On Synergy of Metal, 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 finding real errors in programs. The technique is based on a synergy of three well-known methods: metacompilation, slicing, and symbolic execution. More precisely, we instrument a given program with a code that tracks runs of state machines representing various kinds of errors. Next we slice the program to reduce its size without affecting runs of state machines. And then we symbolically execute the sliced program. 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

The title of this paper refers to two popular bug-finding techniques: metacompilation and symbolic execution. The two techniques use completely different principles leading to different advantages and disadvantages.

Metacompilation \[10,26\] is a static analysis technique looking for various kinds of errors specified by state machines. We explain the technique with use of the state machine \(SM(x)\) of Figure 1 which describes errors in lock manipulation. 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 a transition of the state machine if the execution performs a program statement syntactically subsuming the label of the transition. We would like to decide whether there exists any program execution where an instance of state machine \(SM(x)\) assigned to some lock of the analyzed program reaches an error state. Unfortunately, this is not feasible due to potentially unbounded number of executions and unbounded execution length. Hence, we use static analysis to overapproximate the set of reachable states of state machines.

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 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.

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

\[
\text{Fig. 2. Function \texttt{copy} copying a source string} \ src \ \text{into a buffer} \ dst \ \text{using a lock} \ L \ \text{to prevent parallel writes.}
\]

```c
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) {
10:         dst[i] = src[i];   // \{U,L\}
11:         i++;              // \{U,L\}
12:     }                     // \{U,L\}
13:     if (len > 0) {
14:         unlock(L);       // \{DU,U\}
15:     }                    // \{U,L\}
16:     return dst;          // \{U,RL\}
17: }
```

\[
\text{Fig. 1. State machine} \ SM(x) \ \text{describing errors in manipulation with lock} \ x. \ \text{The nodes} \ U \ \text{and} \ L \ \text{refer to states} \ \text{unlocked} \ \text{and} \ \text{locked}, \ \text{respectively. The other three nodes refer to error states: DU to double unlock, DL to double lock, and RL to return in locked state. The initial node is} \ U.
\]
executions passing lines 1,2,3,4,5,6,7,8, then iterating the \texttt{while}-loop and finally going through lines 13,16. All these paths are also infeasible except the one that performs zero iterations of the \texttt{while}-loop, which is the only real execution leading to the only real locking error in the program.

To sum up, metacompilation is highly flexible, fast, and thus applicable on extremely large software projects (e.g. the Linux kernel). It examines all the code and finds many error reports. Unfortunately, some of the reports are false positives. The main source of false positives is related to the fact that the analysis does not work with data values. In particular, the analysis does not track connections between variable values and states of state machines. A drawback of this approach may be illustrated by the \textit{double unlock} false positive detected in the program of Figure 2 because the analysis does not know that the condition at line 13 holds only if the state machine $SM(L)$ is in state $L$.

At this point, we would like to emphasize that metacompilation actually uses a more sophisticated algorithm enriched with many techniques for partial elimination of false positives (see [26] for details). Metacompilation employs a dedicated language for description of state machines called \textsc{Metal}. The idea of error specification using state machines appears in several tools including the original implementation of metacompilation called \textsc{xgcc} [26], \textsc{Esp} [11] or \textsc{Stanse} [29].

In contrast to metacompilation, \textit{symbolic execution} [27] analyzes each execution path separately. In contrast to standard execution, symbolic execution replaces input data by symbols representing arbitrary values. Executed statements then manipulate expressions over the symbols rather than exact values. For each execution path, symbolic execution builds a formula called \textit{path condition}, which is a necessary and sufficient condition on input data to drive the execution along the path. Whenever a path condition becomes unsatisfiable, the symbolic execution of this path is aborted as the path is unfeasible. The main advantage of symbolic execution is that it works only with feasible executions (assuming that we can decide satisfiability of a path condition) and hence it does not report any false positives. A minor disadvantage is that implementations of this technique usually detect only low-level errors leading to crash. To detect a specific kind of error, the program has to be modified to reduce the error to a detected one (typically violation of an \texttt{assert} statement). The main disadvantage of the technique is its high computation cost. In particular, programs containing loops or recursion have typically large or even infinite number of execution paths and cannot be completely analyzed by symbolic execution. Hence, symbolic execution usually explores only a part of an analyzed program.

