will drive the program into an error state.

In Section 4.5 we present the results of and extensive evaluation of our technique by applying a prototypical SOCA Verifier to the Verisec benchmarking suite as well as to almost 10,000 functions taken from Linux device drivers. For the evaluation of our SOCA technique we chose Linux device drivers compiled for 32-bit Intel architectures (IA32, [Intel Corporation, 2009]), as our application domain. Despite being relatively small in size, device drivers represent an interesting challenge as they implement hardware access and are usually written in a mixture of C code and inlined assembly code. This combination makes them particularly hard to test and analyse with currently available verification tools.

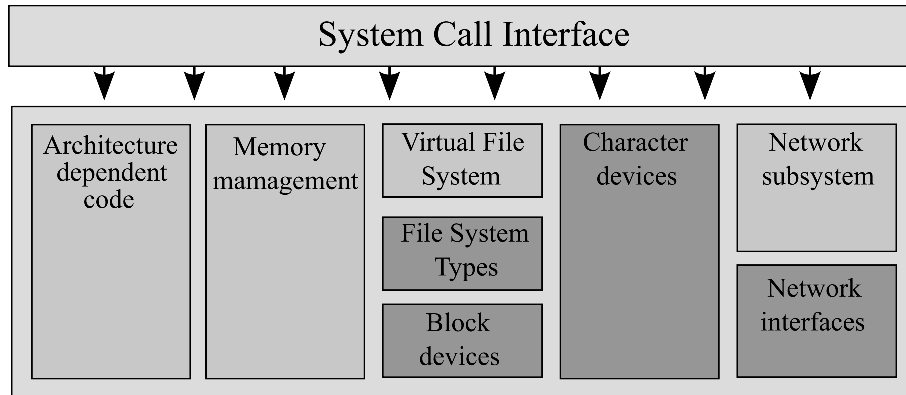

Figure 4.1: Overview of the Linux OS kernel

4.1 Background

The Linux operating system kernel consists of a large and complex code base implementing key tasks such as process management, memory management, file system access, device control and networking for about 20 different computer architectures. As illustrated in Fig. 4.1, it features a relatively small monolithic core of components such as the scheduler and the memory management subsystem. However, the majority of its functionality, shown in dark gray in Fig. 4.1, is implemented in terms of *kernel modules* or *device drivers*[1] that can be built separately from the kernel and loaded at runtime. These modular components of the Linux kernel are responsible, for example, for making a particular physical device attached to the computer respond to a well-defined internal programming interface. Hence, user access to this device can be performed by means of standardised functions provided by the kernel's System Call Interface, which are independent of the particular driver or device. Kernel modules amount to roughly two-thirds (≈200 MBytes of code) of the entire Linux kernel distribution.

Defining memory safety. The scope we are aiming on is to develop a framework that verifies that every pointer in a given program is *(1)* valid in the sense that it never references a memory location outside the address space allocated by or for that program, and *(2)* valid with respect to a given set of API usage rules obtained from the Linux kernel's documentation at every point in program execution the pointer is dereferenced at. In detail the memory safety properties we are interested in may be classified as follows:

(a) Dereferencing invalid pointers. A pointer may not be `NULL`, shall be initialised and shall not point to a memory location outside the address space allocated by or

[1] The terms *kernel module* and *device driver* are used synonymously within this paper.

for the driver under consideration. A violation of this property leads to undefined behaviour. *(b) Uninitialised reads.* Memory cells shall be initialised before they are read. The program's behaviour becomes undefined, otherwise. *(c) Violation of memory permissions.* When a program is loaded into memory, the different segments (cf. [Tool Interface Standards (TIS) Committee, 1995]) of the program file are assigned with different permissions determining whether that section can be read, written or executed. Violations of those permissions may lead to program termination and are usually a sign of erroneous pointer arithmetics. Memory permissions are not always strictly enforced by the operating system. *(d) Buffer overflows.* By "buffer overflow" we mean out-of-bound read and write operations to objects on heap and stack. These errors lead to memory corruption and give way to various security problems. *(e) Memory leaks.* When a program dynamically allocates memory but loses the handle to it, the memory cannot be deallocated anymore. Memory leaks have an especially high impact on the reliability of OS components since they are supposed not to terminate. In a long-term execution, a device driver losing only a few bytes of dynamically allocated heap space per operation becomes a reliability issue. *(f) Proper handling of allocation and deallocation.* The Linux kernel provides several APIs for the dynamic (de)allocation of memory. The kernel's documentation specifies precisely what pairs of functions need to be employed together in order to safely (de)allocate heap space. Furthermore it is specified that the deallocation functions shall not be used more than once on a specific pointer unless it has been re-allocated.

Aliasing in source code and object code. A major issue in the construction of optimising compilers, as well as for source-code-based program analysis and verification tools, is presented by the aliasing problem. Aliasing means that a data location in memory may be accessed through different symbolic names in the program. Considering the C programming language, this usually means that multiple pointer variables in a program are referencing the same data object. Since those aliasing relations between symbolic names and data locations often arise unexpectedly during program execution, they may result in erroneous program behaviours that are particularly hard to trace and to debug.

Let us consider the C program given in Fig. 4.2. The program shows a rather uncomfortable way of implementing an endless loop. It declares a counter i of 32 bit length. In addition, two pointers *p1* and *p2* are used such that *p2* points to i and *p1* to the least significant 16 bits of i. Hence, *p1* and *p2* are pointing to the same memory location. In the loop declaration (l. 8) we are now counting the data object pointed to by *p1* from 0 up to 10 while setting the data object pointed to by *p2*

```
01 #include <stdio.h>
02 #include <sys/types.h>
03
04 int main (void) {
05  int32_t  i, *p2=&i;
06  int16_t *p1=&((int16_t*) &i)[0];
07
08  for (*p1=0; *p1<10; (*p1)++)
09   { *p2=0; }
10
11  printf ("%08x: %d\n", p1, *p1);
12  printf ("%08x: %d\n", p2, *p2);
13  printf ("%08x: %d\n", &i, i);
14  return (0); }
```

Figure 4.2: Example for pointer aliasing in C: `e_loop.c`

```
$ gcc -O2 e_loop.c
$ ./a.out
bfc76f2c: 10
bfc76f2c: 0
bfc76f2c: 0
```

```
$ gcc -O2 e_loop.c
$ ./a.out
bfc7428c: 10
bfc7428c: 10
bfc7428c: 10
```

```
$ gcc -O1 e_loop.c
$ ./a.out
-> does not terminate
```

Figure 4.3: Output of the program given in Fig. 4.2 compiled with *(a)* gcc version 4.1.2 (left) and *(b,c)* gcc version 4.3.1 (right).

to 0 (l. 9) in every loop iteration. The code should loop forever and the `printf()` statements in ll. 11 to 13 should never be reached. However possible behaviours of the program are presented in Fig. 4.3.

The different outcomes of the program execution can be explained as a result of unsound/different assumptions made about pointer aliasing by the developer and the compiler, in connection with different optimisations applied to the code. In the first and second case, the compiler is invoked with the option "-O2", enabling several optimisations along with the assumption that pointers of different types do not alias (in compliance with ISO/IEC 9899:1999).

We may now look at the same program at assembly level. Fig. 4.4 shows an

```
80483ba: xor    %eax,%eax        ;; eax := 0;
80483c4: lea    -0xc(%ebp),%ebx ;; ebx := ebp - 0xc
80483c8: add    $0x1,%eax        ;; eax := eax + 0x00000001
80483cb: cmp    $0x9,%ax         ;; (ax = 9)?
80483cf: movl   $0x0,-0xc(%ebp) ;; *p2 (= ebp - 0xc)       := 0
80483d6: mov    %ax,(%ebx)       ;; *p1 (= ebx = ebp - 0xc) := ax
80483d9: jle    80483c8          ;; if (ax <= 9) goto 80483c8
```

Figure 4.4: Excerpt of the disassembled code from Fig. 4.3.*b*.

excerpt of the assembly code obtained by disassembling the program which produced the output shown in Fig. 4.3.*b* using the `objdump` disassembler. We can easily spot (at instructions 80483cf and 80483d6) that *p1* and *p2* are indeed pointing to the same location in memory. We can also see that **p2* is actually written before **p1*. This is unexpected when looking at the source code, but valid from the compiler's point of view as it assumes that the two pointers do not alias. As another consequence of this assumption, *eax* is never reloaded from the memory location to which *p1* and *p2* point.

The above example shows that source-code-based analysis tools have to decide for a particular semantics of the source language, which may not be the one that is actually used by a compiler to translate the code into an executable. Hence, results obtained by analysing the source code may not necessarily meet a program's runtime behaviour.

While the above example motivates the analysis of compiled programs, doing so does not provide a generic solution for dealing with pointer aliasing. Consider the following lines of C code:

```
01 int i, *p1 = &i, *p2 = NULL;
02 if (condition) { p2 = &i; }
03 ...
```

In this case, after line 2, we cannot determine whether *p1* and *p2* do alias or not, regardless of the program representation we chose. However, we may attempt to do a path-sensitive analysis of the program and consider the path in which *condition* evaluates to true and hence *p1* and *p2* do alias, separated from the path in which *condition* does not hold. Of course this is not feasible in general as programs may have infinitely many paths. Our assumption is that for the application domain of device drivers – relatively short programs – our approach will scale well enough in order to find previously unknown errors. Our results presented in Section 4.5 demonstrate that this is true.

4.2 Valgrind's Intermediate Representation Language

A program under consideration is stored by us in an intermediate representation (IR) borrowed from Valgrind [Nethercote and Seward, 2007], a framework for dynamic binary instrumentation. The IR consists of a set of *basic blocks* containing a group of statements such that all transfers of control to the block are to the first statement in the group. Once the block has been entered, all statements in the group are

```
<reg>       ::= <CPU register number>

<treg>      ::= <Temporary register number>

<type>      ::= <I8 | I16 | I32>

<statement> ::=
   |          <treg>:<type>
   |          PUT      (<reg>)  = <<treg>|<<val>:<type>>>
   | <treg> = GET:type (<reg>)
   |          ST       (<<treg>|<<val>:I32>>) = <<treg>|<<val>:<type>>>
   | <treg> = LD:type  (<<treg>|<<val>:I32>>)
   |          GOTO     (<<treg>|<<val>:I32>>)
   |          IF       (<treg>) <statement>
   |          EXIT
   | <treg> = ADD:type (<<treg>|<<val>:<type>>>, <<treg>|<<val>:<type>>>)
   | <treg> = AND:type (<<treg>|<<val>:<type>>>, <<treg>|<<val>:<type>>>)
   | ...
   | <treg> = Xor:type (<<treg>|<<val>:<type>>>, <<treg>|<<val>:<type>>>)
```

Figure 4.5: Basic syntax of Valgrind's IR language

executed sequentially. Hence, a basic block has exactly one entry point but may have multiple exit points. The IR is basically a typed assembly language in static-single-assignment form [Cytron et al., 1991; Leung and George, 1999] using *temporary registers* and some memory for storing the *guest state*, i.e., registers available in the CPU the program is originally compiled for.

In Valgrind's IR all arithmetic expressions, including address arithmetic, are decomposed into simple expressions with a fixed number of operands using temporary registers for intermediate results. Furthermore, all load and store operations to memory cells as well as to the guest state are made explicit. Hence, normalising a program by transforming it into its IR increases the number of separate instructions as each CPU instruction is usually expanded into multiple IR instructions. However, this proceeding reduces the complexity of the program's representation because IR instructions are relatively simple and free of side effects.

The basic syntax of Valgrind's IR is illustrated in Fig. 4.5. The meaning of the the different constructs in the language is as follows:

treg:type : temporary register declaration

PUT : stores a value or the contents of a temporary register in a CPU register

GET : load a CPU register into a temporary register

IA32 Assembly	IR Instructions

`xor %eax,%eax`	`t9 = GET:I32(0)   ;; t9  := eax`
	`t8 = GET:I32(0)   ;; t8  := eax`
	`t7 = Xor32(t9,t8) ;; t7  := t9 xor t8`
	`PUT(0) = t7       ;; eax := t7`

`lea -0xc(%ebp),%ebx`	`t42 = GET:I32(20)`
	`t41 = Add32(t42,0xFFFFFFF4:I32)`
	`PUT(12) = t41`

Figure 4.6: First two instructions of Fig. 4.4 and their respective IR instructions.

ST : stores a value or the contents of a temporary register at a heap location identified by a value or a temporary register

LD : loads the contents of a heap location identified by a value or a temporary register to a temporary register

GOTO : Makes program execution proceed at the program location identified by a value or a temporary register

IF : Conditional execution of a statement if the first parameter equals 1

EXIT : finish program execution

other operations : Apart from the instructions explained above, the IR language consists of various statements for arithmetical operations and other transformations on temporary registers. These instructions do always have up to four parameters. The result of the operation is stored in a previously declared but not assigned temporary register, preserving the static single assignment form of the IR. To give some examples, we have added the ADD and AND instructions above. Their semantics is self-explanatory.

memory allocation : The IR does not provide mechanisms for allocating or deallocating objects on the heap or stack. Instead, the `GOTO` statement is used to denote jumps to allocators and de-allocators provided by the operating system. Since model checking the operating system's memory management facilities itself is currently not in the scope of our research, we do not translate functions like `malloc()` or `free()` into their IR representation but provide a semantics for the entire function call.

The example for IR instructions given in Fig. 4.6 shows that our chosen intermediate representation consists of a few basic elements such as temporary registers

IA32 Assembly	IR Instructions

`sub $0x8,%esp`	`PUT(60) = 0x8048377:I32 ;; put PC`
	`t4 = GET:I32(16) ;; get ESP into t4`
	`t2 = Sub32(t4,0x8:I32) ;; t2 = t4 - 0x8`
	`PUT(32) = 0x6:I32 ;; EFLAGS: operation`
	`PUT(36) = t4 ;; EFLAGS: first operand`
	`PUT(40) = 0x8:I32 ;; EFLAGS: second operand`
	`PUT(16) = t2 ;; put new ESP`

Figure 4.7: Valgrind IR: `EFLAGS` usage.

denoted with $t{<}n{>}$, *GET* and *PUT* statements to access machine registers identified by integers, as well as arithmetic and boolean operations such as *Add*, *And* and *Xor*. Note that the latter instructions operate on temporary registers or literals only. In addition to those statements, there are also *LD* and *ST* instructions for loading and storing data to and from the main memory, respectively. An important feature of the IR is that all operations and registers are typed. While machine registers are always 8 bits long, temporary registers may have a length of 1, 8, 16, 32 or 64 bits. As a result of this, the statement `t9 = GET:I32(0)` means that *t9* is generated by concatenating the machine registers 0 to 3. As each IR block is in static single assignment form with respect to the temporary registers, *t9* is assigned only once within a single IR block.

Valgrind's IR takes special care of the `EFLAGS` register of Intel x86 microprocessors. The `EFLAGS` register is the status register of these CPUs, containing the current state of the processor. The register may be updated by various instructions. Especially arithmetical operations may update the register's **C**arry, **Z**ero and **S**igned bits depending on the result of the operation. Valgrind's IR does not force the computation of these flags for each arithmetic operation. Instead, IR-instructions storing the parameters of the last operation that may have updated the `EFLAGS` register are generated so that the actual flag assignment may be computed when it is needed at a later point in program execution, i.e. for evaluating a guarded jump statement. An example for these additional IR instructions is given in Fig. 4.7: the IR instructions marked with the comment "`EFLAGS:`" denote the storing of the `Sub32` operation and the two operands in additional machine registers that have no corresponding representation in actual IA32 CPUs.

4.2.1 A semantics for Valgrind's IR

We define a simple operational semantics for for the operations in Valgrind's IR language in terms of bit vector arithmetic.

Definition 1 (Bit Vector) *A bit vector b is a vector of bits with a given length l (or dimension):*

$$b : \{0, ..., l-1\} \rightarrow \{0, 1\}$$

The set of all 2^l bit vectors of length l is denoted by $bvec_l$. The i-th bit of the bit vector b is denoted by b_i [Kroening and Strichman, 2008].

To give a semantics to the different IR instructions we use command-state pairs $\langle c, (t, r, h) \rangle$ where c is a command (i.e. an IR instruction with its parameters) and the triple (t, r, h) represents the program state with t holding the temporary register assignment, r the CPU register assignment and h the current heap. As shown in Definition 3, our semantics is based on three partial functions $Registers$, $TempRegisters$ and $Heap$ representing the program state.

Definition 2 (Basic Definitions)

$$\begin{aligned} Types &= \{I1, I8, I16, I32\} \\ Addresses &= bvec_{32} \\ Values &= bvec_1 \cup bvec_8 \cup bvec_{16} \cup bvec_{32} \end{aligned}$$

Definition 3 (Program State)

$$\begin{aligned} Registers &= Int \rightarrow bvec_8 \\ TempRegisters &= Int \rightarrow (type \in Types, val \in Values \cup \{\bot\}) \\ Heap &= Addresses \rightarrow bvec_8 \end{aligned}$$

$$States = TRegisters \times Registers \times Heap$$

Note that programs compiled for IA32 may make use of 64-bit operations and registers. Common examples for this are Intel's Multi Media Extension (MMX) instructions and registers. However, as the handling of those 64-bit data types is largely equivalent to 32-bit register handling. For the sake of conciseness we omit these types here. Of course, they are supported by the SOCA Verifier. Further

details on Intel's architecture, instructions and register layout can be found in [Intel Corporation, 2009].

CPU register access. CPU registers are accessed via the `PUT` and `GET` instructions. The simplest case for those is probably the `PUT` instruction with only literal parameters, which we use as an example for explaining our notation below.

Definition 4 (PUT with literal parameters)

$$\frac{}{\langle \mathit{PUT}(\text{reg}) = \text{val} \in \mathit{Values}\text{:type} \in \mathit{Types}, (t, r, h)\rangle}$$
$$\rightsquigarrow \begin{cases} (t, [r|\text{reg} : \text{val}], h) & \text{if type} = I8 \\ (t, [r|\langle \text{reg..reg} + 1\rangle : \text{val}], h) & \text{if type} = I16 \\ (t, [r|\langle \text{reg..reg} + 3\rangle : \text{val}], h) & \text{if type} = I32 \end{cases}$$

Let us explain this definition: The `PUT` instruction has three parameters. The first of those is `reg` and denotes the first CPU register we are going to write to. The second parameter is `val`, the value we are going to write to `reg`. The last parameter is `type` and tells us what size the bit vector `val` has, and respectively, how many CPU registers we have to use in order to store it.

The most complicated case arises if `type` equals $I32$ and hence `val` has to be handled as a bit vector of length 32. Since the CPU registers store bit vectors of size 8, we have to store `val` in four of those registers such that the concatenation of those four registers again results in `val`. We write

$$(t, r, h) \rightsquigarrow (t, [r|\langle \text{reg..reg} + 3\rangle : \text{val}], h)$$

which means that only the r component of the originating (t, r, h) is updated by the `PUT` instruction in such a way that after the execution of the `PUT` (denoted by $\rightsquigarrow$), the CPU registers `reg`, `reg` $+ 1$, `reg` $+ 2$ and `reg` $+ 3$ will together hold the value of `val`, and hence

$$r(\text{reg}) \circ r(\text{reg} + 1) \circ r(\text{reg} + 2) \circ r(\text{reg} + 3) = \text{val}$$

holds true. The $\circ$-operator denotes the concatenation of two bit vectors.

