Checking Properties Described by State Machines: On Synergy of Instrumentation, Slicing, and Symbolic Execution

Jiří Slabý, Jan Strejček, and Marek Trtík

Faculty of Informatics, Masaryk University
Botanická 68a, 60200 Brno, Czech Republic
{slaby,strejcek,trtik}@fi.muni.cz

Abstract. We introduce a novel technique for checking properties described by finite state machines. The technique is based on a synergy of three well-known methods: instrumentation, program slicing, and symbolic execution. More precisely, we instrument a given program with a code that tracks runs of state machines representing various properties. Next we slice the program to reduce its size without affecting runs of state machines. And then we symbolically execute the sliced program to find real violations of the checked properties, i.e. real bugs. Depending on the kind of symbolic execution, the technique can be applied as a stand-alone bug finding technique, or to weed out some false positives from an output of another bug-finding tool. We provide several examples demonstrating the practical applicability of our technique.

1 Introduction

There are several successful formalisms for description of program properties. One of the most popular is a *finite state machine (FSM)*. This formalism is simple and still flexible enough to describe many often studied program properties including locking policy in concurrent programs, null-pointer dereferences, resource allocations, and resource leaks. FSM specification is therefore used in many static program analysis tools like XGCC [24], SLAM [4], SDV [3], BLAST [5], ESP [14], or STANSE [27]. All the mentioned tools produce *false positives*, i.e. they report errors that do not correspond to any real error. We now roughly explain the basic principle of static analysis implemented in XGCC, ESP, and STANSE.

1.1 Checking FSM Properties by Static Analysis

Let us consider the state machine $SM(x)$ of Figure 1. It describes a lock manipulation including malign transitions. Intuitively, the state machine represents possible courses of states of a lock referenced by x along an execution of a program. The state of the lock is changed according to the program behavior. Whenever the program contains a statement syntactically subsuming the label of a transition, the transition is fired in the state machine. We would like to

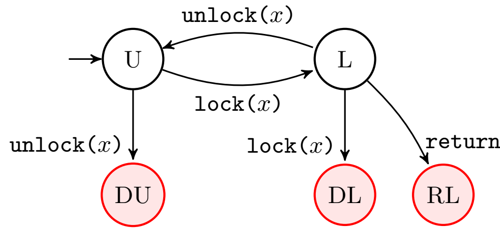

Fig. 1. State machine $SM(x)$ describing errors in manipulation with lock x. The nodes U and L refer to states *unlocked* and *locked*, respectively. The other three nodes refer to error states: DU to *double unlock*, DL to *double lock*, and RL to *return in locked state*. The initial node is U.

decide whether there exists any program execution where an instance of state machine $SM(x)$ reaches an error state for some lock in the program. Unfortunately, this is not feasible due to potentially unbounded number of executions and unbounded execution length. Hence, static analysis tools overapproximate the set of reachable state machine states.

Let us assume that we want to check the program of Figure 2 for errors specified by the state machine $SM(x)$. First, we find all locks in the program and to each lock we assign an instance of the state machine. In our case, there is only one lock pointed to by `L` and thus only one instance $SM($`L`$)$. For each program location, we compute a set overapproximating possible states of $SM($`L`$)$ after executions leading to the location. Roughly speaking, we initialize the set in the initial location to $\{U\}$ and the other sets to $\emptyset$. Then we repeatedly update the sets according to the effect of individual program statements until the fixed point is reached. The resulting sets for the program of Figure 2 are written directly in the code listing as comments.

```
 1: char *copy(char *dst, char *src, int n, int *L) {
 2:     int i, len;                                 // {U}
 3:     len = 0;                                    // {U}
 4:     if (src != NULL && dst != NULL) {           // {U}
 5:         len = n;                                // {U}
 6:         lock(L);                                // {L}
 7:     }                                           // {U,L}
 8:     i = 0;                                      // {U,L}
 9:     while (i < len) {                           // {U,L}
10:         dst[i] = src[i];                        // {U,L}
11:         i++;                                    // {U,L}
12:     }                                           // {U,L}
13:     if (len > 0) {                              // {U,L}
14:         unlock(L);                              // {DU,U}
15:     }                                           // {U,L}
16:     return dst;                                 // {U,RL}
17: }
```

Fig. 2. Function `copy` copying a source string `src` into a buffer `dst` using a lock `L` to prevent parallel writes.

