

PhD Qualifying Dissertation

Validating and Verifying Memory Safety for Concurrent Operating System Code

Jan Tobias Mühlberg
Department of Computer Science
University of York
muehlber@cs.york.ac.uk

June 30, 2006

Abstract

The current practice of finding programming errors in operating system development is by testing and debugging. However, testing techniques are expensive because of their requirement of manual labour. Furthermore, they are susceptible to missing severe errors. This problem can be solved by applying automated verification techniques such as software model checking. Most of these techniques suffer from limitations in analysing pointer programs and dealing with concurrency. Because of this, verification methods available today are not sufficient to cover the class of software defects related to memory safety in concurrent reactive software systems.

In this dissertation we give a review on current practice and state-of-the-art techniques and tools for detecting memory safety errors in computer programs. Furthermore we outline a new approach in validating and verifying memory safety properties for concurrent reactive software systems such as device drivers. Our approach is based on both, program simulation and software model checking. In contrast to previous work on memory safety problems, we intend to provide a fully automatic analysis and verification framework covering a broad range of memory safety related problems.

Contents

1	Introduction	3
2	Problem Statement	4
2.1	An Overview of Errors in Operating Systems	4
2.2	Finding Bugs in Device Drivers	6
2.3	Object Code Verification vs. Source Code Verification	8
2.4	Simulation vs. Verification	9
3	State of the Art and Current Practice	11
3.1	Pointer Programs	11
3.2	Alias Analysis	12
3.3	Abstraction and Partial Order Techniques	14
4	Proposal	16
4.1	Work Package 1: Memory Model	17
4.2	Work Package 2: Abstraction	18
4.3	Work Package 3: Simulation	19
4.4	Work Package 4: Model Checking	20
4.5	Work Package 5: Tool Support	21
4.6	Work Package 6: Evaluation	21
4.7	Work Schedule	22
5	Preliminary Results	24
5.1	Case Study with BLAST	24
5.2	Memory Model and Tool Development	25
	References	27
A	Mühlberg and Lüttgen: BLASTing <i>Linux Code</i>	33

1 Introduction

During the last 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 finding subtle errors in them. Software defects like buffer overflows and deadlocks mainly decrease system's reliability and hence, render them unusable as components of dependable systems that control power plants or are used in avionic systems. In many cases these defects also have security implications like leaking secrets or allowing an attacker to bypass authentication systems. Thus, software defects frequently render nicely designed security infrastructures useless.

This dissertation deals with the problem of programming errors related to memory safety in operating systems. By the term "memory safety" we mean that a given program does not violate basic safety rules of the operating system's programming interface by de-referencing invalid pointers, exceeding boundaries of memory structures or calling de-allocation functions in a wrong context. Such programming errors will, in most cases, result in undefined behaviour of the system under consideration.

As we will show in Sec. 2, there are many cases in which such problems occur. Most of them cannot be found in the operating system kernel but in extensions such as device drivers. However, errors in kernel extensions affect the whole operating system and have deep impact on the reliability of programs at application level. The current practice of finding memory safety related bugs in device driver development is by 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 not being limited to a set of test cases, these methods provide a higher level of confidence in the correctness of a program in respect of a given property.

In Sec. 3 we give a review on verification methods used to detect memory safety problems in computer programs. Since all techniques available today have several shortcomings in terms of completeness, automatisability and efficiency, we propose an approach based on object code abstraction, simulation and model checking to overcome these deficiencies in Sec. 4. In Sec. 5 we finally present the results of our research up to now. In addition to the summary of our findings from a case study on verifying memory safety and locking properties for Linux device drivers using the tool BLAST, we have attached our full paper in App. A.

2 Problem Statement

Today's application software critically depends on the reliability, safety and security of the underlying operating system (OS). Due to their complicated task of managing a system's physical resources, OSs are difficult to develop and even more difficult to debug. Quite frequently major errors remain undiscovered until they are exploited in security attacks or are found "by accident". As recent publications show, most defects causing OSs to crash are not in the system's kernel but in the large number of OS extensions available [57, 14]. In Windows XP, for example, 85% of reported failures are caused by errors in device drivers [5]. Ongoing research shows, that the situation is similar for Linux and BSD. The error rates reported for device drivers are up to seven times higher than in rest of the kernels of these OSs [14].

There are several reasons for the high number of errors in device drivers. Firstly, a device driver is a nondeterministic reactive system. It is continuously responding to different events – i.e., user requests and hardware interrupts – for which neither order nor time of occurrence are predictable in advance. Furthermore, drivers are often required to provide timely responses. In order to do this, they must be able to run in a preemptive OS kernel, where the driver's normal operation may be interrupted at any time.

As Ball points out in [5], drivers run in a highly concurrent environment provided by the OS. 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 OSs 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.

In addition to this, device drivers are frequently written by developers who are less experienced in using the kernel's interface than those who built the OS itself [57]. They tend to ignore side-effects of the kernel's application programming interface (API) and thereby introduce subtle errors that break the OS's safety and security and that are often very difficult to find. All this renders driver development rather difficult.

