Verifying Compiled File System Code

Jan Tobias Mühlberg and Gerald Lüttgen

Software Engineering and Programming Languages Research Group,
University of Bamberg, 96052 Bamberg, Germany.
{jan-tobias.muehlberg, gerald.luettgen}@swt-bamberg.de

Abstract. This paper presents a case study on retrospective verification of the Linux Virtual File System (VFS), which is aimed at checking for 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 verification tool, the SOCA Verifier, which symbolically executes and analyses compiled code. We describe how this verifier deals with complex program features such as memory access, pointer aliasing and computed jumps, while reducing manual modelling to the bare minimum. Our results show that the SOCA Verifier is capable of reliably analysing complex operating system components such as the Linux VFS, thereby going beyond traditional testing tools and into niches that current software model checkers do not reach.

1 Introduction

In the context of the grand challenge proposed to the program verification community by Hoare [16], a mini challenge of building a verifiable *file system* (FS) as a stepping stone was presented by Joshi and Holzmann [17]. As FSs are vital components of operating system kernels, bugs in their code can have disastrous consequences. Unhandled failure may render all application-level programs unsafe and gives way to serious security problems.

This paper applies an analytical approach to verifying an implementation of the *Virtual File System* (VFS) layer [5] within the Linux operating system kernel, using our novel, automated *Symbolic Object-Code Analysis* (SOCA) technique. As described in Sec. 2, the VFS layer is of particular interest since it provides support for implementing concrete FSs such as EXT3 and ReiserFS [5], and encapsulates the details on top of which C POSIX libraries are defined; such libraries in turn provide functions, e.g., *open* and *remove*, that facilitate file access. Our case study aims at checking for violations of API usage rules and memory properties within VFS, and equally at assessing the feasibility of our SOCA technique to reliably analysing intricate operating system components such as the Linux VFS implementation. We are particularly interested in finding

out to what degree the *automatic* verification of complex properties involving pointer safety and the correct usage of locking APIs within VFS is possible.[1]

Since the Linux VFS implementation consists of more than 65k lines of complex C code including inlined assembly and linked dynamic data structures, its verification is not supported by current software model checkers such as BLAST [15] and CBMC [8]. Thus, previous work by us focused on the question whether and how an appropriate model of the VFS can be reverse engineered from its implementation, and whether meaningful verification results can be obtained using model checking on the extracted model [13]. 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 [13] 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 [7]. Our SOCA technique addresses these shortcomings, providing automated verification support that does away with manual modelling and ad-hoc pointer analysis.

The contributions of this paper are threefold. In Sec. 3 we summarise our *SOCA technique* for automated analysis of compiled programs by means of bounded symbolic execution, using the SMT solver *Yices* [11] as execution and verification engine. 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 represent a barrier for us. Our technique is especially designed for programs employing complex heap-allocated data structures and provides full counterexample paths for each bug found. While generating counterexamples is often impossible for static analysis techniques due to precision loss in join and widening operations [14], traditional model checking requires the manual construction of models or the use of techniques such as predicate abstraction [3] which do not work well in the presence of heap-allocated data structures. Hence, symbolic execution is our method of choice over static analysis and model checking. Despite only employing path-sensitive and heap-aware slicing, the SOCA technique scales well for the Linux VFS. Moreover, manual modelling efforts are reduced to a bare minimum, namely to the abstract specification of a program's execution context that specifies input and initial heap content.

The paper's second contribution lies in demonstrating how verification properties can be expressed for symbolic object-code analysis, for which two different approaches are employed in Sec. 4. Firstly, properties may be presented to the SMT solver as assertions on the program's register contents at each execution point. Alternatively, the program may be instrumented during its symbolic execution, by adding test and branch instructions to its control flow graph. Verify-

[1] Doing so is in the remit of Joshi and Holzmann's mini challenge: "researchers could choose any of several existing open-source filesystems and attempt to verify them" [17].

ing 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 last, but not least, contribution is the formal verification of a group of commonly used VFS functions, namely those for creating and removing files and directories, which we report in Sec. 5. By applying symbolic execution and leaving the parameters of these functions as unspecified as possible, our analysis covers low-probability scenarios. In particular, we look for program points where pointers holding invalid values may be de-referenced or where the violation of API usage rules may cause the VFS to deadlock. The experimental results show that the SOCA technique works well on the Linux VFS and that it produces a relatively low number of false-positive counterexamples while achieving high code coverage. Therefore, the absence of any flagged errors contributes to raising confidence in the correctness of the Linux VFS implementation.

