Symbolic Object Code Analysis

Jan Tobias Mühlberg and Gerald Lüttgen

Software Technologies Research Group
University of Bamberg, 96045 Bamberg, Germany.
{jan-tobias.muehlberg, gerald.luettgen}@swt-bamberg.de

Abstract Current software model checkers quickly reach their limits when being applied to verifying pointer safety properties in source code that includes function pointers and inlined assembly. This paper introduces an alternative technique for checking pointer safety violations, called *Symbolic Object Code Analysis* (SOCA), which is based on bounded symbolic execution, incorporates path-sensitive slicing, and employs the SMT solver Yices as its execution and verification engine. Experimental results of a prototypic SOCA Verifier, using the Verisec suite and almost 10,000 Linux device driver functions as benchmarks, show that SOCA performs competitively to source-code model checkers and scales well when applied to real operating systems code and pointer safety issues.

1 Introduction

One challenge when verifying complex software is the proper analysis of pointer operations. A recent study shows that most errors found in device drivers involve *memory safety* [6]. Writing software that is free of memory safety concerns, e.g., free of errors caused by pointers to invalid memory cells, is difficult since many such issues result in program crashes at later points in execution. Hence, a statement causing a memory corruption may not be easily identifiable using conventional validation and testing tools, e.g., *Purify* [31] and *Valgrind* [27].

Today's static verification tools, including *software model checkers* such as [4,7,8,13], are also not of much help: they either assume that programs do "not have wild pointers" [3], perform poorly in the presence of pointers [25], or simply cannot handle certain software. A particular challenging kind of software are operating system (OS) components such as device drivers, which are usually written in C code involving function pointers, pointer arithmetic and inlined assembly. Further issues arise because of platform-specific and compiler-specific details concerning memory layout, padding and offsets [2]. In addition, several approaches to model checking *compiled programs* given in assembly or bytecode [5,23,33,35] and also to integrating *symbolic execution* [17] with model checking [12,11,16,29,34] have recently been presented. However, these are tailored to exploit specific characteristics of certain programming paradigms such as object-oriented programming, or lack support for data structures, function pointers and computed jumps, or require substantial manual modelling effort (cf. Sec. 5).

This paper introduces and evaluates a novel, automated technique to identifying memory safety violations, called *Symbolic Object Code Analysis* (SOCA).

This technique is based on the *symbolic execution* [17] of compiled and linked programs (cf. Sec. 2). In contrast to other verification techniques, SOCA requires only a minimum of manual modelling effort, namely the abstract, symbolic specification of a program's execution context in terms of function inputs and initial heap content. Our extensive evaluation of a prototypic SOCA implementation, also reported in this paper, shows that SOCA performs competitively to state-of-the-art model checkers such as [4,8,13] on programs with "well-behaved" pointers, and that it scales well when applied to "dirty" programs such as device drivers which cannot be properly analysed with source-code model checkers.

Technically, the SOCA technique traverses a program's *object code* in a systematic fashion up to a certain depth and width, and calculates at each assembly instruction a *slice* [36] required for checking the relevant pointer safety properties. It translates such a slice and properties into a bit-vector constraint problem and executes the property checks by invoking the *Yices* SMT solver [10] (cf. Sec. 3). To the best of our knowledge, SOCA is the only program verification technique reported in the literature, that features full support for pointer arithmetics, function pointers and computed jumps. While SOCA is based on existing and well-known techniques, combining and implementing these for object code analysis is challenging. Much engineering effort went into our SOCA implementation, so that it scales to complex real-world OS code.

The particular combination of techniques in SOCA is well suited for checking memory safety. Analysing object code is beneficial in that it inherently considers compiler specifics such as code optimisations, makes memory layout obvious, and does away with the challenge of handling mixed input languages involving assembly code. Symbolic execution, rather than the concrete execution adopted in testing, can handle software functions with many input parameters, whose values are typically not known at compile time. It is the existence of efficient SMT solvers that makes the symbolic approach feasible. Symbolic execution also implies a path-wise exploration, thus reducing the aliasing problem and allowing us to handle even complex pointer operations and computed jumps. In addition, slicing can now be conducted at path-level instead of at program-level, resulting in drastically smaller slices to the extent that abstraction is not necessary for achieving scalability. However, the price of symbolic execution is that it must be bounded and can thus only analyse code up to a finite depth and width.

To evaluate our technique, we have implemented a prototypic SOCA tool, the *SOCA Verifier*, for programs compiled for the 32-bit Intel Architecture (IA32) and performed extensive experiments (cf. Sec. 4). Using the *Verisec* benchmark [21] we show that the SOCA Verifier performs on par with the model checkers *LoopFrog* [19] and *SatAbs* [8] with regards to performance, error detection and false-positive rates. We have also applied the SOCA Verifier to 9296 functions taken from 250 Linux device drivers. Our tool is able to successfully analyse 95% of these functions and, despite the fact that SOCA performs a bounded analysis, 28% of the functions are analysed exhaustively. Therefore, SOCA proves itself to be a capable technique when being confronted with checking pointer-complex software such as OS components.