As we can see, the sets contain two error states: *double unlock* after the `unlock(L)` statement and *return in locked state* in the terminal location. If we analyze the computation of the sets, we can see that the first error corresponds to executions going through lines 1,2,3,4,8, then iterating the `while`-loop and finally passing lines 13,14. These execution paths are not feasible due to the value of `len`, which is set to 0 at line 3 and assumed to satisfy `len > 0` at line 13. Hence, the first error is a *false positive*. The second error corresponds to executions passing lines 1,2,3,4,5,6,7,8, then iterating the `while`-loop and finally going through lines 13,16. All these paths are also infeasible except the one that performs zero iterations of the `while`-loop, which is the only real execution leading to the only real locking error in the program.

To sum up, static analysis tools like XGCC, ESP, and STANSE are highly flexible, fast and thus applicable to extremely large software projects (e.g. the Linux kernel). It examines all the code and finds many error reports. Unfortunately, many of the reports are false positives.[1] As manual sorting of error reports containing a pile of false positives is tedious work, the practical applicability of such tools is limited.

1.2 No False Positives with Symbolic Execution

In contrast to static analysis, test-generation tools based on *symbolic execution* do not suffer from false positives, since the checked program is actually executed (but on symbolic input instead of concrete one). A disadvantage of these tools is that they usually detect only low-level errors representing violations of programming language semantics, i.e. various types of undefined behavior or crashes. To detect violations of other program properties like locking policy, the program has to be modified such that the errors can be detected during the execution. This can be achieved, for example, by introducing a couple of `assert` statements to specific program locations. Another and even more important disadvantage of the tools is extreme computation cost of symbolic execution. In particular, programs containing loops or recursion have typically large or even infinite number of execution paths and cannot be entirely analysed by symbolic execution.

1.3 Our Contribution: A New Technique

In this paper, we introduce a new fully automatic program analysis technique offering flexibility of FSM property specification with zero false positive rate of symbolic execution. The technique symbolically executes only parts of the analysed program having impact on the checked property. The basic idea is very simple:

1. We get state machines describing some program properties. We instrument a given program with a code tracking behavior of the state machines.

[1] We note that XGCC and ESP actually use many techniques for partial elimination of false positives (see [24, 14] for details).

2. The instrumented program is then reduced using method called *slicing* [33]. The sliced program has to meet the criterion to be equivalent to the instrumented program with respect to reachability of error states of tracked state machines. Note that slicing may remove large portions of the code, including loops and function calls. Hence, an original program with an infinite number of execution paths may be reduced to a program with a finite number of execution paths.
3. Finally, we execute the sliced program symbolically to find violations of the checked property.

Our technique may be used in two ways according to the applied symbolic execution tool. If we apply a symbolic executor that prefers to explore more parts of the code (for example by exploring each program loop at most twice), we may use the technique as a general bug-finding technique reporting only real errors. On the contrary, if we use a symbolic executor exploring all execution paths, we may use our technique for classification of error reports produced by other tools (e.g. xgcc or Stanse). For each such an error report, we may instrument the corresponding code only with the state machine describing the reported error. If our technique finds the same error, it is a real one. If our technique explores all execution paths of the sliced code without detecting the error, it is a false positive. If our technique runs out of resources, we cannot decide whether the error is a real one or just a false positive.

We have developed an experimental tool implementing our technique. The tool instruments a program with a state machine describing locking errors (we use a single-purpose instrumentation so far), then it applies an interprocedural slicing to the instrumented code, and it passes the sliced code to symbolic executor Klee [9]. Our experimental results indicate that the technique can indeed classify error reports produced by Stanse applied to the Linux kernel.

We emphasize the synergy of the three known methods combined in the presented technique.

- Instrumentation of a program with a code emulating state machines provides us with simple slicing criteria: we want to preserve values of memory places representing states of state machines. Hence, the sliced program contains only the code relevant to the considered errors specified by state machines.
- Slicing may substantially reduce the size of the code, which in turn may remarkably improve performance of the symbolic execution.
- Application of symbolic execution brings us another benefit. While in standard static analysis, the state machines are associated to syntactic objects (e.g. lock variables appearing in a program), we may associate state machines to actual values of these objects. This leads to a higher precision of error detection.