In this paper, we introduce a new technique offering a flexibility of metacompilation and zero false positive rate of symbolic execution. The basic idea is very simple: we use the concept of state machines to get flexibility in error specification. Then we instrument a given program with a code for tracking behaviors of the state machines. The instrumented program is then reduced using the slicing method introduced in [38]. 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 big 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. Finally, we execute the sliced program symbolically.

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, it can explore only the execution paths iterating
each program loop at most twice), we may use the technique as a general bug-
finding technique reporting only real errors. Note that this approach may miss
errors appearing only on unexplored paths. On the contrary, if we use a sym-
bolic executor exploring all execution paths, we may use our technique for basic
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 that 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 er-
ners (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 execu-
tor KLEE [7]. 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.

– The errors are specified by state machines (inspired by metacompilation)
and a given program instrumented with a code emulating the state ma-
chines. This provides us 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.

– 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 meta-
compilation, 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,
which may potentially result in a detection of real errors missed by meta-
compilation.

The rest of the paper is organized as follows. Sections 2, 3, and 4 deal with
program instrumentation, slicing, and symbolic execution, respectively. Experimental
implementation of our technique and some experimental results are dis-
ssued 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

The purpose of the instrumentation phase of our algorithm is to insert a code implementing a state machine into the analysed program. Nevertheless, 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 and it simultaneously updates instrumented state machines. We show the process using the state machine \( SM(x) \) of Figure 1 and the program consisting of a function \( \text{foo} \) of \( x \) and the function \( \text{copy} \) of Figure 2. The function \( \text{foo} \) calls the function \( \text{copy} \) twice, first with the lock \( L1 \) and then with the lock \( L2 \). The locks guard writes into buffers \( \text{buf1} \) and \( \text{buf2} \) respectively. The function \( \text{foo} \) is a so-called starting function. It is a function where the symbolic execution starts.

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

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

Fig. 3. Function \( \text{foo} \) forms the analysed program together with function \( \text{copy} \).

The instrumentation starts by recognizing the code fragments in the analysed program which manipulate with locks. 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 \( \text{copy} \) (see Figure 2): the call to \( \text{lock} \) at line 6, the call to \( \text{unlock} \) 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 \( \text{copy} \) is the only program variable through which the program manipulates with locks. Using a points-to analysis, we obtain the set \( \{L1, L2\} \) 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 \( \text{smL1} \) and \( \text{smL2} \) for keeping current state of state machines \( SM(L1) \) and \( SM(L2) \), 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 related state machines. Figure 4 shows an implementation of a function \( \text{smGetMachine} \) that maps addresses of locks \( L1 \) and \( L2 \) to addresses of related state machines. We note that the implementation of \( \text{smGetMachine} \) would be more complicated if state machines are associated to dynamically allocated objects.
const int smU = 0;  // state U
const int smL = 1;  // state L
const int smDU = 2;  // state DU
const int smDL = 3;  // state DL
const int smRL = 4;  // state RL

const int smLOCK = 0;  // transition lock(x)
const int smUNLOCK = 1;  // transition unlock(x)
const int smRETURN = 2;  // transition return

int smL1 = smU, smL2 = smU;

int *smGetMachine(int *p) {
    if (p == &L1) return &smL1;
    if (p == &L2) return &smL2;
    return NULL;  // unreachable
}

void smFire(int *SM, int transition) {
    switch (*SM) {
    case smU:
        switch (transition) {
        case smLOCK:
            *SM = smL;
            break;
        case smUNLOCK:
            assert(false);  // double unlock
            break;
        default: break;
        }
        break;
    case smL:
        switch (transition) {
        case smLOCK:
            assert(false);  // double lock
            break;
        case smUNLOCK:
            *SM = smU;
            break;
        case smRETURN:
            assert(false);  // return in locked
            break;
        default: break;
        }
        break;
    default: break;
    }
}