2.1 An Overview of Errors in Operating Systems

There is a large number of commonly found OS errors. An insightful study on this topic has been published in [14]; see Table 1 for a summary of its results. The authors of this study 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 cov-

% 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 1: Results of an empirical study of OS errors [14]

ered by several formal software engineering tools such as SDV [8], an industry strength software model checker for Microsoft Windows device drivers, memory safety remains a major issue. Likewise our case study on the "Berkeley Lazy Abstraction Software verification Tool" (BLAST, c.f. App. A) comes to the result that this state-of-the-art verification toolkit does not cover memory safety in full.

Although memory safety problems have a direct impact on an OS's reliability, programming rules for OS kernels are usually described in an informal way. For example, it is stated in the Linux device driver handbook [20, 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. Throughout this dissertation we distinguish between user/application space programs and kernel space programs. OS extensions such as device drivers run in kernel space. This means that they run in a privileged execution mode, which is implemented in the system's CPU. Furthermore, kernel level programs use a different programming interfaces than application programs. For example, the functions `kmalloc()` and `kfree()` are kernel-space functions which are used to dynamically allocate and de-allocate memory, respectively. They are almost equivalent to `malloc()` and `free()`, which are known from the standard C library of common OSs.

2.2 Finding Bugs in Device Drivers

The problem of 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 [6], these are related to the restricted observability inside OS 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 OS provides several different kernel-level programming interfaces, "which gives rise to many ways in which a driver can misuse these APIs". Most of the API violations do rarely result in immediate failures but leave the OS in an inconsistent state. This may be a crash or improper behaviour at a later time, mostly without revealing the source of the error.

Runtime analysis. While popular runtime analysis tools which target memory safety problems are mainly available for the development of software at the application level, the large domain of OS kernels and device drivers is rarely covered. Toolkits such as Purify [52] and Valgrind [58] 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 [50] 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 "kldb" [43] for Solaris or the Novell Linux Kernel Debugger [49] 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 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 [18]. Formal verification can establish a much higher confidence in a program under consideration by assuring that a certain property holds for all possible executions.

Static analysis and abstract interpretation. 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 via an approach based on system-specific compiler extensions, known as meta-level compilation [28]. This method is implemented in the tool Coverity [24] and was used in an extensive study on OS errors [14]. Also most of the examples for memory safety bugs in the Linux kernel analysed in our case study on BLAST have been detected using this technique.

A further recent attempt to find bugs in OS code is based on abstract interpretation [23] and presented in [11]. 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 overview of the advantages and disadvantages of static analysis versus model checking can be found in [29].

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 [18]. Intensive research in this area has resulted in software model checkers like Bandera [21] for Java programs or SLAM/SDV [8] and BLAST [36] for analysing C source code. The major advantage of these tools over model-based model checkers such as Spin [39] 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 API 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 [47], while BLAST is able to verify a program against a user defined temporal safety specification [36] 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 Sec. 5.1, the BLAST toolkit has several shortcomings related to the detection of memory safety problems and concurrency issues.

2.3 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 [4], 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 software are only visible after compilation [4].

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. Even the condition that 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, shows that object code analysis should be taken into account [4]. 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 [61].

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 [61]. 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" [6]. This means that the tool does not check memory safety but assumes it.

Advantages 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 [65, 3]. Furthermore, the use of function pointers and computed jumps (`set jmp()` and `long jmp()` 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 [61], 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. At least in the step of abstracting an intermediate representation – a model – from the program under consideration, all instructions need to be taken into account. Because of this, verifying object code is much more difficult than verifying a program in its high-level source code.

In order to analyse an object code program, it needs to be disassembled first. As van Emmerik explains in [59], this step requires the separation of data from code, which is not given in machine code programs. In [41], Horspool and Marovac¹ show that this problem is undecidable in general, thus requiring approximation. However, recent work such as [65, 3] shows, 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.

2.4 Simulation vs. Verification

One of the major limitations of software verification techniques based on exhaustive state-space exploration such as model checking is the state-explosion problem [32]. Since software systems like OSs and device drivers are highly complex systems with infinite-state behaviour, exhaustively verifying them consumes massive amounts of memory and CPU time. Especially in the area of memory safety ver-

¹As cited by van Emmerik in [59].

ification, where several address values may be required to be introduced into the model [2], the verification process will easily exceed the limitations of a programmer's patience. Furthermore, one has to take into account that CPU time for allocating and de-allocating memory is usually considered as overhead. However, in order to verify that a program is memory safe, we are mainly interested in this overhead but can safely abstract away from most of the application logic: This means that we do not need to run sections of a program that compute results the user is interested in. We can safely abstract away from these computations as long as we can assure that control never reaches a situation in which memory safety is violated. Therefore, tools for formal software development may gain in practicality if they provide a fast model generation and offer the functionality to use this model either in exhaustive analysis or in a fast simulation of system behaviour.

3 State of the Art and Current Practice

In this section we will give a concise outline on ongoing research, techniques and tools for the analysis of computer programs. We centre around pointer programs, alias analysis and abstraction techniques.

3.1 Pointer Programs

The analysis of pointer programs has been an important but still unsolved issue in Computer Science for more than thirty years. In [63], 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 part 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 [4] 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 [12]. 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 [12]². The basic idea of Burstall's work provides a "store-based" operational semantics [42] for heap usage by modelling the heap used by a program

²As cited by Reynolds in [53].

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 [38]. The approach has been applied in recent research on verifying pointer programs using separation logic with spatial conjunction [53, 54, 44] and on proof automation by providing integration in existing theorem proving infrastructures [45]. 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, OSs, 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 compute the effect of procedures on the global heap from their effect on sub-heaps [55].