2 Pointers, Aliasing & Intermediate Representation

The verification technique developed in this paper aims at ensuring that every pointer in a given program is valid in the sense that it *(i)* never references a memory location outside the address space allocated by or for that program, and *(ii)* respects the usage rules of the Application Programming Interfaces (APIs) employed by the program. There exist several categories of memory safety properties — *(1) dereferencing invalid pointers*: a pointer may not be `NULL`, shall be initialised, and shall not point to a memory location outside the address space allocated by or for the program; *(2) uninitialised reads*: memory cells shall be initialised before they are read; *(3) violation of memory permissions*: when the program is loaded into memory, its segments are assigned with permissions that determine whether a segment can be read, written or executed; *(4) buffer overflows*: out-of-bounds read and write operations to objects on the heap and stack, which may lead to memory corruption and give way to various security problems; *(5) memory leaks*: when a program dynamically allocates memory but loses the handle to it, the memory cannot be deallocated anymore; *(6) proper handling of allocation and deallocation*: OSs usually provide several APIs for the dynamic (de)allocation of memory, whose documentation specifies precisely what pairs of functions are to be employed, and how.

Aliasing in source & object code. A major issue for analysing pointer programs is aliasing. Aliasing means that a data location in memory may be accessed through different symbolic names. Since aliasing relations between symbolic names and data locations often arise unexpectedly during program execution, they may result in erroneous program behaviours that are particularly hard to trace and debug. To illustrate this, the following C program shows a complicated way of implementing an infinite loop:

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

At least three different outcomes of the program's execution can occur as a result of varying assumptions made about pointer aliasing by the developer and the compiler, as well as compiler optimisations applied to the code. In the following listing we give the output of the program when compiled with gcc version 4.1.2 (left) and gcc version 4.3.1 (middle and right).

```
$ gcc -O2 e_loop.c       $ gcc -O2 e_loop.c       $ gcc -O1 e_loop.c
$ ./a.out                $ ./a.out                $ ./a.out
bfc76f2c: 10             bfc7428c: 10             -> does not terminate
bfc76f2c: 0              bfc7428c: 10
bfc76f2c: 0              bfc7428c: 10
```

More surprises are revealed when disassembling the program that produced the output shown in the middle of the above listing:

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

One can see at instructions 80483cf and 80483d6 that *p1* and *p2* are pointing to the same location in memory, and that **p2* is actually written before **p1*. This is unexpected when looking at the program's source code but valid from the compiler's point of view since it assumes that the two pointers are pointing to different data objects. As another consequence of this assumption, register *eax* is never reloaded from the memory location to which *p1* and *p2* point.

This example shows that source-code-based analysis has to decide for a particular semantics of the source language, which may not be the one that is used by a compiler. Hence, results obtained by analysing the source code may not meet a program's runtime behaviour. While this motivates the analysis of compiled programs, doing so does not provide a generic solution for dealing with pointer aliasing, as aliasing relationships may depend on runtime conditions.

Intermediate representation. A program is stored by us in an intermediate representation (IR) borrowed from *Valgrind* [27], a framework for dynamic binary instrumentation. The IR consists of a set of *basic blocks* containing a group of statements such that all transfers of control to the block are to the first statement in the group. Once the block is entered, its statements are executed sequentially until an *exit* statement is reached. An exit is always denoted as `goto <t>`, where `<t>` is either a constant or a temporary register that determines the next program location to be executed. Guarded jumps are written as `if (<b>) goto <t>`, where `<b>` is a temporary register of type boolean, which has previously been assigned within the block.

The listing below depicts an example for assembly statements and their corresponding IR statements. It shows how, e.g., 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. All operands used in the first block of the example are 4 bytes, or 32 bits, in size.

IA32 Assembly	**IR Instructions**

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

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

As can be seen, the IR is essentially a typed assembly language in static-single-assignment form [22], and employs *temporary registers*, which are denoted as $t<n>$, and the *guest state*. The guest state consists of the contents of the registers that are available in the architecture for which the program under analysis is compiled. While machine registers are always 8 bits long, temporary registers may be 1, 8, 16, 32 or 64 bits in length. As a result of this, statement `t9 = GET:I32(0)` means that *t9* is generated by concatenating machine registers 0 to 3. Since each IR block is in static-single-assignment form with respect to the temporary registers, *t9* is assigned only once within a single IR block. As a valuable feature for analysing pointer safety, Valgrind's IR makes all load and store operations to memory cells explicit.

3 SOCA – Symbolic Object Code Analysis

This section introduces our new approach to verifying memory safety in compiled and linked programs, to which we refer as *Symbolic Object Code Analysis* (SOCA). The basic idea behind our approach employs well-known techniques including *symbolic execution* [17], *SMT solving* [20] and *program slicing* [36]. However, combining these ideas and implementing them in a way that scales to real applications, such as Linux device drivers, is challenging and the main contribution of this paper.

Starting from a program's given entry point, we automatically translate each instruction of the program's *object code* into Valgrind's IR language. This is done lazily, i.e., as needed, by iteratively following each program path in a depth-first fashion and resolving target addresses of computed jumps and return statements. We then generate systems of bit-vector constraints for the path under analysis, which reflect the path-relevant register content and heap content of the program. In this process we employ a form of program slicing, called *path-sensitive and heap-aware program slicing*, which is key to SOCA's scalability and makes program abstraction unnecessary. Finally, we invoke the SMT solver *Yices* [10] to check the satisfiability of the resulting constraint systems and thus the validity of the path. This approach allows us to instrument the constraint systems on-the-fly as necessary, by adding constraints that express, e.g., whether a pointer points to an allocated address.

