Verifying Compiled File System Code*

Jan Tobias Mühlberg and Gerald Lüttgen

Software Technologies Research Group, The University of Bamberg, Germany

Abstract. This article presents a case study on retrospective verification of the Linux Virtual File System (VFS), which is aimed at checking violations of API usage rules and memory properties. Since VFS maintains dynamic data structures and is written in a mixture of C and inlined assembly, modern software model checkers cannot be applied. Our case study centres around our novel automated software verification tool, the SOCA Verifier, which symbolically executes and analyses compiled code. We describe how this verifier deals with complex features such as memory access, pointer aliasing and computed jumps in the VFS implementation, while reducing manual modelling to a minimum. Our results show that the SOCA Verifier is capable of analysing the complex Linux VFS implementation reliably and efficiently, thereby going beyond traditional testing tools and into niches that current software model checkers do not reach. This testifies to the SOCA Verifier's suitability as an effective and efficient bug-finding tool during the development of operating system components.

Keywords: model checking; symbolic execution; case study; Linux virtual file system; object code analysis

1. Introduction

In the context of the grand challenge proposed to the program verification community by Hoare [Hoa03], a mini challenge of building a verifiable *file system* (FS) as a stepping stone was presented by Joshi and Holzmann [JH07]. As FSs are vital components of operating system kernels, bugs in their code can have disastrous consequences. Unhandled failures may render all application-level programs unsafe and gives way to serious security problems.

This article applies an analytical approach to verifying an implementation of the *Virtual File System* (VFS) layer [BC05] within the Linux operating system kernel, using our novel, automated *Symbolic Object-Code Analysis* (SOCA) technique. The VFS layer is of particular interest since it lends support for implementing concrete FSs such as EXT3 and ReiserFS [BC05], and encapsulates the details on top of which C POSIX libraries are defined; such libraries in turn provide functions, e.g., *open* and *remove*, which facilitate file access. Our case study aims at checking for violations of API usage rules and memory properties within the Linux VFS implementation, and equally at assessing the feasibility of our SOCA technique to analysing

---

Correspondence and offprint requests to: Software Technologies Research Group, Wilhelmsplatz 3, The University of Bamberg, 96047 Bamberg, Germany. {jan-tobias.muehlberg,gerald.luetzgen}@swt-bamberg.de

* An extended abstract of this article has appeared in the proceedings of SBMF 2009: “Formal Methods: Foundations and Applications”, volume 5902 of Lecture Notes in Computer Science, pages 306–320, Springer, 2009.

intricate operating system components such as the VFS reliably. 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 the Linux VFS implementation is possible. Doing so is in the spirit of Joshi and Holzmann’s mini challenge as “researchers could choose any of several existing open-source filesystems and attempt to verify them” [JH07].

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 [HJM⁺02] and CBMC [CKL04]. Thus, previous work focused on the question whether and how an appropriate model of the VFS can be reversely engineered from its implementation, and whether meaningful verification results can be obtained using model checking on the extracted model [GLMS09]. 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 [GLMS09] 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 [CJMS06]. Our SOCA technique addresses these shortcomings by providing automated verification support that does away with manual modelling and ad-hoc pointer analysis.

The contributions of this article are fourfold. Firstly, we summarise our *SOCA technique* for automated analysis of compiled programs by means of bounded symbolic execution, using the SMT solver *Yices* [DdM06] as execution and verification engine for our SOCA Verifier that prototypically implements the technique [ML10, Müh09]. Analysing the object code enables us to bypass limitations of software model checkers with respect to the accepted input language, so that analysing code sections written in inline assembly does not present a barrier for us. Our technique is especially designed for software employing complex heap-allocated data structures and provides full counterexample paths for each bug found. While generating counterexamples is often impossible for static analysis techniques due to precision loss in join and widening operations [GR06], traditional software model checking [Hol03] requires the manual construction of models or the use of techniques such as predicate abstraction [BMMR01] which do not work well in the presence of heap-allocated data structures. Hence, *symbolic execution* [Kin76] is our method of choice over static analysis and model checking for the Linux VFS case study presented in this article. Despite only employing *path-sensitive and heap-aware slicing*, the SOCA technique scales well for the Linux VFS and provides reliable results efficiently. Moreover, manual modelling efforts are reduced to a minimum, namely to the abstract modelling of a program’s execution context that specifies input and initial heap content.

The article’s second contribution lies in demonstrating how verification properties involving pointers and locks can be expressed at object-code level, 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 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.

Our third contribution is the formal verification of a group of commonly used VFS functions, namely those for mounting FSs and creating, removing and administrating files and directories. By applying symbolic execution and leaving the parameters of these functions as unspecified as possible, our analysis covers also low-probability scenarios. In particular, we look for program locations where pointers holding invalid values may be dereferenced or where the violation of API usage rules may cause the VFS to deadlock. Our 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.

Our last, but not least, contribution is in providing evidence for the effectiveness and reliability of the SOCA technique by conducting an “archaeological” case study on the Linux VFS. 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. Thus, adding automated verification to the toolset of kernel software developers promises to improve the quality assurance process for operating system kernels significantly.

Figure 1. VFS environment and data structures, where arcs denote pointers.