The rest of the paper is organized as follows. Sections 2, 3, and 4 deal with program instrumentation, program slicing, and symbolic execution, respectively. Experimental implementation of our technique and some experimental results are discussed in Section 5. Section 6 is devoted to related work while Section 7

indicates some directions driving our future research. Finally, the last section summarizes presented results.

2 Instrumentation

In our algorithm, the purpose of the instrumentation is to insert a code implementing a state machine into the analysed program. Nonetheless, the semantics of the program being instrumented must not be changed. A result of this phase is therefore a new program that still has the original functionality but it also simultaneously updates instrumented state machines. We show the process using the state machine $SM(x)$ of Figure 1 and a program consisting of two functions: `copy` of Figure 2 and `foo` of Figure 3. The function `foo` calls `copy` twice, first with the lock `L1` and then with the lock `L2`. The locks protect writes into buffers `buf1` and `buf2` respectively. The function `foo` is a so-called *starting function*. It is a function where the symbolic execution starts.

```
char *buf1, *buf2;
int L1, L2;

void foo(char *src, int n) {
   copy(buf1, src, n, &L1);
   copy(buf2, src, n, &L2);
}
```

Fig. 3. Function `foo` forms the analysed program together with function `copy`.

The instrumentation starts by recognizing code fragments which manipulate with locks in the analysed program. More precisely, we look for all those code fragments matching edge labels of the state machine $SM(x)$ of Figure 1. The analysed program contains three such fragments, all of them in function `copy` (see Figure 2): the call to `lock` at line 6, the call to `unlock` at line 14, and the `return` statement at line 16.

Next we determine a set of all locks that are manipulated by the program. From the recognized code fragments, we find out that a pointer variable `L` in `copy` is the only program variable through which the program manipulates with locks. Using a points-to analysis, we obtain set $\{\texttt{L1}, \texttt{L2}\}$ of all possible locks the program manipulates with.

We introduce a unique instance of the state machine $SM(x)$ for each lock in the set. More precisely, we define two integer variables `smL1` and `smL2` to keep the current state of state machines $SM(\texttt{L1})$ and $SM(\texttt{L2})$, respectively. Further, we need to specify a mapping from locks to their state machines. The mapping is basically a function (preferably with constant complexity) from addresses of program objects (i.e. the locks) to addresses of corresponding state machines. Figure 4 shows an implementation of a function `smGetMachine` that maps addresses of locks `L1` and `L2` to addresses of corresponding state machines. We note that the implementation of `smGetMachine` would be more complicated if state machines are associated to dynamically allocated objects.

```
 1: const int smU  = 0;      // state U
 2: const int smL  = 1;      // state L
 3: const int smDU = 2;      // state DU
 4: const int smDL = 3;      // state DL
 5: const int smRL = 4;      // state RL
 6:
 7: const int smLOCK   = 0; // transition lock(x)
 8: const int smUNLOCK = 1; // transition unlock(x)
 9: const int smRETURN = 2; // transition return
10:
11: int smL1 = smU, smL2 = smU;
12:
13: int *smGetMachine(int *p) {
14:    if (p == &L1) return &smL1;
15:    if (p == &L2) return &smL2;
16:    return NULL;          // unreachable
17: }
18:
19: void smFire(int *SM, int transition) {
20:    switch (*SM) {
21:    case smU:
22:       switch (transition) {
23:       case smLOCK:
24:          *SM = smL;
25:          break;
26:       case smUNLOCK:
27:          assert(false); // double unlock
28:          break;
29:       default: break;
30:       }
31:       break;
32:    case smL:
33:       switch (transition) {
34:       case smLOCK:
35:          assert(false); // double lock
36:          break;
37:       case smUNLOCK:
38:          *SM = smU;
39:          break;
40:       case smRETURN:
41:          assert(false); // return in locked
42:          break;
43:       default: break;
44:       }
45:       break;
46:    default: break;
47:    }
48: }
```

Fig. 4. Implementation of the state machine (`smFire`) and its identification (`smGetMachine`).

Besides `smGetMachine`, Figure 4 contains also many constants and a function `smFire` implementing the state machine $SM(x)$. Further, Figure 4 declares variables `smL1` and `smL2` and initializes them to the initial state of the state machine. Note that we represent both states of the machine and names of transitions by integer constants. Also keep in mind that the pointer argument `SM` of `smFire` function points to an instrumented state machine, whose transition has to be fired.