Similar to the above definition of the `PUT` instruction with a literal operand, we can now easily give a semantics for more complicated cases such as `PUT` and `GET` with temporary registers being used as operands:

Definition 5 (PUT with temporary registers)

$$\frac{t(\text{treg}).val \neq \bot}{\langle \texttt{PUT}(\text{reg}) = \text{treg}, (t, r, h)\rangle}$$
$$\rightsquigarrow \begin{cases} (t, [r|\text{reg} : t(\text{treg}).val], h) & \text{if } t(\text{treg}).type = I8 \\ (t, [r|\langle \text{reg..reg} + 1\rangle : t(\text{treg}).val], h) & \text{if } t(\text{treg}).type = I16 \\ (t, [r|\langle \text{reg..reg} + 3\rangle : t(\text{treg}).val], h) & \text{if } t(\text{treg}).type = I32 \end{cases}$$

Definition 6 (GET with temporary registers)

$$\frac{t(\text{treg}).type = \text{type} \wedge t(\text{treg}).val = \bot}{\langle \text{treg} = \texttt{GET} : \text{type}(\text{reg}), (t, r, h)\rangle}$$
$$\rightsquigarrow \begin{cases} ([t|\text{treg}.val : r(\text{reg})], r, h) & \text{if type} = I8 \\ ([t|\text{treg}.val : r(\langle \text{reg..reg} + 1\rangle)], r, h) & \text{if type} = I16 \\ ([t|\text{treg}.val : r(\langle \text{reg..reg} + 3\rangle)], r, h) & \text{if type} = I32 \end{cases}$$

Byte ordering. Importantly, Valgrind provides support for multiple different CPU architectures including our target architecture IA32, but also Motorola's PowerPC CPUs and Acorn's ARM processors. As a result, tools building upon Valgrind's internals have to take special care in order to interpret word-aligned register and memory access correctly. For example, the IA32 supports only the use of the little-endian format for storing word-aligned data, which means that the least significant byte of a word-aligned data object is stored at the lowest address. PowerPC and ARM, on the other hand, support both, little-endian and big-endian (most significant byte first). However, in order to simplify logical and arithmetical operation that are carried out on temporary registers, we want those registers to hold values in the more natural big-endian format only, leaving byte-ordering conversions to the `PUT` and `GET` instructions, respectively. For the sake of simplicity, the semantic definitions of Valgrind's IR language in this section are given for big-endian architectures.

Arithmetic functions. Besides `PUT` and `GET`, Valgrind's IR provides a large set of logical and arithmetical functions ranging from negation over widening, narrowing, bit-shifting and logical conjunction and disjunction to addition, multiplication and division. All these instructions require a fixed number of temporary registers or literals as parameters and store the output in a temporary registers. Since conventions for the widths of input and output bit-vectors as well as operation-specific information, i.e. on overflow handling, are provided in Valgrind's public header files, we only give an example for the `ADD` instruction here:

Definition 7 (ADD with temporary registers)

$$\frac{t(\text{sum}).\mathit{type} = \text{type} \wedge t(\text{sum}).\mathit{val} = \bot \wedge \; t(\text{add1}).\mathit{val} \neq \bot \wedge t(\text{add2}).\mathit{val} \neq \bot}{\langle \text{sum} = \mathtt{ADD} : \text{type}(\text{add1}, \text{add2}), (t, r, h)\rangle}$$

$$\rightsquigarrow \begin{cases} ([t|\text{sum}.\mathit{val} : (t(\text{add1}) + t(\text{add2})) \; \mathit{mod} \; 2^{8}], r, h) & \text{if type} = I8 \\ ([t|\text{sum}.\mathit{val} : (t(\text{add1}) + t(\text{add2})) \; \mathit{mod} \; 2^{16}], r, h) & \text{if type} = I16 \\ ([t|\text{sum}.\mathit{val} : (t(\text{add1}) + t(\text{add2})) \; \mathit{mod} \; 2^{32}], r, h) & \text{if type} = I32 \end{cases}$$

In the above definition + denotes arithmetic addition of two bit-vectors. As the resulting bit-vector is required to have the same size as the parameters, we truncate the result using the modulo operation. Other arithmetic functions can be defined along the lines of `ADD`.

Memory access Memory access is similar to register access. Here, `ST` (store) corresponds with `PUT` and `LD` (load) resembles the `GET` instruction. However, `LD` and `ST` are used to access the main memory of the computer system. The major difference to `PUT` and `GET` is that memory addresses are provided as 32-bit-wide parameters, either as literals or as temporary registers whose content has been computed by instructions preceding the current memory access. Hence, in different executions of the same code fragment, the location addressed by `LD` and `ST` is not static as with `PUT` and `GET`.

Again, we start with a simple case, namely `ST` with literal operands:

Definition 8 (ST with literal operands)

$$\frac{}{\langle ST(\text{addr} \in Values : I32) = \text{val} \in Values : \text{type}, (t, r, h)\rangle} \rightsquigarrow \begin{cases} (t, r, [h|\text{addr} : \text{val}]) & \text{if type} = I8 \\ (t, r, [h|\langle \text{addr..addr} + 1\rangle : \text{val}]) & \text{if type} = I16 \\ (t, r, [h|\langle \text{addr..addr} + 3\rangle : \text{val}]) & \text{if type} = I32 \end{cases}$$

More commonly found are cases where temporary registers are used as parameters to the instructions:

Definition 9 (ST with temporary registers)

$$\frac{t(\text{addr}).type = I32 \wedge t(\text{addr}).val \neq \bot \wedge \\ t(\text{src}).val \neq \bot}{\langle ST(\text{addr}) = \text{src}, (t, r, h)\rangle} \rightsquigarrow \begin{cases} (t, r, [h|t(\text{addr}).\text{val} : t(\text{src}).val]), & \text{if } t(\text{src}).type = I8 \\ (t, r, [h|\langle t(\text{addr}).val..t(\text{addr}).val + 1\rangle : t(\text{src}).val]) & \text{if } t(\text{src}).type = I16 \\ (t, r, [h|\langle t(\text{addr}).val..t(\text{addr}).val + 3\rangle : t(\text{src}).val]) & \text{if } t(\text{src}).type = I32 \end{cases}$$

Definition 10 (LD with temporary registers)

$$\frac{t(\text{target}).type = \text{type} \wedge t(\text{target}).val = \bot \wedge \\ t(\text{addr}).type = I32 \wedge t(\text{addr}).val \neq \bot}{\langle \text{target} = LD : \text{type}(\text{addr}), (t, r, h)\rangle} \rightsquigarrow \begin{cases} ([t|\text{target}.val : h(t(\text{addr}).val)], r, h) & \text{if type} = I8 \\ ([t|\text{target}.val : h(\langle t(\text{addr}).val..t(\text{addr}).val + 1\rangle)], r, h) & \text{if type} = I16 \\ ([t|\text{target}.val : h(\langle t(\text{addr}).val..t(\text{addr}).val + 3\rangle)], r, h) & \text{if type} = I32 \end{cases}$$

Memory allocation and de-allocation. For supporting memory allocation and de-allocation using APIs such as `malloc()` and `free()` as defined for ANSI-C, we extend the program state by a function *HeapLocations*. This function provides a mapping from addresses to meta-information on the respective memory cell. Note that the *HeapLocations* function has no meaning for the execution of the program under consideration and does not influence its results. Instead, it provides additional information that is usually hidden inside the operating system's memory management facilities. Hence, the information stored here may vary with the properties to be checked.

Definition 11 (Heap Locations)

$$\frac{HeapLocations = \quad Addresses \rightarrow (alloc : Bool, init : Bool \\ start \in Addresses, size \in bvec_{32})}{States = \quad TRegisters \times Registers \times Heap \times HeapLocations}$$

In the context of this thesis we are interested in checking whether a particular pointer may only point to an address that belongs to a previously allocated location of the heap, and whether the respective memory cells have been initialised, i.e. written to, before they are read. Furthermore the start address and size of that location are required in order to be able to identify out-of-bounds access or invalid use of the de-allocators provided by the runtime environment of the program. According to Definition 11 we use *alloc*, *init*, *start* and *size* to retain the above information, respectively. The *HeapLocations* is denoted wit l in the command-state pairs of the semantic definitions given below. The command-state pair has to be extended to $\langle c, (t, r, h, l)\rangle$.

Below we give semantic definitions for a generic allocator *MALLOC* and de-allocator *FREE*:

Definition 12 (MALLOC)

$$\frac{\begin{array}{l} t(\text{addr}).type = I32 \wedge t(\text{addr}).val = \bot \wedge \\ t(\text{size}).type = I32 \wedge t(\text{size}).val \neq \bot \wedge \\ ((l(\text{loc..}(\text{loc}+t(\text{size}).val-1)).alloc = false \vee \text{loc} = 0) \sqcap \text{loc} = \\ 0) \end{array}}{\langle \text{addr} = GOTO\ \ MALLOC(\text{size}), (t, r, h, l)\rangle}$$

$$\rightsquigarrow \begin{cases} ([t|\text{addr}.val = 0], r, h, l) & \text{if } t(\text{size}).val = 0 \wedge \text{loc} = 0 \\ ([t|\text{addr}.val = loc], r, h, [l|\langle \text{loc..}(\text{loc} + t(\text{size}).val - 1)\rangle : & \\ \quad (true, false, \text{loc}, t(\text{size}).val)) & \text{if } t(\text{size}).val \neq 0 \wedge \text{loc} = 0 \end{cases}$$

Here the $\sqcap$-operator denotes a non-deterministic choice between the two cases

$$l(\text{loc..}(\text{loc} + t(\text{size}).val - 1)).alloc = false \vee \text{loc} = 0$$

success or failure due to lack of free memory or fragmentation and

$$\text{loc} = 0$$

non-deterministic failure.

Definition 13 (FREE)

$$\frac{t(\text{addr}).\mathit{type} = I32 \wedge t(\text{addr}).\mathit{val} \neq \bot}{\langle \mathit{GOTO\ FREE}(\text{addr}), (t, r, h, l)\rangle}$$
$$\rightsquigarrow \begin{cases} (t, r, h, l) & \text{if } t(\text{addr}).\mathit{val} = 0 \\ (t, r, h, [l|\langle t(\text{addr}).\mathit{val}..(t(\text{addr}).\mathit{val} + l(t(\text{addr}).\mathit{val}).\mathit{size} - 1)\rangle : \\ \quad (\mathit{false}, \mathit{false}, 0, 0)]) & \text{else} \end{cases}$$

Of course, extending the definition command-state pair also requires us to provide a new definition of the `ST` operation:

Definition 14 (ST with temporary registers)

$$\frac{\begin{array}{l} t(\text{addr}).\mathit{type} = I32 \wedge t(\text{addr}).\mathit{val} \neq \bot \wedge \\ t(\text{src}).\mathit{val} \neq \bot \end{array}}{\langle \mathit{ST}(\text{addr}) = \text{src}, (t, r, h, l)\rangle}$$
$$\rightsquigarrow \begin{cases} (t, r, [h|t(\text{addr}).\text{val} : t(\text{src}).\mathit{val}], \\ \quad [l|t(\text{addr}).\text{val}.\mathit{init} : \mathit{true}]), & \text{if } t(\text{src}).\mathit{type} = I8 \\ (t, r, [h|\langle t(\text{addr}).\mathit{val}..t(\text{addr}).\mathit{val} + 1\rangle : t(\text{src}).\mathit{val}], \\ \quad [l|\langle t(\text{addr}).\mathit{val}..t(\text{addr}).\mathit{val} + 1\rangle.\mathit{init} : \mathit{true}]) & \text{if } t(\text{src}).\mathit{type} = I16 \\ (t, r, [h|\langle t(\text{addr}).\mathit{val}..t(\text{addr}).\mathit{val} + 3\rangle : t(\text{src}).\mathit{val}], \\ \quad [l|\langle t(\text{addr}).\mathit{val}..t(\text{addr}).\mathit{val} + 3\rangle.\mathit{init} : \mathit{true}]) & \text{if } t(\text{src}).\mathit{type} = I32 \end{cases}$$

All other instructions require minor changes only as they do not perform updates to *HeapLocations*. For conciseness we do not present these minor modifications here.

When looking at real operating system kernels, we will notice that there are usually additional allocators and de-allocators available. Also functions mapping and unmapping parts of the file system or memory from devices attached to the system bus into the address space of the program to be analysed, are currently considered as if they were performing allocation, de-allocation as well as initialisation of memory cells. Furthermore, those functions may have additional parameters identifying a particular area of the heap in which the newly allocated memory chunk should be placed in, or control other aspects of the allocator's behaviour. For the sake of simplicity we ignore these details here. Of course, an implementation of our analysis framework has to account for some of those details while others may be irrelevant with respect to the properties we want to check.

Please note that the preconditions of the semantic definitions above only consider integrity properties of the intermediate representation. Let us for example have another look at the `LD` instruction. Its preconditions are:

$$t(\text{treg}).\mathit{type} = \text{type} \wedge t(\text{treg}).\mathit{val} = \bot$$

We only require the type of the target register matching the type of the load instruction and the target register not being previously initialised. The first precondition makes sure that we are neither loosing some bits of the result nor adding uninitialised data to the program's execution. The latter condition guarantees that the static single assignment form of the IR is preserved.

As we are reasoning about pointer safety,

$$t(\mathrm{addr}).val \neq 0$$

would be another important safety property of the program, expressing that the address to be dereferenced shall not hold the value `NULL`. However, since `NULL` is a valid register assignment that solely has a special semantics with respect to pointer operations, it is not a integrity property of the IR.

4.3 Symbolic Execution

In this section we introduce a novel approach to verifying properties in software components based on *bounded path-sensitive symbolic execution of compiled and linked programs* as illustrated in Fig. 4.8. The basic idea behind our approach employs well-known techniques including symbolic program execution, SMT solving and program slicing. However, implementing it in a way that renders the techniques scalable up to the application domain of Linux device drivers is a challenging task. As shown in the illustration, we automatically translate a program given in its *object code* into an *intermediate representation* (IR), borrowed from the Valgrind binary instrumentation framework [Nethercote and Seward, 2007], by iteratively following each program path and resolving all target addresses of computed jumps and return statements. From the IR we generate systems of bit-vector constraints for each execution path, which reflect the path-relevant register and heap contents of the program under analysis. We then employ the *Yices* SMT solver [Dutertre and de Moura, 2006] to check the satisfiability of the resulting constraint systems and thus the validity of the path. This approach also allows us to add in a range of pointer safety properties, e.g., whether a pointer points to an allocated address, as simple assertions over those constraint systems.

In contrast to other methods for software verification, our technique does not employ program abstraction but only *path-sensitive and heap-aware program slicing*, which means that our slices are not computed over the entire program but only over a particular path during execution. Furthermore, we do not consider the heap as one big data object but compute slices with respect to those heap locations that are data-flow dependents of a location in a program path for which a property is being checked. A safe over-approximation is used for computing these slices. In addition, our technique leaves most of the program's input (initially) unspecified in order to allow the SMT solver to search for subtle inputs that will drive the program into an error state. Obviously, our analysis by symbolic execution cannot be complete: the search space has to be bounded since the total number of execution paths and the number of instructions per path in a program is potentially infinite. However, our experimental results will show that this boundedness is not a restriction in practice: many programs are relatively "shallow" and may still be analysed either exhaustively or up to an acceptable depth.

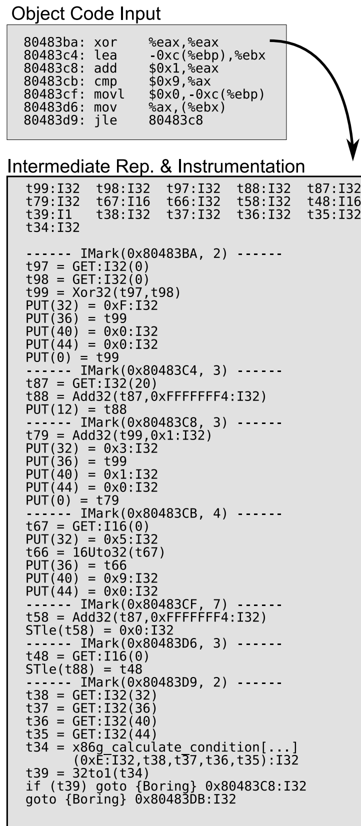

Figure 4.8.a: Starting from a given function entry point, each instruction is translated into IR.

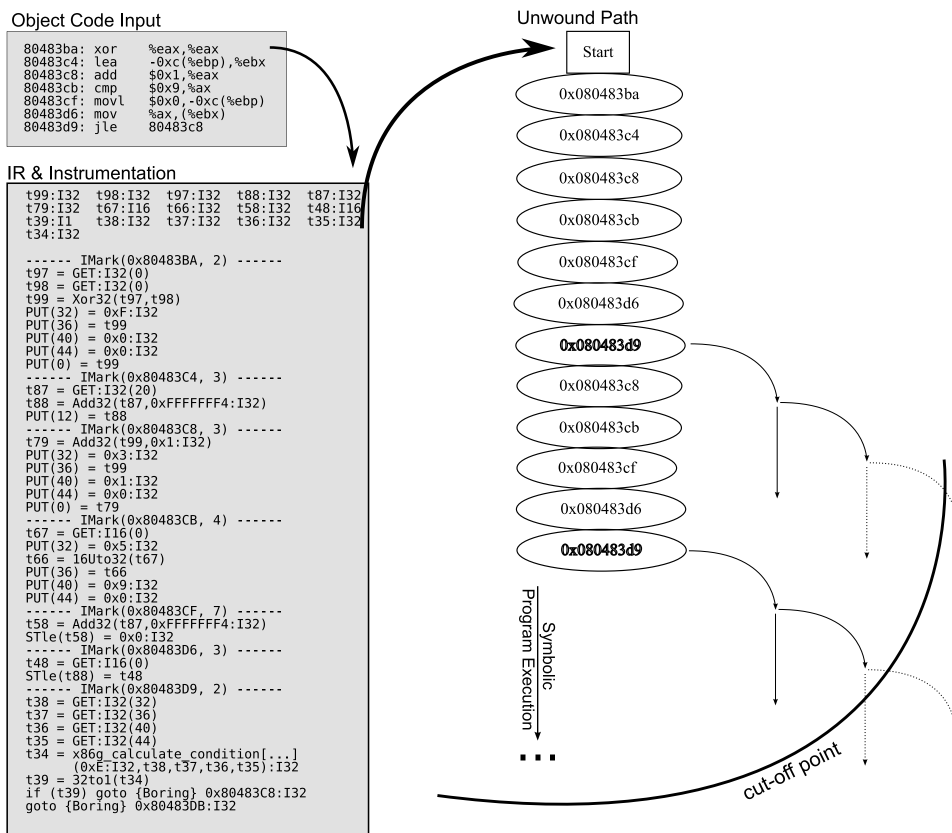

Figure 4.8.b: In order to construct paths, the IR is systematically traversed in depth-first fashion up to a certain width and and depth.