Fig. 4. Implementation of the state machine (smFire) and its identification (smGetMachine).
Figure 4 contains also many constants and a function \texttt{smFire} implementing
the state machine \texttt{SM}(x). Further, Figure 4 declares variables \texttt{smL1} and
\texttt{smL2} and initialize them to the initial state of the state machine. Note that we
represent both states of the machine and names of transitions by integer con-
stants. Also note that the pointer argument \texttt{SM} of \texttt{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 pro-
gram. For each fragment we know its related transition of the state machine and
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 \texttt{smGetMachine} and then we fire the transition by calling
the function \texttt{smFire}. The instrumented version of the original program consists
of the code of Figure 4 and the instrumented version of the original functions
\texttt{foo} and \texttt{copy} given in Figure 5, where the instrumented lines are highlighted
by *.

```c
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(src, buf1, n, &L1);
    copy(src, buf2, n, &L2);
}
```

Fig. 5. Functions \texttt{foo} and \texttt{copy} instrumented by calls of \texttt{smFire} function.
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, i.e. the loop copying the characters, does not affect states of the instrumented state machines. Symbolic execution of such a code is very expensive. Therefore, we use the slicing technique \[38\] 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 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 consists of 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

Fig. 6. Functions foo and copy after slicing.
of state machines into error states. In other words, the slicing criteria for our running example are pairs \((27, \{\text{smL1, smL2}\}), (35, \{\text{smL1, smL2}\}), (41, \{\text{smL1, smL2}\})\), where the number refers 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 changed by the slicing). Note that the sliced code contains neither the \texttt{while}-loop nor the \texttt{lock} and \texttt{unlock} commands.

It is important to note that some slicing techniques, including the one in \cite{38} that we use, do not consider inputs for which the original program does not halt. Therefore, there is no way to guarantee that a sliced program will fail to halt whenever the original program fails to halt. This is the only principal source of potential false positives in our technique.

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 \texttt{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 \texttt{foo}. The execution eventually reaches the function \texttt{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 \texttt{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 \texttt{smGetMachine} and \texttt{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 \texttt{smGetMachine} and \texttt{smFire}. Three of the explored paths are marked with the label \texttt{bug}. These paths reach the second assertion in function \texttt{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 \texttt{copy} in a locked state. The remaining explored paths of Figure 7 miss the assertions in the function \texttt{smFire}. It means that the original program contains only one locking error, namely \texttt{return in locked state}.

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

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. It works on a vast majority of the code we used. 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 [41], namely Prepare, Slicer, and Kleerer. The framework provides us with the C compiler clang. We also use an existing symbolic executor for LLVM called Klee [7].

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 [11]. 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 [38]. 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 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 [31]. 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 [31] 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.

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.
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. 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.

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

A description of program properties in METAL language and meta-level compilation is discussed in [9,10,13,26]. The technique presented in [10] found a thousands of bugs in real system code. It provides an easy description of properties to be checked for and a fast analysis. Nevertheless, it suffers from false positives. Since false positive rate has huge impact on practical usability, an important part of the technique are 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 [14] provides an
automatic inference of some temporal properties based on statistical analysis of
assumed programmer’s beliefs. The ESP technique uses a similar language
to MetaL for properties description. It implements an interprocedural dataflow
algorithm based on for error detection and an abstract simulation pruning
algorithm for false positives suppression. STANSE, a static analysis tool also
uses state machines for description of checked program properties. The descrip-
tion is based on parametrised abstract syntax trees. Although this tool found
hundreds of real bugs in the Linux kernel, it suffers from a high false positive
rate since its false positive suppression algorithms are very limited.