SOCA leaves most of a program's input and initial heap content unspecified in order to allow the SMT solver to search for inputs that may reveal pointer errors. 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 may be infinite. Our experimental results (cf. Sec 4) show that this boundedness is not a restriction in practice: many interesting programs, such as Linux device driver functions, are relatively "shallow" and may still be analysed either exhaustively or to an acceptable extent.

Translating IR into Yices constraints. To translate IR statements into bit-vector constraint systems for Yices, we have defined a simple operational semantics for Valgrind's IR language. Due to space constraints we cannot present

this semantics here and refer the reader to [24] instead. Instead, we focus directly on examples illustrating this translation.

As a first example we consider the `PUT(0) = t7` statement from the example above. Intuitively, the semantics of *PUT* is to store the value held by *t7* to the guest state, in registers 0 to 3 (i.e., *r0* to *r3* below):

IR Instruction	**Constraint Representation**

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

Here, the `bv-extract` operation denotes bit-vector extraction. Note that the IA32 CPU registers are assigned in reverse byte order, while arithmetic expressions in Yices are implemented for bit-vectors that have their 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 translate the content of temporary registers when accessing the guest state.

Similar to the *PUT* instruction, we can express *GET*, i.e., loading a value from the guest state, as the concatenation of bit-vectors, and the *Xor* and *Add* instructions in terms of bit-vector arithmetic:

IR Instruction	**Constraint Representation**
`t9 = GET:I32(0)`	`(define t9::(bitvector 32) (bv-concat` `(bv-concat r3 r2) (bv-concat r1 r0))`
`t7 = Xor32(t9,t8)`	`(define t7::(bitvector 32) (bv-xor t9 t8))`
`t41 = Add32(t42,` `0xFFFFFFF4:I32)`	`(define t88::(bitvector 32)` `(bv-add t87 (mk-bv 32 4294967284)`

More challenging to implement are the IR instructions *ST* (*store*) and *LD* (*load*) which facilitate memory access. The main difference of these instructions to *PUT* and *GET* is that the target of *ST* and the source of *LD* are variable and may only be computed at runtime. To include these statements in our framework we have to express them in a flexible way, so that the SMT solver can identify cases in which safety properties are violated. In Yices we declare a function *heap* as our representation of the program's memory. An exemplary *ST* statement `ST(t5) = t32` can be expressed in terms of updates of that function:

IR Instruction	**Constraint Representation**

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

Since the above *ST* instruction stores the content of a 32-bit variable in four separate 8-bit memory cells, we have to perform four updates of *heap*. Byte-ordering conventions apply in the same way as explained for *PUT*. Constraints for the *LD* instruction are generated analogous to *GET*.

Encoding pointer safety assertions. Being able to translate each object code instruction into constraints allows us to express the safety pointer properties given in Sec. 2 in terms of assertions within the constraint systems. The simplest case of such an assertion is a null-pointer check. For the *ST* instruction in the above example, we state this assertion as `(assert (= t5 (mk-bv 32 0)))`. If the resulting constraint system is satisfiable, Yices will return a possible assignment to the constraint system variables representing the program's input. This input is constructed such that it will drive the program into a state in which $t5$ holds the value `NULL` at the above program point.

However, many memory safety properties demand additional information to be collected about a program's current execution context. In particular, answering the question whether a pointer may point to an "invalid" memory area requires knowledge which cells are currently allocated. We retain this information by adding a function named *heaploc* to our memory representation:

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

This allows us to express assertions stating that, e.g., pointer $t5$ has to point to an allocated address at the program location where it is dereferenced, as:

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

All other pointer safety properties mentioned in Sec. 2 may be encoded along the lines of those two examples. Most of them require further additional information to be added to the *heaploc* function. To reduce the size and search space of the resulting constraint systems we check assertions one-by-one with a specialised *heaploc* function for each property. The full details of our generation of constraint systems can be found in [24].

Path-sensitive slicing. To ensure scalability of our SOCA technique, we do not run Yices on an entire path's constraint system. Instead we compute a slice [36] of the constraint system containing only those constraints that are relevant to the property to be checked at a particular program location.

The approach to path-sensitive program slicing in SOCA employs an algorithm based on system dependence graphs as introduced in [14]. Our slices are extracted using conventional slicing criteria (L, var) denoting a variable *var* that is used at program location L but, in contrast to [14], over the single path currently being analysed instead of the program's entire control flow. The slice is then computed by collecting all statements on which *var* is data dependent by tracing the path backwards, starting from L up to the program's entry point. While collecting flow dependencies is relatively easy for programs that do only use CPU registers and temporary registers, it becomes difficult when dependencies to the heap and stack are involved.

Handling memory access in slicing. Consider the following two IR statements: `01 ST(t5) = t32; 02 t31 = LD:I32(t7)`. To compute a slice for the slicing criterion $(02, t31)$ we have to know whether the store statement `ST` may affect the value of $t31$, i.e., whether $t5$ and $t7$ may alias. We obtain this information by using Yices to iteratively explore the potential address range that can be accessed via $t5$. This is done by making Yices find a satisfying model e for $t5$, as described below. When reading a model, which is represented by Yices as a bit-vector, we compute its integer representation and further satisfying models e' such that $e > e'$ or $e < e'$ holds, until the range is explored.