However, with the emergence of programming languages such as Java, store-less semantics for heap access have been developed [10]. By abstracting away from specific memory addresses, these heap representations provide a conceptual and compact view on the memory usage of a program.

3.2 Alias Analysis

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 [56, 63]. 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 [27, 26], Deutsch provides a very exact alias analysis for high-level programs based on a store-less semantics and abstract interpretation (c.f. Sec. 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 [60], 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 [64], but makes conservative assumptions about aliasing for several cases in which the analysis will fail.

In [9], the use of CCURED [48] 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 practise.

Another major restriction for alias analysis techniques based on abstractions of high-level programming languages lies in the control flow of many programs. Since alias analysis techniques need to follow the program execution in order to determine memory allocations, deallocations and pointer statements, 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 linker are an interesting examples for this. In [25], 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 [35]. Another recent approach for an memory analysis algorithm based on the inspection of object code is given by Balakrishnan and Reps in [2]. 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.

Two interesting tools that operate directly on object code have been proposed in [65] and [1]. Both tools are able to analyse machine code programs in absence of source code or debugging information. While approach taken by [65] is

focusing on fully automatic but not exhaustive program analysis of user defined memory safety properties, the latter is mainly meant as a support tool for the manual analysis of machine code programs. However, both approaches do not cover concurrent execution runs as they are required for the analysis of OS components. Nevertheless, they provide valuable ideas and inspiration for further research.

For the analysis and verification of highly concurrent software systems such as OS kernels and device drivers, none of the mentioned technique and tools will be sufficient. This is because they are not designed for analysing concurrent program execution and will therefore not be able to deal with memory races arising from interleaving executions.

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 [16], current software systems, especially in the domain of OS verification, are infinite-state systems. Constructing their state space leads to the state explosion problem as explained by Godefroid in [32]. 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 [17].

Predicate abstraction. Most abstraction techniques currently used in software model checking are based on the work of Graf and Saïdi in [34]. The author’s approach employs abstract interpretation [22] 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 [7, 37] and implemented in tools such as SLAM [8] and BLAST [36]. 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 App. A, 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 [32]. 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 [39] for communication protocols as well as VeriSoft for verifying software systems [33, 13]. The VeriSoft approach is particularly interesting. As summarised in [13] 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 [46, 30, 31]. These approaches aim on the reduction of the total amount of test cases by identifying and removing cases, which are already covered by others.

4 Proposal

The goal of the proposed research project is to provide novel tools and techniques for validating and verifying memory safety properties for concurrent reactive OS extensions such as device drivers. In contrast to previous work on model checking memory safety, such as [9], we will interpret pointer invalidity in a more general way, which is not restricted on the detection of `NULL`-pointer de-references. Analysis tools covering safety issues where invalid pointers may not be `NULL` are only available in the form of manual inspection suites such as [2, 1, 19]. Unlike their approach, we intend to provide a fully automatic simulation and model checking framework. A major challenge for this is to deal with an extensive memory model consisting of possibly unbounded numbers of pointer values and complex control flow structures in an automatic verification toolkit. Another example of strongly related work on safety checks for machine code has been published by Xu et al. in [65]. We distinguish from their work by developing techniques that do not require the manual annotation of a program with typestate information. Furthermore, we aim to overcome their restrictions regarding pointer arithmetic and array references. The second big challenge of the proposed research lies in its difference to [1] and [65]: We are focusing on verifying memory safety for concurrent programs, which is not covered by previous work at all.

The research will aim at the development of efficient methods for the automatic abstraction of intermediate representations from concurrent machine code programs such as OS extensions. These intermediate representations, models, need to preserve information that is relevant for an exhaustive analysis of memory safety properties of a program under consideration. We intend to analyse the models using simulation runs and model checking techniques. Furthermore, we propose to develop an automated tool for the abstraction, simulation and verification of object code programs. The results of the research will be evaluated by case study and by comparison with other verification tools. To the knowledge of the author, automated verification techniques based on object code programs, and specialised on memory safety properties for highly concurrent reactive systems such as device drivers, are not available yet.

In general, there are two classes of memory safety problems we are interested in. Firstly, we want to deal with invalid pointer operations taking place in memory areas allocated by a device driver under consideration. Under this limitation we do not consider the content of memory areas which have been passed to the driver from other parts of the OS kernel. The following list gives an overview on error conditions that will be detected by our tool:

- de-referencing `NULL`-pointers
- exceeding memory boundaries

- accessing unallocated memory
- accessing code segments
- memory races

We expect our results to be helpful for the development of drivers that use only memory which has been allocated by the driver under consideration itself.

We intend to extend our approach towards covering a second class of memory safety problems. This class includes drivers that make extensive use of techniques such as Direct Memory Access or memory mapped I/O [20, pp. 412 ff.] and work on memory ranges that are externally modified by other parts of the OS or the underlying hardware. In this case a device driver needs to make sense of memory provided directly by hardware components such as network adaptors or disk drive controllers. Covering problems emerging from misinterpretation of these memory areas will require us to investigate means of include more complex descriptions of the driver's environment in our models. The approach is considered as a valuable add-on for the proposal given in the remainder of this section.