Program analysis tools based on symbolic execution 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. There are also hybrid techniques combining symbolic execution with a complementary static analysis. Symbolic execution can be accelerated by a compositional approach based on function summaries. Another approach to effective symbolic execution introduced in is based on recording of already seen behaviour and pruning its repetition. The following techniques focus on reaching a specific program location. Fitnex, a search strategy implemented in Pex, guides a path exploration to a particular target location using fitness function. The function measures how close an already discovered feasible path is to the target. The LESE approach introduces symbolic variables for the number of times each loop was executed. The symbolic variables are linked with features of a known grammar generating inputs. Using these links, the grammar can control the numbers of loop iterations performed on a generated input. A technique presented in analyses loops on-the-fly, i.e. during simultaneous concrete and symbolic executions of a program for a concrete input. The loop analysis infers variables that are modified by a constant value in each loop iteration. These variables are used to build loop summaries expressed in a form of pre and post conditions. An algorithm constructs a nontrivial necessary condition on input values to drive the program execution to a given location. A technique presented in introduces a pair of counters for two different paths around loop for each recurrent variable. Each counter keeps an information about the number of iterations around one path since the last iteration around the other one. Finally, 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, and some other techniques reduce input space using a given input grammar.

The interprocedural static slicing was introduced by Weiser. But nowadays, there are many different approaches to program slicing. They are surveyed by several authors. Applications of slicing include program debugging, reverse engineering and regression testing.
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. 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 [31] (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: specification of errors with state machines, slicing, and symbolic execution. We currently do not know about any technique combining arbitrary two of the three methods. 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 Jan Strejček has been supported by The Czech Science Foundation (GACR), grants No. P202/10/1469 and P202/12/G061. Jiří Slabý and Marek Trtík have been supported by The Czech Science Foundation (GACR), grant No. 102/09/H042.

References

1. S. Anand, P. Godefroid, and N. Tillmann. Demand-driven compositional symbolic execution. In Proceedings of the TACAS’08, LNCS, pages 367–381. Springer, 2008.
2. L.O. Andersen. Program analysis and specialization for the C programming language. PhD thesis, University of Copenhagen, 1994.
3. N. E. Beckman, A. V. Nori, S. K. Rajamani, R. J. Simmons, S. Tetal, and A. V. Thakur. Proofs from tests. In Proceedings of the ISSTA ’08, ISSTA ’08, pages 3–14. ACM, 2008.
4. D. W. Binkley and K. B. Gallanger. Program slicing. In *Advances in Computers*. Academic Press San Diego, 1996.

5. N. Bjørner, N. Tillmann, and A. Voronkov. Path feasibility analysis for string-manipulating programs. In *Proceedings of the TACAS ’09*, TACAS ’09, pages 307–321. Springer-Verlag, 2009.

6. P. Boonstoppel, C. Cadar, and D. Engler. RWset: attacking path explosion in constraint-based test generation. In *Proceedings of the TACAS’08/ETAPS’08*, TACAS’08/ETAPS’08, pages 351–366. Springer-Verlag, 2008.

7. C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of OSDI’08*, OSDI’08, pages 209–224. USENIX Association, 2008.

8. 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.

9. B. Chelf, S. Hallem, and D. Engler. How to write system-specific, static checkers in metal. In *Proceedings of the PASTE 2002*, PASTE ’02, pages 51–60. ACM, 2002.

10. 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.

11. M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *PLDI 2002*, volume 37,5 of *SIGPLAN*. ACM Press, 2002.

12. A. De Lucia. Program slicing: methods and applications. In *Source Code Analysis and Manipulation*, pages 142–149. IEEE, 2001.

13. D. Engler, B. Chelf, A. Chou, and Hallem S. Checking system rules using systemspecific, programmer-written compiler extensions. In *Proceedings of the OSDI 2000*, OSDI 2000, pages 1–16, 2000.

14. 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 the ACM Symposium on Operating Systems Principles*, SOSP ’01. ACM, 2001.

15. P. Godefroid. Compositional dynamic test generation. In *Proceedings of the POPL ’07*, POPL ’07, pages 47–54. ACM, 2007.