It remains to instrument the recognized code fragments in the original program. For each fragment we know its related transition of the state machine and

```
  char *buf1, *buf2;
  int L1, L2;

  char *copy(char *dst, char *src, int n, int *L) {
     int i, len;
     len = 0;
     if (src != NULL && dst != NULL) {
        len = n;
*       smFire(smGetMachine(L), smLOCK);
        lock(L);
     }
     i = 0;
     while (i < len) {
        dst[i] = src[i];
        i++;
     }
     if (len > 0) {
*       smFire(smGetMachine(L), smUNLOCK);
        unlock(L);
     }
*    smFire(smGetMachine(L), smRETURN);
     return dst;
  }

  void foo(char *src, int n) {
     copy(buf1, src, n, &L1);
     copy(buf2, src, n, &L2);
  }
```

Fig. 5. Functions `foo` and `copy` instrumented by calls of `smFire` function.

we also know what objects the fragment manipulates with (if any). Therefore, we first retrieve an address of state machine related to manipulated objects (if any) by using the function `smGetMachine` and then we fire the transition by calling the function `smFire`. The instrumented version of the original program consists of the code of Figure 4 and the instrumented version of the original functions `foo` and `copy` given in Figure 5, where the instrumented lines are highlighted by `*`. Note that in our example, the instrumented state variables `smL1` and `smL2` directly correspond to the program locks `L1` and `L2` respectively. In general, however, states of a state machine of a more complex property need not necessarily correspond to values of a particular program variable. Therefore, we apply this general approach to our example too.

3 Slicing

Let us have a look at the instrumented program in Figure 5. We can easily observe, that the main part of the function `copy`, the loop copying the characters, does not affect states of the instrumented state machines. Symbolic execution of such a code is known to be very expensive, moreover in this case it is yet unneeded. Therefore, we use the slicing technique from [33] to eliminate such a code from the instrumented program.

The input of the slicing algorithm is a program to be sliced and a so-called *slicing criteria*. A slicing criterion is a pair of a program location and a set

```
 1: char *buf1, *buf2;
 2: int L1, L2;
 3:
 4: char *copy(char *dst, char *src, int n, int *L) {
 5:    int len;
 6:    len = 0;
 7:    if (src != NULL && dst != NULL) {
 8:       len = n;
 9:       smFire(smGetMachine(L), smLOCK);
10:    }
11:    if (len > 0) {
12:       smFire(smGetMachine(L), smUNLOCK);
13:    }
14:    smFire(smGetMachine(L), smRETURN);
15:    return dst;
16: }
17:
18: void foo(char *src, int n) {
19:    copy(buf1, src, n, &L1);
20:    copy(buf2, src, n, &L2);
21: }
```

Fig. 6. Functions `foo` and `copy` after slicing.

of program variables. The slicing algorithm removes program statements that do not affect any slicing criterion. More precisely, for each input data passed to both original and sliced programs, values of the variable set of each slicing criterion at the corresponding location are always equal in both programs. Our analysis is interested only in states of the instrumented automata, especially in locations corresponding to errors. Hence, the slicing criterion is a pair of a location preceding an `assert` statement in `smFire` function and the set of all variables representing current states of the corresponding state machines. The slicing criteria then comprises all such pairs.

In the instrumented program of Figures 4 and 5, we want to preserve variables `smL1` and `smL2`. We put slicing criteria into the lines of code detecting transitions of state machines into error states. In other words, the slicing criteria for our running example are pairs (27,{`smL1`,`smL2`}), (35,{`smL1`,`smL2`}), and (41,{`smL1`,`smL2`}), where the numbers refer to lines in the code of Figure 4. The result of the slicing procedure is presented in Figures 4 and 6 (the code in the former Figure shall not be changed by the slicing). Note that the sliced code contains neither the `while`-loop nor the `lock` and `unlock` commands.

It is important to note that some slicing techniques, including the one in [33] that we use, do not consider inputs for which the original program does not halt. As a result, an input causing an infinite run of the original program can induce a finite run in the sliced program. Moreover, the finite run can contain locations not visited by the infinite run. This is the only principal source of potential false positives in our technique.