The remainder of this article is organised as follows. In Sec. 2 we introduce the Linux Virtual File System and mention challenges that automated verification techniques have to face when being applied to the VFS. Our SOCA technique is outlined in Sec. 3 to the extent required for understanding this article’s VFS case study; a full account of SOCA is given in [Müh09, ML10]. In Sec. 4 we introduce our verification properties and the modelling of a symbolic execution environment for our case study in detail. The actual application of our prototypical SOCA Verifier to five important functions from Linux’ VFS implementation is described in Sec. 5, together with an account of the results. In Sec. 6 we present the “archaeological” case study in which the SOCA Verifier is applied to historical releases of the VFS. Related work is discussed in Sec. 7, and Sec. 8 presents our conclusions and directions for future work.

2. The Linux Virtual File System

This section introduces the Linux FS architecture and, in particular, the *Virtual File System* (VFS) layer; the reader is referred to [BC05, CRKH05] for a more detailed description. An overview of the VFS internals and data structures is presented in Fig. 1.

The Linux FS architecture consists of multiple layers. The most abstract layer is the *application* layer which refers to the user programs; this is shown as “process” in Fig. 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()`, as well as administrative functions such as ‘mount FS’ `mount()`, ‘unmount FS’ `umount()`, and the `stat()` function which returns a file’s status. 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 [CRKH05]. 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 inconsistent state, effectively crashing the operating system.

Each *specific file system*, such as EXT3 and ReiserFS [BC05], then implements the processing that supports 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* that realise access control for physical media.

The most relevant data structures in the VFS are *dentries*, *inodes* and *super blocks*. As shown in Fig. 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 usually created when a FS is mounted and during path-lookups, i.e., when access to a particular file is requested by a process at the application layer. The request is passed to the specific FS implementation, which will search for the requested file object on the physical medium and returns information on the directory hierarchy leading to the file. The VFS caches this information in dentry objects. 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. When a FS is unmounted, the dentry objects related to files on the FS are destroyed. The full definition of the *dentry* structure resides in `include/linux/dcache.h` of the Linux source hierarchy which can be obtained from <http://www.kernel.org/>.

Information specific to one file, e.g., whether it is a regular file, directory or device, or the file’s location on a physical medium, is stored in *inode* data structures. The *inode* structure also includes a link to the relevant *super block*, backward links to the *dentries* referencing the inode, file permissions, file type, file size, function pointers referencing operations for use on inodes by the VFS, call-backs to the underlying specific FS, device-specific information, and information about how the file is memory-mapped, e.g., how it links to file objects that capture the data needed to support file descriptors in user space. The struct *inode* is defined in `include/linux/fs.h`.

Last but not least, the *super_block* data structure works as an index to the FS, describing the FS’s abstract properties, such as its type (e.g., EXT3 [BC05]), the physical device on which it resides, its total size, its mount point, and a pointer to the root dentry. The struct *super_block* is defined in `include/linux/fs.h`.

In our initial attempt to verify the VFS [GLMS09], 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 standard model checking [CJMS06, Hol03]. Hence, much effort was put into discovering relations between the different data structures employed by the VFS. The focus of this article 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 symbolically executed and analysed by an SMT solver, thus fully automating verification.

3. The SOCA Technique

One of the disadvantages of today’s software model checking tools such as BLAST [HJM⁺02] and SLAM/SDV [BR01] results from their restriction to analysing *source code*. They 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 in from libraries and the use of inlined assembly are common examples. In addition, many errors, especially in operating system components, arise because of platform-specific and compiler-specific details such as byte alignment in memory and registers, memory layout, padding between structure fields and offsets [BRMT08]. Thus, software model checkers including BLAST and SLAM/SDV assume either that the program under consideration “does not have wild pointers” [BBC⁺06] or, as is shown in [ML06], perform poorly when analysing such software.

Figure 2. Illustration of the SOCA technique: *I* – Object code input; *II* – Intermediate representation and instrumentation; *III* – Depth-first construction of paths; *IV* – Generation of constraint systems; *V* – Checking satisfiability of constraint systems; *S* – Path-sensitive and heap-aware program slicing.

3.1. Outline of SOCA

In this article we employ 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. 2. As is shown in the illustration, we automatically translate a program given in its *object code* (presented as an assembly listing, step *I*) into an *intermediate representation* (IR, step *II*) which is borrowed from the Valgrind binary instrumentation framework [NS07]. This translation is conducted by iteratively following each program path (step *III*) 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 (step *IV*), which reflect the path-relevant register and heap contents of the program under analysis. We then employ the *Yices* SMT solver [DdM06] to check the satisfiability of the resulting constraint systems and thus the validity of a path (step *V*). This approach also allows us to add a range of pointer safety properties, e.g., whether a pointer points to an allocated address, as simple assertions on those constraint systems. More complex properties such as preconditions for functions can be expressed by instrumenting the program. In the context of the case study presented here, we use instrumentation to check locking properties. These involve the state of heap cells that are referenced by pointers which are passed as parameters to the respective functions of the Linux kernel’s Locking APIs. Our instrumentations are also performed on the IR, and therefore access to the source code of a program unit under verification is not required.

In contrast to other methods of software verification, our technique does not employ program abstraction but only *path-sensitive and heap-aware program slicing* (step *S* in Fig. 2) inspired by [BR06, FMC⁺07, JM05]. This means that our slices are not computed over the entire program but only over a particular path during

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

Figure 3. Intel assembly instruction and its respective IR statements (types omitted).

execution. Furthermore, we do not consider the heap as one big data object; instead we 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. 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 for the Linux VFS implementation reported in Sec. 5 will show that this boundedness is not a restriction in practice; many functions in operating system kernels are relatively “shallow” and may still be analysed either exhaustively or up to an acceptable depth. In the following, we explain some of the components of the SOCA technique in more detail. We refer the reader to [Müh09, ML10] for a full account of SOCA.