Figure 4.8: Illustration of the SOCA technique.

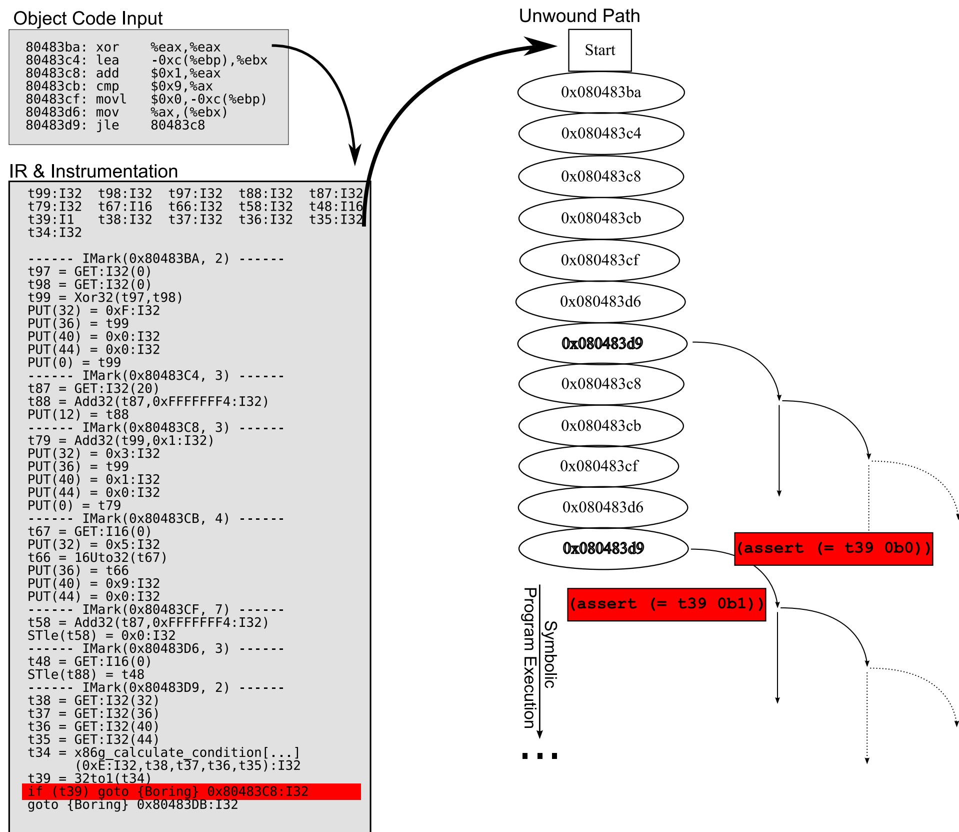

Figure 4.8.*c*: To decide which paths of the program are feasible, assertions are generated at decision points. For program statements facilitating memory access, we also generate assertions expressing the relevant pointer-safety properties at this instruction.

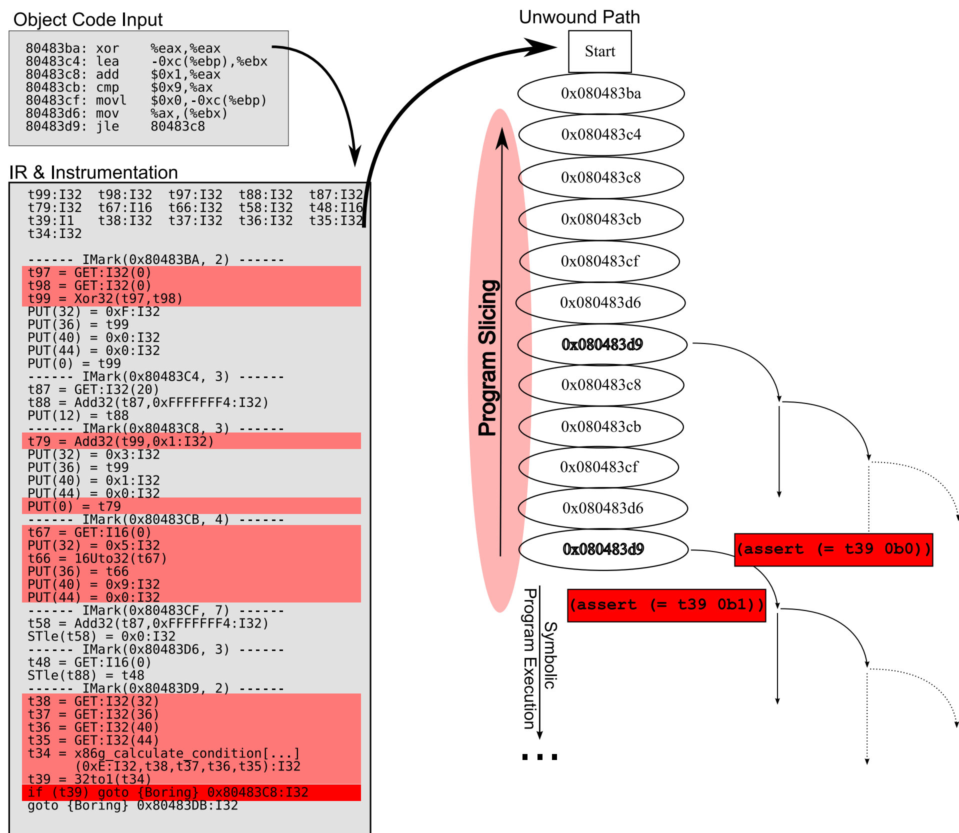

Figure 4.8.*d*: For each assertion we compute a path-sensitive program slicing containing only those program statements that affect the decision variable or pointer, and hence are required for checking the satisfiability of assertions.

Figure 4.8: Illustration of the SOCA technique.

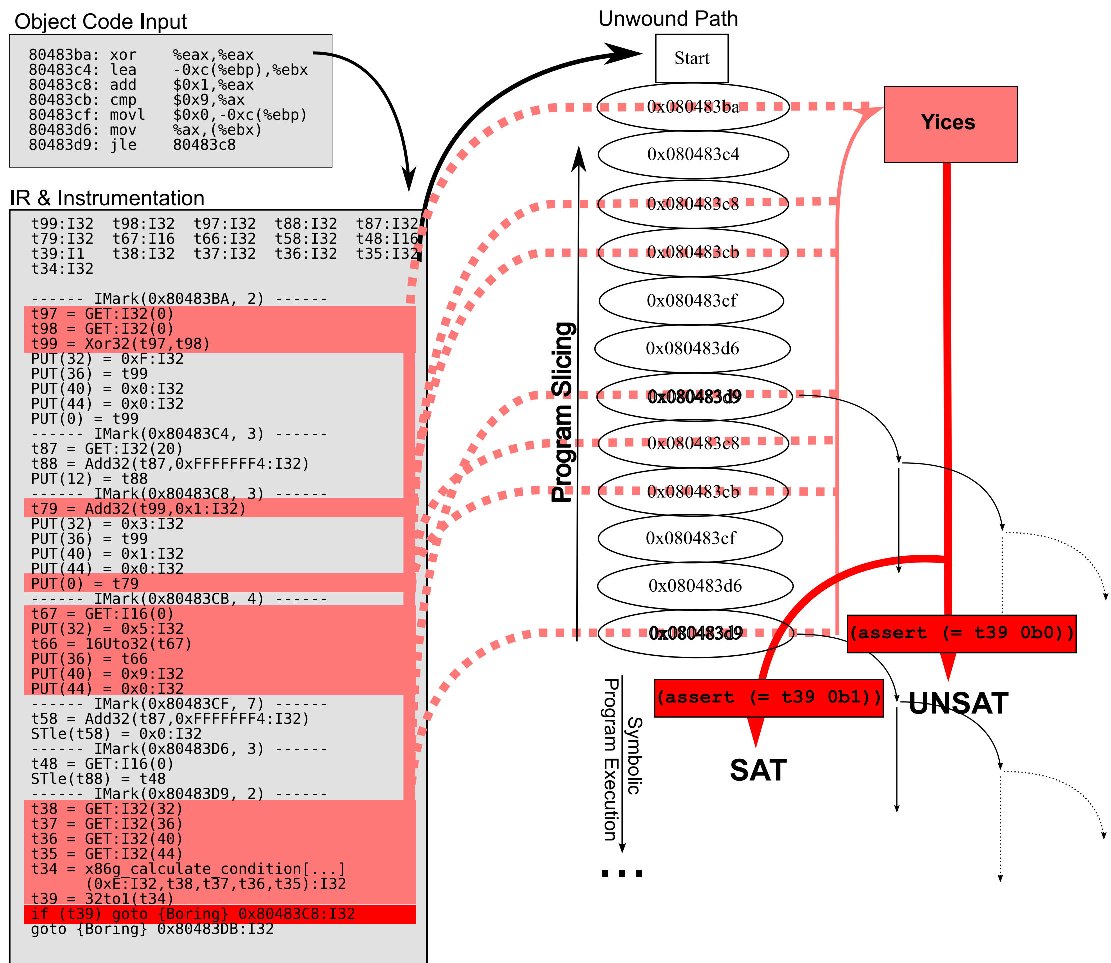

Figure 4.8.*e*: The slice and assertions are translated into a bit-vector constraint problem, which then checked for satisfiability by invoking the Yices SMT solver.

Figure 4.8: Illustration of the SOCA technique.

Using the operational semantics for Valgrind's IR language as outlined in the previous section, we are now able to translate IR instructions into bit-vector constraint systems for Yices[2] [Dutertre and de Moura, 2006]. Given that the IR is in static single assignment form we can simply translate an instruction such as the first *PUT* statement from Fig. 4.6 as follows:

IR Instruction	**Constraint Representation**

`PUT(0) = t7`	`(define r0::(bitvector 8)(bv-extract 31 24 t7))`
	`(define r1::(bitvector 8)(bv-extract 23 16 t7))`
	`(define r2::(bitvector 8)(bv-extract 15  8 t7))`
	`(define r3::(bitvector 8)(bv-extract  7  0 t7))`

Note that the CPU registers are assigned in "reverse byte order", i.e. with the least significant 8 bits in *r0* and the most significant bits in *r3*, to the temporary registers. That is because the above constraints are generated from a binary compiled for IA32 which uses this particular encoding, while arithmetic expressions in Yices are implemented for bit-vectors that have the most significant bit at position 0. Since access operations to the guest state may be 8, 16, 32 or 64 bit aligned, we have to use two different encodings here.

Similar to the *PUT* instruction we can express *GET* or the *Xor* and *Add* instructions in terms of bit-vector constraints for Yices:

IR Instruction	**Constraint Representation**
`t9 = GET:I32(0)`	`(define t9::(bitvector 32) (bv-concat` ` (bv-concat r3 r2) (bv-concat r1 r0))`
`t7 = Xor32(t9,t8)`	`(define t7::(bitvector 32) (bv-xor t9 t8))`
`t41 = Add32(t42,` ` 0xFFFFFFF4:I32)`	`(define t41::(bitvector 32)` ` (bv-add t42 (mk-bv 32 4294967284)`

Since our analysis handles loops by unrolling them while exploring a path, a single instruction might appear multiple times in that path. Furthermore, the IR is in static single assignment form only with respect to the temporary registers within a

[2]The syntax of Yices' input language is explained at `http://yices.csl.sri.com/`.

single IR block. Hence, we have to be more precise when generating variable names. The rule applied by the implementation of our technique appends the instruction's location and the invocation number to each variable.

While the translation method explained above can be applied for operations working on registers and temporary registers only, it cannot be used for operations accessing the heap or stack. To explain this, let us consider the two IR statements:

```
01  STle(t5) = t32
02  t31 = LDle:I32(t7)
```

The semantics of those two statements is quite similar to that of PUT and GET. In order to be as close as possible to the actual IA32 architecture, we define the underlying memory representation as an array of memory cells of eight bits each that are accessed using an index of 32 bit length. Now we can define that the ST will update the memory cells indexed by $t5..t5 + 3$ by storing the value held by the 32-bit-wide temporary register $t32$. Of course, in order to do this, $t32$ needs to be disassembled into 8-bit-wide bit-vectors in the same way as shown for the PUT instruction above. Respectively, the LD instruction will write the concatenation of the memory cells indexed by $t7..t7 + 3$ to $t31$. Byte ordering issues apply in the same way as explained for register access above.

The main difference of these instructions to PUT and GET is that the target of the store or the source of the load instruction is variable and may be computed at runtime. In order to include these statements in our symbolic execution framework we have to express them in a very flexible way in order to allow the SMT solver to identify cases in which safety properties are violated.

Our representation of the main memory in Yices is that of a function from 32 bit wide bit-vectors (the pointer) to bit-vectors of size 8 (the memory cell, respectively). We write:

```
(define heap::(-> (bitvector 32) (bitvector 8)))
```

Now the above store instruction can be expressed as an update of that function:

```
(define heap.0::(-> (bitvector 32) (bitvector 8))
 (update heap   ((bv-add t5 (mk-bv 32 3))) (bv-extract  7  0 t32)))
(define heap.1::(-> (bitvector 32) (bitvector 8))
 (update heap.0 ((bv-add t5 (mk-bv 32 2))) (bv-extract 15  8 t32)))
(define heap.2::(-> (bitvector 32) (bitvector 8))
 (update heap.1 ((bv-add t5 (mk-bv 32 1))) (bv-extract 23 16 t32)))
(define heap.3::(-> (bitvector 32) (bitvector 8))
 (update heap.2 ((bv-add t5 (mk-bv 32 0))) (bv-extract 31 24 t32)))
```

Constraints for the load instruction are generated analogous to the GET as explained above.

Encoding safety assertions. Being able to translate the entire program into constraints makes it rather easy to express our properties given in Section 4.1 in terms of assertions on the resulting constraint systems. The simplest case for such an assertion is a null-pointer check. For the store instruction in the above example, we could state this assertion as:

```
(assert (= t5 (mk-bv 32 0)))
```

If the resulting constraint system is satisfiable, Yices will return an evidence, i.e. a possible input assignment that will drive the program into a state in which $t5$ will be `NULL` at the above program point.

However, most memory safety properties require additional information to be known about the program's current execution context. In particular, answering the question whether a pointer may point to an "invalid" memory area requires us to know, which cells are currently allocated. We retain this information by adding a function named *HeapLocations* to our model that is updated whenever memory is allocated or de-allocated:

```
(define heaploc::(-> (bitvector 32) (record alloc::bool init::bool
 start::(bitvector 32) size::(bitvector 32))))
```

We can now express a property saying that the pointer $t5$ has to point to an allocated address at the program point where it is dereferenced as:

```
(assert (= (select (heaploc t5) alloc) false))
```

All other properties mentioned in Section 4.1 may be expressed along the lines of those two examples. Most of them require further additional information, such as the API that has been used to allocate or deallocate some memory cells, to be added to the *HeapLocations* function. In order to reduce the size and search space of the resulting constraint systems, we check assertions one-by-one with a specialised *HeapLocations* function for each property.

Symbolic execution. The core component of our symbolic execution framework translates a given program starting from some entry point into its intermediate representation and then into bit-vector constraints. There are three cases in which we have to call Yices in order to check the generated constraints for satisfiability: *(a)* a given statement is a computed jump, e.g. `goto t7` or a function return. In those cases we have to compute the target address of the jump or return statement in order to be able to continue analysing this path of the program. *(b)* the statement contains a guard for a jump statement, e.g. `if (t13) goto 0x80483C8:I32`. Here we have to check whether the guarding condition may evaluate to true or false in order to be able to follow only branches for which the guard is satisfiable. *(c)* The last and most interesting case occurs when a temporary register is dereferenced as a pointer, e.g. `STle(t7) = t12`. In that case we want to check whether our memory safety assertions are satisfiable. This is done as described above.

However, in any case we do not run Yices on an entire path's constraint system. Instead we compute a path-sensitive slice of that path. Program slicing, introduced in [Weiser, 1981], is a technique for automatically selecting only those parts of a program that may affect the values computed at some point of interest, based on its control and data flow. Within the last years, various slicing techniques have been developed. A comprehensive survey on these techniques is given in [Tip, 1994]. The approach to program slicing used in this paper employs a slicing algorithm based on program dependence graphs as introduced in [Ottenstein and Ottenstein, 1984] and extended for slicing multi-procedure programs in [Horwitz et al., 1990], using the notion of a system dependence graph. In difference to conventional slicing as discussed above, our slices are computed over a single path instead of the entire program's control flow. In that aspect, our approach to program slicing is similar to what has been introduced as *dynamic slicing* in [Korel and Laski, 1990] and *path slicing* in [Jhala and Majumdar, 2005]. By contrast with those approaches' methods, we use conventional slicing criteria (L, var) denoting a variable var that is used at program location L. Slicing criteria for dynamic slicing and path slicing are given in terms of a well defined input to a program or a (potentially infeasible)

counterexample trace, as well as a location of interest and a set of variables. In difference to that, our approach aims to compute inputs that will lead to a particular path being executed. Hence, we leave the program's input initially unspecified. The slice is then computed by collecting all statements of which *var* is data dependent by tracing the path backwards, starting from L up to the program entry point. While collecting flow dependencies is relatively easy for programs that do only use CPU registers (and temporary registers in our IR), it becomes difficult when dependencies to the heap and stack are involved.

Handling memory access in slicing. Let us have a second look at the LD and ST statements from page 66. In order to compute a small slice for $(02, t31)$ we have to know whether the store statement in l. 1 may affect the value of $t31$, i.e., whether $t5$ and $t7$ may alias. We obtain this information by using Yices to iteratively compute the potential address range that can be accessed through $t5$. This is done by making Yices compute an evidence, i.e. a possible assignment, e for $t5$, and the computing further evidences e' such that $e > e'$ or $e < e'$ holds, until the range is explored. Of course this is an over-approximation as not the entire range may be addressable by the pointer. However, using this abstraction presents a trade-off concerning only the computation of minimal slices. That means, instead of computing and storing all satisfying assignments for a particular pointer (2^{32} in the worst case), we are able to keep the number of Yices runs as well as the amount of data to store small. As a drawback, our technique may produce unnecessarily large slices in the presence of symbolic pointers. Nevertheless, our approach is conservative with respect to the property to be verified.

We store those ranges in a memory tree, an idea borrowed from [Ferdinand et al., 2007], a model handling memory accesses and their access widths dynamically. The approach uses a binary tree structure where each node is labelled with an interval denoting the boundaries of the memory cells it represents. A leaf is labelled with a set of points denoting the program points defining the memory cells represented by the leaf.

By computing the address range possibly accessed by a pointer used in a load statement, i.e. $t7$ in our case, and traversing the memory tree looking for memory intervals overlapping with the range of $t7$, we can now determine which store operations may affect the result of the load operation. Despite being conservative when computing address ranges, our experience shows that most memory access operations end up having very few dependencies as most pointers evaluate to a concrete address and not a range.