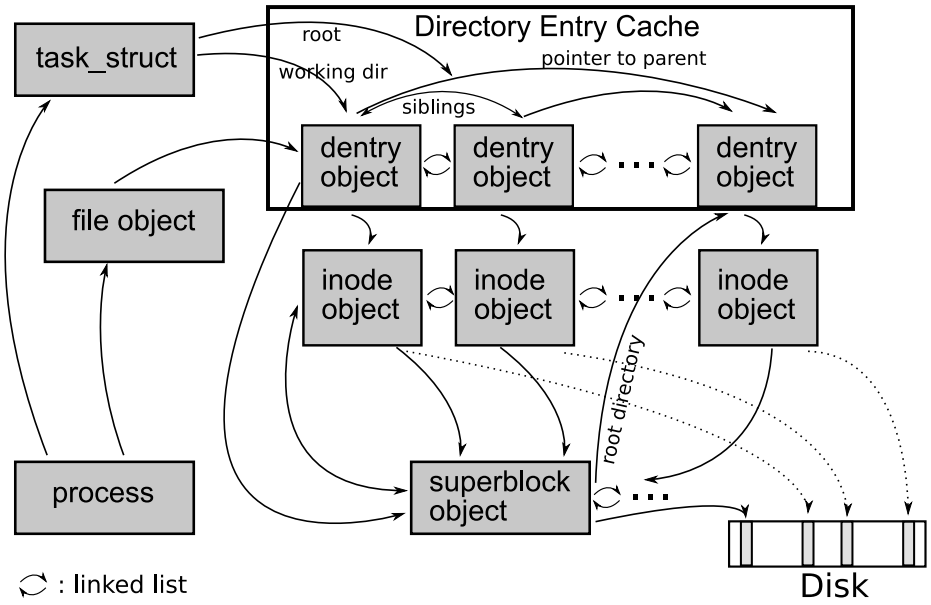

Fig. 1. VFS environment and data structures, where arrows denote pointers.

2 The Linux Virtual File System

This section introduces the Linux FS architecture and, in particular, the *Virtual File System* layer; the reader is referred to [5] 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 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()`. The next lower layer is the *system call interface* which propagates requests for system resources from applications in user space to the kernel, e.g., to the VFS.

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

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

The most relevant data structures in the VFS are *superblocks*, *dentries* and *inodes*. As shown in Fig. 1, all of them are linked by various pointers inside the structures. In addition, the data structures consist of sets of function pointers that are used to transparently access functionality provided by the underlying FS implementation. The most frequently used data objects in the VFS are dentries. The *dentry* data structures collectively describe the structure of all currently mounted FSs. Each dentry contains a file's name, a link to the dentry's parent, the list of subdirectories and siblings, hard link information, mount information, a link to the relevant super block and locking structures. It also carries a reference to its corresponding inode and a reference count that reflects the number of processes currently using the dentry. Dentries are hashed to speed up access; the hashed dentries are referred to as the *Directory Entry Cache*, or *dcache*, which is frequently consulted when resolving path names.

In our initial verification attempt to the VFS [13], our work was focused on manually abstracting these data structures and their associated control flow, so as to obtain a sufficiently small model for automated verification via model checking. Hence, much effort was put into discovering relations between the different data structures employed by the VFS [13]. The focus of this paper differs in the sense that *no* models of data structures, memory layout or control flow are derived from the implementation. Instead, each path of the compiled program is translated automatically into a corresponding constraint system which is then analysed by an SMT solver, thus fully automating the verification process.