3.2. Valgrind’s IR language

Valgrind’s IR language is a typed assembly language in static-single-assignment form [CFR⁺91, LG99]. It uses *temporary registers* and some memory for storing the *guest state*. The guest state comprises of the contents of the registers that are available in the architecture for which the program under analysis is compiled. The IR language 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, the statements in that block are executed sequentially until an *exit* statement is reached. An exit is always denoted as `goto <target>`, where `<target>` is either a constant or a temporary register that determines the next program location to be executed. Guarded jumps are written as `if (<condition>) goto <target>`, where `<condition>` is a temporary register of type boolean, which has previously been assigned within the basic block.

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 and to the guest state are made explicit. While normalising a program by transforming it into its IR increases the number of instructions, it reduces the complexity of the program’s representation because IR instructions are relatively simple and side-effect free. An example for an assembly statement and its respective IR statements is given in Fig. 3. The figure shows how the `xor` statement is decomposed into explicitly loading (`GET`) the source register 0 into the temporary registers `t8` and `t9`, performing the `xor` operation into the temporary register `t7`, followed by storing (`PUT`) the result back to the guest state. All operands used in this example are 4 bytes, or 32 bits, in size.

3.3. From IR to bit-vector constraints

Having a sequence of instructions decoded in the above way enables us to generate a bit-vector constraint system for that sequence. An example with respect to the above IR instructions is given in Fig. 4. It illustrates how the `GET` instruction can be implemented in Yices as the concatenation (`bv-concat`) of byte-aligned CPU registers from the guest state to word-aligned temporary registers. In the example we write `t9::(bitvector 32)` to declare `t9` a bit-vector variable of length 32. The parameter of the `GET:I32` instruction, i.e., CPU registers 0 to 3, are denoted as `r0` to `r3` in the constraint representation. The `PUT` instruction is handled as a bit-vector extraction `bv-extract <end> <start> <source>`; here, `<start>` and `<end>` define the range of bits from `<source>` that is to be copied to the target bit-vector. Translating IR statements that operate

IR Instruction	Constraint Representation
t9 = GET:I32(0)	(define t9::(bitvector 32) (bv-concat (bv-concat r3 r2) (bv-concat r1 r0)))
t8 = GET:I32(0)	(define t8::(bitvector 32) (bv-concat (bv-concat r3 r2) (bv-concat r1 r0)))
t7 = Xor32(t9,t8)	(define t7::(bitvector 32) (bv-xor t9 t8))
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))

Figure 4. IR statements from Fig. 3 and their constraint representations in Yices.

on temporary registers only, such as the `Xor32` in Fig. 4, is straightforward since Yices provides support for a broad variety of arithmetical and logical operations on bit-vectors.

Note that CPU registers are assigned in “reverse byte order” to the temporary registers, i.e., the least significant 8 bits are in `r0`, and the most significant bits are in `r3`. This is because the above constraints are generated from a binary compiled for Intel 32-bit CPUs (IA32), 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 encodings here.

Furthermore, the IR is in static-single-assignment form only with respect to the temporary registers within a single IR block. Therefore, we have to be careful when generating variable names for Yices: we simply append the instruction’s location and the invocation number to each variable. Finally, since our analysis handles loops by unrolling them while exploring a path, a single instruction might appear multiple times in the path.

3.4. Heap-aware program slicing

Most difficulties in program analysis arise from the need to analyse accesses to a program’s heap and stack. Valgrind’s IR language provides two instructions, `LD` and `ST`, for loading and storing values from and to memory, respectively. While these instructions are in principle as easily mapped to constraints as the above `GET` and `PUT` instructions, handling them in the analysis phase requires care: including the entire 32-bit address space of a program into the constraint system and performing access operations on pointer variables that hold potentially symbolic address values quickly becomes infeasible.

Our SOCA technique tackles this scalability problem by employing *heap-aware program slicing*: for each pointer used along a program’s execution path we compute its potential target address range. When checking a property regarding some value obtained by dereferencing a particular pointer `p`, we only add those store instructions and their dependents to the constraint system that may have updated the value pointed to by `p`. The slicing mechanism used here is inspired by the algorithms presented in [BR06, FMC⁺07, JM05]. Our adaptation focuses on computing dynamic slices over a given program path, rather than static slices over a whole program.

4. 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 that are relevant for the Linux VFS implementation [BC05, CRKH05].

```

01 /* include/asm-i386/current.h */
02
03 extern struct task_struct
04     *vfs_analysis_current (void);
05 #define current vfs_analysis_current()
06 /* fs/namei.c */
07
08 #ifdef current
09 #undef current
10 struct task_struct
11     *vfs_analysis_current (void);
12 #define current vfs_analysis_current()
13 #endif
14 static struct task_struct vfs_current;
15 static struct fs_struct  vfs_fsstruct;
16 static struct dentry     dentries[15];
17 static struct inode      inodes[15];
18 static struct dentry
19     *root = &dentries[0], *pwd = &dentries[1],
20     *altroot = &dentries[2];
19 static struct vfsmount  rootmnt, pwdmnt,
20     altrootmnt;
21 static struct super_block sb;
22 void vfs_analysis_current_init (void)
23 {
24     int i;
25
26     for(i = 0; i < 15; i++)
27     {
28         dentries[i].d_inode = &inodes[i];
29         inodes[i].i_op = NULL;
30         inodes[i].i_sb = &sb;
31     }
32
33     rootmnt.mnt_mountpoint = root;
34     rootmnt.mnt_root       = root;
35     rootmnt.mnt_sb         = &sb;
36
37     pwdmnt.mnt_mountpoint  = root;
38     pwdmnt.mnt_root       = root;
39     pwdmnt.mnt_sb         = &sb;
40
41     altrootmnt.mnt_mountpoint = root;
42     altrootmnt.mnt_root      = root;
43     altrootmnt.mnt_sb       = &sb;
44
45     vfs_fsstruct.root       = root;
46     vfs_fsstruct.pwd       = pwd;
47     vfs_fsstruct.altroot    = altroot;
48     vfs_fsstruct.rootmnt    = &rootmnt;
49     vfs_fsstruct.pwdmnt     = &pwdmnt;
50     vfs_fsstruct.altrootmnt = &altrootmnt;
51
52     vfs_current.fs = &vfs_fsstruct;
53
54     return;
55 }
56
57 void soca_ERROR (void)
58 { return; }
59
60 extern struct task_struct
61     *vfs_analysis_current (void);
62
63 struct task_struct
64     *vfs_analysis_current (void)
65 {
66     return (&vfs_current);
67 }