4.4 Complications and Optimisations

Handling computed jumps. A major issues when analysing compiled programs arises from the extensive use of code pointers and jump target computations. While most source-code based approaches simply ignore function pointers, this cannot be done when analysing object since code jump computations are too widely used here. Two examples for this are:

```
01  ;; Return statement:
02  t8 = GET:I32(16)
03  t9 = LDle:I32(t8)
04  t26 = Add32(t8,0x4:I32)
05  PUT(16) = t26
06  goto {Return} t9
```

```
01  ;; Call to a library function:
02  t0 = LDle:I32(0x80495D8:I32)
03  goto {Call} t0
```

In both cases the target address of the jump has to be loaded from the memory and may differ in multiple invocations of the same instruction from different program contexts. In our approach, jump target addresses are determined in the same way as addresses for load and store operations. This is done by computing slices for $(06, t9)$ or $(03, t0)$ for the return statement or the function call, respectively and then iteratively computing the address ranges for the two pointers.

If Yices returns only one possible target address, we extend the program's control flow representation and the current path dynamically with the instruction blocks reachable for that target. On the other hand, if $t9$ or $t0$ are symbolic pointers, we terminate the path at this point since following each possible address would lead to an explosion in the number of paths, and also to unsound results since many pointer assignments may be due to missing information in the initial memory state, and hence may actually be infeasible in practice. However, the latter case happens rarely, practically only in case a function to be analysed gets a function pointer passed as its argument. We show in Section 4.5 that only a small percentage of drivers of our sample exhibit this behaviour, while the majority of drivers can be analysed exhaustively despite this limitation.

Optimising GET and PUT statements. One major problem with respect to the scalability of our approach arises from the vast number of GET and PUT statements shown in Fig. 4.6. The reason for this is in our adaptation of Valgrind's IR: temporary registers are usually stored in the guest state at the end of each CPU instruction and may be reloaded in several following instructions. In fact, Valgrind is able to optimise the IR in a way that removes a majority of those statements.

However, in order to simplify the handling of jumps, we decided to turn this optimisation off. This allows each IR block to be entered at various points and hence saves us time and memory for translating and maintaining multiple IR blocks holding subsets of the instructions of another block. However the frequent de- and re-composing of temporary registers into 8-bit-wide guest-state registers and back into temporary registers introduces lots of additional variables in the SMT solver and makes it run out of memory rather quickly.

An efficient way around this issue is to optimise unnecessary GET and PUT operations away based on the actual path we are analysing. Let us look at another piece of IR obtained from the example program shown in Fig. 4.4:

```
;; 0x80483cb (cmp)            ;; 0x80483d9 (jle)
t25 = GET:I16(0)              t49 = GET:I32(32)
IR-NoOp                       t50 = GET:I32(36)
PUT(32) = 0x5:I32             t51 = GET:I32(40)
t43 = 16Uto32(t25)            t52 = GET:I32(44)
PUT(36) = t43                 t53 = x86g_calculate_condition[mcx=0x13]
PUT(40) = 0x9:I32              {0x808c940}(0xE:I32,t49,t50,t51,t52):I32
PUT(44) = 0x0:I32             t48 = 32to1(t53)
...                           if (t48) goto {Boring} 0x80483C8:I32
```

We see that the *cmp* instruction is decomposed into several instructions. Four of those are PUT statements storing values to registers of the guest state. The same registers are read by the GET statements at the beginning of the *jle* instruction and there are no further write operations to these registers in between, while the temporary registers are in static single assignment form in any case. However, we can also see that the temporary registers written to the guest state have the same size as the ones that are read; hence they will hold the same values and no byte-ordering conversions are required. Hence, we may simply remove the affected PUT and GET statements by assigning, for example $t50 = t43$, or go even further and replace the temporary register $t50$ in the *x86g_calculate_condition* statement with $t43$.

There are several cases where this optimisation is not possible. Examples for this are code sequences in which a 32-bit value is written to the guest state and a 16-bit value is read at a later point in the program flow from the same register. In those cases the changes of the byte-ordering performed by PUT and GET operations are required to preserve the semantics of the program we are analysing.

Practical results show that this simple optimisation reduces the memory consumption of Yices for large constraint systems ($>$ 10,000 constraints) by up to 90%.

Hence it prevents a large quantity of Yices runs from terminating without returning a result due to timeouts or memory exhaustion.

Determining a valid initial memory state. Another challenge when implementing symbolic execution as an SMT problem is given by the enormous search space that may result from leaving the program's initial memory state undefined. As a result, the SMT solver tends to run out of memory regularly, or slows down the whole analysis. Furthermore, even unsound results in pointer computations are possible as those regularly employ fixed values taken from the initial heap or stack of the binary program.

To make our approach scale to the desired application domain, we compute an initial memory tree from the information given in the device driver's object code. For all loadable program sections assigned in the binary (cf. [Tool Interface Standards (TIS) Committee, 1995]), we create leave nodes in the memory dependency tree as explained in Section 4.3. If our analysis determines that a particular address or range of addresses is accessed by a pointer within a slice, we generate constraints for the initial memory cell assignment of that particular range of addresses and prepend them to the constraints in the slice before passing the entire constraint system to the SMT solver.

As we explain in Chapter 5, OS components including functions taken from device drivers, make regularly use of an external data environment consisting of heap objects allocated and initialised by other modules of the OS. Hence, this data environment cannot be inferred from the information available in the program binary. In Chapter 5 we show that data environments can easily be embedded into the analysis by adding just a few lines of C code as a preamble to our analysis. However, doing so requires one to have specific knowledge of the employed data objects employed by a function to be analysed. Hence, doing so is not a difficult task in general but could not be done for the large number of functions analysed in Section 4.5. As a result of this, our analysis reports higher ratios of false-positive errors than initially expected.

4.5 Experimental Results

In order to evaluate the SOCA technique with respect to its ability to correctly identify pointer safety issues as well as to evaluate its performance when analysing operating system components, the SOCA Verifier, which implements our technique, was developed. In this section we outline the SOCA Verifier's architecture and report on extensive experiments conducted by applying the SOCA verifier to a benchmark

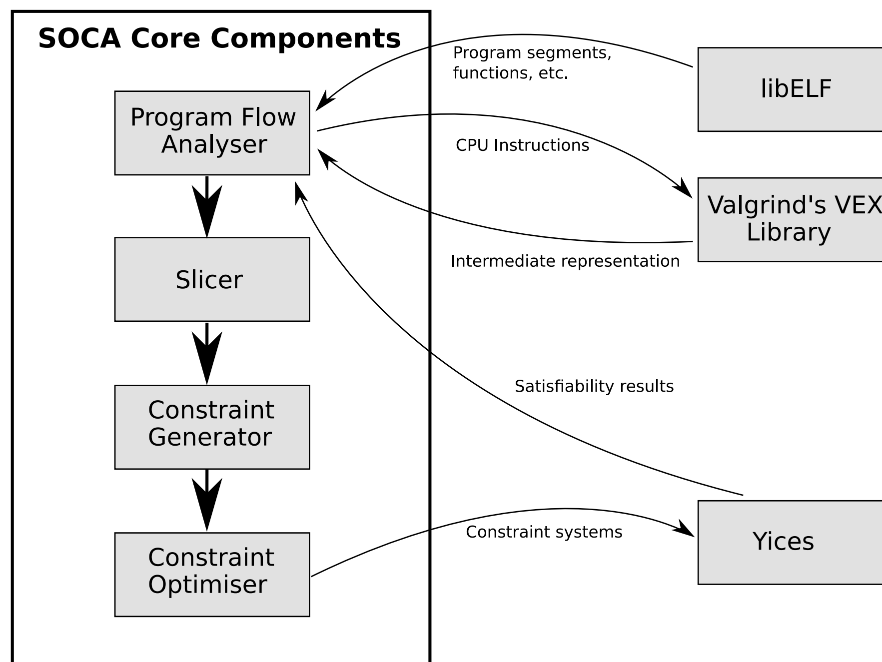

Figure 4.9: Software architecture of the SOCA Verifier

suite for software model checkers as well as to a large set of Linux device drivers.

4.5.1 Tool Development

The current implementation of the SOCA Verifier is written in C, mainly for facilitating integration with Valgrind's VEX library (cf. [Nethercote and Seward, 2007], [Valgrind, 2009]). In Fig. 4.9 we outline the Verifiers software architecture. The components developed for this thesis are those labelled as "SOCA Core components", comprising of a total of 15,000 lines of code (LOC). We interface with three external components that are used for parsing binary program files in the ELF format (*libELF*, [Koshy, 2009]), translating CPU instructions into IR (*Valgrind's VEX Library*, [Valgrind, 2009]) and for solving bit-vector constraint problems (*Yices*, [Dutertre and de Moura, 2006]). All these components are available for multitude of different computer architectures. Hence, we believe that the SOCA Verifier can be easily adapted to check programs for platforms other than IA32.

As shown in Fig. 4.9, the core components of the SOCA Verifier comprise a *Program Flow Analyser*, a *Slicer*, a *Constraint Generator* and a *Constraint Optimiser*. The Program Flow Analyser is the central component of our tool. It consists of about 4300 LOC implementing the systematic traversal of the object code in a depth-first manner, passing every instruction reachable from a given program entry point to the VEX library in order to obtain its IR. The Flow Analyser then iden-

tifies control dependencies and data dependencies for each IR statement based on traditional data flow analysis (cf. [Nielson et al., 1999]), and generates assertions for branching conditions and pointer dereferences. The assertions are then used as slicing criteria by the Slicer (1400 LOC), which computes a path-slice for the slicing criterion with respect to the path currently analysed. Slices are passed to the Constraint Generator and further to the Constraint Optimiser which transform the IR statements of a slice into bit-vector constraints for Yices as explained above. These two components are the biggest part of the SOCA Verifier, consisting of 5800 LOC which is due to the multitude of different IR instructions that have to be translated into constraints.

For the purpose of analysing operating system components, our implementation of the Constraint Generator is fairly complete with respect to the supported IR statements. We currently support 74 out of about 110 instructions commonly used in optimised driver binaries. Floating point arithmetic (which is not used within the Linux kernel), operations working on 64-bit registers and a large number of CPU extensions recently integrated into IA32 processors for multimedia acceleration, are largely unsupported at the moment. However, with the existing tool framework we have available implementing a new CPU instruction usually takes not more than 30 LOC and can be done within hours. Hence, we believe that our tool can easily be completed and even extended to cope with new application domains such as analysing application level programs rather than operating system components.

4.5.2 Small Benchmarks: Verisec

For enabling qualitative comparison of our technique with other tools we applied the SOCA verifier to the Verisec benchmark suite [Ku et al., 2007]. Verisec consists of 298 test cases (149 faulty programs and 149 corresponding fixed programs) for buffer overflow vulnerabilities taken from various open source programs. These test cases are given in terms of C source code and provide a configurable buffer size, set to 4 in the experiments conducted by us. The test cases had to be compiled to binaries using `gcc` in order to be analysed by the SOCA verifier. In previous work [Kroening et al., 2008; Ku et al., 2007] the benchmark suite has been used to evaluate the C-code model checkers SatAbs [Clarke et al., 2005] and LoopFrog [Kroening et al., 2008]. For comparison of our technique, we use the metrics proposed in [Zitser et al., 2004][3]: in Table 4.1 we report the *detection rate* $R(d)$, the *false positive rate* $R(f)$ and the *discrimination rate* $R(\neg f|d)$. The latter is defined as the ratio of test cases for which an error is correctly reported, while it is, also correctly, not reported in

[3] We do not use Zitser's test suite as it is not publicly available.

Table 4.1: Detection rate $R(d)$, false positive rate $R(f)$ and discrimination rate $R(\neg f|d)$ for SatAbs, LoopFrog and SOCA

	$R(d)$	$R(f)$	$R(\neg f\|d)$
SatAbs (from [Ku, 2008])	0.36	0.08	n/a
LoopFrog (from [Kroening et al., 2008])	1.0	0.26	0.74
SOCA	0.66	0.23	0.81

the corresponding fixed test case. Hence tools are penalised for not finding bugs, but also for not reporting a fixed program as safe.

As the above table shows, our technique reliably detects the majority of buffer overflow errors in the benchmarking suite. However, our detection rate is still lower than the one reported for the LoopFrog tool. An explanation for this can be found in the nature of the given test cases: in most test cases, static arrays are declared together with other program variables at the beginning of a `main()` function. This program setup renders the benchmarking suite easily comprehensible for source-code based verification tools since the bounds of the different data objects are clearly visible in the source-code representation. However, in the object code obtained by compiling the test case, the boundaries of data objects are not visible anymore. For example, an array consisting of four one-byte elements followed by a 32-bit index variable results in an 8-byte data section in the binary only, making it virtually impossible to discriminate between the array and the index variable. While source-code based techniques may be able to identify an overflow error in this scenario as soon as the array is accessed at a position greater than three, the SOCA technique will only be able to notice it when an access exceeding the bounds of the program's data segment (i.e. at indices greater than 7) occurs. This renders our tool less efficient for analysing programs with small, statically declared buffers.

However, the SOCA technique still shows a lower false positive rate and a better discrimination rate than the other tools. Remarkable is also that the SOCA verifier failed for only four cases of the Verisec suite: once due to memory exhaustion and three times due to unimplemented features in our tool which can easily be added by investing more development efforts. According to Ku [Ku, 2008], the SatAbs tool crashed in 73 three cases and timed out in another 87 cases. Ku's experiments were conducted with a timeout of 30 minutes. As shown in Fig. 4.10.a, the runtime of the SOCA verifier exceed this time in only 7 cases.

Despite having used a benchmark suite providing examples which are in favour of source-code analysis, our results show that object-code analysis as implemented in the SOCA Verifier can compete with state-of-the-art source-code checkers. However,

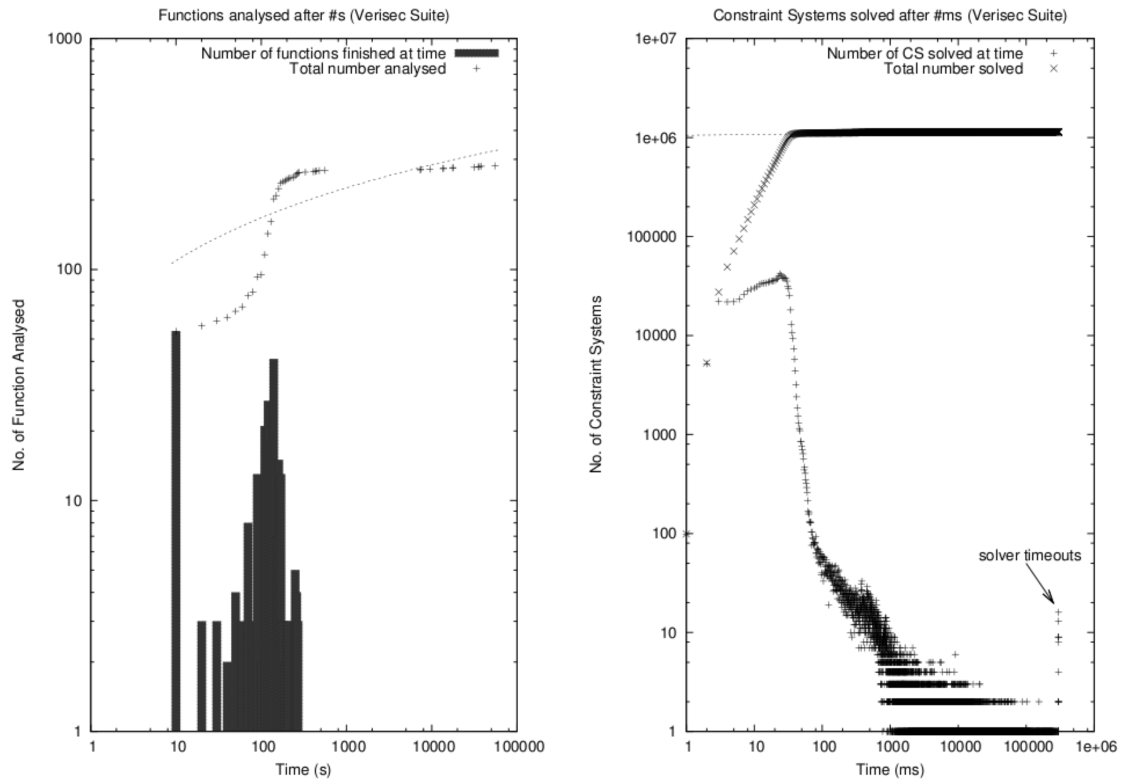

Figure 4.10: Performance results for the Verisec suite. Left: *(a)* numbers of test cases verified by time. Right: *(b)* numbers of constraint systems solved by time.

as our tool analysis object code, it can be employed in a much wider application domain. Unfortunately, benchmarking suites that include dynamic allocation and provide examples of pointer safety errors other than buffer overflows are, to our knowledge, not available.

In addition to the above comparison with other verification tools, Fig. 4.10 gives an overview of the SOCA Verifier's general performance for small-scale programs. Fig. 4.10.a shows the CPU times consumed for analysing the different test cases in the Verisec suite. It can be seen that the vast majority of test cases is analysed within less than three minutes per test case. Only in 38 cases this time is exceeded due to extensive loop unrolling. However, as presented in Table 4.2, the average computation time consumed per test case is 18.5 minutes. In total, about 92 CPU hours have been used. Employing a 16-core compute box with 2.3 GHz clock speed per CPU and a total of 256 GB of RAM, the experiment was conducted in about 6 hours.

In Fig. 4.10.b we show the behaviour of Yices for solving the constraint systems generated by the SOCA Verifier. For the Verisec suite, a total of 11,994,834 con-

Table 4.2: Performance statistics for the Verisec suite

	average	standard deviation	min	max	total
per test case					
total runtime	18m30s	1h33m	162ms	15h21m	91h54m
slicing time	28s150ms	41s808ms	28ms	5m15s	2h19m
Yices time	17m59s	1h33m	110ms	15h20m	89h19m
no. of CS	4025.11	173.76	11	8609	11994834
pointer operations	8.73	37.74	4	242	2603
per Yices invocation					
runtime	267ms	4s986ms	1ms	5m	88h59m
CS size	891.64	7707.95	0	368087	
memory usage	6.82MB	46.54MB	3.81MB	2504.36MB	

straint systems are solved in 89 hours. With the timeout for Yices set to 5 minutes, the solver timed out for 34 constraint systems, while 96% of the generated constraint systems were solved in less than one second. A total of 2,250,878 (19%) constraint systems is used to express verification properties, while the other constraint systems were required to correctly follow the program's control flow, i.e., to decide branching conditions and resolve computed jumps. Again, average timings, constraint system sizes and memory consumptions are given in Table 4.2.

4.5.3 Large-Scale Benchmarks: Linux Device Drivers

In order to evaluate the performance and scalability of the SOCA Verifier, a large set of 9296 functions originating from 250 Linux device drivers of version 2.6.26 of the Linux OS compiled for IA32, is analysed. The selection criterion for the drivers is to consider only those drivers that require only functionality provided by the kernel and not by other drivers. This selection has been made because our tool chain does currently not support analysing multiple drivers at once, however, implementing this feature should be trivial.

The tool chain used in our experiments employs `nm` to obtain a list function symbols present in the `.text` section of a given device driver object. The driver object is then statically linked against the Linux kernel to resolve undefined symbols in the driver, i.e., functions provided by the OS kernel that are called by the driver's functions. The SOCA technique is then applied on the resulting binary file to analyse each of the driver's functions separately.