3 The SOCA Technique

One of the disadvantages of today's model checking tools results from their restriction to the analysis of 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 for this. In addition, many errors, especially in operating system components, arise because of platform-specific and compiler-specific details such as the byte-alignment in memory and registers, memory-layout, padding between structure fields and offsets [1]. Thus, software model checkers including BLAST [15] and SLAM/SDV [4] assume either that the program under consideration "does not have wild pointers" [2] or, as we show in [20], perform poorly when analysing such software.

Analysis outline. In this paper 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 shown in the illustration, we automatically translate a program given in its *object code* into an *intermediate representation* (IR), borrowed from the Valgrind binary instrumentation framework [21], by iteratively following each program path and resolving all target addresses of computed jumps and return statements. From the IR we generate systems of bit-vector constraints for each execution path, which reflect the path-relevant register and heap contents of the program under analysis. We then employ the *Yices* SMT solver [11] to check the satisfiability of the resulting constraint systems and thus the validity of the path. This approach also allows us to add in a range of pointer safety properties, e.g., whether a pointer points to an allocated address, as simple assertions over those constraint systems, while more complex properties such as preconditions for functions can be expressed by instrumenting the program. These instrumentations are also performed on the IR, and whence access to the source code is not required.

In contrast to other methods for software verification checking, our technique does not employ program abstraction but only *path-sensitive and heap-aware program slicing*, which means that our slices are not computed over the entire

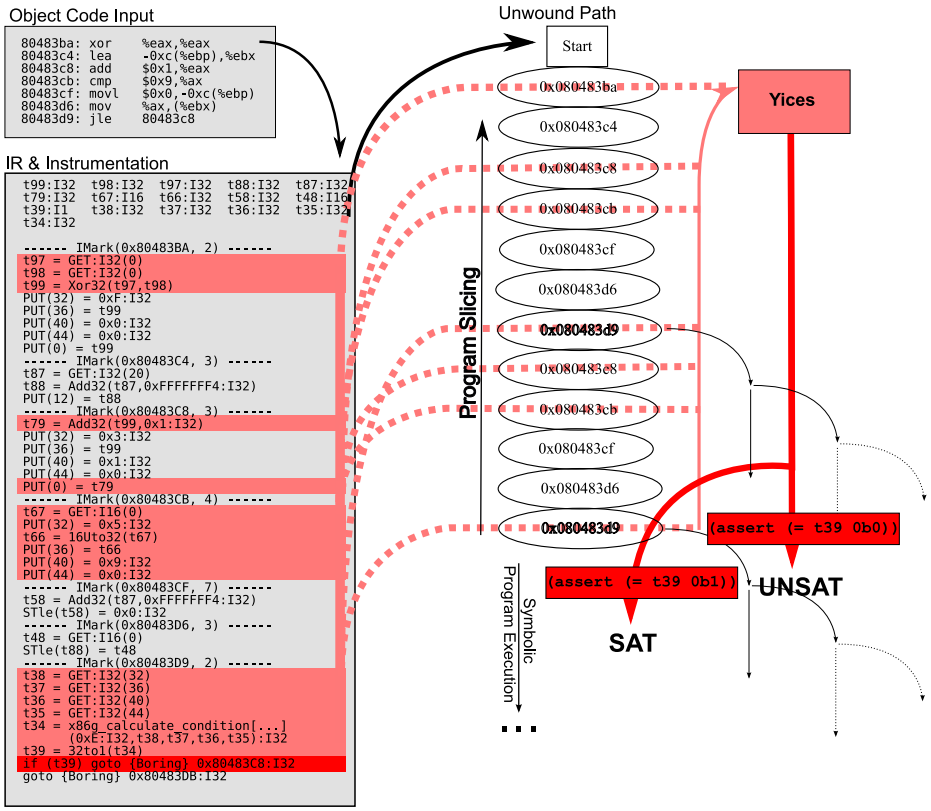

Fig. 2. Illustration of the SOCA technique.

program but only over a particular path during execution. Furthermore, we do not consider the heap as one big data object but compute slices in respect of 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 on the Linux VFS reported in Sec. 5 will show that this boundedness is not a restriction in practice: many programs are relatively "shallow" and may still be analysed either exhaustively or up to an acceptable depth.