16. P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based whitebox fuzzing. In *Proceedings of the PLDI ’08*, pages 206–215. ACM, 2008.

17. P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *Proceedings of the PLDI ’05*, pages 213–223. ACM, 2005.

18. P. Godefroid, M. Y. Levin, and D. A. Molnar. Active property checking. In *Proceedings of the EMSOFT ’08*, pages 207–216. ACM, 2008.

19. P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *Proceedings of the Network Distributed Security Symposium (NDSS)*, pages 151–166, 2008.

20. P. Godefroid, X. Levin, and X. Elkarablieh. Precise pointer reasoning for dynamic test generation. In *Proceedings of the ISSTA’09*, ISSTA ’09, pages 129–140. ACM, 2009.

21. P. Godefroid and D. Luchau. Automatic partial loop summarization in dynamic test generation. In *Proceedings of the ISSTA ’11*, pages 23–33. ACM, 2011.

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 the POPL ’10*, pages 43–56. ACM, 2010.

23. B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. SYNERGY: a new algorithm for property checking. In *Proceedings of the SIGSOFT ’06/FSE-14*, pages 117–127. ACM, 2006.
24. A. Gupta, R. Majumdar, and A. Rybalchenko. From tests to proofs. In Proceedings of the TACAS ’09, pages 262–276. Springer-Verlag, 2009.
25. R. Gupta, M. J. Harrold, and M. L. Soffa. An approach to regression testing using slicing. In Proceedings of IEEE Conference on Software Maintenance, pages 299–308. IEEE Computer Society Press, 1992.
26. S. Hallem, B. Chelf, Y. Xie, and D. R. Engler. A system and language for building system-specific, static analyses. In Proceedings of the PLDI 2002, PLDI ’02, pages 69–82. ACM, 2002.
27. J. C. King. Symbolic execution and program testing. Communications of the ACM, 19(7):385–394, 1976.
28. 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 the TACAS ’09, pages 178–181. Springer-Verlag, 2009.
29. J. Obdržálek, J. Slabý, and M. Trtík. STANSE: Bug-finding Framework for C Programs. In Proceeding of MEMICS 2011, LNCS. Springer, 2012. To appear.
30. J. Obdržálek and M. Trtík. Efficient loop navigation for symbolic execution. In Proceedings of the ATVA ’11, pages 453–462. LNCS, 2011.
31. D. Ramos and D. Engler. Practical, low-effort equivalence verification of real code. In Proceedings of Computer Aided Verification, CAV’11, pages 669–685. Springer-Verlag, 2011.
32. T. Reps, S. Horowitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In Proceedings of the POPL 1995, pages 49–61. ACM, 1995.
33. P. Saxena, P. Poosankam, S. McCamant, and D. Song. Loop-extended symbolic execution on binary programs. In Proceedings of the ISSTA ’09, pages 225–236. ACM, 2009.
34. K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In Proceedings of the ESEC/FSE-13, pages 263–272. ACM, 2005.
35. J. Strejček and M. Trtík. Abstracting path conditions. Technical report, 2011. http://www.fi.muni.cz/~xtrtik2/apc.pdf.
36. N. Tillmann and J. de Halleux. Pex – white box test generation for .NET. In Proceedings of the TAP ’08, LNCS, pages 134–153. Springer, 2008.
37. F. Tip. A survey of program slicing techniques. JOURNAL OF PROGRAMMING LANGUAGES, 3:121–189, 1995.
38. Mark Weiser. Program slicing. IEEE Transactions on Software Engineering, 10(4):352–357, 1984.
39. X. T. Xie, N. Tillmann, J. de Halleux, and W. Schulte. Fitness-guided path exploration in dynamic symbolic execution. In Proceedings of the DSN ’09, pages 359–368, 2009.
40. R. G. Xu, P. Godefroid, and R. Majumdar. Testing for buffer overflows with length abstraction. In Proceedings of the ISSTA ’08, pages 27–38. ACM, 2008.
41. LLVM. http://llvm.org/