```

Figure 5. An execution environment for the VFS.

4.1. Modelling the environment

One problem for program verification arises when program functions make use of an external data environment, i.e., dereference pointers to data structures that are not created by the function under analysis. This is particularly common in case of the VFS since 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 that contain function pointers for accessing the specific FS underneath the VFS layer. In general, 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 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-positive error reports 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 Linux kernel’s C source code as shown in Fig. 5. We assign a local `task_struct` (line 14; also cf. `include/linux/sched.h` in the Linux source hierarchy) which defines the logical context for a symbolically executing kernel process. This logical context includes the working directory and a list of 15 dentries and inodes describing the FS’s mount point and a simple directory hierarchy (lines 16 and 17). The data objects are partially initialised by a handcrafted function `vfs_analysis_current_init()` (lines 22 ff.) which is used as a preamble in our analysis process. We further define a function `soca_ERROR()`

(lines 57 and 58) which plays the role of an “error label” for program instrumentation as will be explained in Sec. 4.3. 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. It is worth mentioning that our modelling of the external environment of Fig. 5 was conducted iteratively, by successively adding details to the initial memory state while carefully avoiding to be over-restrictive: we only reduced the number of false-positive warnings by eliminating impossible initial memory states, while *not* restricting possible behaviour.

4.2. Pointer safety properties

We now turn our attention to pointer safety properties which we will verify for the Linux VFS implementation using our SOCA technique. The SOCA Verifier analyses binary programs in the ELF format [Too95], which is widely used in Unix-like operating systems. The standard [Too95] specifies a program binary file consisting of various sections to which a pointer may point. Important sections are the `.text` section, which contains the actual object code of the program, and the `.data` section, which contains global data objects. Obviously, the program binary also has other sections such as the symbol table, or static data which are, however, invalid as jump targets.

For the purpose of our case study, we are interested in checking three basic safety properties for every pointer that is dereferenced 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 (either a call, a return or a computed jump), it may only point inside the `.text` section of the kernel binary.

A check of the above properties on the IR is performed by computing an over-approximation of the address range to which the considered pointer may point, i.e., the pointer may address any memory cell in the range allowed 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.

4.3. 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 data structures shared between concurrently running kernel threads. The Locking APIs of Linux that are 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 much attention in the automated verification community [BR01, HJM⁺02, WBKW07, XA07]. For software systems like the Linux kernel with its fine grained locking approach, conducting these checks is non-trivial. Due to the vast number of locks employed within the VFS, locks are passed by reference to the Locking API’s functions, which involves additional pointer safety issues. The data structures implementing a lock are dynamically allocated and deallocated, and have to be explicitly initialised. Furthermore, these data structures are more complex than one would naively expect. They involve additional information regarding interrupt handling, the process and CPU holding the lock, and in some cases even a list of waiting processes. The Linux kernel provides several functions for obtaining a lock with respect to this additional information. Most of these functions are implemented in inlined assembly. A precise analysis of pointer aliasing relationships, which is known to be undecidable in general, would be required to prove programs to be free of locking errors.

Table 1. Experimental Results I: Code statistics by VFS function analysed

	creat	unlink	mkdir	rmdir	rename	stat	mount	umount	totals
no. of instructions	3602	3143	3907	3419	4929	2908	4018	3271	29197
lines in source code	1.4k	1.2k	1.6k	1.4k	2k	1.2k	1.8k	1.3k	11.9k
no. of paths	279	149	212	318	431	73	371	433	2266
min. path length	91	41	87	72	72	54	58	54	41
max. path length	4138	3218	5319	3017	5910	2418	7038	8019	8019
pointer operations	2537	2190	2671	2466	4387	2071	5817	8393	30532
concrete	2356	2134	2458	2368	3989	2012	5533	8349	29199
symbolic	181	56	213	98	398	59	284	44	1333
locking operations	287	231	391	319	451	212	411	415	2717

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, and an error message for the path is raised if the error location is reachable.

5. Applying the SOCA Verifier to the Linux VFS

To conduct our case study of verifying the Linux VFS implementation, we use a prototypic implementation of our SOCA technique, called the SOCA Verifier. It is written in C, mainly to facilitate integration with the Valgrind VEX library [NS07]. For applying it to the VFS, we used the VFS implementation of version 2.6.18.8¹ 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.