While our technique is in principle capable of tracing into all functions called by

the target function, there are a few cases where we decided not to do so. Instead, the current implementation of the SOCA technique provides a set of built-in instrumentations for certain functions of the kernel. The rationale behind this is that various functions used by the driver perform I/O operations that have no meaning with respect to the analysis since we do not include a model of the underlying physical devices a driver is supposed to operate in our symbolic execution runs. The most common example for this are the `printk()` function, the kernel's equivalent for `printf()`, which is used to write out messages. Our instrumentation of this function does only dereference all given parameters and checks the alignments and null-termination of strings the parameters point to. However, the code that actually prints the message is omitted. A second group of functions we provide instrumentations for, are those used for memory (de-)allocation. That is because the different (de-)allocation APIs provided by the kernel are assumed to behave the same with respect to our heap model. Furthermore, functions like `mmap()` are considered as simple memory allocation as well. Finally, all functions influencing the concurrent behaviour of a driver are replaced with stubs as well. That is because calls to the scheduler or the locking of resources are irrelevant for the sequential program executions our work focuses on. As most locking APIs get a pointer to a particular lock passed as their arguments, we do check the validity of those pointers. Our instrumentations are done on the level of the IR, and hence no source code is required to perform the analysis of any given function.

The bounds for the SOCA Verifier were set to a maximum of 1000 paths to be analysed, where a single instruction may appear at most 1000 times per path, thereby effectively bounding the number of loop iterations or recursions to that depth. The Yices SMT solver was set to a timeout of 300 seconds per invocation.

General results. Our test suite consists of a total of 9296 functions taken from 250 Linux device drivers. The promising result of our work is that 95.3% of the functions in the sample could be analysed without failure in our tool chain. In 67.5% of the sample the exhaustion of execution bounds led to an early termination of the analysis. However, the analysis reached a considerable depth in those cases, analysing paths with a length of up to 22,577 CPU instructions. Most interestingly, 27.8% of those functions could be analysed exhaustively. Here exhaustiveness means, that none of the bounds regarding the number of paths, the path length or the SMT solver's timeout where ever reached. As shown in Fig. 4.11.*a*, in the majority of cases, our analysis returns a result in less than 10 min, while the constraint systems generated by our tool can usually be solved in less than 500 ms, and the timeout for Yices (set to 5 min) is hardly ever reached (cf. 4.11.*b*). The analysis was

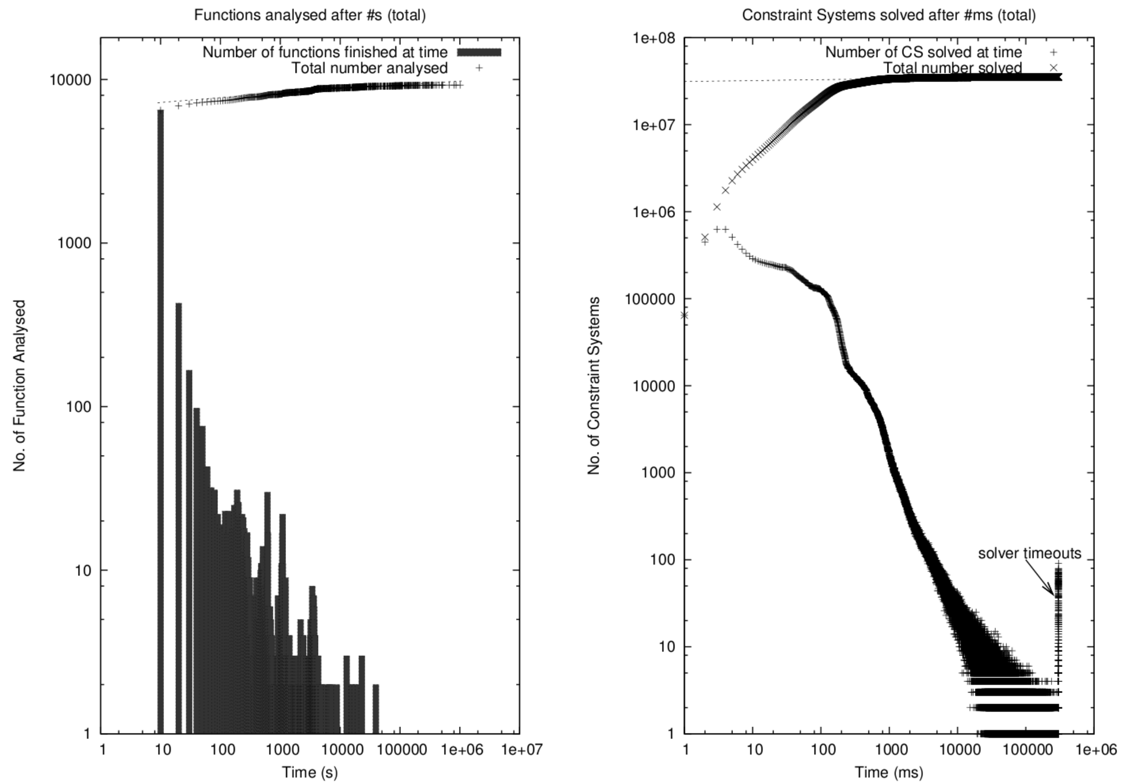

Figure 4.11: Performance results for the Kernel modules. Left: *(a)* numbers of test cases verified by time. Right: *(b)* numbers of constraint systems solved by time.

carried out on a 1.5 GHz 8-core PC with and 12 GB of RAM and on a 16-core PC with 2.3 GHz clock frequency and 256 GB of RAM. As we were not exclusively using these machines – especially the 16-core PC was under heavy loads from other experiments, we cannot determine the total CPU-hours used by our experiments and all measures presented here are absolute times measured by our tool and Yices that may include sleep times due to scheduling. The total time consumed for conducting our experiment amounts to 9058 hours, we assume that this is equivalent to about 4500 CPU-hours on exclusively used machines.

In 0.98% (91 functions) of the sample functions our tool may have produced unsound results due to non-linear arithmetic in the generated constraint systems, which is currently not decidable by Yices. Our SOCA Verifier failed in 5.6% (522 functions) of the cases due to memory exhaustion, missing support for particular instructions or functions in our tool or Valgrind, as well as due to crashes of Yices. We believe that all those issues can be solved by investing substantial effort in tool development.

Table 4.3: Performance statistics for the Kernel modules

	average	standard deviation	min	max	total
per test case					
total runtime	58m28s	7h56m	21ms	280h48m	9058h32m
slicing time	8m35s	2h13m	0	95h39m	1329h46m
Yices time	48m36s	7h28m	0	280h30m	7531h51m
no. of CS	3591.14	9253.73	0	53449	33383239
pointer operations	99.53	312.64	0	4436	925277
no. of paths	67.50	221.17	1	1000	627524
max path lengths	727.22	1819.28	1	22577	
per Yices invocation					
runtime	845ms	8s765ms	1ms	5m2s	8295h56m
CS size	4860.20	20256.77	0	7583410	
Memory usage	5.75MB	14.76MB	3.81MB	3690.00MB	

Error reports and false positives. In this case study, our analysis of the device drivers is focused on identifying possible null-pointer dereferences. The SOCA Verifier revealed a total of 887 program locations at which a pointer may hold the value `NULL` when it is dereferenced. Since our approach is based on unrolling loops, it may report a single error location multiple times, namely as often as the loop is unrolled. For the results presented here, the bound for loop unrolling is set to 1000 – indeed, in a few cases, a single program location was reported up to 1000 times. The SOCA verifier issued a total of 472,351 warnings during the experiment conducted here. However, only a small subset of these error traces has been analysed in detail yet. That is because doing so currently requires one to manually establish a mapping from the error trace and heap content reported by our tool and with respect to the program's object code representation, to the source code and then decide whether the reported initial heap state may actually be generated by the execution environment the function under analysis may be executed in. In general this is comprises of several hours of work per program trace, which is currently not automated at all. Provided that many functions utilised in this case study make use of external data environments that have not modelled explicitly (i.e. not as in Chapter 5), this case study can be expected to show a substantially higher false-positive-rate than the comparison using the Verisec suite in Section 4.5.2.

Chapter 5

Beyond Pointer Safety: The Linux Virtual File System

In the context of the grand challenge proposed to the program verification community by Hoare [Hoare, 2003], a mini challenge of building a verifiable *file system* (FS) as a stepping stone was presented by Joshi and Holzmann [Joshi and Holzmann, 2007]. As FSs are vital components of operating system kernels, bugs in their code can have disastrous consequences. Unhandled failure may render all application-level programs unsafe and gives way to serious security problems.

In this chapter, we apply an analytical approach to verifying an implementation of the *Virtual File System* (VFS) layer [Bovet and Cesati, 2005] within the Linux operating system kernel, using our novel, automated *Symbolic Object-Code Analysis* (SOCA) technique explained in Chapter 4. As described in Section 5.1, the VFS layer is of particular interest since it provides support for implementing concrete FSs such as EXT3 and ReiserFS [Bovet and Cesati, 2005], and encapsulates the details on top of which C POSIX libraries are defined; such libraries in turn provide functions, e.g., *open* and *remove*, that facilitate file access. Our case study aims at checking for violations of API usage rules and memory properties within VFS, and equally at assessing the feasibility of our SOCA technique to reliably analysing intricate operating system components such as the Linux VFS implementation. We are particularly interested in finding out to what degree the *automatic* verification of complex properties involving pointer safety and the correct usage of locking APIs within VFS is possible.[1]

Since the Linux VFS implementation consists of more than 65k lines of complex C code including inlined assembly and linked dynamic data structures, its verification is not supported by current software model checkers such as BLAST [Henzinger

[1]Doing so is in the remit of Joshi and Holzmann's mini challenge: "researchers could choose any of several existing open-source filesystems and attempt to verify them" [Joshi and Holzmann, 2007].

et al., 2002a] and CBMC [Clarke et al., 2004]. Thus, previous work by us focused on the question whether and how an appropriate model of the VFS can be reverse engineered from its implementation, and whether meaningful verification results can be obtained using model checking on the extracted model [Galloway et al., 2009]. This proved to be a challenging task since automated techniques for extracting models from C source code do not deal with important aspects of operating system code, including macros, dynamic memory allocation, function pointers, architecture-specific and compiler-specific code and inlined assembly. Much time was spent in [Galloway et al., 2009] on extracting a model by hand and validating this model via reviews and simulation runs, before it could be proved to respect data-integrity properties and to be deadlock-free using the SMART model checker [Ciardo et al., 2006]. Our SOCA technique addresses these shortcomings, providing automated verification support that does away with manual modelling and ad-hoc pointer analysis.

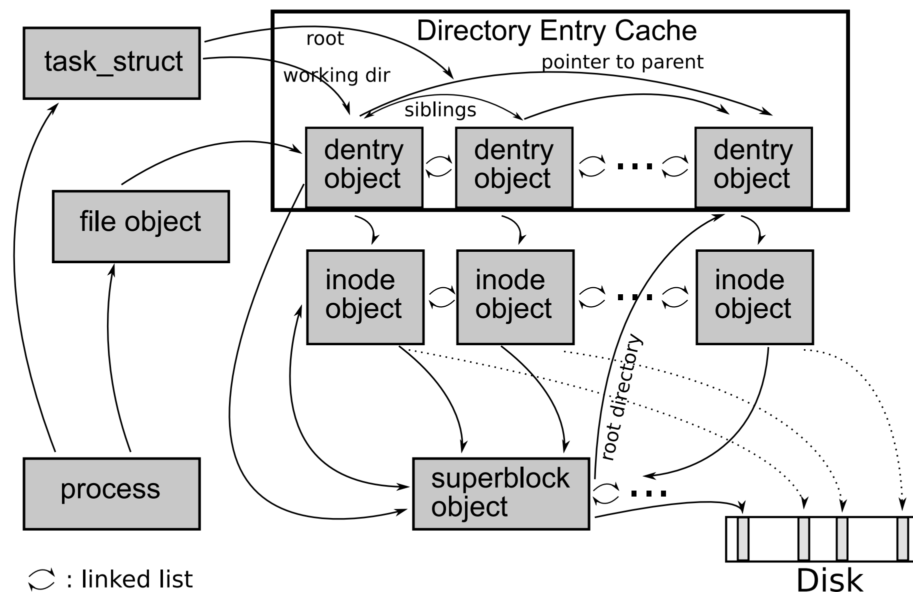

Figure 5.1: VFS environment and data structures, where arcs denote pointers.

5.1 The Linux Virtual File System

This section introduces the Linux FS architecture and, in particular, the *Virtual File System* layer; the reader is referred to [Bovet and Cesati, 2005] for a more detailed description. An overview of the VFS internals and data structures is presented in Fig. 5.1.

The Linux FS architecture consists of multiple layers. The most abstract is the *application* layer which refers to the user programs; this is shown as "process" in Fig.

5.1. Its functionality is constructed on top of the file access mechanisms offered by the *C POSIX library*, which provides functions facilitating file access as defined by the POSIX Standard, e.g., open file `open()`, delete file `remove()`, make directory `mkdir()` and remove directory `rmdir()`. The next lower layer is the *system call interface* which propagates requests for system resources from applications in user space to the kernel, e.g., to the VFS.

The *Virtual File System* layer is an indirection layer, providing the data structures and interfaces needed for system calls related to a standard Unix FS. It defines a common interface that allows many kinds of specific FSs to coexist, and enables the default processing needed to maintain the internal representation of a FS. The VFS runs in a highly concurrent environment as its interface functions may be invoked by multiple, concurrently executing application programs. Therefore, mechanisms implementing mutual exclusion are widely used to prevent inconsistencies in VFS data structures, such as atomic values, mutexes, reader-writer semaphores and spinlocks. In addition, several global locks are employed to protect the global lists of data structures while entries are appended or removed. To serve a single system call, typically multiple locks have to be obtained and released in the right order. Failing to do so could drive the VFS into a deadlock or an undefined state, effectively crashing the operating system.

Each *specific file system*, such as EXT3 and ReiserFS, then implements the processing supporting the FS and operates on the data structures of the VFS layer. Its purpose is to provide an interface between the internal view of the FS and physical media, by translating between the VFS data structures and their on-disk representations. Finally, the lowest layer contains *device drivers* which implement access control for physical media.

The most relevant data structures in the VFS are *superblocks*, *dentries* and *inodes*. As shown in Fig. 5.1, all of them are linked by various pointers inside the structures. In addition, the data structures consist of sets of function pointers that are used to transparently access functionality provided by the underlying FS implementation. The most frequently used data objects in the VFS are dentries. The *dentry* data structures collectively describe the structure of all currently mounted FSs. Each dentry contains a file's name, a link to the dentry's parent, the list of subdirectories and siblings, hard link information, mount information, a link to the relevant super block and locking structures. It also carries a reference to its corresponding inode and a reference count that reflects the number of processes currently using the dentry. Dentries are hashed to speed up access; the hashed dentries are referred to as the *Directory Entry Cache*, or *dcache*, which is frequently consulted when resolving path names.

In our initial verification attempt to the VFS [Galloway et al., 2009], our work was focused on manually abstracting these data structures and their associated control flow, so as to obtain a sufficiently small model for automated verification via model checking. Hence, much effort was put into discovering relations between the different data structures employed by the VFS [Galloway et al., 2009]. The focus of this chapter differs in the sense that *no* models of data structures, memory layout or control flow are derived from the implementation. Instead, each path of the compiled program is translated automatically into a corresponding constraint system which is then analysed by an SMT solver, thus fully automating the verification process.

5.2 VFS Execution Environment and Properties

This section discusses our model of the VFS execution environment and also presents the pointer safety properties and locking API usage rules relevant for the Linux VFS implementation.

Modelling the environment. One problem for program verification arises when program functions make use of an external data environment, i.e., de-reference pointers to data structures that are not created by the function under analysis. This is particularly common in case of the VFS as the majority of the VFS code operates on dentries that are assigned either when an FS is mounted or during previous path-lookup operations. The problem becomes particularly awkward since all these data structures are organised as linked lists which contain function pointers for accessing the specific file system underlying the VFS layer. This is because symbolic execution can easily cope with symbolic data objects of which only a pointer to the beginning of the structure is defined, while the remainder of the structure is left unspecified. However, in the case of linked data structures, some unspecified component of a given data object may be used as a pointer to another object. Treating the pointer symbolically will not only result in many false warnings since the pointer may literally point to any memory location, but may also dramatically increase the search space.

In our case study we “close” the VFS system to be analysed by defining a small number of dentries and associated data structures as static components of the kernel binary. As far as necessary, these data structures are directly defined in the VFS C source code by assigning a static `task_struct` (cf. `include/linux/sched.h` in the Linux source hierarchy) defining the logical context, including the working directory and a list of 15 dentries describing the FS’s mount point and a simple directory

hierarchy. The data objects are partially initialised by a handcrafted function that is used as a preamble in our analysis process. Note that the actual parameters to the VFS interface functions and the majority of data fields in the predefined data objects are still treated as symbolic values. Our modelling of the external environment is conducted by successively adding details to the initial memory state while carefully avoiding being over-restrictive. We only intend to reduce the number of false warnings by eliminating impossible initial memory states to be considered in our analysis.

Pointer safety properties. We check three basic safety properties for every pointer that is de-referenced along an execution path:

1. The pointer does not hold value `NULL`.
2. The pointer only points to allocated data objects.
3. If the pointer is used as a jump target (call, return or computed jump), it may only point inside the `.text` section of the kernel binary, which holds the actual program code. Obviously, the program binary also has other sections such as the symbol table or static data which are, however, invalid as jump targets.

A check of the above properties on the IR is performed by computing an over-approximation of the address range the pointer may point to. That is, we assume that the pointer may address any memory cell between the maximal and minimal satisfying model determined by the constraint system for that pointer. For programs involving only statically assigned data we can directly evaluate the above properties by checking *(a)* whether the address range is assigned in the program binary and *(b)* whether it belongs to appropriate program sections for the respective use of the pointer. If dynamic memory allocation is involved, we keep track of objects and their respective locations currently allocated within the program's constraint representation. Checking the above properties is then performed as an assertion check within Yices.

Locking API usage rules. Being designed for a range of multiprocessor platforms, the Linux kernel is inherently concurrent. Hence, it employs various mechanisms implementing mutual exclusion, and primarily locking, to protect concurrently running kernel threads. The locking APIs used within the VFS are mainly spinlocks and semaphores, and each of the VFS structures contains pointers to at least one lock. In addition to these per-object locks, there exist global locks to protect access to lists of objects.

At a high level of abstraction, all locking APIs work in a similar fashion. If a kernel thread attempts to acquire a particular lock, it waits for this lock to become available, acquires it and performs its critical actions, and then releases the lock. As a result of this, a thread will wait forever if it attempts to acquire the same lock twice without releasing it in-between. Checking for the absence of this problem in single- and multi-threaded programs has recently attracted a lot of attention in the automated verification community [Ball and Rajamani, 2001; Henzinger et al., 2002a; Witkowski et al., 2007; Xie and Aiken, 2007]. For software systems like the Linux kernel with its fine grained locking approach, conducting these checks is non-trivial since locks are passed by reference and due to the vast number of locks employed. A precise analysis of pointer aliasing relationships would be required to prove programs to be free of this sort of errors, which is known to be an undecidable problem in general.