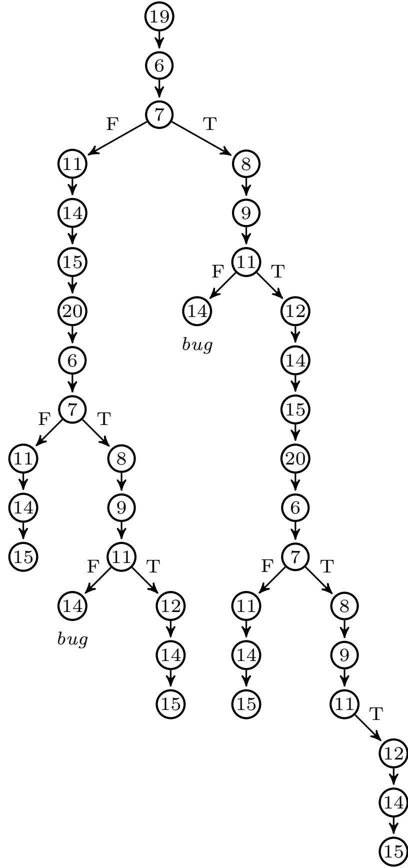

Fig. 7. Symbolic execution tree of the sliced program of Figure 6.

4 Symbolic Execution

This is the final phase of our technique. We symbolically execute the sliced program from the entry location of the starting function. Symbolic execution explores real program paths. Therefore, if it reaches some of the assertions inside function `smFire`, then we have found a bug.

Our running example nicely illustrates the crucial role of slicing to feasibility of symbolic execution. Let us first consider symbolic execution of the original program. It starts at the entry location of the function `foo`. The execution eventually reaches the function `copy`. Note that value of the parameter `n` is symbolic. Therefore, symbolic execution will fork into two executions each time we reach line 9 of Figure 2. One of the executions skips the loop at lines 9–12, while the other enters it. If we assume that the type of `n` is a 32-bit integer, then the symbolic execution of one call of `copy` explores more then 2^{31} real paths.

By contrast, the sliced program does not contain the loop, which generated the huge number of real paths. Therefore, a number of real paths explored by the symbolic execution is exactly 6. Figure 7 shows the symbolic execution tree of the sliced program of Figure 6. We left out vertices corresponding to lines in called functions `smGetMachine` and `smFire`. Note that although the parameter `n` has a symbolic value, it can only affect the branching at line 11. Moreover, the parameter `L` always has a concrete value. Therefore, we do not fork symbolic execution at branchings inside functions `smGetMachine` and `smFire`. Two of the explored paths are marked with the label *bug*. These paths reach the second assertion in function `smFire` (see Figure 4) called from line 14 of the sliced program. In other words, the paths are witnesses that we can leave the function `copy` in a locked state. The remaining explored paths of Figure 7 miss the assertions in the function `smFire`. It means that the original program contains only one locking error, namely *return in locked state*.

It might be the case for some program and checked property, that the sliced code still contains loops. Then the subsequent symbolic execution can be very costly due to the well-known path explosion problem. Fortunately, there have been done some work tackling the problem [18, 19, 22, 30, 34].

5 Implementation and Experimental Results

To verify applicability of the presented technique, we have developed an experimental implementation. Our experimental tool works with programs in C and, for the sake of simplicity, it detects only locking errors described by a state machine very similar to $SM(x)$ of Figure 1. The instances of the state machine are associated with arguments of `lock` and `unlock` function calls. Note that the technique currently works only for the cases where a lock is instantiated only once during the run of the symbolic executor, which is the most frequent case. However we plan to add a support even for the rest. The main part of our implementation is written in three modules for the LLVM framework [35], namely `Prepare`, `Slicer`, and `Kleerer`. The framework provides us with a C compiler CLANG. We also use an existing symbolic executor for LLVM called KLEE [9].

Instrumentation of a given program proceeds in two steps. Using a C preprocessor, the original program is instrumented with function calls `smFire` located just above statements changing states of state machines. The program is then translated by CLANG into LLVM bytecode [35]. Optimizations are turned off as required by KLEE. The rest of the instrumentation (e.g. adding global variables and changing the code to work with them) is done on the LLVM code using the module `Prepare`.

The module `Slicer` implements a variant of the inter-procedural slicing algorithm by Weiser [33]. To guarantee correctness and to improve performance of slicing, the algorithm employs points-to analysis by Andersen [2].