To use Yices as efficiently as possible when searching for satisfying models, we employ stepwise adding or retracting of constraints. Since we remember only the maximal and minimal satisfying models for a given pointer, this is an over-approximation because not the entire address range may be addressable by that pointer. However, using this abstraction presents a trade-off concerning only the size of the computed slices and not their correctness, and helps us to keep the number of Yices runs and the amount of data to be stored small.

By computing the potential address range accessed by a pointer used in a load statement, $t7$ in our example, and looking for memory intervals overlapping with the range of $t7$, we can now determine which store operations may affect the result of the load operation above. Despite being conservative when computing address ranges, our experience shows that most memory access operations end up having few dependencies; this is because most pointers evaluate to a concrete value, i.e., the constraint system has exactly one satisfying model, rather than a symbolic value which represents potentially many concrete values.

Handling computed jumps. A major challenge when analysing compiled programs arises from the extensive use of function pointers, jump tables and jump target computations. While most source-code-based approaches simply ignore function pointers [4,8,13], this cannot be done when analysing object code since jump computations are too widely deployed here. The most common example for a computed jump is the *return* statement in a subroutine. To perform a return, the bottom element of the stack is loaded into a temporary register, e.g., $t1$, followed by a `goto t1` statement, which effectively sets the value of the program counter to $t1$. In our approach, jump target addresses are determined in the same way as addresses for load and store operations, i.e., by computing a slice for each jump target and then using Yices to determine satisfying models for the target register.

Optimising GET & PUT statements. A potential problem with respect to the scalability of our approach arises from the vast number of *GET* and *PUT* statements in IR code. In particular, the frequent de-/re-composing of word-aligned temporary registers into guest registers and back into temporary registers introduces lots of additional variables in the SMT solver. These *GET* and *PUT* statements are introduced into our IR in order to make the IR block generated for a single CPU instruction reentrant with respect to the guest state. Thereby the need to repeat the translation from object code to IR whenever an

instruction is used in a different execution context is avoided, at the expense of having to deal with larger constraint systems.

An efficient way around this issue is to optimise unnecessary *GET* and *PUT* operations away, based on a *reaching definition* analysis for a given register and path. Practical results show that this simple optimisation greatly reduces the memory consumption of Yices for large constraint systems. We can apply the same optimisations to memory accesses in cases where the address arguments to *LD* and *ST* evaluate to constant values. From our experience, dealing with unnecessary *GET*, *PUT*, *LD* and *ST* statements, by performing the above optimisations on IR level for an entire execution path, results in more efficient constraint systems and shorter runtimes of SOCA and Yices than when allowing Valgrind to perform similar optimisations at basic-block level.

Determining a valid initial memory state. Another challenge when implementing symbolic execution as an SMT problem is given by the enormous search space that may result from leaving the program's initial memory state undefined. OS components, including functions taken from device drivers, make regular use of an external data environment consisting of heap objects allocated and initialised by other OS modules. Hence, this data environment cannot be inferred from the information available in the program binary. In practice, data environments can often be embedded into our analysis without much effort, by adding a few lines of C code as a preamble, as is shown in [26].

4 Experimental Results

To evaluate our SOCA technique regarding its ability to identify pointer safety issues and to judge its performance when analysing OS components, we have implemented SOCA in a prototypic tool, the *SOCA Verifier*. The tool comprises 15,000 lines of C code and took about one person-year to build; details of its architecture can be found in [24]. This section reports on extensive experiments we conducted in applying the SOCA Verifier to a benchmark suite for software model checkers and to a large set of Linux device drivers. All experiments were carried out on a 16-core PC with 2.3 GHz clock speed and 256 GB of RAM, running 16 instances of the SOCA Verifier in parallel. However, an off-the-shelf PC with 4 GB of RAM is sufficient for everyday use, when one must not verify thousands of programs concurrently to meet a paper submission deadline.

4.1 Experiments I: The Verisec Benchmark

To enable a qualitative comparison of the SOCA Verifier to other tools, we applied it to the Verisec benchmark [21]. Verisec consists of 298 test programs (149 faulty programs – *positive* test programs – and 149 corresponding fixed programs – *negative* test programs) for buffer overflow vulnerabilities, taken from various open source programs. These test cases are given in terms of C source code which we compiled into object code using *gcc*, and are provided with a configurable buffer size which we set to 4. The bounds for the SOCA

Verifier were set to a maximum of 100 paths to be analysed, where a single instruction may appear at most 500 times per path. Yices was configured to a timeout of 300 seconds per invocation. Of these bounds, only the timeout for Yices was ever reached.

Table 1. Comparison of SatAbs, LoopFrog and SOCA

	$R(d)$	$R(f)$	$R(\neg f\|d)$
SatAbs (from [21])	0.36	0.08	n/a
LoopFrog (from [19])	1.0	0.26	0.74
SOCA	0.66	0.23	0.81

In previous work [19,21], Verisec was used to evaluate the C-code model checkers *SatAbs* [8] and *LoopFrog* [19]. To enable a transparent comparison, we adopted the metrics proposed in [38]: in Table 1 we report the *detection rate* $R(d)$, the *false-positive rate* $R(f)$, and the *discrimination rate* $R(\neg f|d)$. The latter is defined as the ratio of positive test cases for which an error is correctly reported, plus the negative test case for which the error is correctly not reported, to all test cases; hence, tools are penalised for not finding bugs and for not reporting a sound program as safe.