Valgrind's IR language. Valgrind's IR language is a typed assembly language in static-single-assignment form [9, 19] using *temporary registers* and some mem-

ory for storing the *guest state*, i.e., the registers available in the architecture for which the program under analysis is compiled. The 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.

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`

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

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`, and performing the xor operation followed by storing (`PUT`) the result back to the guest state.

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))`

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

From IR to bit-vector constraints. Having a sequence of instructions decoded in the above way makes it relatively easy to generate a bit-vector constraint system for that sequence. An example with respect to the above IR

instructions is given in Fig. 4, illustrating how the `GET` instruction can be implemented in Yices as the concatenation (`bv-concat`) of byte-aligned CPU registers (i.e., the parameter of the `GET:I32` instruction, which is denoted as `r0` to `r3` in the constraint representation) from the guest state to word-aligned temporary registers. The `PUT` instruction is handled as bit-vector extraction (`bv-extract <end> <start> <source>`), respectively.

Note that CPU registers are assigned in "reverse byte order" to the temporary registers, i.e. with the least significant 8 bits in *r0* and the most significant bits 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 for the temporary registers within a single IR block. Hence, we have to be more precise 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.

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 systems and performing access operation on pointer variables that hold potentially symbolic address values quickly becomes infeasible. Our approach tackles this 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 de-referencing 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 interprocedural algorithm presented in [12]; our adaptation focuses on computing dynamic slices over a given program path.

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 relevant for the Linux VFS implementation.

Modelling the environment. One problem for program verification arises when program functions make use of an external data environment, i.e., dereference pointers to data structures that are not created by the function under

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

In our case study we "close" the VFS system to be analysed by defining a small number of dentries and associated data structures as static components of the kernel binary. As far as necessary, these data structures are directly defined in the VFS C source code by assigning a static `task_struct` (cf. `include/linux/sched.h` in the Linux source hierarchy) defining the logical context, including the working directory and a list of 15 dentries describing the FS's mount point and a simple directory hierarchy. The data objects are partially initialised by a handcrafted function that is used as a preamble in our analysis process. Note that the actual parameters to the VFS interface functions and the majority of data fields in the predefined data objects are still treated as symbolic values. Our modelling of the external environment is conducted by successively adding details to the initial memory state while carefully avoiding to be over-restrictive. We only intend to reduce the number of false warnings by eliminating impossible initial memory states to be considered in our analysis.

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

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

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

keep track of objects and their respective locations currently allocated within the program's constraint representation. Checking the above properties is then performed as an assertion check within Yices.

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

At a high level of abstraction, all locking APIs work in a similar fashion. If a kernel thread attempts to acquire a particular lock, it waits for this lock to become available, acquires it and performs its critical actions, and then releases the lock. As a result of this, a thread will wait forever if it attempts to acquire the same lock twice without releasing it in-between. Checking for the absence of this problem in single- and multi-threaded programs has recently attracted a lot of attention in the automated verification community [4, 15, 24, 23]. For software systems like the Linux kernel with its fine grained locking approach, conducting these checks is non-trivial since locks are passed by reference and due to the vast number of locks employed. A precise analysis of pointer aliasing relationships would be required to prove programs to be free of this sort of errors, which is known to be an undecidable problem in general.

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

5 Applying the SOCA Verifier to the VFS

The current implementation of the SOCA Verifier is written in C, mainly for facilitating integration with the Valgrind VEX library [21]. For applying it to the VFS, we used the VFS implementation of version 2.6.18.8 of the Linux kernel, compiled with gcc 4.3.3 for the Intel Pentium-Pro architecture. All configuration options of the kernel were left as defaults. Our experiments were then carried out on an Intel Core 2 Quad machine with 2.83 GHz and 4 GBytes of RAM, typically analysing three VFS functions in parallel.

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

were chosen so that code coverage is maximised, while execution time is kept reasonably small.