The module `Kleerer` performs a final processing of the sliced bytecode before it is passed to KLEE. In particular, the module adds to the bytecode a function `main` that calls a *starting function*. The `main` function also allocates a symbolic

File Function	Running Time (s)					Sliced	Result	Factual State
	Comp.	Instr.	Slic.	SE	Total			
`fs/jfs/super.c` `jfs_quota_write`	1.25	0.18	0.15	5.09	6.67	67.8% 369/119	BUG	BUG
`drivers/net/qlge/qlge_main.c` `qlge_set_mac_address`	2.70	0.72	26.75	13.28	43.45	66.5% 1333/447	BUG	BUG
`drivers/hid/hidraw.c` `hidraw_read`	1.06	0.18	0.14	Timeout		67.0% 666/220	TO	BUG
`drivers/net/ns83820.c` `queue_refill`	1.76	0.29	1.72	0.62	4.39	72.9% 1212/329	FP	FP
`drivers/usb/misc/sisusbvga/sisusb_con.c` `sisusbcon_set_palette`	1.50	0.24	0.27	17.19	19.20	76.0% 2936/705	FP	FP
`fs/jffs2/nodemgmt.c` `jffs2_reserve_space`	1.04	0.18	0.22	Timeout		46.8% 677/360	TO	FP
`kernel/kprobes.c` `pre_handler_kretprobe`	0.32	0.09	0.51	2.43	3.35	66.3% 202/68	ME	FP

Table 1. Experimental results. The table presents running time of preprocessing and compilation (**Comp.**), instrumentation including points-to analysis (**Instr.**), slicing (**Slic.**), symbolic execution (**SE**), and the total running time. The column **Sliced** presents the ratio of instructions sliced away from the instrumented LLVM code and the exact number of instructions before/after slicing. The column **Result** specifies the result of our tool: BUG means that the tool found a real error, FP means that the analysis finished without error found (i.e. the original error report is a false positive), TO that the symbolic execution did not finish in time and ME denotes an occurrence of memory error. The last column specifies the factual state of the error report.

memory for each parameter of the starting function. Size of the allocated memory is determined by the parameter type. Plus, when the parameter is a pointer, the size is multiplied by 4000. For example, 4 bytes are allocated for an integer and 16000 bytes for an integer pointer. Further, for the pointer case, we pass a pointer to the middle of the allocated memory (functions might dereference memory at negative index). The idea behind is explained in [28]. Finally, the resulting bytecode is symbolically executed by KLEE. If a symbolic execution touches a memory out of the allocated area, we get a *memory error*. To remedy this inconvenience, we plan to implement the same on-demand memory handling UCKLEE [28] does.

5.1 Experiments

We have performed our experiments on several functions of the Linux kernel 2.6.28, where the static analyzer STANSE reported some error. More precisely, STANSE reported an error trace starting in these functions. We consulted the errors with kernel developers to sort out which are false positives and which are

real errors. All the selected functions (and all functions transitively called from them) contain no assembler (in some cases, it has been replaced by an equivalent C code) and no external function calls after slicing.

We ran our experimental tool on these functions. All tests were performed on a machine with an Intel E6850 dual-core processor at 3 GHz and 6 GiB of memory, running Linux. We specified KLEE parameters to time out after 10 seconds spent in an SMT solver and after 300 seconds of an overall running time. Increasing these times brings no real effect in our environment. We do not pass optimize option for KLEE because it causes KLEE to crash for most of the input.

Table 1 presents results of our tool on selected functions. The table shows compilation, instrumentation, slicing, symbolic execution, and the overall running time. Further, the table presents the ratio of instructions that were sliced away from the instrumented LLVM code. The last two columns specify the results of our analysis and the real state confirmed by kernel developers. The table clearly shows that the bottleneck of our technique is the symbolic execution. However, if we did not slice the code, the only function completely executed in time would be `sisusbcon_set_palette`, computed in 20.64s.

Although the results have no statistical significance, it is clear that the technique can in principle classify error reports produced by other tools like STANSE. If our technique reports an error, it is a real one. If it finishes the analysis without any error detected, the original error report is a false positive. The analysis may also not finish in a given time, which is usually caused by loops in the sliced code. Finally, it may report a memory error mentioned above.