As Table 1 testifies, the SOCA Verifier reliably detects the majority of buffer overflow errors in the benchmark, and has a competitive false-positive rate and a better discrimination rate than the other tools. Remarkable is also that the SOCA Verifier failed for only four cases of the Verisec suite: once due to memory exhaustion and three times due to missing support for certain IR instructions in our tool. Only our detection rate is lower than the one reported for LoopFrog. An explanation for this is the nature of Verisec's test cases where static arrays are declared globally. This program setup renders Verisec easily comprehensible for source-code verification tools since the bounds of data objects are clearly identifiable in source code. In object code, however, the boundaries of data objects are not visible anymore. This makes the SOCA Verifier less effective when analysing programs with small, statically declared buffers.

Hence, despite having used a benchmark providing examples that are in favour of source code analysis, our results show that object code analysis, as implemented in the SOCA Verifier, can compete with state-of-the-art source-code model checkers. However, as our tool analyses object code, it can be employed in a much wider application domain. Unfortunately, benchmarks that include dynamic allocation and provide examples of pointer safety errors other than buffer overflows are, to the best of our knowledge, not publicly available.

Fig. 1*(a)* displays the CPU times consumed for analysing each test case in the Verisec benchmark. The vast majority of test cases is analysed by the SOCA Verifier within less than three mins per case. As shown in Table 2, the average computation time consumed per test case is 18.5 mins. In total, about 92 CPU hours were used. The memory consumption of both, the SOCA Verifier and Yices

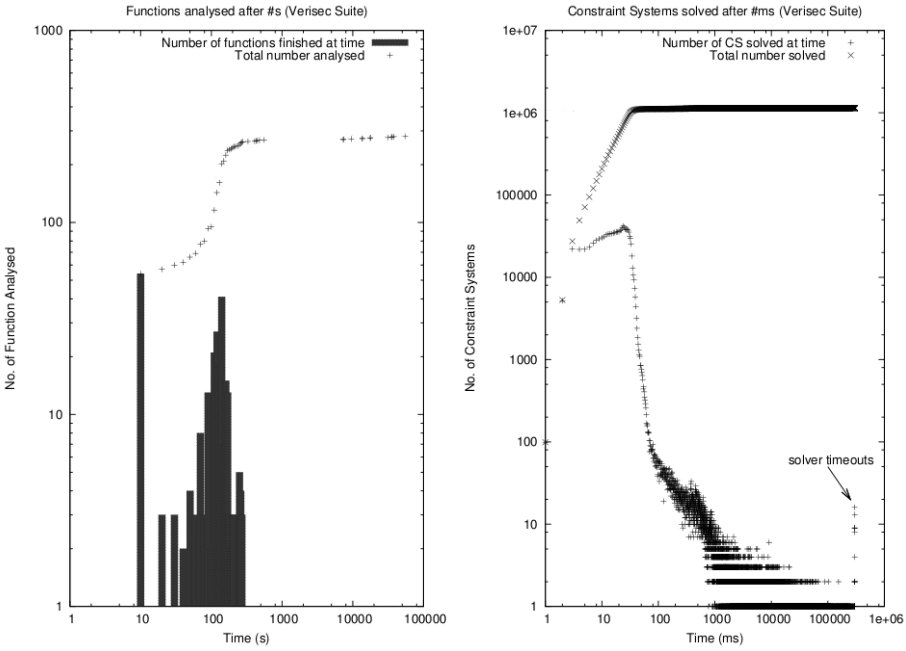

Figure 1. Performance results for the Verisec benchmark. *(a)* Numbers of test cases verified by time (left). *(b)* Numbers of constraint systems solved by time (right).

together, amounts to an average of only 140 MBytes and a maximum of about 3 GBytes, which is a memory capacity that is typically available in today's PCs. Notably, Ku reported in [21] that the SatAbs tool crashed in 73 cases and timed

Table 2. Performance statistics for the Verisec suite

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

out in another 87 cases with a timeout of 30 mins. The runtime of the SOCA Verifier exceeds this time in only 7 cases.

In Fig. 1*(b)* we show the behaviour of Yices for solving the constraint systems generated by the SOCA Verifier. For the Verisec suite, a total of 11,994,834 constraint systems were solved in 89 hours. 2,250,878 (19%) of these systems express verification properties, while the others were required for computing control flow, e.g., for deciding branching conditions and resolving computed jumps. With the timeout for Yices set to 5 mins, the solver timed out on 34 constraint systems, and 96% of the constraint systems were solved in less than one second. Thus, the SOCA Verifier's qualitative performance is competitive with state-of-the-art software model checkers. In addition, it is sufficiently efficient to be used as an automated debugging tool by software developers, both regarding time efficiency and space efficiency.