5.1. Statistics and performance

Our experimental results are summarised in three tables. Table 1 provides a statistical overview of the VFS code. We report the total *number of machine instructions* that have been translated into IR by following each VFS 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, header files, macro definitions and comments). The next values in the table present the numbers of explored paths and, respectively, the lengths of the shortest and longest explored paths, in numbers of instructions. The *pointer* and *locking operations* resemble the numbers of concrete and symbolic pointer dereferences and lock/unlock operations encountered along the analysed paths, respectively.

In Table 2 we report the performance of the SOCA Verifier, showing the total time needed for analysing the VFS functions and our tool’s maximum memory consumption. As can be seen, the verification times are in the order of 1.5 to 4 hours. This is much less time than a typical kernel developer spends on manual debugging, which is the only possibility given the lack of tools checking for kernel software. On modern multi-core PCs, the execution time of the SOCA Verifier can be further reduced by running several Yices sessions in parallel.

The maximum memory consumption listed in Table 2 is an estimate generated by simply summing up our tool’s and Yices’ maximum memory usages as given in Table 3; however, in practice these may not necessarily hit their peak memory at the same time. Most importantly, the peak memory consumption never exceeds

¹ Linux 2.6.18.8 is available at <http://git.kernel.org/?p=linux/kernel/git/stable/linux-2.6.18.y.git>.

Table 2. Experimental Results II: SOCA Verifier statistics

	creat	unlink	mkdir	rmdir	rename	stat	mount	umount	totals
total time	2h27m	1h22m	2h42m	1h34m	3h45m	57m	3h12m	2h57m	18h56m
max. memory (SOCA)	1.03G	752M	1.15G	743M	1.41G	798M	1.60G	1.41G	1.60G
max. mem. (SOCA + Yices)	1.79G	800M	1.92G	791M	2.18G	846M	2.85G	2.17G	2.85G
exec. bound exhausted	yes	yes	yes	yes	yes	no	yes	yes	yes
path bound exhausted	no	no	no	no	no	no	no	no	no
paths reaching end	154	112	165	215	182	62	300	152	1342
assertions checked	13.4k	12.4k	15.8k	11.8k	21.9k	11.7k	29.7k	52.1k	168.8k
ratio of failed checks	0.043	0.012	0.041	0.019	0.049	0.011	0.021	0.014	0.026

Table 3. Experimental Results III: Yices statistics

	creat	unlink	mkdir	rmdir	rename	stat	mount	umount	totals
total Yices calls	27533	21067	31057	20988	44439	19878	46219	90612	101.8k
total time spent in Yices	2h22m	1h11m	2h22m	1h24m	3h8m	49m	2h57m	2h44m	16h57m
average time	311ms	192ms	271ms	198ms	376ms	149ms	230ms	109ms	230ms
standard deviation	3.7s	0.9s	5.2s	1.4s	5.9s	0.9s	3.1s	5.1s	3.3s
max CS size in vars	450k	97k	450k	95k	450k	94k	530k	450k	530k
average CS size in vars	2844	2871	2871	2862	2939	2868	2896	2764	2864
standard deviation	14619	8948	14618	8898	16052	8917	22653	14653	13669
max. memory consumption	766M	48M	766M	48M	766M	48M	1258M	766M	1258M

2.85 GBytes, which is a memory capacity that is typically available in today’s PCs. The next two rows of Table 2 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 ratio of failed checks. Paths not reaching a return statement in the target function are terminated either due to bound exhaustion or a property being violated that does not permit continuation of that path.

Finally, we outline in Table 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 coverage and condition coverage of 60% to 80% in this case study. The main reason for this seemingly low percentage is that VFS functions often implement multiple 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 to quality assurance.

The above tables only give a glimpse of the total scale of experiments that we conducted for this case study. Depending on how detailed or coarse the execution environment was 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 was the total number of “symbolic” pointers; a symbolic pointer is a pointer whose exact value cannot be determined at the point at which it is dereferenced. 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. The impact of these symbolic pointers becomes apparent when comparing our analysis results for the functions `mount()` and `unmount()`: for `unmount()` the SOCA Verifier checked almost twice as many assertions in roughly the same time as for `mount()`. This is because our analysis encountered a total of 44 symbolic pointers in `unmount()`, but 284 symbolic pointers in `mount()`, rendering the constraint systems generated for `unmount()` smaller than those generated for `mount()`.

As can be seen, symbolic values are bad for the performance of the SOCA technique and, in general, for any other automated formal analysis method since search spaces are increased. However, symbolic values are important for driving the analysis into paths that may be hard to reach in testing-based approaches to system verification.