6 Related Work

There are many tools checking properties described by finite state machines. They produce both kinds of reports, real error as well as false positives. The technique of XGCC presented in [11, 12, 16, 24] found a thousands of bugs in real system code. It provides a language METAL for easy description of properties to be checked. XGCC suffers from false positives despite usage of false positive suppression algorithms like killing variables and expressions, synonyms, false path pruning, and others. Besides the suppression algorithms, bug-reports from the tool are further ranked according to their probability of being real errors. There are generic and statistical ranking algorithms ordering bug-reports. An extension introduced in [17] provides an automatic inference of some temporal properties based on statistical analysis of assumed programmer's beliefs. The ESP [14] technique uses a similar language to METAL for properties description. It implements an interprocedural dataflow algorithm based on [29] for error detection and an abstract simulation pruning algorithm for false positives suppression. STANSE [27], a static analysis tool also uses state machines for description of checked program properties. The description is based on parametrised abstract syntax trees. Finally, CEGAR [13] based tools like SLAM [4], SDV [3], or BLAST [5], do not produce false positives, in theory. However, to achieve an ap-

propriate efficiency and scalability for a practical use, the implementation of the CEGAR loop is typically unsound.

Program analysis tools based on symbolic execution [25] mainly discover low-level bugs like division by zero, illegal memory access, assertion failure etc. These tools typically do not have problems with false positives, but they have problems with scalability to large programs. There has been developed a lot of techniques improving the scalability to programs used in practice. Modern techniques are mostly hybrid: they usually combine symbolic execution with concrete one [20, 21, 31]. There are also hybrid techniques combining symbolic execution with a complementary static analysis [22, 26]. Symbolic execution can be accelerated by a compositional approach based on function summaries [1, 18]. Another approach to effective symbolic execution introduced in [8–10] is based on recording of already seen behavior and pruning its repetition. There is an orthogonal line of research which tries to improve the symbolic execution for programs with some special types of inputs. Some techniques deal with programs manipulating strings [7, 34], and some other techniques reduce input space using a given input grammar [19, 30].

The interprocedural static slicing was introduced by Weiser [33]. But nowadays, there are many different approaches to program slicing. They are surveyed by several authors [6, 15, 32]. Applications of slicing include program debugging, reverse engineering and regression testing [23].

7 Future Work

Our future work has basically three independent directions.

First, we plan to run our tool to classify all lock-related error reports produced by Stanse on the Linux kernel. The results should provide a better image of practical applicability of the technique. To get a relevant data, we should solve some practical issues like a correct detection of starting functions, automatic replacement of assembler, treatment of external function calls, etc. We should also implement an on-demand memory allocation to Klee as discussed in Section 5 or use a different executor.

The second direction is to adopt or design some convenient way for specification of arbitrary state machines. It may be a dedicated language similar to Metal [12]. Then we plan to implement an instrumentation treating these state machines. In particular, the instrumentation should correctly handle state machines associated with dynamically allocated objects.

Finally, we would also like to examine performance of our technique as a stand-alone error-detection tool. To this point, we have to use a symbolic executor aiming for maximal code coverage. In particular, such an executor has to suppress execution paths that differ from explored paths only in number of loop iterations. Unfortunately, we do not know about any publicly available symbolic executor of this kind. However, it seems that UcKlee [28] (which is not public as of now) has been designed for a similar purpose.

8 Conclusion

We have presented a novel technique combining three standard methods (instrumentation, slicing, and symbolic execution) to check program properties given in form of finite state machines. We have discussed a synergy of the three methods. Moreover, our experimental results indicate that the technique can recognize some false positives and some real errors in error reports produced by other error-detection tools.

Acknowledgements The authors are supported by the Czech Science Foundation, project No. P202/12/G061.