4.1 Work Package 1: Memory Model

Since our work centres around reasoning about allocations, de-allocations and memory access in programs, we need to provide a semantics for these operations. Particularly, we intend to develop a memory model that provides a generalisation for registers, heap and stack in commonly used computer architectures. Since we are not interested in storing application data but only need to preserve important attributes of pointer values and the shape of a possibly unbounded set of dynamically allocated entities, our model will require much less space than the memory consumed by the program to be analysed. To simplify program simulation, we intend to construct models that combine relevant parts of a program's control flow with information related to its memory usage.

The most natural way of modelling memory operations that have been obtained from analysing object code is by using a store-based operational semantics. As we are going to take this approach, we do not abstract away from using distinguishable heap addresses. While this proceeding may increase the size of the model substantially, it allows us to obtain memory shape information even if a program under consideration builds up complex linked structures without allocating space for every virtual entity separately. These shape information can be used to generate more abstract representations of a program under consideration, which are mainly required for program verification using model checking.

We expect that using efficient representations of heap cells and operations on them, the model will reveal interesting information about the heap usage of machine code programs during program simulation.

Main research questions:

- What properties of dynamically and statically allocated memory need to be preserved for simulating a sequential program’s memory usage behaviour?
- What additional information do we need for simulating concurrent behaviour?
 - How can atomic memory access be modelled in contrast to divisible operations?
 - Under what conditions does memory have visible effects for other threads of control?
 - How can sequential memory access be guaranteed?
- What information is required to make inconsistencies or other problems detected in the model feasible in the program?
- How can the model be implemented efficiently?
- What are the capabilities and limitations of the resulting model?

4.2 Work Package 2: Abstraction

Since the proposed verification system will provide both, simulation and model checking facilities, we intend to use a two-layered abstraction scheme.

In a first abstraction pass, we will construct the input for the simulation process. Therefore we intend to adapt program slicing [62] and techniques for tracking pointer-valued and integer-valued data objects as discussed in [2] to identify pointer variables and relevant sections of the control flow of an object code program under consideration. Using this information, we intend to construct a new program that contains all parts of the original program’s control flow that are relevant for allocation, de-allocation, pointer manipulation and memory access. However, memory operations will not be directly executed but operate on a symbolic heap representation, our memory model, as explained in Sec. 4.1. By reducing the program under consideration in this way, we intend to reduce its complexity by an order of magnitude and provide a testing environment that is not as susceptible to destructive heap corruption as debugging-based approaches are.

The second stage of the abstraction phase will be invoked in order to model check memory access of the program under consideration. Based on the numeric abstraction of heap and control flow generated in pass one, we intend to use abstract interpretation in an abstraction model-check refinement loop as implemented in recent software model checkers.

Main research questions:

- How can we safely re-introduce an approximation for separating data from control in order to identify pointer variables?
- Under what conditions will the effects of write operations be seen in other threads of concurrent program execution? What invisible memory operations, e.g., need to be preserved?
- What parts of a program's control flow need to be preserved in an intermediate program representation? In what cases can we safely reduce the control flow and construct highly abstract representations?
- Is it possible to generate abstract executions from the programs control flow in order to guide the simulation process?
- Is the outlined two-layered abstraction sufficient for generating models for model checking memory safety?

4.3 Work Package 3: Simulation

Sequential Programs and Coverage Criteria. Firstly, we are interested in exploring sequential executions of an intermediate representation as generated in the first abstraction phase by simulation runs. While simulation will not be exhaustive, we intend to apply coverage criteria and coverage analysis in order to achieve a high condition and path coverage on the model when compared to random simulation runs. Therefore we need to generate abstract program executions representing memory safety related parts of program executions and that can be used within our model to perform a guided simulation. This approach is closely related to temporal logic based approaches for test generation and research on model based testing as explained in [40] and [51], respectively. However, we have to investigate whether these technologies are fully applicable for our domain.

Concurrent Programs and Partial Order Techniques. In a second stage of research on model-based simulation of system behaviour, we will consider concurrent program executions. This requires the automatic generation on a concurrent operating environment for the model under consideration. Since the domain

of our practical work will be restricted to Linux device drivers, we intend to generate this environment from the interface specification given by the kernel API and the driver's source respectively. The simulation will assume that any interrupt can occur at any time. By this we make sure that a driver under consideration cannot end up in an erroneous state due to unexpected or rare behaviour of its environment. Because of the many possible interleaving in concurrent driver executions, it can be expected that automatically generated abstract program executions will be considerably large. We therefore intend to apply partial order techniques to reduce the number of abstract executions to those which are provably sufficient to check our properties.

Main research questions:

- How can we efficiently simulate memory access behaviour of sequential and concurrent programs?
- Do program abstraction, generation of abstract executions and program simulation lead to a high path coverage and decision coverage?
- How reliable are results found by this technique?
- Can partial order techniques be applied in order to reduce the number of concurrent executions we have to cover?

4.4 Work Package 4: Model Checking

As the second approach in exploring abstractions generated from object code programs, we intend to apply model checking techniques. Since model checking is much more expensive than simulation in terms of computing power, we will apply further abstractions to our model. In order to do this, different techniques for abstraction and model checking need to be evaluated. We expect that our research can leverage most from partitioning models and from reducing the size of linked data structures to only a few elements. Furthermore, we will investigate whether novel and efficient algorithms for state space generation such as "Saturation" [15] can be used in order to make exhaustive verification of memory safety properties possible.