Table 4. Experimental Results IV: Evaluating the effectiveness of SOCA

#	Commit Key	Type	No. of Functions	Results		Unrelated Traces
				<i>before</i>	<i>after</i> patch	
1	322ee5b36eac42e762526b0df7fa432beba6e7a0	B	1	+/-	-	2
2	745ca2475a6ac596e3d8d37c2759c0fbe2586227	B	1	-/-	-	1
3	acd0c935178649f72c44ec49ca83bee35ce1f79e	B	2	+/-	-	1
4	ad775f5a8faa5845377f093ca11caf577404add9	B	2	-/-	-	3
5	d0185c0882d76b8126d4a099c7ac82b3b216d103	B	2	+/-	-	1
6	e0e817392b9acf2c98d3be80c233dddb1b52003d	B	1	+/-	-	2
7	2a737871108de9ba8930f7650d549f1383767f8b	BF	5	+/-, 0/0	-	1
8	4ea3ada2955e4519befa98ff55dd62d6dfbd1705	BP	3	+/-, 2/1	-	0
9	520c85346666d4d9a6fcaaa8450542302dc28b91	BP	2	+/-, 0/2	-	3
10	6c5daf012c9155aafd2c7973e4278766c30dfad0	BP	2	+/-, 0/2	-	1
11	7ed7fe5e82c9fc8473974fbd7389d169b8f17c77	BP	1	+/-, 0/0	-	2
12	08ce5f16ee466ffc5bf243800deeccd77d9eaf50	F	2	0/0	-	2
13	22d2b35b200f76085c16a2e14ca30b58510fcbe7	F	1	does not compile	-	
14	4a19542e5f694cd408a32c3d9dc593ba9366e2d7	F	1	does not compile	-	
15	73241ccca0f7786933f1d31b3d86f2456549953a	FP	5	does not compile	-	
16	2f38d70fb4e97e7d00e12eaac45790cf6ebd7b22	P	4	0/0	-	0
17	acb0c854fa9483fa85e377b9f342352ea814a580	P	3	0/0	-	2
18	acfa4380efe77e290d3a96b11cd4c9f24f4fbb18	P	4	0/3	-	0
19	cb59861f03a626196a23fdef5e20ddbb8cca6466	P	1	0/1	-	0
20	cdb70f3f74b31576cc4d707a3d3b00d159cab8bb	P	1	0/1	-	2
21	6ea36ddbd1abfe867f1e874a8312bfd811e5fd2c	P	1	does not compile	-	
22	214fda1f6e1b8ef2a5292b0372744037fc80d318	P	2	does not compile	-	
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.

5.2. Errors and false positives

All errors detected by the SOCA Verifier can actually be reproduced in the code, provided that other kernel components match the behaviour of the employed model of the execution environment. This is because our verification technique does, by its definition, not include infeasible paths.

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. Obviously, we could have made our model of the execution environment more precise by iteratively excluding these false-positives. However, due to the complex nature of the linked data structures passed to the VFS functions, the refined environment model would have become as unwieldy as the actual environment. We think that the environment model chosen by us provide a good compromise between modelling effort and false-positive rate.

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.

6. Evaluating the Effectiveness of SOCA

With the goal of further evaluating the effectiveness of SOCA as a bug-finding tool, we conducted a second case study which applied 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 almost 20 years, the Linux kernel is an ideal candidate for the sort of “archaeological” study presented here. The question that 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?

6.1. 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 repository² 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 implementation which involves 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 changed data structures and function interfaces. An overview of our sample is given in Table 4. The *commit keys* given in the table are references to Linux’s source code repository.

6.2. Case study setup

To conduct our case study we compiled two versions of the Linux kernel for each of the 23 patches. More precisely, we compiled 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 was the kernel source configured for the IA32 architecture using the default configuration shipped with the kernel sources. The SOCA Verifier was 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, SOCA was applied here without modelling an execution environment for the functions checked. The modelling step was 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 were set to the values used in the previous case study. The error traces reported by our tool were manually checked for validity, i.e., whether the expected error had been found or whether traces not related to the subject matter of the patch or false-positive traces were reported.

A detailed account of the results of the case study is given in Table 4. When analysing a patch that is supposed to *fix a bug*, we expected 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 $\neg_-$ in the *Results* column of Table 4. With $\neg_+$ and $\neg_+$ 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 $\neg_-$ if no error was detected at all. For patches introducing new features or implementing performance improvements we expected the pre-patched kernel and the patched kernel to be free of errors and denote that with $\%_0$ in Table 4. If SOCA issues false-positive errors for these cases, we write $\frac{pre}{post}$, 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, were produced by the SOCA Verifier, we give the number of those reported error traces in column *Unrelated Traces*. Table 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.

6.3. 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

² The source repository of Linux 2.6 is available at <http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git>.

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 6 of the 11 performance improvements and feature introductions as safe (54.5%), 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 potential error trace.

Initially we expected the lack of an execution environment to result in a high number of potential error traces. During our experiments it turned out that the majority of functions modified between two releases of the Linux kernel are relatively short when compared to the complex functions analysed in our initial case study, and perform memory access only within a narrow scope of addresses. Only in two cases did the analysed functions perform a traversal of the dentry cache. Therefore omitting the modelling step did not have much impact on the number of potential error traces and the performance of the SOCA Verifier.

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 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 had available, the actual verification was conducted within less than 6 hours. These figures show that the SOCA technique can be 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.

7. Related Work

A comprehensive survey on automated techniques for formal software verification can be found in [DKW08], where also approaches to model-checking the source code of operating system components [BR01, CCG⁺04, HJM⁺02] are discussed. In theory, these are able to prove a FS implementation to be, e.g., free of deadlock, by checking the proper use of locking mechanisms. However, modern model checkers such as BLAST [HJM⁺02] require extensive manual preprocessing and are not able to deal with general pointer operations [ML06]. Recent work [Ser09] shows further that, again in contrast to our SOCA Verifier, BLAST cannot analyse programs with multiple locks since its specification language does not permit the specification of observer automata for API safety rules with respect to function parameters.