References

1. S. Anand, P. Godefroid, and N. Tillmann. Demand-driven compositional symbolic execution. In *Proceedings of TACAS*, pages 367–381. Springer, 2008.
2. L.O. Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, DIKU, University of Copenhagen (report 94/19), 1994.
3. T. Ball, E. Bounimova, R. Kumar, and V. Levin. SLAM2: Static driver verification with under 4% false alarms. In *Proceedings of FMCAD*, pages 35–42. IEEE, 2010.
4. T. Ball, V. Levin, and S. K. Rajamani. A decade of software model checking with SLAM. *Journal on Commun. ACM*, 54(7), 2011.
5. D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker BLAST. *Journal on Software Tools for Technology Transfer 9(5), 505-525*, 2007.
6. D. W. Binkley and K. B. Gallanger. Program slicing. *Advances in Computers*, 43, 1996.
7. N. Bjørner, N. Tillmann, and A. Voronkov. Path feasibility analysis for string-manipulating programs. In *Proceedings of TACAS*, pages 307–321. Springer, 2009.
8. P. Boonstoppel, C. Cadar, and D. Engler. RWset: attacking path explosion in constraint-based test generation. In *Proceedings of TACAS*, pages 351–366. Springer, 2008.
9. C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of OSDI*, pages 209–224. USENIX Association, 2008.
10. C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically generating inputs of death. *ACM Trans. Inf. Syst. Secur.*, 12:1–38, 2008.
11. B. Chelf, S. Hallem, and D. Engler. How to write system-specific, static checkers in metal. In *Proceedings of PASTE*, pages 51–60. ACM, 2002.
12. A. Chou, B. Chelf, D. Engler, and M. Heinrich. Using meta-level compilation to check FLASH protocol code. *ACM SIGOPS Oper. Syst. Rev.*, 34(5):59–70, 2000.
13. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proceedings of CAV*, pages 154–169. Springer, 2000.
14. M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *Proceedings of PLDI*, volume 37,5 of *SIGPLAN*. ACM Press, 2002.
15. A. De Lucia. Program slicing: methods and applications. In *Proceedings of SCAM*, pages 142–149. IEEE Computer Society, 2001.

16. D. Engler, B. Chelf, A. Chou, and Hallem S. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of OSDI*, pages 1–16. ACM, 2000.
17. D. Engler, D. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of SOSP*, pages 57–72. ACM, 2001.
18. P. Godefroid. Compositional dynamic test generation. In *Proceedings of POPL*, pages 47–54. ACM, 2007.
19. P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based whitebox fuzzing. In *Proceedings of PLDI*, pages 206–215. ACM, 2008.
20. P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *Proceedings of PLDI*, pages 213–223. ACM, 2005.
21. P. Godefroid, M. Y. Levin, and D. A. Molnar. Active property checking. In *Proceedings of EMSOFT*, pages 207–216. ACM, 2008.
22. P. Godefroid, A. V. Nori, S. K. Rajamani, and S. D. Tetali. Compositional may-must program analysis: unleashing the power of alternation. In *Proceedings of POPL*, pages 43–56. ACM, 2010.
23. R. Gupta, M. J. Harrold, and M. L. Soffa. An approach to regression testing using slicing. In *Proceedings of ICSM*, pages 299–308. IEEE, 1992.
24. S. Hallem, B. Chelf, Y. Xie, and D. R. Engler. A system and language for building system-specific, static analyses. In *Proceedings of PLDI*, pages 69–82. ACM, 2002.
25. J. C. King. Symbolic execution and program testing. *Communications of ACM*, 19(7):385–394, 1976.
26. A. V. Nori, S. K. Rajamani, S. Tetali, and A. V. Thakur. The Yogi project: Software property checking via static analysis and testing. In *Proceedings of TACAS*, pages 178–181. Springer, 2009.
27. J. Obdržálek, J. Slabý, and M. Trtík. STANSE: Bug-finding Framework for C Programs. In *Proceeding of MEMICS*, volume 7119, pages 167–178. Springer, 2011.
28. D. Ramos and D. Engler. Practical, low-effort equivalence verification of real code. In *Proceedings of CAV*, pages 669–685. Springer, 2011.
29. T. Reps, S. Horowitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of POPL*, pages 49–61. ACM, 1995.
30. P. Saxena, P. Poosankam, S. McCamant, and D. Song. Loop-extended symbolic execution on binary programs. In *Proceedings of ISSTA*, pages 225–236. ACM, 2009.
31. K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *Proceedings of ESEC/FSE*, volume 30, pages 263–272. ACM, 2005.
32. F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–189, 1995.
33. Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.
34. R. G. Xu, P. Godefroid, and R. Majumdar. Testing for buffer overflows with length abstraction. In *Proceedings of ISSTA*, pages 27–38. ACM, 2008.
35. LLVM. `http://llvm.org/`.