In our approach, locking properties are checked by instrumenting locking related functions in their IR in such a way that a guarded jump is added to the control flow of the program, passing control to a designated "error location" whenever acquiring an already locked lock structure is attempted or an unlocked lock is released. Our symbolic analysis is then used to evaluate whether the guard may possibly be true or not, and an error message for the path is raised if the error location is reachable.

5.3 Applying the SOCA Verifier to the VFS

For applying the SOCA Verifier to the VFS, we used the VFS implementation of version 2.6.18.8 of the Linux kernel, compiled with gcc 4.3.3 for the Intel Pentium-Pro architecture. All configuration options of the kernel were left as defaults. Our experiments were then carried out on an Intel Core 2 Quad machine with 2.83 GHz and 4 GBytes of RAM, typically analysing three VFS functions in parallel.

The bounds for the SOCA Verifier were set to a maximum of 1000 paths to be analysed, where a single program location may appear at most 1000 times per path, thereby effectively bounding the number of loop iterations or recursions to that depth. The Yices SMT solver was set to a timeout of 60 seconds per invocation, which was never reached in our experiments. All these bounds were chosen so that code coverage is maximised, while execution time is kept reasonably small.

Statistics and performance. Our experimental results are summarised in three tables. Table 5.1 provides a statistical overview of the VFS code. We report the total *number of machine instructions* that have been translated into IR by follow-

Table 5.1: Experimental Results I: Code statistics by VFS function analysed

	`creat`	`unlink`	`mkdir`	`rmdir`	`rename`	totals
no. of instructions	3602	3143	3907	3419	4929	19000
lines in source code	1.4k	1.2k	1.6k	1.4k	2k	7.6k
no. of paths	279	149	212	318	431	1389
min. path length	91	41	87	72	72	41
max. path length	4138	3218	5319	3017	5910	5910
pointer operations	2537	2190	2671	2466	4387	14251
concrete	2356	2134	2458	2368	3989	13305
symbolic	181	56	213	98	398	946
locking operations	287	231	391	319	451	1679

Table 5.2: Experimental Results II: SOCA Verifier statistics

	`creat`	`unlink`	`mkdir`	`rmdir`	`rename`	totals
total time	2h27m	1h22m	2h42m	1h34m	3h45m	11h50m
max. memory (SOCA)	1.03G	752M	1.15G	743M	1.41G	1.41G
max. mem. (SOCA+Yices)	1.79G	800M	1.92G	791M	2.18G	2.18G
exec. bound exhausted	yes	yes	yes	yes	yes	yes
path bound exhausted	no	no	no	no	no	no
paths reaching end	154	112	165	215	182	828
assertions checked	13.4k	12.4k	15.8k	11.8k	21.9k	75.3k
ratio of failed checks	0.043	0.012	0.041	0.019	0.049	0.033

ing each function's control flow. The *lines in source code* give an estimate of the checked implementation's size as the size of the C functions involved (excluding type definitions and header files, macro definitions, etc.). The next values in the table present the numbers of paths and, respectively, the lengths of the shortest and longest paths, in instructions explored by our verifier with respect to the calling context of the analysed function. The *pointer* and *locking operations* resemble the numbers of pointer de-references and lock/unlock operations encountered along the analysed paths, respectively.

Table 5.3: Experimental Results III: Yices statistics

	`creat`	`unlink`	`mkdir`	`rmdir`	`rename`	totals
total Yices calls	27533	21067	31057	20988	44439	145k
total time spent in Yices	2h22m	1h11m	2h22m	1h24m	3h8m	10h28m
average time	311ms	192ms	271ms	198ms	376ms	248ms
standard deviation	3.7s	0.9s	5.2s	1.4s	5.9s	4.8s
max CS size in vars	450k	97k	450k	95k	450k	450k
average CS size in vars	2844	2871	2871	2862	2939	2877
standard deviation	14619	8948	14618	8898	16052	13521
max. memory consumption	766M	48M	766M	48M	766M	766M

In Table 5.2 we report the performance of the SOCA Verifier, showing the total time needed for analysing the kernel functions and our tool's maximum *memory consumption*. The maximum memory consumption of our tool together with the Yices solver engine is an estimate generated by summing up our tool's and Yices' maximum memory usage as given in Table 5.3; however, these may not necessarily hit their peak memory at the same time. The next two rows denote whether the analysis bounds were reached. We also report the number of paths reaching the end of the function analysed, the total number of assertions checked and the percentage of failed checks. Paths not reaching a return statement in the target function are terminated either due to bound exhaustion, or due to a property being violated that does not permit continuation of that path.

Finally, we outline in Table 5.3 the usage and behaviour of the SMT solver Yices, by reporting the number of times Yices was called when checking a particular VFS function and the total and average time spent for SMT solving. We also give the size of the checked constraint systems (CS) in boolean variables, as output by Yices and show the maximum amount of memory used by Yices.

Our analyses usually achieve a statement and condition coverage of 60% to 80% in this case study.[2] The main reason for this, at-first-sight low percentage, is that VFS functions often implement multiple different behaviours of which only a few are reachable for the given execution environment. For example, the implementation of the `creat()` system call resides mainly in the `open_namei()` function alongside different behaviours implementing the `open()` system call. Taking this into account, the coverage achieved by the SOCA Verifier is remarkably high when compared to testing-based approaches.

It should be noted that the above tables can only give a glimpse of the total scale of experiments that we have conducted for this case study.[2] Depending on how

[2]A complete account of the experiments will be made available on the SOCA website located at `http://swt-bamberg.de/soca/`.

detailed or coarse the execution environment is specified, we experienced run times reaching from a few minutes up to several days, achieving different levels of statement and condition coverage (ranging from 20% to 80%) and different error ratios (ranging from 0 to 0.5). The discriminating value in all these experiments is the total number of "symbolic" pointers; a symbolic pointer is a pointer where the exact value cannot be determined at the point at which it is de-referenced. This usually happens when the entire pointer or some component of it (e.g., its base or offset) is retrieved from an incompletely specified component of the execution environment or directly from the input to the analysed function. While these symbolic values are generally bad for the performance of the SOCA technique since slicing is rendered inefficient and search spaces are increased, they are important for driving the analysis into paths that may be hard to reach in testing-based approaches to system validation.

Errors and false positives. As our verification technique does not include infeasible paths, all errors detected by the SOCA Verifier can actually be reproduced in the code, provided that other kernel components match the behaviour of our employed execution environment.

In advance of the experiments reported in this chapter, we had tested our implementation of the SOCA technique on a variety of hand-crafted examples and also on the *Verisec* suite [Ku et al., 2007] which provides 280 examples of buffer overflow vulnerabilities taken from application programs. In all these cases we experienced low false-positive rates of less than 20%. However, as these examples represent closed systems not using external data objects, they are handled more efficiently by the SOCA Verifier than the VFS which makes heavy use of external data objects.

Our above result tables show that our analysis approach detects a number of errors of about 3% of the total number of checked assertions in each VFS function analysed. We have inspected each reported error in detail and discovered that all of them are due to an imprecisely specified execution environment. As explained in the previous section, specifying a valid but non-restrictive environment is particularly hard as all VFS functions operate on data structures that are allocated and assigned by other kernel sub-systems before the VFS functions are executed. As most of these structures form multiple lists, modelling them manually is tedious and error-prone. Therefore, our strategy was to leave many fields of those structures initially unspecified and successively add as much detail as necessary to eliminate false positives. This proved to be a good way to specify valid and at the same time non-restrictive execution environments.

Not having discovered any real errors in the analysed VFS code contributes to our high confidence in the Linux kernel and is to be expected; the VFS consists of

a well established and extensively used and tested code base, which is under active development for many years. Indeed, our primary goal when setting up this case study was not to find errors in the VFS code but to use the VFS as a complex, real-world verification project for stress-testing our SOCA Verifier. With respect to this task, our results demonstrate that the SOCA Verifier is capable of reliably and efficiently analysing the complex Linux VFS implementation on off-the-shelf hardware.

5.4 Evaluating the Effectiveness of SOCA

With the goal of further evaluating the effectiveness of SOCA as a bug-finding tool, we conduct a second case study which applies our SOCA Verifier to consecutive releases of the Linux kernel's VFS implementation. With its publicly available source code, well documented bug reports and patches, and a release history reaching back for almost 20 years, the Linux kernel is an ideal candidate for the sort of "archaeological" study presented here. The question which we pursue is: If the Linux developers would have had the SOCA Verifier available, what ratio of newly introduced bugs could have been detected automatically, and hence, could have been fixed immediately?

5.4.1 Choice of VFS versions

For this case study we chose to analyse 23 patches committed to the current "stable" 2.6 development branch of the Linux kernel. The source repository[3] contains all contributions committed to Linux 2.6 between April 2005 (Linux 2.6.12-rc2) and February 2010 (Linux 2.6.33-rc7). Our selection is made by choosing all commits affecting the VFS and in which null-pointer issues are addressed, according to the documentation of the patch. Due to the previously explained high complexity of the VFS, involving linked data structures and computed jumps, restricting this case study to null-pointers does not render the study trivial. The subject matter of the 23 patches considered here varies from actual bug fixes, to performance enhancements, to the implementation of new features. Hence, the patches differ substantially in size, ranging from a few lines of code modifications in one file, up to 300 lines of code modifications that are distributed over several files and changing data structures and function interfaces. An overview of our sample is given in Table 5.4. The *commit keys* given in the table are references to Linux's source code repository.

[3]The source repository of Linux 2.6 is available at `http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git`.

5.4.2 Case study setup

To conduct our case study we compile two versions of the Linux kernel for each of the 23 patches. More precisely, we compile one kernel binary using the source code directly before a patch was committed, and a second binary from the sources that include the patch. In all cases is the kernel source configured for the IA32 architecture using the default configuration shipped with the kernel sources. The SOCA Verifier is then applied to the functions affected by a particular patch in each of the two kernel binaries compiled with respect to that patch. In difference to the first case study presented in Sec. 5.3, SOCA is applied here without modelling an execution environment for the functions checked. The modelling step has been omitted due to the large number of functions and kernels to be analysed, and especially due to the changes in function interfaces and the kernel's data structures between these releases. The bounds for the SOCA Verifier are set to the values used in the previous case study. The error traces reported by our tool are manually checked for validity, i.e., whether the expected error has been found or whether traces not related to the subject matter of the patch or false-positive traces are reported.

A detailed account of the results of the case study is given in Table 5.4. When analysing a patch that is supposed to *fix a bug*, we expect the SOCA Verifier to report an error trace for that bug in the kernel binary compiled from the pre-patched source, and also to report the patched version of the kernel to be free of that bug. We denote this success case with a + in the *Results* column of Table 5.4. With ${}^{+}/_{+}$ and ${}^{-}/_{+}$ we denote that the error was detected in the pre-patched kernel but also in the patched kernel, or that the error was only reported for the patched kernel, respectively. We write ${}^{-}/_{-}$ if no error was detected at all. For patches introducing new features or implementing performance improvements we expect the pre-patched kernel and the patched kernel to be free of errors and denote that with 0 in Table 5.4. If SOCA issues false-positive errors for these cases, we write ${}^{pre}/_{post}$, where *pre* and *post* denote the numbers of false-positive errors raised for the pre-patched kernel and the patched kernel, respectively. If error traces that are not related to the patch under consideration, are produced by the SOCA Verifier, we give the number of those reported error traces in column *Unrelated Traces*. Table 5.4 contains seven cases where the kernel source failed to compile for the pre-patched kernel, the patched kernel, or both. Obviously, SOCA could not be applied to these kernels.

Table 5.4: Experimental Results IV: Evaluating the effectiveness of SOCA

#	Commit Key	Type	No. of Functions	Results	Unrelated Traces
1	`08ce5f16ee466ffc5bf243800deeecd77d9eaf50`	F	2	0	2
2	`214fda1f6e1b8ef2a5292b0372744037fc80d318`	P	2	does not compile	
3	`22d2b35b200f76085c16a2e14ca30b58510fcbe7`	F	1	does not compile	
4	`2a737871108de9ba8930f7650d549f1383767f8b`	BF	5	+, 0	1
5	`2f38d70fb4e97e7d00e12eaac45790cf6ebd7b22`	P	4	0	0
6	`322ee5b36eac42e762526b0df7fa432beba6e7a0`	B	1	+	2
7	`4a19542e5f694cd408a32c3d9dc593ba9366e2d7`	F	1	does not compile	
8	`4ea3ada2955e4519befa98ff55dd62d6dfbd1705`	BP	3	$^{+}/_{+}$, $^{2}/_{1}$	0
9	`520c85346666d4d9a6fcaaa8450542302dc28b91`	BP	2	+, $^{0}/_{2}$	3
10	`6c5daf012c9155aafd2c7973e4278766c30dfad0`	BP	2	+, $^{0}/_{2}$	1
11	`6ea36ddbd1abfe867f1e874a8312bfd811e5fd2c`	P	1	does not compile	
12	`73241ccca0f7786933f1d31b3d86f2456549953a`	FP	5	does not compile	
13	`745ca2475a6ac596e3d8d37c2759c0fbe2586227`	B	1	0	1
14	`7ed7fe5e82c9fc8473974fbd7389d169b8f17c77`	BP	1	+, 0	2
15	`acb0c854fa9483fa85e377b9f342352ea814a580`	P	3		
16	`acd0c935178649f72c44ec49ca83bee35ce1f79e`	B	2	+	1
17	`acfa4380efe77e290d3a96b11cd4c9f24f4fbb18`	P	4	$^{0}/_{3}$	0
18	`ad775f5a8faa5845377f093ca11caf577404add9`	B	2	$^{-}/_{-}$	3
19	`cb59861f03a626196a23fdef5e20ddbb8cca6466`	P	1	$^{0}/_{1}$	0
20	`cdb70f3f74b31576cc4d707a3d3b00d159cab8bb`	P	1	$^{0}/_{1}$	2
21	`d0185c0882d76b8126d4a099c7ac82b3b216d103`	B	2	+	1
22	`e0e817392b9acf2c98d3be80c233dddb1b52003d`	B	1	+	2
23	`e6c6e640b8b258dc7f60533e81f050d15fc0a9af`	P	1	does not compile	

Commit Key: A patch referenced by `<commitkey>` can be viewed at
`http://git.kernel.org/?p=linux/kernel/git/torvalds/`
`linux-2.6.git;a=commit;h=<commitkey>`.
Type: B – bug fixed; F – feature introduced; P – performance improved

5.4.3 Results

The most important result of this case study relates to the patches labelled as bug fixes. The SOCA Verifier reliably reports 8 out of 10 of the pre-patched kernels as buggy and the corresponding 8 patched kernels as safe. This means that 80% of the total number of null-pointer bugs fixed in these kernel releases were successfully detected by the SOCA Verifier.

In a similar way, SOCA reports 5 of the 11 performance improvements and feature introductions as safe (45.5%), which is to be expected when considering the extensive amount of peer-review done for each patch submission by the kernel's developers. Especially for the patches implementing performance improvements, the SOCA Verifier reports a number of false-positive error traces on the patched kernels. This result can be explained when looking at the code modifications introduced by these patches: most of them *remove* "superfluous" pointer checks from the code.

This means, the kernel's developers consider these checks as redundant because the patched functions will never be called with certain types of invalid parameters. However, since we check these functions in isolation, i.e., without considering all possible calling contexts, we cannot verify that an invalid pointer is never used as an argument to the function, and hence report a potentially error trace.

The experiments conducted here consumed a total of 41.71 hours of CPU time on an exclusively used Intel Xeon 8-core PC with 2.6 GHz per core and 12 GBytes of memory. Of this time 9.67 hours were used for sequentially compiling 32 Linux kernel binaries. The remaining 32 CPU hours were consumed by the SOCA Verifier for checking a total of 68 functions, which was done by invoking seven SOCA processes in parallel. The memory consumption of the SOCA processes always stayed below 2 GBytes per process. Hence, the SOCA Verifier can be used on a modern off-the-shelf PC without limitations. By exploiting the parallel machine architecture we have available, the actual verification was conducted within less than 6 hours. These figures show that the SOCA technique can be very well applied as a unit-level bug-finding tool during software development in large projects. The effective time needed for verifying the small set of components usually modified within a single commit is typically shorter than the time the developer has to wait for compiling the project.

5.5 Related Work on File System Verification

The verification of file system implementations is studied in [Butterfield and Catháin, 2009; Damchoom and Butler, 2009; Ferreira and Oliveira, 2009; Galloway et al., 2009; Kim and Kim, 2009; Yang et al., 2006, 2004]. In [Yang et al., 2004], model checking is used within the systematic testing of EXT3, JFS and ReiserFS. The employed verification system consists of an explicit-state model checker running the Linux kernel, a file system test driver, a permutation checker that verifies that a file system can always recover, and a recovery checker using the *fsck* recovery tool. The verification system starts with an empty file system and recursively generates successive states by executing system calls affecting the file system under analysis. After each step, the verification system is interrupted, and *fsck* is used to check whether the file system can recover to a valid state. In contrast to this, our work focuses on checking a different class of properties, namely pointer safety and locking properties. Thanks to our SOCA technique we can analyse these properties precisely and feed back detailed error traces together with specific initial heap state information leading to the error.

In [Kim and Kim, 2009] an empirical study applying *concolic testing* [Sen et al.,

2005] to the multi-sector read operation of a flash memory implementation is presented. Concolic testing relies on performing concrete execution on random inputs while collecting path constraints along executed paths. The constraints are then used to compute new inputs driving the program along alternative paths. In difference to this approach, SOCA uses symbolic execution to explore all paths and concretises only in order to resolve computed jumps. Concrete execution in SOCA may also be employed to set up the environment for symbolic execution. [Kim and Kim, 2009] discusses the advantages and weaknesses of concolic testing on the domain of low-level file system verification as compared to model checking. The authors conclude that their approach achieved several experimental goals, namely automated test case generation, high code coverage and the detection of bugs, but suffers from limitations including the low speed of the analysis and the lack of support for array index variables in their tool chain.

A model in the process algebra CSP that covers the concurrent aspects of flash memory is described in [Butterfield and Catháin, 2009]. The authors focused on developing a low-level model covering the internal behaviour of *Open NAND* flash devices. They Apply the FDR model checker to prove the consistency of this model with a specification of the external interface of the device. While the authors detected several deadlocks and sources of missinterpretation in the models, the analysis could only be partially completed as the specifications proved to be too complex for being analysed with FDR in full.

In [Damchoom and Butler, 2009] a model of a flash-based file store developed in Event-B is given. In this paper, the authors centre on discussing their use of refinement in feature augmentation and as structural refinement. The goal of their work is to simplify the process of model construction and to relate an abstract file system model with the flash specification. The paper explains further, how machine decomposition can be applied to separate parts of the file system layer from the interface layer in a complex file system model.

Finally, the application of theorem proving techniques to build a formal methods tool chain and apply it to an abstract file system model is presented in [Ferreira and Oliveira, 2009]. The paper shows how different formal methods and tools, including Alloy, VDM++ and HOL may be glued together by relation modelling. It also advocates transparent integration and automation of formal methods in software development processes.