7.1. Verification of file system implementations

The verification of FS implementations is studied in [BC09, DB09, FO09, GLMS09, KK09, YST⁺06, YTEM04]. In [YTEM04], 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 FS test driver, a permutation checker verifying that a FS can always recover, and a recovery checker using the *fsck* recovery tool. The verification system starts with an empty FS and recursively generates successive states by executing system calls affecting the FS under analysis. After each step, the verification system is interrupted, and *fsck* is used to check whether the FS 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 [KK09] an empirical study applying *concolic testing* [SMA05] to the multi-sector read operation of a flash memory FS 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 contrast 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. [KK09] discusses the advantages and weaknesses of concolic testing on the domain of low-level FS 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 which covers concurrent aspects of flash memory is described in [BC09]. The authors focused on developing a low-level model covering the internal behaviour of *Open NAND* flash devices. They applied the FDR model checker [Ros94] to prove the consistency of this model to a specification of the external interface of the device. While the authors detected several deadlocks and sources of misinterpretation in the models, the analysis could only be completed partially as the specifications proved to be too complex for being analysed with FDR in full.

In [DB09] a model of a flash-based file store developed in Event-B is given. In the paper, the authors centre on discussing their use of refinement in feature augmentation and to extend abstract model components with more design details. The goal of their work is to simplify the process of model construction and to relate an abstract FS model to the flash specification.

Finally, the application of theorem proving techniques to build a formal methods tool chain and apply it to an abstract FS model is presented in [FO09]. 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.

7.2. Verification via symbolic execution

Symbolic execution was introduced by King [Kin76] 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. This is exactly what our SOCA technique does, albeit not for testing but for systematic, powerful memory safety analysis. A recent approach using symbolic execution to derive inputs that make a given program crash has been proposed in [CGP⁺06]. In contrast to our work, [CGP⁺06] 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 been published in recent years, including DART [GKS05] and SATURN [XA07]. 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 language agnostic tool in the spirit of DART is SAGE [GdHN⁺08], 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 this article. 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.

A bounded model checker for C source code based on symbolic execution and SAT solving is SATURN [XA07]. This tool is specialised on checking locking properties and null-pointer dereferences. The authors show that their tool scales for analysing the entire Linux kernel. Unlike the SOCA Verifier, the approach in [XA07] 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 unsound results but scale better than SOCA.

8. Conclusions and Future Work

The initial motivation for our SOCA technique to automated program verification was to explore the possibilities of symbolic execution for analysing compiled programs. 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 showed in this article, 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.

One could argue that, as a drawback of object-code analysis, the verification has to be redone for every target platform. For SOCA we consider this as a minor issue because the actual verification is fully automatic. Furthermore our second case study demonstrated that SOCA can be applied to finding errors introduced between two releases of a program by checking the modified functions in isolation. Here, verification took only three times longer than compiling the code.

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 into our SOCA implementation so that it scales to complex real-world operating system code such as the Linux VFS implementation. The SOCA Verifier is expected to scale even better for programs employing fewer external data structures than the VFS does; for example, the majority of Linux device drivers including actual FS implementations satisfies this criterion.

Regarding future work, we wish to extend the SOCA Verifier so as to be able to analyse concurrent programs. This would help checking the VFS implementation for erroneous behaviour that is only exhibited when multiple kernel threads interact. In addition, the SOCA Verifier should be integrated into widely used operating software development environments so that counterexamples found in object code can be presented in source code to the developer. Future work should also involve further case studies applying the SOCA Verifier to other FSs, e.g., flash FSs [Hyn08], and other operating systems, e.g., the FreeRTOS kernel [Bar10].

Acknowledgements

We would like to thank the anonymous reviewers of this article and our SBMF 2009 paper on which this article is based, for their valuable comments. We also thank Andy Galloway and Radu I. Siminiceanu for their contributions to our previous work on the Linux VFS, and Jim Woodcock and Daniel Kroening for their insightful comments made during the examination of the first author's PhD thesis.

References

- [Bar10] R. Barry. FreeRTOS: A portable, open source, mini real time kernel, 2010. <http://www.freertos.org/>.
- [BBC⁺06] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. *SIGOPS Operating Systems Review*, 40(4):73–85, 2006.
- [BC05] D. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly, 2005.
- [BC09] A. Butterfield and A. Ó. Catháin. Concurrent models of flash memory device behaviour. In *SBMF '09*, vol. 5902 of *LNCS*, pp. 70–83. Springer, 2009.
- [BMMR01] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. *SIGPLAN Notices*, 36(5):203–213, 2001.
- [BR01] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN '01*, vol. 2057 of *LNCS*, pp. 102–122. Springer, 2001.
- [BR06] G. Balakrishnan and T. Reps. Recency-abstraction for heap-allocated storage. In *SAS '06*, vol. 4134 of *LNCS*, pp. 221–239. Springer, 2006.
- [BRMT08] G. Balakrishnan, T. Reps, D. Melski, and T. Teitelbaum. WYSINWYX: What You See Is Not What You eXecute. In *VSTTE '08*, vol. 4171 of *LNCS*, pp. 202–213. Springer, 2008.
- [CCG⁺04] S. Chaki, E. Clarke, A. Groce, J. Ouaknine, O. Strichman, and K. Yorav. Efficient verification of sequential and concurrent C programs. *FMSD*, 25(2-3):129–166, 2004.
- [CFR⁺91] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. on Programming Languages and Systems*, 13(4):451–490, 1991.
- [CGP⁺06] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically generating inputs of death. In *CCS '06*, pp. 322–335. ACM, 2006.