Main research questions:

- What abstractions and model checking tools can efficiently work on the memory model used in our simulation environment?

- Can we provide an efficient framework for exhaustive verification of memory safety properties for concurrent reactive software systems?
- What are the limitations of the approach? How does it scale with problem size?

4.5 Work Package 5: Tool Support

We intend to develop a prototypical toolkit performing abstraction, simulation and model checking on device drivers for the Linux kernel. Since this toolkit is crucial for the evaluation of the applicability of our results in industrial software development, we need to provide a tool that covers a widely used development platform. We therefore intend to implement the algorithms and techniques for the 64-bit processor architecture developed in cooperation by Intel and Hewlett-Packard (IA-64), implemented by processors such as the Itanium. While, in comparison with RISC architectures, the tool development for this platform might be more difficult due to the huge instruction set of the IA-64, we assume that providing a tool for this architecture will provide us with a larger set of example drivers that can be used in evaluation. Furthermore, we expect that kernel developers from the Linux community find some interest in the tool and provide us with feedback on coverage and performance issues.

4.6 Work Package 6: Evaluation

Development goals. At the present time, the Coverity tool seems to provide the most efficient static analysis suite available. As shown in [14] it has been used to detect hundreds of software defects related to memory safety in the Linux and BSD OS kernels. Achieving a higher level of problem coverage and exhaustiveness as the static analysis framework used in their work constitutes the main development goal. However, higher exhaustiveness usually comes along with increased execution times that may be unacceptable for software developers. Therefore, our second goal is to provide a toolkit that is considered to be useful by the Linux kernel community. We intend to achieve this goal by making a fair trade-off between efficiency and completeness by providing both simulation and model checking.

Comparison with other tools. In order to evaluate the results of our research, we will compare our approach with those of other research projects. As explained above, the key to this evaluation will be the toolkit implemented during the project. Unfortunately, a tool-based comparison of software verification tools will turn out to be rather difficult since neither commonly used evaluation metrics nor case studies on these tools are presently available. Furthermore, only very few tools

cover the area of memory safety problems in concurrently running OS kernel code to an extent as we intend to. While this situation may change during the next years, we currently assume that we will have to accomplish an extensive survey of our tool and perhaps two of the others such as Coverity [24], CodeSurfer/x86 [19] and possibly BLAST [36]. This evaluation may be performed by checking whether the selected tools can detect a set of errors in Linux device drivers that are known beforehand.

Quantitative evaluation. Another important way to evaluate our work will be based on automatically analysing a range of Linux device drivers and on releasing the toolkit to the Linux development community. Doing this, we expect to gather interesting information on the problem coverage, the efficiency and the usability of our tool. Furthermore, this will provide us with valuable feedback from the Linux community.

4.7 Work Schedule

In Fig. 1 I give an overview of the six work packages explained above. I intend to spend the next three months in further progression of our memory model. This work is a prerequisite for the next packages, the development of techniques for object code abstraction and the simulation of system behaviour related memory safety properties. We expect to be able to publish on first results of our modelling and simulation approach in October 2006. The development of our simulation and verification toolkit is also supposed to start in this autumn. The tool development is scheduled as a continuous task which takes place throughout the research project. By July 2007, I intend to provide a working prototype which performs automated safety checks on Linux device drivers. By this time I will focus on work package four, the adaption of model checking techniques for our verification framework. Furthermore, I expect to be able to start the evaluation phase for our simulation environment at this point. I intend to improve the toolkit permanently during the evaluation phase. Furthermore, I will implement model checking support during this time. By June 2008 I expect to finish our research and spend the final three months in a writing-up phase. I anticipate that, because of our generous planing of parts five and six, I will probably have sufficient time to research possibilities of analysing memory access under consideration of detailed models of the OS and the underlying hardware.

Risk assessment. I consider work packages two and three – abstraction and simulation – as the most challenging and also most critical parts of this research project. This is because all further research depends on them. However, in case

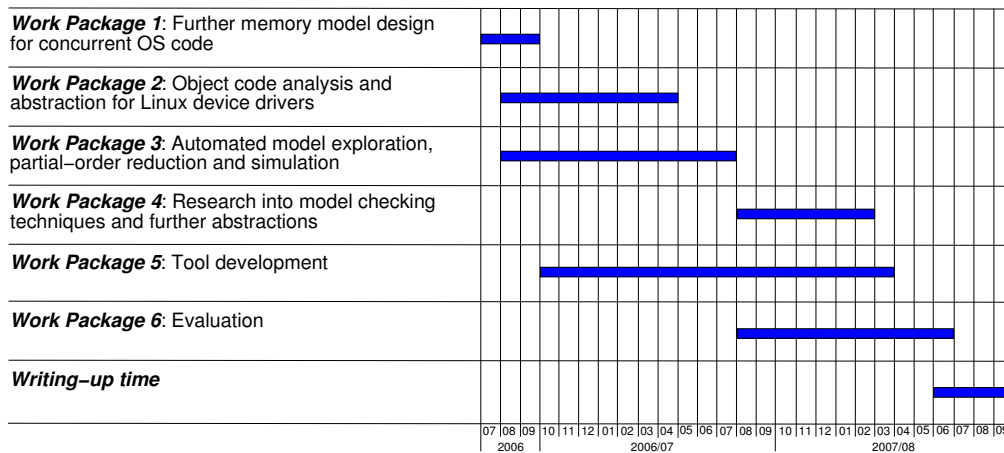


Figure 1: Workflow for the proposed research.

we will not be able to successfully finish this stage in time, it will be possible to shorten the phases of tool development and evaluation. These parts of the project are currently planned quite generous and reducing them will have consequence only in terms of not developing a industry strength toolkit but a prototype. Therefore, the impact on the scientific value of our research is minimised.