Chapter 6

Summary and Conclusions

This thesis focusses on identifying pointer safety related errors in computer programs. We make five contributions in this area:

BLASTing Linux Code. In Chapter 3, we present a case study on the software model checker BLAST. We exposed BLAST to analysing 16 different operating system code examples of programming errors related to memory safety and locking behaviour. In our experience, BLAST is rather difficult to apply by a practitioner during operating system software development. This is because of (i) its limitations with respect to reasoning about pointers, (ii) several issues regarding usability, including bugs in within the program itself, and (iii) a lack of consistent documentation. Especially in the case of memory safety properties, massive changes to the source code were necessary which essentially requires one to know about a bug beforehand. However, it must be mentioned that BLAST was not designed as a memory debugger. Indeed, BLAST performed considerably better during our tests with locking properties; however, modifications on the source code were still necessary in most cases.

Symbolic Object Code Analysis. Our second contribution, given in Chapter 4, is in introducing Symbolic Object Code Analysis, a technique for verifying pointer safety properties by bounded symbolic execution of compiled programs. More precisely, the SOCA technique (i) systematically traverses the object code in a depth-first fashion up to a certain depth and width, (ii) calculates at each assembly instruction a slice required for checking the relevant pointer-safety properties at this instruction, (iii) translates the slice and properties into a bit-vector constraint problem, and (iv) executes the checks by invoking the Yices SMT solver.

Evaluation of SOCA. Our third contribution, also in Chapter 4, is in introducing the SOCA Verifier as a prototypical implementation of the SOCA technique. By

means of extensive experimental results of the SOCA Verifier, using the Verisec suite and almost 10,000 Linux device driver functions as benchmarks, we show not only that SOCA performs competitively to current source-code model checkers but that it also scales well when applied to real operating systems code and pointer safety issues. SOCA effectively explores semantic niches of software that current software verifiers do not reach.

Beyond Memory Safety: VFS. Our fourth contribution is given in an further case study applying the SOCA technique to the Linux Virtual File System (VFS) in Chapter 5. We demonstrate how complex verification properties including information on heap-allocated data structures as well as pre- and post conditions of functions, can be expressed for symbolic object-code analysis, for which two different approaches are employed. Firstly, properties may be presented to the SMT solver as assertions on the program's register contents at each execution point. Alternatively, the program may be instrumented during its symbolic execution, by adding test and branch instructions to its control flow graph. Verifying a particular property then involves checking for the reachability of a specific code section. While the first approach allows us to express safety properties on pointers, we use the latter technique for checking preconditions of kernel API functions reflecting particular API usage rules.

Effectiveness of SOCA. Our fifth contribution is in providing evidence for the effectiveness and reliability of the SOCA technique by conducting an "archaeological" case study on the Linux VFS in Chapter 5. We apply the SOCA Verifier to VFS functions obtained from 32 releases of the Linux kernel, showing that up to 80% of null-pointer related bugs fixed between these releases can be detected automatically. We demonstrate further that the SOCA Verifier can be applied as an efficient, unit-level bug-finding tool since the effective time needed for verifying the set of software components modified between two releases is typically shorter than the time needed for compiling the project. Therefore, adding automated software verification to the tool set of kernel software developers promises to significantly improve the quality assurance process for operating system kernels.

Verification of the VFS. Our last, but not least, contribution is the formal verification of a group of commonly used VFS functions, namely those for creating and removing files and directories. By applying symbolic execution and leaving the parameters of these functions as unspecified as possible, our analysis covers low-probability scenarios. In particular, we look for program points where pointers

holding invalid values may be de-referenced or where the violation of API usage rules may cause the VFS to deadlock. The experimental results show that the SOCA technique works well on the Linux VFS and that it produces a relatively low number of false-positive counterexamples while achieving high code coverage. Therefore, the absence of any flagged errors contributes to raising confidence in the correctness of the Linux VFS implementation.

6.1 Conclusions

The initial motivation for our SOCA technique to automated program verification was to explore the feasibility of using symbolic execution for analysing compiled programs with respect to pointer safety properties. Indeed, object-code analysis is the method of choice for dealing with programs written in a combination of programming languages such as C and inlined assembly. This is particularly true for operating system code which is often highly platform specific and makes extensive use of programming constructs such as function pointers. As we show in this chapter, these constructs can be dealt with efficiently in path-wise symbolic object-code analysis, while they are usually ignored by static techniques or by source-code-based approaches.

While the ideas behind the SOCA technique, namely symbolic execution, path-sensitive slicing and SMT solving, are well-known, the way in which these are integrated into the SOCA Verifier is novel. Much engineering effort went also into our SOCA implementation so that it scales to complex real-world operating system code such as the Linux device drivers analysed in this paper.

The main reasons for this scalability are in the structure of programs in the application domain of Linux device drivers as well as in the proceeding followed by the SOCA technique. Firstly, device drivers are relatively small programs consisting of generally short functions with small data spaces, rendering a search-based analysis possible. We expect our technique to be applicable for large-scale application software. However, this may require major adaption, probably including the use of program abstraction. Hence, doing so may result in having to deal with different classes of false-positive results. Currently a valid counterexample-trace can be produced for each violation of a safety property. That means, our technique issues false-positive error reports only due to imprecisely defined initial memory states or function parameters that are not to be expected in real program execution.

Secondly, our choice of path-wise analysing compiled code contributes a great deal to the results presented above. That is because having a program representation with explicit memory access operations and exploring each path separately

in a symbolic execution setting turned out to be sufficient for efficiently handling a majority of pointer aliasing problems and computed jumps, other approaches are not able to cope well with.

6.2 Open Issues and Future Work

There are several open issues to be addressed in future work. The most pressing problem is to gain the ability of automatically dealing with device drivers with large data spaces and drivers that make use of complex, pointered data structures as their input. In Chapter 5 we show that in general, constraints on the driver's input can be easily prepended to the constraint systems and may even be considered by the slicer. However, this has to be done manually. Extracting the required information from the binary or from the public C header files describing the interface implemented by the driver, remains an open problem. Knowing whether the input is supposed to be a cyclic list or a tree and at which offsets pointers are supposed to be would reduce the number of false-positive errors found by our approach substantially. We think that current work on shape analysis [Calcagno et al., 2009], [Yang et al., 2007] may provide results that can be integrated into our tool.

Another important and probably quickly achievable goal is to provide debugger integration for our tool in such a way that analysis results can be presented as an error trace in a program debugger, together with an program input that would lead to a segmentation fault or similar when the program is executed.

We are also aiming too parallelising our analysis approach in order to benefit from currently available multi-CPU and multi-core PCs. This should be relatively easy to achieve as constraint generation and constraint solving are already performed in separated processes, and the constraint generation is much faster than the solving. Hence, multiple paths could be explored by one constraint generating process while several instances of the SMT solver are employed to boost analysis performance.

The probably biggest challenge is in regard of handling concurrency in the driver to be analysed. Device drivers run in a highly concurrent environment in which their interface functions may be invoked from multiple concurrently executing application programs. Hence, computer architectures supporting symmetric multi-processing, as well as normal process preemption caused by scheduling on single processor machines, gives rise to the indeterminate sequencing of the respective threads. Therefore, mechanisms implementing mutual exclusion are widely used in order to prevent inconsistencies arising in this context. Our approach currently ignores all memory safety issues arising from interleaved writing to the heap and much more research is required in this area. We believe that techniques such as partial order reduction

[Godefroid, 1994], [Flanagan and Godefroid, 2005] may constitute a way to deal with the potentially infinitely large number of possible interleavings that have to be considered in a "concurrent symbolic execution" setting.

Last but not least, the work presented within this thesis has not revealed a previously unknown error in an operating system component. Although we know from our experiments with the Verisec suite that our technique produces relatively few false positives, we have no indication regarding the false-positive rate in the device driver benchmark presented in Section 4.5.3. We have detected 887 program locations at which potential null-pointers may be dereferenced. While we have not checked whether those are real errors or whether they are the result of a too loosely specified execution environment, our second case study on the Linux VFS in Sec. 5.4 gives an indication on SOCA's effectiveness as a bug-finding tool. To substantiate this, future research should also aim at extending the SOCA Verifier to support, i.e., properties related to real-time components of operating system kernels kernels, for which additional case studies would be required. A particularly worthwhile project would be another "archaeological" study in the spirit of the one conducted in Sec. 5.4 on projects like FreeRTOS [Barry, 2010] or implementations of flash file systems [Hynix Semiconductor et al., 2008]: tracing that code's development line over several releases and checking whether timing-related and scheduling-related errors that were introduced or removed in subsequent releases can be found by SOCA, could give a clear indication on the applicability of the SOCA technique in the area of embedded and real-time systems.

Bibliography

Balakrishnan, G. and Reps, T. (2004). Analyzing memory accesses in x86 executables. In *CC '04*, volume 2985 of *LNCS*, pages 5–23, Heidelberg. Springer.

Balakrishnan, G. and Reps, T. (2005). Recovery of variables and heap structure in x86 executables. Technical report, University of Wisconsin, Madison.

Balakrishnan, G. and Reps, T. (2006). Recency-abstraction for heap-allocated storage. In *SAS '06*, volume 4134 of *LNCS*, pages 221–239, Heidelberg. Springer.

Balakrishnan, G., Reps, T., Melski, D., and Teitelbaum, T. (2008). WYSINWYX: What You See Is Not What You eXecute. In *VSTTE '08*, volume 4171 of *LNCS*, pages 202–213, Heidelberg. Springer.

Ball, T. (2005). The verified software challenge: A call for a holistic approach to reliability. In *VSTTE '08*, volume 4171 of *LNCS*, pages 42–48, Heidelberg. Springer.

Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., Rajamani, S. K., and Ustuner, A. (2006). Thorough static analysis of device drivers. *SIGOPS Oper. Syst. Rev.*, 40(4):73–85.

Ball, T., Majumdar, R., Todd Millstein, T., and Rajamani, S. K. (2001). Automatic predicate abstraction of C programs. In *PLDI '01*, pages 203–213, New York. ACM.

Ball, T. and Rajamani, S. K. (2001). Automatically validating temporal safety properties of interfaces. In *SPIN '01*, volume 2057 of *LNCS*, pages 102–122, Heidelberg. Springer.

Barry, R. (2010). FreeRTOS: a portable, open source, mini real time kernel. `http://www.freertos.org/`.

Beyer, D., Chlipala, A. J., Henzinger, T. A., Jhala, R., and Majumdar, R. (2004). Invited talk: The BLAST query language for software verification. In *PEPM '04*, pages 201–202, New York. ACM.

Beyer, D., Henzinger, T. A., Jhala, R., and Majumdar, R. (2005). Checking memory safety with BLAST. In *FASE '05*, volume 3442 of *LNCS*, pages 2–18, Heidelberg. Springer.

Bovet, D. and Cesati, M. (2005). *Understanding the Linux Kernel*. O'Reilly, 3rd edition.

Boyer, R. S. and Yu, Y. (1996). Automated proofs of object code for a widely used microprocessor. *J. ACM*, 43(1):166–192.

Bozga, M., Iosif, R., and Laknech, Y. (2003). Storeless semantics and alias logic. In *PEPM '03*, pages 55–65, New York. ACM.

Breuer, P. T. and Pickin, S. (2006). Abstract interpretation meets model checking near the 10^6 LOC mark. In *AVIS'06*, ENTCS, pages 5–11. Elsevier.

Brummayer, R., Biere, A., and Lonsing, F. (2008). BTOR: Bit-precise modelling of word-level problems for model checking. In *SMT '08/BPR '08*, pages 33–38, New York. ACM.

Burstall, R. M. (1972). Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence*, pages 23 – 50.

Butterfield, A. and Catháin, A. Ó. (2009). Concurrent models of flash memory device behaviour. In *SBMF '09*, volume 5902 of *LNCS*, pages 70–83, Heidelberg. Springer.

Cadar, C., Ganesh, V., Pawlowski, P. M., Dill, D. L., and Engler, D. R. (2006). EXE: Automatically generating inputs of death. In *CCS '06*, pages 322–335, New York. ACM.

Calcagno, C., Distefano, D., O'Hearn, P., and Yang, H. (2009). Compositional shape analysis by means of bi-abduction. *SIGPLAN Not.*, 44(1):289–300.

Chandra, S., Godefroid, P., and Palm, C. (2002). Software model checking in practice: an industrial case study. In *ICSE '02*, pages 431–441, New York. ACM.

Chou, A., Yang, J., Chelf, B., Hallem, S., and Engler, D. R. (2001). An empirical study of operating system errors. In *SOSP '01*, pages 73–88, New York. ACM.

Ciardo, G., Jones, R. L., Miner, A. S., and Siminiceanu, R. I. (2006). Logic and stochastic modeling with SMART. *Perform. Eval.*, 63(6):578–608.

Clarke, E., Kroening, D., and Lerda, F. (2004). A tool for checking ANSI-C programs. In *TACAS '04*, volume 2988 of *LNCS*, pages 168–176, Heidelberg. Springer.

Clarke, E. M., Emerson, E. A., and Sistla, A. P. (1983). Automatic verification of finite state concurrent system using temporal logic specifications: a practical approach. In *POPL '83*, pages 117–126, New York. ACM.

Clarke, E. M., Grumberg, O., and Long, D. E. (1992). Model checking and abstraction. In *POPL '92*, pages 343–354, New York. ACM.

Clarke, E. M., Grumberg, O., and Peled, D. A. (2000). *Model checking.* MIT Press.

Clarke, E. M., Kroening, D., Sharygina, N., and Yorav, K. (2005). SATABS: SAT-based predicate abstraction for ANSI-C. In *TACAS '05*, volume 3440 of *LNCS*, pages 570–574, Heidelberg. Springer.

Corbet, J., Rubini, A., and Kroah-Hartmann, G. (2005). *Linux Device Drivers.* O'Reilly, 3rd edition.

Corbett, J. C., Dwyer, M. B., Hatcliff, J., Laubach, S., Păsăreanu, C. S., Robby, and Zheng, H. (2000). Bandera: extracting finite-state models from Java source code. In *ICSE '00*, pages 439–448, Washington. IEEE.

Cousot, P. (1996). Abstract interpretation. *Symposium on Models of Programming Languages and Computation, ACM Computing Surveys*, 28(2):324 – 328.

Cousot, P. and Cousot, R. (2002). On abstraction in software verification. In *CAV '02*, volume 2404 of *LNCS*, pages 37–56, Heidelberg. Springer.

Coverity, Inc. (2009). Coverity, Inc. `http://www.coverity.com` [24 September 2009].

Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. (1991). Efficiently computing static single assignment form and the control dependence graph. *ACM TOPLAS*, 13(4):451–490.

Damchoom, K. and Butler, M. (2009). Applying event and machine decomposition to a flash-based filestore in Event-B. In *SBMF '09*, volume 5902 of *LNCS*, pages 134–152, Heidelberg. Springer.

Debray, S., Muth, R., and Weippert, M. (1998). Alias analysis of executable code. In *POPL '98*, pages 12–24, New York. ACM.

Deutsch, A. (1992). A storeless model of aliasing and its abstractions using finite representations of right-regular equivalence relations. In *International Conference on Computer Languages '92*, pages 2–13, Washington. IEEE.

Deutsch, A. (1994). Interprocedural may-alias analysis for pointers: beyond k-limiting. In *PLDI '94*, pages 230–241, New York. ACM.

D'Silva, V., Kroening, D., and Weissenbacher, G. (2008). A survey of automated techniques for formal software verification. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 27(7):1165–1178.

Dutertre, B. and de Moura, L. (2006). The Yices SMT solver. Technical Report 01/2006, SRI International. Available at `http://yices.csl.sri.com/tool-paper.pdf` [24 September 2009].

Engler, D. R., Chelf, B., Chou, A., and Hallem, S. (2000). Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI '00*. USENIX Association.

Ferdinand, C., Martin, F., Cullmann, C., Schlickling, M., Stein, I., Thesing, S., and Heckmann, R. (2007). New developments in WCET analysis. In *Program Analysis and Compilation, Theory and Practice*, volume 4444 of *LNCS*, pages 12–52, Heidelberg. Springer.

Ferreira, M. A. and Oliveira, J. N. (2009). An integrated formal methods tool-chain and its application to verifying a file system model. In *SBMF '09*, volume 5902 of *LNCS*, pages 153–169, Heidelberg. Springer.

Flanagan, C. and Godefroid, P. (2005). Dynamic partial-order reduction for model checking software. In *POPL '05*, pages 110–121, New York. ACM.

Gälli, M., Greevy, O., and Nierstrasz, O. (2006). Composing unit tests. In *2nd International Workshop on Software Product Line Testing*, pages 16–22.

Gälli, M., Nierstrasz, O., and Wuyts, R. (2004). Partial ordering unit tests by coverage sets. Technical Report IAM-03-013, Institut für Informatik und Angewandte Mathematik, University of Bern, Switzerland.

Galloway, A., Lüttgen, G., Mühlberg, J. T., and Siminiceanu, R. (2009). Model-checking the Linux Virtual File System. In *VMCAI '09*, volume 5403 of *LNCS*, pages 74–88, Heidelberg. Springer.

Galloway, A., Mühlberg, J. T., Siminiceanu, R., and Lüttgen, G. (2007). Model-checking part of a Linux file system. Technical Report YCS-2007-423, Department of Computer Science, University of York, UK.

Godefroid, P. (1994). Partial-order methods for the verification of concurrent systems – an approach to the state-explosion problem. Dissertation, Department of Computer Science, University of Liege, Belgium.

Godefroid, P. (1997). Model checking for programming languages using VeriSoft. In *POPL '97*, pages 174–186, New York. ACM.

Godefroid, P., de Halleux, P., Nori, A. V., Rajamani, S. K., Schulte, W., Tillmann, N., and Levin, M. Y. (2008). Automating software testing using program analysis. *IEEE Software*, 25(5):30–37.

Godefroid, P., Klarlund, N., and Sen, K. (2005). DART: Directed automated random testing. In *PLDI '05*, pages 213–223, New York. ACM.

Graf, S. and Hassen Saïdi, H. (1997). Construction of abstract state graphs with PVS. In *CAV '97*, volume 1254 of *LNCS*, pages 72–83, Heidelberg. Springer.

Gupta, S. K. and Sharma, N. (2003). Alias analysis for intermediate code. In *Proceedings of the First Anual GCC Developers' Summit, 25–27 May, 2003, Ottawa, Canada.* `http://www.linux.org.uk/~ajh/gcc/gccsummit-2003-proceedings.pdf` [24 September 2009].

Henzinger, T. A., Jhala, R., and Majumdar, R. (2004). Race checking by context inference. In *PLDI '04*, pages 1–13, New York. ACM.

Henzinger, T. A., Jhala, R., Majumdar, R., Necula, G. C., Sutre, G., and Weimer, W. (2002a). Temporal-safety proofs for systems code. In *CAV '02*, volume 2402 of *LNCS*, pages 382–399, Heidelberg. Springer.

Henzinger, T. A., Jhala, R., Majumdar, R., and Sanvido, M. A. A. (2003). Extreme model cecking. In *International Symposium on Verification: Theory and Practice*, volume 2772 of *LNCS*, pages 232–358, Heidelberg. Springer.

Henzinger, T. A., Jhala, R., Majumdar, R., and Sutre, G. (2002b). Lazy abstraction. In *POPL '02*, pages 58–70, New York. ACM.

Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580.

Hoare, T. (2003). The verifying compiler: A grand challenge for computing research. *J. ACM*, 50(1):63–69.

Holzmann, G. J. (2003). *The SPIN Model Checker*. Addison-Wesley Longman.

Horspool, R. N. and Marovac, N. (1980). An approach to the problem of detranslation of computer programs. *Computer Journal*, 23(3):223–229.

Horwitz, S., Reps, T., and Binkley, D. (1990). Interprocedural slicing using dependence graphs. *ACM TOPLAS*, 12(1):26–60.

Hynix Semiconductor et al. (2008). Open NAND flash interface specification, revision 2.0. Technical report, ONFI. http://www.onfi.org.

Intel Corporation (2009). Intel 64 and IA-32 architectures software developer's manuals. Technical report. http://www.intel.com/products/processor/manuals/index.htm [24 September 2009].

Jhala, R. and Majumdar, R. (2005). Path slicing. *SIGPLAN Not.*, 40(6):38–47.

Jie, H. and Shivaji, S. (2004). Temporal Safety Verification of AVFS using BLAST. Project report submitted at the Baskin School of Engineering, University of California, Santa Cruz, USA. http://www.cse.ucsc.edu/classes/cmps203/Fall04/finalreports/Shivaji_CMPS203.pdf [24 September 2009].

Josh Berdine, C. C. and O'Hearn, P. W. (2005). Symbolic execution with separation logic. In *APLAS '05*, volume 3780 of *LNCS*, pages 52–68, Heidelberg. Springer.

Joshi, R. and Holzmann, G. J. (2007). A mini challenge: Build a verifiable filesystem. *Formal Aspects of Computing*, 19(2):269–272.

Kim, M. and Kim, Y. (2009). Concolic testing of the multi-sector read operation for flash memory file system. In *SBMF '09*, volume 5902 of *LNCS*, pages 251–265, Heidelberg. Springer.

King, J. C. (1976). Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394.

Kirchner, F. (2005). Store-based operational semantics. In *Seizièmes Journées Francophones des Langages Applicatifs*. INRIA.

KMDB (2009). Solaris modular debugger guide, chapter 7: Kernel execution control. http://docs.sun.com/app/docs/doc/816-5041 [24 September 2009].

Kolb, E., Sery, O., and Weiss, R. (2009). Applicability of the BLAST model checker: An industrial case study. In *PSI '09*, LNCS, Heidelberg. Springer. To appear.

Korel, B. and Laski, J. (1990). Dynamic slicing of computer programs. *J. Syst. Softw.*, 13(3):187–195.

Koshy, J. (2009). LibElf. http://wiki.freebsd.org/LibElf [24 September 2009].

Kroening, D., Sharygina, N., Tonetta, S., Tsitovich, A., and Wintersteiger, C. M. (2008). Loop summarization using abstract transformers. In *ATVA '08*, volume 5311 of *LNCS*, pages 111–125, Heidelberg. Springer.

Kroening, D. and Strichman, O. (2008). *Decision Procedures*. Springer, Heidelberg.

Ku, K. (2008). *Software Model-Checking: Benchmarking and Techniques for Buffer Overflow Analysis*. PhD thesis, University of Toronto.

Ku, K., Hart, T. E., Chechik, M., and Lie, D. (2007). A buffer overflow benchmark for software model checkers. In *ASE '07*, pages 389–392, New York. ACM.

Kuncak, V. and Rinard, M. (2004). On spatial conjunction as second-order logic. Technical report, Computer Science and AI Lab, MIT, Cambridge, USA.

Leung, A. and George, L. (1999). Static single assignment form for machine code. In *PLDI '99*, pages 204–214, New York. ACM.

Leven, P., Mehler, T., and Edelkamp, S. (2004). Directed error detection in C++ with the assembly-level model checker StEAM. In *Model Checking Software*, volume 2989 of *LNCS*, pages 39–56, Heidelberg. Springer.

Mehta, F. and Nipkow, T. (2005). Proving pointer programs in higher-order logic. *Inf. Comput.*, 199(1-2):200 – 227.

Memon, A. M., Pollack, M. E., and Soffa, M. L. (2001). Hierarchical GUI test case generation using automated planning. *IEEE Trans. Softw. Eng.*, 27(2):144–155.

Microsoft Corporation (2004). Static driver verifier: Finding bugs in device drivers at compile-time. Website. Available online at `http://www.microsoft.com/whdc/devtools/tools/SDV.mspx`; [24 September 2009].

Mong, W. S. (2004). Lazy abstraction on software model checking. Project report submitted at the Department of Computer Science, University of Toronto, Canada. Available online at `http://www.cs.toronto.edu/~arie/csc2108conf/mong.pdf` [24 September 2009].

Mühlberg, J. T. and Lüttgen, G. (2007a). BLASTing Linux code. In *FMICS '06*, volume 4346 of *LNCS*, pages 211–226, Heidelberg. Springer.

Mühlberg, J. T. and Lüttgen, G. (2007b). BLASTing Linux Code. Technical Report YCS-2007-417, Department of Computer Science, University of York, UK.

Mühlberg, J. T. and Lüttgen, G. (2009). Verifying compiled file system code. In *SBMF '09*, volume 5902 of *LNCS*, pages 306–320, Heidelberg. Springer.

Mühlberg, J. T. and Lüttgen, G. (2010). Symbolic object code analysis. Technical report appeared in "Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik" 85/2010, Faculty of Information Systems and Applied Computer Science, The University of Bamberg. ISSN 0937-3349, `http://www.opus-bayern.de/uni-bamberg/volltexte/2010/236/`.

Necula, G. C., McPeaki, S., and Weimer, W. (2005). CCured: type-safe retrofitting of legacy code. *ACM TOPLAS*, 27(3):477–526.

Nethercote, N. and Fitzhardinge, J. (2004). Bounds-checking entire programs without recompiling. In *SPACE '04*, New York. ACM. `http://www.diku.dk/topps/space2004/space_final/nethercote-fitzhardinge.pdf` [24 September 2009].

Nethercote, N. and Seward, J. (2007). Valgrind: A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100.

Nielson, F., Nielson, H. R., and Hankin, C. (1999). *Principles of Program Analysis.* Springer, Heidelberg.

NLKD (2008). Novell Linux kernel debugger (NLKD). `http://forge.novell.com/modules/xfmod/project/?nlkd` [24 September 2009].

Noll, T. and Schlich, B. (2008). Delayed nondeterminism in model checking embedded systems assembly code. In *Hardware and Software: Verification and Testing*, volume 4899 of *LNCS*, pages 185–201, Heidelberg. Springer.

Ottenstein, K. J. and Ottenstein, L. M. (1984). The program dependence graph in a software development environment. In *SDE '84*, pages 177–184, New York. ACM.

Perens, B. (2009). Electric fence. `http://perens.com/FreeSoftware/ElectricFence/` [24 September 2009].

Purify (2009). IBM Rational Purify. `http://www.ibm.com/software/awdtools/purify/` [24 September 2009].

Păsăreanu, C. S., Mehlitz, P. C., Bushnell, D. H., Gundy-Burlet, K., Lowry, M., Person, S., and Pape, M. (2008). Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In *ISSTA '08*, pages 15–26, New York. ACM.

Reynolds, J. C. (2000). Intuitionistic reasoning about shared mutable data structure. In *Millennial Perspectives in Computer Science*, pages 303–321. Palgrave.

Reynolds, J. C. (2002). Separation logic: A logic for shared mutable data structures. In *LICS '02*, pages 55–74, Washington. IEEE.

Rinetzky, N., Bauer, J., Reps, T., Sagiv, M., and Wilhelm, R. (2005). A semantics for procedure local heaps and its abstractions. In *POPL '05*, pages 296–309, New York. ACM.

Sagiv, M., Reps, T., and Wilhelm, R. (1998). Solving shape-analysis problems in languages with destructive updating. *ACM TOPLAS*, 20(1):1–50.

Schlich, B. and Kowalewski, S. (2006). [mc]square: A model checker for microcontroller code. In *ISOLA '06*, pages 466–473, Washington. IEEE.

Sen, K., Marinov, D., and Agha, G. (2005). CUTE: A concolic unit testing engine for C. In *ESEC/FSE-13*, pages 263–272, New York. ACM.

Sery, O. (2009). Enhanced property specification and verification in BLAST. In *FASE '09*, volume 5503 of *LNCS*, pages 456–469, Heidelberg. Springer.

Swift, M., Bershad, B., and Levy, H. (2005). Improving the reliability of commodity operating systems. *ACM Trans. Comput. Syst.*, 23(1):77–110.

Tip, F. (1994). A survey of program slicing techniques. Technical report, Centre for Mathematics and Computer Science, Amsterdam, The Netherlands.

Tool Interface Standards (TIS) Committee (1995). Executable and linking format (ELF) specification version 1.2. Technical report.

Valgrind (2009). Valgrind – debugging and profiling Linux programs. `http://valgrind.org/` [24 September 2009].

van Emmerik, M. J. (2003). Type inference based decompilation. Phd confirmation report, School of Information Technology and Electrical Engineering, University of Queensland, Australia.

Venet, A. (1999). Automatic analysis of pointer aliasing for untyped programs. *Sci. Comput. Program.*, 35(2-3):223–248.

Visser, W., Havelund, K., Brat, G., Park, S. J., and Lerda, F. (2003). Model checking programs. *FMSD*, 10(2):203–232.

Wahab, M. (1998). Object Code Verification. Dissertation, University of Warwick, UK.

Weiser, M. (1981). Program slicing. In *ICSE '81*, pages 439–449, Washington. IEEE.

Wilhelm, R., Sagiv, M., and Reps, T. (2000). Shape analysis. In *CC '00*, volume 1781 of *LNCS*, pages 1–16, Heidelberg. Springer.

Wilson, R. P. and Lam, M. S. (1995). Efficient context-sensitive pointer analysis for C programs. In *PLDI '95*, pages 1–12, New York. ACM.

Witkowski, T., Blanc, N., Kroening, D., and Weissenbacher, G. (2007). Model checking concurrent Linux device drivers. In *ASE '07*, pages 501–504, New York. ACM.

Xie, Y. and Aiken, A. (2007). SATURN: A scalable framework for error detection using boolean satisfiability. *ACM TOPLAS*, 29(3):16.

Xu, Z., Miller, B. P., and Reps, T. (2000). Safety checking of machine code. In *PLDI '00*, pages 70–82, New York. ACM.

Yang, H., Lee, O., Calcagno, C., Distefano, D., and O'Hearn, P. (2007). On scalable shape analysis. Technical Report RR-07-10, Queen Mary, University of London.

Yang, J., Sar, C., Twohey, P., Cadar, C., and Engler, D. R. (2006). Automatically generating malicious disks using symbolic execution. In *Security and Privacy*, pages 243–257, Washington. IEEE.

Yang, J., Twohey, P., Engler, D. R., and Musuvathi, M. (2004). Using model checking to find serious file system errors. In *OSDI*, pages 273–288. USENIX.

Yu, D. and Shao, Z. (2004). Verification of safety properties for concurrent assembly code. In *ICFP '04*, pages 175–188, New York. ACM.

Zitser, M., Lippmann, R., and Leek, T. (2004). Testing static analysis tools using exploitable buffer overflows from open source code. *SIGSOFT Softw. Eng. Notes*, 29(6):97–106.

Author Index

Agha, G. 27, 94
Aiken, A. 26, 86

Balakrishnan, G. 12, 20, 22, 24, 28, 29
Ball, T. 12, 13, 16, 18, 23, 24, 29, 86
Barry, R. 99
Bauer, J. 21
Bershad, B. 12, 13
Beyer, D. 25, 31, 32, 36, 40
Biere, A. 25
Binkley, D. 68
Blanc, N. 86
Bounimova, E. 12, 18, 29
Bovet, D. 81, 82
Boyer, R. S. 27
Bozga, M. 21
Brat, G. 25
Breuer, P. T. 19
Brummayer, R. 25
Burstall, R. M. 20
Bushnell, D. H. 26
Butler, M. 93, 94
Butterfield, A. 93, 94

Cadar, C. 26, 93
Calcagno, C. 21, 27, 98
Catháin, A. Ó. 93, 94
Cesati, M. 81, 82
Chandra, S. 24
Chechik, M. 14, 74, 89
Chelf, B. 6, 11, 12, 16, 17, 19, 30
Chlipala, A. J. 36, 40
Chou, A. 6, 11, 12, 16, 17, 19, 30
Ciardo, G. 82
Clarke, E. 24, 82
Clarke, E. M. 19, 22–24, 74
Cook, B. 12, 18, 29
Corbet, J. 13, 16, 17
Corbett, J. C. 24
Cousot, P. 19, 23
Cousot, R. 19
Cullmann, C. 69
Cytron, R. 50

Damchoom, K. 93, 94
de Halleux, P. 27
de Moura, L. 14, 45, 61, 65, 73
Debray, S. 22
Deutsch, A. 21
Dill, D. L. 26
Distefano, D. 21, 27, 98
D'Silva, V. 19
Dutertre, B. 14, 45, 61, 65, 73
Dwyer, M. B. 24

Edelkamp, S. 25
Emerson, E. A. 22
Engler, D. R. 6, 11, 12, 16, 17, 19, 26, 30, 93

Ferdinand, C. 69
Ferrante, J. 50
Ferreira, M. A. 93, 94
Fitzhardinge, J. 14
Flanagan, C. 99

Gälli, M. 24
Galloway, A. 82, 84, 93
Ganesh, V. 26
George, L. 50
Godefroid, P. 22–24, 26, 27, 99
Graf, S. 23
Greevy, O. 24
Grumberg, O. 19, 23, 24
Gundy-Burlet, K. 26
Gupta, S. K. 22

Hallem, S. 6, 11, 12, 16, 17, 19, 30
Hankin, C. 74
Hart, T. E. 14, 74, 89
Hassen Saïdi, H. 23
Hatcliff, J. 24
Havelund, K. 25
Heckmann, R. 69
Henzinger, T. A. 12, 23–25, 30–32, 36, 40, 82, 86
Hoare, C. A. R. 20
Hoare, T. 81
Holzmann, G. J. 24, 81
Horspool, R. N. 26, 29
Horwitz, S. 68
Hynix Semiconductor et al. 99

Intel Corporation 14, 45, 54
Iosif, R. 21

Jhala, R. 12, 23–25, 30–32, 36, 40, 68, 82, 86
Jie, H. 32
Jones, R. L. 82
Josh Berdine, C. C. 27
Joshi, R. 81

Kim, M. 27, 93, 94
Kim, Y. 27, 93, 94
King, J. C. 26
Kirchner, F. 20
Klarlund, N. 26
Kolb, E. 32
Korel, B. 24, 68
Koshy, J. 73
Kowalewski, S. 25, 26
Kroah-Hartmann, G. 13, 16, 17
Kroening, D. 19, 24, 53, 74, 75, 82, 86
Ku, K. 14, 74, 75, 89
Kuncak, V. 20

Laknech, Y. 21
Lam, M. S. 21
Laski, J. 24, 68
Laubach, S. 24
Lee, O. 21, 98
Leek, T. 74
Lerda, F. 24, 25, 82
Leung, A. 50
Leven, P. 25
Levin, M. Y. 27
Levin, V. 12, 18, 29
Levy, H. 12, 13
Lichtenberg, J. 12, 18, 29
Lie, D. 14, 74, 89
Lippmann, R. 74
Long, D. E. 23
Lonsing, F. 25
Lowry, M. 26
Lüttgen, G. 27, 31, 32, 82, 84, 93

Majumdar, R. 12, 23–25, 30–32, 36, 40, 68, 82, 86
Marinov, D. 27, 94
Marovac, N. 26, 29
Martin, F. 69
McGarvey, C. 12, 18, 29
McPeaki, S. 25, 31
Mehler, T. 25
Mehlitz, P. C. 26
Mehta, F. 20
Melski, D. 12, 20, 28
Memon, A. M. 24
Microsoft Corporation 25
Miller, B. P. 29
Miner, A. S. 82
Mong, W. S. 32
Mühlberg, J. T. 27, 31, 32, 82, 84, 93
Musuvathi, M. 93
Muth, R. 22

Necula, G. C. 12, 23–25, 30, 31, 40, 82, 86
Nethercote, N. 14, 45, 49, 61, 73
Nielson, F. 74
Nielson, H. R. 74
Nierstrasz, O. 24
Nipkow, T. 20
Noll, T. 25
Nori, A. V. 27

O'Hearn, P. 21, 27, 98
O'Hearn, P. W. 27
Oliveira, J. N. 93, 94
Ondrusek, B. 12, 18, 29
Ottenstein, K. J. 68
Ottenstein, L. M. 68

Palm, C. 24

Pape, M. 26
Park, S. J. 25
Păsăreanu, C. S. 24
Pawlowski, P. M. 26
Peled, D. A. 19, 24
Perens, B. 18
Person, S. 26
Pickin, S. 19
Pollack, M. E. 24
Păsăreanu, C. S. 26

Rajamani, S. K. 12, 16, 18, 23, 24, 27, 29, 86
Reps, T. 12, 19–22, 24, 27–29, 68
Reynolds, J. C. 20, 27
Rinard, M. 20
Rinetzky, N. 21
Robby 24
Rosen, B. K. 50
Rubini, A. 13, 16, 17

Sagiv, M. 19, 21, 27
Sanvido, M. A. A. 31
Sar, C. 93
Schlich, B. 25, 26
Schlickling, M. 69
Schulte, W. 27
Sen, K. 26, 27, 94
Sery, O. 25, 32
Seward, J. 45, 49, 61, 73
Shao, Z. 27
Sharma, N. 22
Sharygina, N. 24, 74, 75
Shivaji, S. 32
Siminiceanu, R. 82, 84, 93
Siminiceanu, R. I. 82
Sistla, A. P. 22
Soffa, M. L. 24
Stein, I. 69
Strichman, O. 53
Sutre, G. 12, 23–25, 30, 31, 40, 82, 86
Swift, M. 12, 13

Teitelbaum, T. 12, 20, 28
Thesing, S. 69
Tillmann, N. 27
Tip, F. 68
Todd Millstein, T. 23
Tonetta, S. 74, 75
Tool Interface Standards (TIS) Committee 14, 47, 72
Tsitovich, A. 74, 75
Twohey, P. 93

Ustuner, A. 12, 18, 29

van Emmerik, M. J. 29
Venet, A. 21
Visser, W. 25

Wahab, M. 28, 29
Wegman, M. N. 50
Weimer, W. 12, 23–25, 30, 31, 40, 82, 86
Weippert, M. 22
Weiser, M. 24, 68
Weiss, R. 32
Weissenbacher, G. 19, 86
Wilhelm, R. 19, 21, 27
Wilson, R. P. 21
Wintersteiger, C. M. 74, 75
Witkowski, T. 86
Wuyts, R. 24

Xie, Y. 26, 86
Xu, Z. 29

Yang, H. 21, 27, 98
Yang, J. 6, 11, 12, 16, 17, 19, 30, 93
Yorav, K. 24, 74
Yu, D. 27
Yu, Y. 27

Zadeck, F. K. 50
Zheng, H. 24
Zitser, M. 74