- [CJMS06] G. Ciardo, R. L. Jones, A. S. Miner, and R. I. Siminiceanu. Logic and stochastic modeling with SMART. *Performance Evaluation*, 63(6):578–608, 2006.
- [CKL04] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *TACAS '04*, vol. 2988 of *LNCS*, pp. 168–176. Springer, 2004.
- [CRKH05] J. Corbet, A. Rubini, and G. Kroah-Hartmann. *Linux Device Drivers*. O'Reilly, 3rd edition, 2005.
- [DB09] K. Damchom and M. Butler. Applying event and machine decomposition to a flash-based filestore in Event-B. In *SBMF '09*, vol. 5902 of *LNCS*, pp. 134–152. Springer, 2009.
- [DdM06] B. Dutertre and L. de Moura. The Yices SMT solver. Technical Report 01/2006, SRI International, 2006. <http://yices.csl.sri.com/tool-paper.pdf>.
- [DKW08] V. D'Silva, D. Kroening, and G. Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 27(7):1165–1178, 2008.
- [FMC⁺07] C. Ferdinand, F. Martin, C. Cullmann, M. Schlickling, I. Stein, S. Thesing, and R. Heckmann. New developments in WCET analysis. In *Program Analysis and Compilation, Theory and Practice*, vol. 4444 of *LNCS*, pp. 12–52. Springer, 2007.
- [FO09] M. A. Ferreira and J. N. Oliveira. An integrated formal methods tool-chain and its application to verifying a file system model. In *SBMF '09*, vol. 5902 of *LNCS*, pp. 153–169. Springer, 2009.
- [GdHN⁺08] P. Godefroid, P. de Halleux, A. V. Nori, S. K. Rajamani, W. Schulte, N. Tillmann, and M. Y. Levin. Automating software testing using program analysis. *IEEE Software*, 25(5):30–37, 2008.
- [GKS05] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *PLDI '05*, pp. 213–223. ACM, 2005.
- [GLMS09] A. Galloway, G. Lüttgen, J. T. Mühlberg, and R. Siminiceanu. Model-checking the Linux Virtual File System. In *VMCAI '09*, vol. 5403 of *LNCS*, pp. 74–88. Springer, 2009.
- [GR06] B. S. Gulavani and S. K. Rajamani. Counterexample driven refinement for abstract interpretation. In *TACAS '06*, vol. 3920 of *LNCS*, pp. 474–488. Springer, 2006.
- [HJM⁺02] T. A. Henzinger, R. Jhala, R. Majumdar, G. C. Necula, G. Sutre, and W. Weimer. Temporal-safety proofs for systems code. In *CAV '02*, vol. 2402 of *LNCS*, pp. 382–399. Springer, 2002.
- [Hoa03] T. Hoare. The verifying compiler: A grand challenge for computing research. *J. ACM*, 50(1):63–69, 2003.
- [Hol03] G. J. Holzmann. *The SPIN Model Checker*. Addison-Wesley Longman, 2003.
- [Hyn08] Hynix Semiconductor et al. Open NAND flash interface specification, revision 2.0. Technical report, ONFI, 2008. <http://www.onfi.org>.
- [JH07] R. Joshi and G. J. Holzmann. A mini challenge: Build a verifiable filesystem. *Formal Aspects of Computing*, 19(2):269–272, 2007.
- [JM05] R. Jhala and R. Majumdar. Path slicing. *SIGPLAN Notices*, 40(6):38–47, 2005.
- [Kin76] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.
- [KK09] M. Kim and Y. Kim. Concolic testing of the multi-sector read operation for flash memory file system. In *SBMF '09*, vol. 5902 of *LNCS*, pp. 251–265. Springer, 2009.
- [LG99] A. Leung and L. George. Static single assignment form for machine code. In *PLDI '99*, pp. 204–214. ACM, 1999.
- [ML06] J. T. Mühlberg and G. Lüttgen. BLASTing Linux code. In *FMICS '06*, vol. 4346 of *LNCS*, pp. 211–226. Springer, 2006.
- [ML10] J. T. Mühlberg and G. Lüttgen. Symbolic object code analysis. Technical Report 85/2010, Faculty of Information Systems and Applied Computer Sciences, The University of Bamberg, 2010.
- [Müh09] J. T. Mühlberg. *Model Checking Pointer Safety in Compiled Programs*. PhD thesis, Department of Computer Science, University of York, 2009.
- [NS07] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Notices*, 42(6):89–100, 2007.
- [Ros94] A. W. Roscoe. Model-checking CSP. In *A classical mind: Essays in honour of C. A. R. Hoare*, pp. 353–378. Prentice Hall, 1994.
- [Ser09] O. Sery. Enhanced property specification and verification in BLAST. In *FASE '09*, vol. 5503 of *LNCS*, pp. 456–469. Springer, 2009.
- [SMA05] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *ESEC/FSE-13*, pp. 263–272. ACM, 2005.
- [Too95] Tool Interface Standard (TIS) Committee. Executable and linking format (ELF) specification version 1.2. Technical report, 1995.
- [WBKW07] T. Witkowski, N. Blanc, D. Kroening, and G. Weissenbacher. Model checking concurrent Linux device drivers. In *ASE '07*, pp. 501–504. ACM, 2007.
- [XA07] Y. Xie and A. Aiken. SATURN: A scalable framework for error detection using boolean satisfiability. *ACM Trans. on Programming Languages and Systems*, 29(3):article 16, 1–43, 2007.
- [YST⁺06] J. Yang, C. Sar, P. Twohey, C. Cadar, and D. R. Engler. Automatically generating malicious disks using symbolic execution. In *Security and Privacy*, pp. 243–257. IEEE, 2006.
- [YTEM04] J. Yang, P. Twohey, D. R. Engler, and M. Musuvathi. Using model checking to find serious file system errors. In *OSDI*, pp. 273–288. USENIX, 2004.