5 Preliminary Results

In this section we outline our research achievements up to now. These include a case study on the software model checking toolkit BLAST as well as ongoing work on the development of a memory model and simulation environment for software validation.

5.1 Case Study with BLAST

In order to evaluate the abilities of state-of-the-art software model checkers to deal with properties we are interested in, a case study on the tool BLAST[36] was conducted in cooperation with Gerald Lüttgen. BLAST is a popular and actively developed toolkit that implements an advanced abstraction algorithm, called "lazy abstraction" [37], 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*.

In the paper we investigate to which extent software model checking as implemented in BLAST can aid a practitioner during OS software development. To do so, we analyse whether BLAST is able to detect errors that have been reported for recent releases of the Linux kernel. We consider programming errors related to *memory safety* and *locking behaviour*. The code examples utilised in the paper 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. 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 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 www.cs.york.ac.uk/~muehlber/blast/.

Our experiments on memory safety show that BLAST is able to find the programming error discovered by the Coverity checker; however, it does so with 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.

For reference, the full paper is attached in App. A. It has been submitted to the 11th International Workshop on Formal Methods for Industrial Critical Systems (FMICS).

5.2 Memory Model and Tool Development

During the last months I have developed a rather simplistic simulation environment for dynamic allocation, de-allocation and memory access in sequential programs. The system has been implemented in C and allows us to run abstract programs which are also given in the form of C code. Up to now, program abstractions need to be provided by hand. Therefore we applied program slicing to simple programs working on linked data structures. Indeed, this is only useful for simulating the behaviour of small constructed examples.

The environment follows a store-based approach on modelling a program’s heap: It is represented as a set of memory areas. Each of them has a distinct start address and end address. Furthermore the system provides a mapping from addresses to a limited number of stored values such as pointers or counters used in loops.

In several small examples our model turned out to be sufficient for simulating abstract programs which operate on complex linked data structures such as doubly linked lists and trees. We were able to detect several faults including accessing invalid addresses, exceeding of area boundaries and memory leakages. However, the purpose of the tool is not to actually validate programs but to experiment with abstract store-based program representations. Presently we are not able to give

precise statements on the soundness of the results obtained from simulation runs. We attempted to use the BLAST toolkit in order to model check the simulation environment including an abstract program. This failed because BLAST appeared to be unable to operate on our symbolic heap representation.

Currently our simulation environment does not support concurrent program simulation because it neither provides an infrastructure for representing locks on memory resources, nor does it support a scheduling environment. We intend to introduce these requirements for simulating concurrent executions into the environment shortly. However, the major challenge of our approach lies in the automatic generation of abstract program representations. Especially distinguishing between address information and other integer values is considered as rather difficult. We intend to solve this problem by applying multi-layered data-flow analysis and program slicing.

References

- [1] Balakrishnan, G., Gruian, R., Reps, T., and Teitelbaum, T. CodeSurfer/x86 – a platform for analyzing x86 executables. In *Proc. Int. Conf. on Compiler Construction*, vol. 3443 of *LNCS*, pp. 250 – 254, June 2005.
- [2] Balakrishnan, G. and Reps, T. Analyzing memory accesses in x86 executables. In *Proc. Int. Conf. on Compiler Construction*, vol. 2985 of *LNCS*, pp. 5 – 23, January 2004.
- [3] Balakrishnan, G. and Reps, T. Recovery of variables and heap structure in x86 executables. Technical report, University of Wisconsin, Madison, September 2005.
- [4] Balakrishnan, G., Reps, T., Melski, D., and Teitelbaum, T. WYSINWYX: What You See Is Not What You eXecute. In *VSTTE*, 2005.
- [5] Ball, T. The verified software challenge: A call for a holistic approach to reliability. In *VSTTE*, 2005.
- [6] Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., Rajamani, S. K., and Ustuner, A. Thorough static analysis of device drivers. In *EuroSys*, 2006.
- [7] Ball, T., Majumdar, R., Todd Millstein, T., and Rajamani, S. K. Automatic predicate abstraction of C programs. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation, Snowbird, Utah, USA*, pp. 203 – 213, New York, NY, USA, 2001. ACM Press.
- [8] Ball, T. and Rajamani, S. K. Automatically validating temporal safety properties of interfaces. In *SPIN '01: Proceedings of the 8th international SPIN workshop on Model checking of software, Toronto, Ontario, Canada*, pp. 103 – 122, New York, NY, USA, 2001. Springer-Verlag New York, Inc.
- [9] Beyer, D., Henzinger, T. A., Jhala, R., and Majumdar, R. Checking memory safety with BLAST. In M.Cerioli, editor, *Proceedings of the 8th International Conference on Fundamental Approaches to Software Engineering (FASE 2005), Edinburgh, April 2-10, volume 3442 of Lecture Notes in Computer Science*, vol. 3442 of *LNCS*, pp. 2 – 18, Berlin, Germany, 2005. Springer-Verlag.
- [10] Bozga, M., Iosif, R., and Laknech, Y. Storeless semantics and alias logic. In *PEPM '03: Proceedings of the 2003 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation*, pp. 55 – 65, New York, NY, USA, 2003. ACM Press.
- [11] Breuer, P. T. and Pickin, S. Abstract interpretation meets model checking near the 10^6 LOC mark. In *ENTCS: Proceedings of the Fifth International Workshop on Automatic Verification of Infinite State Systems, AVIS'06, 01 April 2006, Vienna, Austria*, pp. 5 – 11. Elsevier, 2006.