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

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

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

Table 2. Experimental Results II: SOCA Verifier statistics

	`creat`	`unlink`	`mkdir`	`rmdir`	`rename`	totals
total time	2h27m	1h22m	2h42m	1h34m	3h45m	11h50m
max. memory (SOCA)	1.03G	752M	1.15G	743M	1.41G	1.41G
max. mem. (SOCA + Yices)	1.79G	800M	1.92G	791M	2.18G	2.18G
exec. bound exhausted	X	X	X	X	X	X
path bound exhausted	-	-	-	-	-	-
paths reaching end	154	112	165	215	182	828
assertions checked	13.4k	12.4k	15.8k	11.8k	21.9k	75.3k
ratio of failed checks	0.043	0.012	0.041	0.019	0.049	0.033

Table 3. Experimental Results III: Yices statistics

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

In Table 2 we report the performance of the SOCA Verifier, showing the total time needed for analysing the kernel functions and our tool's maximum *memory consumption*. The maximum memory consumption of our tool together with the Yices solver engine is an estimate generated by summing up our tool's and Yices' maximum memory usage as given in Table 3; however, these may not necessarily hit their peak memory at the same time. The next two rows denote whether the analysis bounds were reached. We also report the number of paths reaching the end of the function analysed, the total number of assertions checked and the percentage of failed checks. Paths not reaching a return statement in the target function are terminated either due to bound exhaustion, or due to a property being violated that does not permit continuation of that path.

Finally, we outline in Table 3 the usage and behaviour of the SMT solver Yices, by reporting the number of times Yices was called when checking a particular VFS function and the total and average time spent for SMT solving. We also give the size of the checked constraint systems (CS) in boolean variables, as output by Yices and show the maximum amount of memory used by Yices.

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

It should be noted that the above tables can only give a glimpse of the total scale of experiments that we have conducted for this case study.[2] Depending on how detailed or coarse the execution environment is specified, we experienced run times reaching from a few minutes up to several days, achieving different levels of statement and condition coverage (ranging from 20% to 80%) and different error

[2] A complete account of the experiments will be published in the first author's forthcoming PhD thesis and on the SOCA website located at `http://swt-bamberg.de/soca/`.

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

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

In advance of the experiments reported in this paper, we had tested our implementation of the SOCA technique on a variety of hand-crafted examples and also on the *Verisec* suite [18] which provides 280 examples of buffer overflow vulnerabilities taken from application programs. In all these cases we experienced low false-positive rates of less than 20%. However, as these examples represent closed systems not using external data objects, they are handled more efficiently by the SOCA Verifier than the VFS which makes heavy use of external data objects.

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

Not discovering 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.

6 Related Work

A survey on automated techniques for formal software verification can be found in [10]. Verification approaches employing predicate abstraction to model-check the source code of operating system components are presented in [4, 6, 15]. In theory, these are able to prove a file system implementation to be, e.g., free of

deadlock, by checking the proper use of locking mechanisms. However, modern model checkers such as BLAST [15] require extensive manual preprocessing and are not able to deal with general pointer operations [20]. Recent work [22] shows further that, again in contrast to our verifier, BLAST cannot analyse programs with multiplicities of locks since its specification language does not permit the specification of observer automatons for API safety rules with respect to function parameters.

A bounded model checker for C source code based on symbolic execution and SAT solving is SATURN [24]. This tool is specialised on checking locking properties and null-pointer de-references. The authors show that their tool scales for analysing the entire Linux kernel. Unlike the SOCA Verifier, the approach in [24] computes function summaries instead of adding the respective code to the control flow, unwinds loops a fixed number of times and does not handle recursion. Hence, it can be expected to produce more unsound results but scale better than our SOCA technique.