4.2 Experiments II: Linux Device Drivers

To evaluate the scalability of the SOCA Verifier, a large set of 9296 functions originating from 250 Linux device drivers of version 2.6.26 of the Linux kernel compiled for IA32 was analysed by us. Our experiments employed the Linux utility `nm` to obtain a list of function symbols present in a device driver. By statically linking the driver to the Linux kernel we resolved undefined symbols in the driver, i.e., functions provided by the OS kernel that are called by the driver's functions. The SOCA technique was then applied on the resulting binary file to analyse each of the driver's functions separately. The bounds for the SOCA Verifier were set to a maximum of 1000 paths to be analysed, where a single instruction may appear at most 1000 times per path, thereby effectively bounding the number of loop iterations or recursions to that depth. Moreover, Yices was configured to a timeout of 300 seconds per invocation.

Table 3. Performance statistics for the Linux device drivers

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

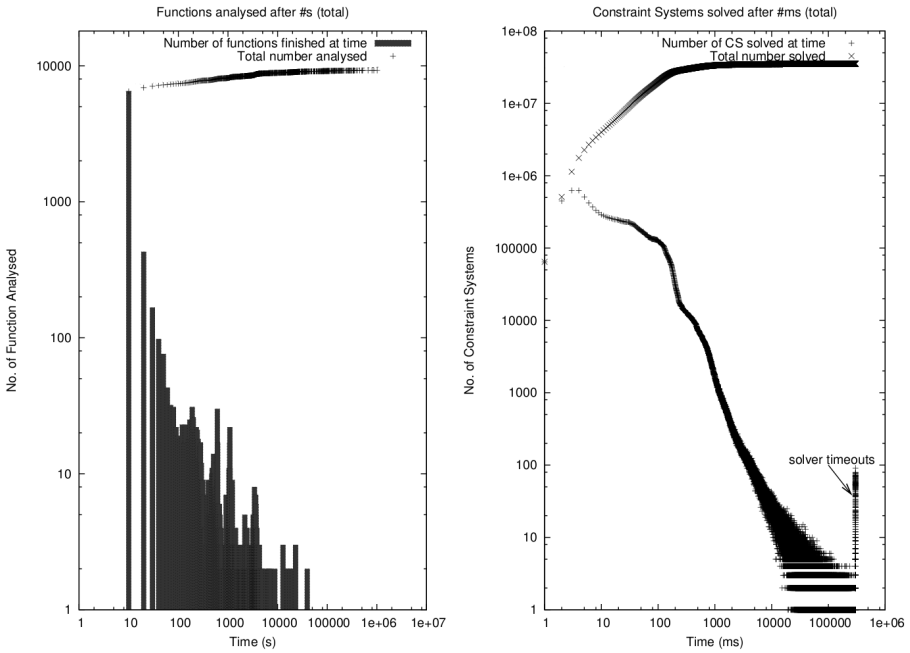

Figure 2. Performance results for the Linux device drivers. *(a)* Numbers of test cases verified by time (left). *(b)* Numbers of constraint systems solved by time (right).

Our results in Table 3 show that 94.4% of the functions in our sample could be analysed by the SOCA Verifier. In 67.5% of the functions the exhaustion of execution bounds led to an early termination of the analysis. However, the analysis reached a considerable depth even in those cases, analysing paths of lengths of up to 22,577 CPU instructions. Interestingly, 27.8% of the functions could be analysed exhaustively, where none of the bounds regarding the number of paths, the path lengths, or the SMT solver's timeout were reached. As depicted in Fig. 2*(a)*, the SOCA Verifier returned a result in less than 10 mins in the majority of cases, while the generated constraint systems were usually solved in less than 500 ms. The timeout for Yices was hardly ever reached (cf. Fig. 2*(b)*).

As an aside, it should be mentioned that in 0.98% (91 functions) of the sample Linux driver functions, the SOCA Verifier may have produced unsound results due to non-linear arithmetic within the generated constraint systems, which is not decidable by Yices. In addition, our verifier failed in 5.6% of the cases (522 functions) due to either memory exhaustion, missing support for particular assembly instructions in our tool or Valgrind, or crashes of Yices.

Our evaluation shows that the SOCA Verifier scales up to real-world OS software while delivering very good performance. Being automatic and not restricted to analysing programs available in source code only, the SOCA Verifier is an efficient tool that is capable of aiding a practitioner in debugging pointer-

complex software such as OS components. The application of the SOCA Verifier is, however, not restricted to verifying memory safety. In [26] we presented a case study on retrospective verification of the Linux Virtual File System (VFS) using the SOCA Verifier for checking violations of API usage rules such as deadlocks caused by misuse of the Linux kernel's `spinlock` API.

5 Related Work

There exists a wealth of related work on automated techniques for formal software verification, a survey of which can be found in [9]. We focus here on more closely related work, namely on *(i)* model checking bytecode and assembly languages, *(ii)* approaches combining model checking with symbolic execution, and *(iii)* program slicing.

Model checking bytecode & assembly languages. In recent years, several approaches to model checking *compiled programs* by analysing bytecode and assembly code have been presented. In [32,35], *Java PathFinder* (*JPF*) for model checking Java bytecode was introduced. *JPF* generates the state space of a program by monitoring a virtual machine. Model checking is then conducted on the states explored by the virtual machine, employing collapsing techniques and symmetry reduction for efficiently storing states and reducing the size of the state space. These techniques are effective because of the high complexity of *JPF* states and the specific characteristics of the Java memory model. In contrast, the SOCA technique to verifying object code involves relatively simple states and, in difference to Java, the order of data within memory is important in IA32 object code. Similar to *JPF*, *StEAM* [23] model checks compiled C++ programs by using a modified Internet Virtual Machine to generate a program's state space. In addition, *StEAM* implements heuristics to accelerate error detection.