- [12] Burstall, R. M. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence*, pp. 23 – 50, 1972.
- [13] Chandra, S., Godefroid, P., and Palm, C. Software model checking in practice: an industrial case study. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pp. 431 – 441, New York, NY, USA, 2002. ACM Press.
- [14] Chou, A., Yang, J., Chelf, B., Hallem, S., and Engler, D. R. An empirical study of operating system errors. In *Symposium on Operating Systems Principles*, pp. 73 – 88, 2001.
- [15] Ciardo, G., Lüttgen, G., and Siminiceanu, R. Saturation: An efficient iteration strategy for symbolic state-space generation. In *7th Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2001)*, vol. 2031 of *Lecture Notes in Computer Science*, pp. 328 – 342, Genova, Italy, April 2001. Springer-Verlag.
- [16] Clarke, E. M., Emerson, E. A., and Sistla, A. P. Automatic verification of finite state concurrent system using temporal logic specifications: a practical approach. In *POPL '83: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pp. 117 – 126, New York, NY, USA, 1983. ACM Press.
- [17] Clarke, E. M., Grumberg, O., and Long, D. E. Model checking and abstraction. In *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Albuquerque, New Mexico, United States*, pp. 343 – 354, New York, NY, USA, 1992. ACM Press.
- [18] Clarke, E. M., Grumberg, O., and Peled, D. A. *Model checking*. MIT Press, 2000.
- [19] CodeSurfer/x86. <http://www.grammatech.com/research/cs-x86/index.html>.
- [20] Corbet, J., Rubini, A., and Kroah-Hartmann, G. *Linux Device Drivers*. O'Reilly, Sebastopol, CA, USA, 3rd edition, 2005.
- [21] Corbett, J. C., Dwyer, M. B., Hatcliff, J., Laubach, S., Păsăreanu, C. S., Robby, and Zheng, H. Bandera: extracting finite-state models from java source code. In *International Conference on Software Engineering*, pp. 439 – 448, 2000.
- [22] Cousot, P. Abstract interpretation. *Symposium on Models of Programming Languages and Computation, ACM Computing Surveys*, 28(2):324 – 328, June 1996.
- [23] Cousot, P. and Cousot, R. On abstraction in software verification. In Brinksma, E. and Larsen, K., editors, *Proceedings of the 14th International Conference on Computer Aided Verification (CAV 2002), Copenhagen, Denmark*, vol. 2404 of *Lecture Notes in Computer Science*, pp. 37 – 56, Berlin, Germany, July 2002. Springer-Verlag.

- [24] Coverity, Inc. <http://www.coverity.com>.
- [25] Debray, S., Muth, R., and Weippert, M. Alias analysis of executable code. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, San Diego, California, United States*, pp. 12 – 24, New York, NY, USA, 1998. ACM Press.
- [26] Deutsch, A. A storeless model of aliasing and its abstractions using finite representations of right-regular equivalence relations. In *Proceedings of the 1992 International Conference on Computer Languages*, April 1992.
- [27] Deutsch, A. Interprocedural may-alias analysis for pointers: beyond k-limiting. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pp. 230 – 241, New York, NY, USA, 1994. ACM Press.
- [28] Engler, D. R., Chelf, B., Chou, A., and Hallem, S. Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions. In *Proceedings of the Fourth Symposium on Operating System Design and Implementation*, October 2000.
- [29] Engler, D. R. and Musuvathi, M. Static analysis versus software model checking for bug finding. In *Proceedings of the Fifth International Conference on Verification, Model Checking and Abstract Interpretation*, January 2004.
- [30] Gälli, M., Greevy, O., and Nierstrasz, O. Composing unit tests. In *2nd International Workshop on Software Product Line Testing*, pp. 16 – 22, 2006.
- [31] Gälli, M., Nierstrasz, O., and Wuyts, R. Partial ordering unit tests by coverage sets. Technical report iam-03-013, Institut für Informatik und Angewandte Mathematik, University of Bern, Switzerland, 2004.
- [32] Godefroid, P. 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, 1994.
- [33] Godefroid, P. Model checking for programming languages using verisoft. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 174 – 186, New York, NY, USA, 1997. ACM Press.
- [34] Graf, S. and Hassen Saïdi, H. Construction of abstract state graphs with pvs. In *CAV '97: Proceedings of the 9th International Conference on Computer Aided Verification*, pp. 72 – 83, London, UK, 1997. Springer-Verlag.
- [35] Gupta, S. K. and Sharma, N. Alias analysis for intermediate code. In *Proceedings of the First Annual GCC Developers' Summit, May 25 – 27, 2003, Ottawa, Canada, 2003*. Available online at <http://www.linux.org.uk/~ajh/gcc/gccsummit-2003-proceedings.pdf>; visited 3rd November 2005.