Actual file system implementations were studied by Engler et al. in [25, 26]. In [26], model checking is used within the systematic testing of EXT3, JFS and ReiserFS. The employed verification system consists of an explicit-state model checker running the Linux kernel, a file system test driver, a permutation checker which verifies that a file system can always recover, and a recovery checker using the *fsck* recovery tool. The verification system starts with an empty file system and recursively generates successive states by executing system calls affecting the file system under analysis. After each step, the verification system is interrupted, and *fsck* is used to check whether the file system can recover to a valid state. In contrast to this, our work focuses on checking a different class of properties, namely pointer safety and locking properties. Thanks to our memory model we can analyse these properties precisely and feed back detailed error traces together with specific initial heap state information leading to the error.

7 Conclusions and Future Work

The initial motivation for our SOCA technique to automated program verification was to explore the possibilities of using 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 show in this paper, these constructs can be dealt with efficiently in path-wise symbolic object-code analysis, while they are usually ignored by static techniques or by source-code-based approaches.

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

References

[1] Balakrishnan, G., Reps, T., Melski, D., and Teitelbaum, T. WYSINWYX: What You See Is Not What You eXecute. In *VSTTE '08*, vol. 4171 of *LNCS*, pp. 202–213. Springer, 2008.

[2] 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 '06*, number 4, pp. 73–85, USA, 2006. ACM.

[3] Ball, T., Majumdar, R., Millstein, T., and Rajamani, S. K. Automatic predicate abstraction of C programs. *SIGPLAN Not.*, 36(5):203–213, 2001.

[4] Ball, T. and Rajamani, S. K. Automatically validating temporal safety properties of interfaces. In *SPIN '01*, vol. 2057 of *LNCS*, pp. 102–122. Springer, 2001.

[5] Bovet, D. and Cesati, M. *Understanding the Linux Kernel*. O'Reilly, 2005.

[6] Chaki, S., Clarke, E., Groce, A., Ouaknine, J., Strichman, O., and Yorav, K. Efficient verification of sequential and concurrent C programs. *FMSD*, 25(2-3):129–166, 2004.

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

[8] Clarke, E., Kroening, D., and Lerda, F. A tool for checking ANSI-C programs. In *TACAS '04*, vol. 2988 of *LNCS*, pp. 168–176. Springer, 2004.

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

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

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

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

[13] Galloway, A., Lüttgen, G., Mühlberg, J. T., and Siminiceanu, R. Model-checking the Linux Virtual File System. In *VMCAI '09*, vol. 5403 of *LNCS*, pp. 74–88. Springer, 2009.

[14] Gulavani, B. S. and Rajamani, S. K. Counterexample driven refinement for abstract interpretation. In *TACAS '06*, vol. 3920 of *LNCS*, pp. 474–488. Springer, 2006.

[15] Henzinger, T. A., Jhala, R., Majumdar, R., Necula, G. C., Sutre, G., and Weimer, W. Temporal-safety proofs for systems code. In *CAV '02*, vol. 2402 of *LNCS*, pp. 382–399. Springer, 2002.

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

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

[18] Ku, K., Hart, T. E., Chechik, M., and Lie, D. A buffer overflow benchmark for software model checkers. In *ASE '07*, pp. 389–392, USA, 2007. ACM.

[19] Leung, A. and George, L. Static single assignment form for machine code. In *PLDI '99*, pp. 204–214, USA, 1999. ACM.

[20] Mühlberg, J. T. and Lüttgen, G. BLASTing Linux code. In *FMICS '06*, vol. 4346 of *LNCS*, pp. 211–226. Springer, 2006.

[21] Nethercote, N. and Seward, J. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *PLDI '07*, vol. 42, pp. 89–100, USA, 2007. ACM.

[22] Sery, O. Enhanced property specification and verification in BLAST. In *FASE '09*, vol. 5503 of *LNCS*, pp. 456–469. Springer, 2009.

[23] Witkowski, T., Blanc, N., Kroening, D., and Weissenbacher, G. Model checking concurrent Linux device drivers. In *ASE '07*, pp. 501–504, USA, 2007. ACM.

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

[25] Yang, J., Sar, C., Twohey, P., Cadar, C., and Engler, D. R. Automatically generating malicious disks using symbolic execution. In *Security and Privacy*, pp. 243–257. IEEE, 2006.

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