BTOR [5] and *[mc]square* [28,33] are tools for model checking assembly code for micro-controllers. They accept assembly code as their input, which may either be obtained during compilation or, as suggested in [33], by disassembling a binary program. Since the problem of disassembling a binary program is undecidable in general, the SOCA technique focuses on the verification of binary programs without the requirement of disassembling a program at once.

All the above tools are explicit model checkers that require a program's entire control flow to be known in advance of the analysis. As we have explained in Sec. 3, this is not feasible in the presence of computed jumps. The SOCA technique has been especially designed to deal with OS components that make extensive use of jump computations.

Combining model checking with symbolic execution. *Symbolic execution* was introduced by King [17] as a means of improving program testing by covering a large class of normal executions with a single 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 survey on recent trends in symbolic execution with an emphasis on program analysis and test generation is given in [30].

Several frameworks for integrating symbolic execution with model checking have been developed, including *Symbolic JPF* [29] and *DART* [11]. *Symbolic JPF* is a successor of the previously mentioned *JPF*. *DART* implements directed and automated random testing to generate test drivers and harness code to simulate a program's environment. The tool accepts C programs and automatically extracts function interfaces from source code. Such an interface is used to seed the analysis with a well-formed random input, which is then mutated by collecting and negating path constraints while symbolically executing the program under analysis. 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* [12], 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 [26]. *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.

DART-like techniques, also known as *concolic testing*, are described in [16,34]. These techniques rely on performing concrete executions on random inputs while collecting path constraints along executed paths. These constraints are then used to compute new inputs that drive the program along alternative paths. In difference to this approach, SOCA uses symbolic execution to explore all paths and concretises only for resolving computed jumps.

Another bounded model checker for C source code based on symbolic execution and SAT solving is SATURN [37]. This tool is specialised on checking locking properties and null-pointer de-references and is thus not as general as SOCA. The authors of [37] show that their tool scales to analysing the entire Linux kernel. Unlike the SOCA Verifier, their approach 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.

Program slicing. An important SOCA ingredient other than symbolic execution is *path-sensitive* slicing. *Program slicing* was introduced by Weiser [36] as a technique for automatically selecting only those parts of a program that may affect the values of interest computed at some point of interest. Different to conventional slicing, our slices are computed over a single path instead of an entire program, similar to what has been introduced as *dynamic slicing* in [18] and *path slicing* in [15]. In contrast to those approaches, we use conventional slicing criteria and leave a program's input initially unspecified. In addition, while collecting program dependencies is relatively easy at source code level, it becomes difficult at object code level when dependencies to the heap and stack are involved. The technique employed by SOCA for dealing with the program's heap and stack is a variation of the *recency abstraction* described in [1].

6 Conclusions and Future Work

This paper presented the novel SOCA technique for automatically checking memory safety of pointer-complex software. Analysing object code allows us to handle software, e.g., OS software, which is written in a mix of C and inlined assembly. Together with SOCA's symbolic execution, this simplifies pointer analysis when being confronted with function pointers, computed jumps and pointer aliasing. SOCA achieves scalability by adopting path-sensitive slicing and the efficient SMT solver Yices. While the SOCA ingredients are well-known, the way in which we integrated these for automated object code analysis is novel. Much effort went into engineering our SOCA Verifier, and extensive benchmarking showed that it performs on par with state-of-the-art software model checkers and scales well when applied to Linux device driver functions. Our verifier explores semantic niches of software, especially OS software, which currently available model checkers and testing tools do not reach. Obviously, the lack of abstraction makes SOCA less useful for programs manipulating unbounded data structures.

Future work shall be pursued along several orthogonal lines. Firstly, since device driver functions may be invoked concurrently, we plan to extend SOCA to handle concurrency. To the best of our knowledge, the verification of concurrent programs with full pointer arithmetic and computed jumps is currently not supported by any automated verification tool. Secondly, we intend to evaluate different search strategies for exploring the paths of a program, employing heuristics based on, e.g., *coverage criteria*. Thirdly, as some inputs of device drivers functions involve pointered data structures, we wish to explore whether *shape analysis* can inform SOCA in a way that reduces the number of false positives raised. Fourthly, the SOCA Verifier shall be interfaced to the *gnu debugger* so that error traces can be played back in a user-friendly form, at source code level.

Acknowledgements. We thank the anonymous reviewers for their valuable comments, especially for pointing out some recent related work.

References

1. Balakrishnan, G. and Reps, T. Recency-abstraction for heap-allocated storage. In *SAS '06*, vol. 4134 of *LNCS*, pp. 221–239. Springer, 2006.
2. 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.
3. 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. *SIGOPS Oper. Syst. Rev.*, 40(4):73–85, 2006.
4. Ball, T. and Rajamani, S. Automatically validating temporal safety properties of interfaces. In *SPIN '01*, vol. 2057 of *LNCS*, pp. 103–122. Springer, 2001.
5. Brummayer, R., Biere, A., and Lonsing, F. BTOR: Bit-precise modelling of word-level problems for model checking. In *SMT '08*, pp. 33–38. ACM, 2008.
6. Chou, A., Yang, J., Chelf, B., Hallem, S., and Engler, D. An empirical study of operating system errors. In *SOSP '01*, pp. 73–88. ACM, 2001.