- [36] Henzinger, T. A., Jhala, R., Majumdar, R., Necula, G. C., Sutre, G., and Weimer, W. Temporal-safety proofs for systems code. In *CAV '02: Proceedings of the 14th International Conference on Computer Aided Verification*, pp. 526 – 538, London, UK, 2002. Springer-Verlag.
- [37] Henzinger, T. A., Jhala, R., Majumdar, R., and Sutre, G. Lazy abstraction. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, Portland, Oregon*, pp. 58 – 70, New York, NY, USA, 2002. ACM Press.
- [38] Hoare, C. A. R. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576 – 580, 1969.
- [39] Holzmann, G. J. *The SPIN Model Checker*. Addison-Wesley Longman, Amsterdam, 2003.
- [40] Hong, H. S., Lee, I., Sokolsky, O., and Ural, H. A temporal logic based theory of test coverage and generation. In *TACAS '02: Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 327 – 341, London, UK, 2002. Springer-Verlag.
- [41] Horspool, R. N. and Marovac, N. An approach to the problem of detranslation of computer programs. *Computer Journal*, 23(3):223 – 229, 1980.
- [42] Kirchner, F. Store-based operational semantics. In *Seizièmes Journées Franco-phones des Langages Applicatifs*. INRIA, 2005.
- [43] Solaris modular debugger guide, chapter 7: Kernel execution control. <http://docs.sun.com/app/docs/doc/816-5041>.
- [44] Kuncak, V. and Rinard, M. On spatial conjunction as second-order logic. Technical report, Computer Science and AI Lab, MIT, Cambridge, USA, 2004.
- [45] Mehta, F. and Nipkow, T. Proving pointer programs in higher-order logic. *Inf. Comput.*, 199(1-2):200 – 227, 2005.
- [46] Memon, A. M., Pollack, M. E., and Soffa, M. L. Hierarchical gui test case generation using automated planning. *IEEE Trans. Softw. Eng.*, 27(2):144 – 155, 2001.
- [47] Microsoft Corporation. Static driver verifier: Finding bugs in device drivers at compile-time. Website, April 2004. Available online at <http://www.microsoft.com/whdc/devtools/tools/SDV.aspx>; [3 November 2005].
- [48] Necula, G. C., McPeak, S., and Weimer, W. CCured: type-safe retrofitting of legacy code. In *Symposium on Principles of Programming Languages*, pp. 128 – 139, 2002.

- [49] Novell Linux kernel debugger (NLKD). <http://forge.novell.com/modules/xfmod/project/?nlkd>.
- [50] Perens, B. Electric fence. <http://perens.com/FreeSoftware/ElectricFence/>.
- [51] Prenninger, W. and Pretschner, A. Abstractions for model-based testing. In *TACoS: International Workshop on Test and Analysis of Component Based Systems*, 2004.
- [52] IBM Rational Purify. <http://www-306.ibm.com/software/awdtools/purify/>.
- [53] Reynolds, J. C. Intuitionistic reasoning about shared mutable data structure. In *Proceedings of the Symposium in Celebration of the Work of C.A.R. Hoare*, 2000.
- [54] Reynolds, J. C. Separation logic: A logic for shared mutable data structures. In *LICS '02: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pp. 55 – 74, Washington, DC, USA, 2002. IEEE Computer Society.
- [55] Rinetzky, N., Bauer, J., Reps, T., Sagiv, M., and Wilhelm, R. A semantics for procedure local heaps and its abstractions. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 296 – 309, New York, NY, USA, 2005. ACM Press.
- [56] Sagiv, M., Reps, T., and Wilhelm, R. Solving shape-analysis problems in languages with destructive updating. *ACM Trans. Program. Lang. Syst.*, 20(1):1–50, 1998.
- [57] Swift, M., Bershad, B., and Levy, H. Improving the reliability of commodity operating systems. In *SOSP*, 2003.
- [58] Valgrind – debugging and profiling Linux programs. <http://valgrind.org/>.
- [59] van Emmerik, M. J. Type inference based decompilation. Phd confirmation report, School of Information Technology and Electrical Engineering, University of Queensland, Australia, February 2003.
- [60] Venet, A. Automatic analysis of pointer aliasing for untyped programs. *Sci. Comput. Program.*, 35(2-3):223 – 248, 1999.
- [61] Wahab, M. Object Code Verification. Dissertation, University of Warwick, UK, December 1998.
- [62] Weiser, M. Program slicing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pp. 439 – 449, Piscataway, NJ, USA, 1981. IEEE Press.
- [63] Wilhelm, R., Sagiv, M., and Reps, T. Shape analysis. In *International Conference on Compiler Construction*, number 1781 in LNCS, pp. 1 – 16. Springer Verlag, 2000.

- [64] Wilson, R. P. and Lam, M. S. Efficient context-sensitive pointer analysis for c programs. In *PLDI '95: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation, La Jolla, CA, USA*, pp. 1 – 12, New York, NY, USA, 1995. ACM Press.
- [65] Xu, Z., Miller, B. P., and Reps, T. Safety checking of machine code. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pp. 70 – 82, New York, NY, USA, 2000. ACM Press.

A Mühlberg and Lüttgen: BLASTing *Linux Code*