7. 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.
8. Clarke, E., Kroening, D., Sharygina, N., and Yorav, K. SATABS: SAT-based predicate abstraction for ANSI-C. In *TACAS '05*, vol. 3440 of *LNCS*, pp. 570–574. Springer, 2005.
9. 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.
10. Dutertre, B. and de Moura, L. The Yices SMT solver. Technical Report 01/2006, SRI, 2006. `http://yices.csl.sri.com/tool-paper.pdf`.
11. Godefroid, P., Klarlund, N., and Sen, K. DART: Directed automated random testing. In *PLDI '05*, pp. 213–223. ACM, 2005.
12. Godefroid, P., Levin, M. Y., and Molnar, D. Automated whitebox fuzz testing. In *NDSS '08*. Internet Society, 2008.
13. Henzinger, T., Jhala, R., Majumdar, R., Necula, G., Sutre, G., and Weimer, W. Temporal-safety proofs for systems code. In *CAV '02*, vol. 2402 of *LNCS*, pp. 526–538. Springer, 2002.
14. Horwitz, S., Reps, T., and Binkley, D. Interprocedural slicing using dependence graphs. *ACM TOPLAS*, 12(1):26–60, 1990.
15. Jhala, R. and Majumdar, R. Path slicing. *SIGPLAN Not.*, 40(6):38–47, 2005.
16. Kim, M. and Kim, Y. 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.
17. King, J. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.
18. Korel, B. and Laski, J. Dynamic slicing of computer programs. *J. Syst. Softw.*, 13(3):187–195, 1990.
19. Kroening, D., Sharygina, N., Tonetta, S., Tsitovich, A., and Wintersteiger, C. Loop summarization using abstract transformers. In *ATVA '08*, vol. 5311 of *LNCS*, pp. 111–125. Springer, 2008.
20. Kroening, D. and Strichman, O. *Decision Procedures*. Springer, 2008.
21. Ku, K. Software model-checking: Benchmarking and techniques for buffer overflow analysis. Master's thesis, University of Toronto, 2008.
22. Leung, A. and George, L. Static single assignment form for machine code. In *PLDI '99*, pp. 204–214. ACM, 1999.
23. Leven, P., Mehler, T., and Edelkamp, S. Directed error detection in C++ with the assembly-level model checker StEAM. In *Model Checking Software*, vol. 2989 of *LNCS*, pp. 39–56. Springer, 2004.
24. Mühlberg, J. T. *Model Checking Pointer Safety in Compiled Programs*. PhD thesis, Department of Computer Science, University of York, 2009.
25. Mühlberg, J. T. and Lüttgen, G. BLASTing Linux code. In *FMICS '06*, vol. 4346 of *LNCS*, pp. 211–226. Springer, 2006.
26. Mühlberg, J. T. and Lüttgen, G. Verifying compiled file system code. In *SBMF '09*, vol. 5902 of *LNCS*, pp. 306–320. Springer, 2009.
27. Nethercote, N. and Seward, J. Valgrind: A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, 2007.
28. Noll, T. and Schlich, B. Delayed nondeterminism in model checking embedded systems assembly code. In *Hardware and Software: Verification and Testing*, vol. 4899 of *LNCS*, pp. 185–201. Springer, 2008.
29. Păsăreanu, C., Mehlitz, P., Bushnell, D., Gundy-Burlet, K., Lowry, M., Person, S., and Pape, M. Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In *ISSTA '08*, pp. 15–26. ACM, 2008.

30. Păsăreanu, C. and Visser, W. A survey of new trends in symbolic execution for software testing and analysis. *Software Tools for Technology Transfer*, 11(4):339–353, 2009.
31. Rational Purify. IBM Corp., `http://www.ibm.com/software/awdtools/purify/`.
32. Rungta, N., Mercer, E., and Visser, W. Efficient testing of concurrent programs with abstraction-guided symbolic execution. In *SPIN '09*, vol. 5578 of *LNCS*, pp. 174–191. Springer, 2009.
33. Schlich, B. and Kowalewski, S. [mc]square: A model checker for microcontroller code. In *ISOLA '06*, pp. 466–473. IEEE, 2006.
34. Sen, K., Marinov, D., and Agha, G. CUTE: A concolic unit testing engine for C. In *ESEC/FSE-13*, pp. 263–272. ACM, 2005.
35. Visser, W., Havelund, K., Brat, G., Joon, S., and Lerda, F. Model checking programs. *Formal Methods in System Design*, 10(2):203–232, 2003.
36. Weiser, M. Program slicing. In *ICSE '81*, pp. 439–449. IEEE, 1981.
37. Xie, Y. and Aiken, A. SATURN: A scalable framework for error detection using boolean satisfiability. *ACM TOPLAS*, 29(3):16, 2007.
38. Zitser, M., Lippmann, R., and Leek, T. Testing static analysis tools using exploitable buffer overflows from open source code. *SIGSOFT Softw. Eng. Notes*, 29(6):97–106, 2004.