An Approach to Static-Dynamic Software Analysis

Pablo González de Aledo[1] and Ralf Huuck[2]

[1] University of Cantabria
pabloga@teisa.unican.es ⋆
[2] DATA61⋆⋆ and UNSW, Sydney, Australia
ralf.huuck@nicta.com.au

Abstract. Safety-critical software in industry is typically subjected to both dynamic testing as well as static program analysis. However, while testing is expensive to scale, static analysis is prone to false positives and/or false negatives. In this work we propose a solution based on a combination of static analysis to zoom into potential bug candidates in large code bases and symbolic execution to confirm these bugs and create concrete witnesses. Our proposed approach is intended to maintain scalability while improving precision and as such remedy the short comings of each individual solution. Moreover, we developed the SEEKFAULT tool that creates local symbolic execution targets from static analysis bug candidates and evaluate its effectiveness on the SV-COMP loop benchmarks. We show that a conservative tuning can achieve a 98% detecting rate in that benchmark while at the same time reducing false positive rates by around 50% compared to a singular static analysis approach.

1 Introduction

Quality assurance for safety-critical systems is no longer only challenged by hardware reliability and the complexity of the environment theses systems are operating in, but to a large degree these systems are suffering from the growth of their software bases. In the automotive space a state-of-the-art car contains over 50 million lines of source code. And although the industry has stringent quality assurance processes in place, complies to strict standards such as ISO 26262 and uses restrictive coding guidelines such as MISRA [1], safety is continuously challenged. As shown by Miller et al. [2, 3] modern automobiles are open to numerous attack vectors, almost exclusively being exposed by software bugs such as buffer overruns, null pointer dereferences and command injections.

Industry has extensive experience designing and testing safety-critical systems. However, the growing software sizes pose problems to many of the existing quality assurance methods and processes. This includes common practices such as dynamic

⋆ Pablo has been supported by the Australian Government as a Fellowship student, by NICTA as a Research Intern and by project TEC2011-28666-C04-02 and grant BES-2012-055572, awarded by the Spanish Ministry of Economy and Competitivity.

⋆⋆ Formally NICTA, funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.

software testing and static program analysis. Software testing does not scale well and is both expensive as well time consuming. On the other hand, static analysis scales well but suffers from both *false positives* and/or *false negatives*. This means, there are spurious warnings not relating to actual defects and instances of software bugs that are part of checked defect classes, but missed. Both false positives as well as false negatives are a serious concern in industry.

In this work we present a first approach that bridges some of the gaps between the existing techniques and their shortcomings. In particular, we present a combination of static program analysis and symbolic dynamic execution to minimize false positives and false negatives, while at the same time maintaining scalability. The core idea is to use static program analysis to broadly zoom in on a potential software defect and treat that as a bug candidate. Next, we make this bug candidate a precise target for symbolic execution. This has a number of advantages:

1. Scalability is maintained by a broad static analysis pass that zooms in on bug candidates.
2. Fine grained symbolic execution has a concrete target as opposed to an unguided crawl, which allows for additional symbolic execution heuristics.
3. Performance can be tuned, by relaxing static analysis constraints and removing false negatives at the expenses of potential false positives, which in turn can potentially be ruled out by symbolic execution. Conversely, only true positives can be reported where symbolic execution provides a concrete witness execution.

Moreover, we present the first steps of an integrated tool called *SEEKFAULT* using static analysis and symbolic execution, which borrows some of the underlying technology of the respective tools Goanna[4] and Forest[5]. As a first benchmark we apply SEEKFAULT to the loop category of the SV-COMP verification competition set. Relaxing static analysis to allow for over-approximations we are able to detect 98% of the defect cases in SV-COMP while reducing the false positive rate by over 50% compared to a single static analysis approach.

The remainder of this paper is structured as follows: In Section 2 we present the overall ideas and their relation with existing work. In Section 3 we give a brief introduction to the formal verification based static analysis we employ as we well as our symbolic execution framework for C/C++ code. This is followed by an explanation of our new combined analysis framework and its architecture in Section 4. We present our initial results from the tight integration of static analysis and symbolic execution in Section 5 and conclude with a summary and future work in Section 6.

2 Overview and Related Work

Software testing is extensively used to validate the safety-critical system (or a unit thereof) against requirements to ensure coverage of both the requirements and the actual code. However, it is well understood that only a fraction of the actual semantic behavior can be realistically tested and many of the known vulnerabilities result from corner cases of a particular input leading to vulnerabilities [2]. For instance, even when a path with a particular division is covered, the semantic case where that

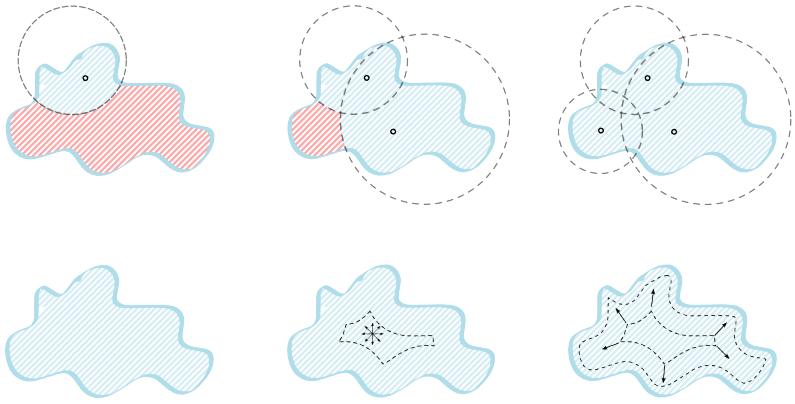

Fig. 1. Top Row: static analysis coverage; bottom row: symbolic execution coverage.

divisor happens to become zero might not. As a result even full traditional test coverage does not equate to full semantic coverage.

There are various approaches to remedy this shortcoming including symbolic execution [6] and concolic testing [7, 8]. These techniques increase semantic coverage criteria by treating inputs and undetermined return values symbolically. As such this enables the investigation of an execution not only for a single value, but for a symbolic range of values at once. However, as shown in industry case studies [9, 10] symbolic techniques are prone to scalability limitations and current tools are not well suited yet for deep embedded applications. As such their adoption in safety-critical industries has so far been limited.

One widely used technology in safety-critical industries is static program analysis. Static analysis approximates the behavior of source code and detects common coding violations such as the ones defined by MISRA, but also possible software bugs that lead to runtime errors such as null pointer dereferences, memory leaks, and buffer overruns. Many commercial tools are based on earlier academic work and are routinely used in industry [11, 12, 4, 13]. However, while static analysis is scalable to very large code bases it approximates program behavior and as such is prone to false positives (false alarms) and/or false negatives (missed bugs).

Static analysis and dynamic testing can be seen as two different approximation methods to cover the program semantics as shown in Figure 1. While (sound) static analysis over-approximates the program behavior and as such allows false positives, dynamic analysis under-approximates program behaviour leaving false negatives.

Our approach combines both techniques and approximations to obtain a solution that is more scalable than symbolic execution, yet more more precise than static program analysis. The main idea is to use static analysis to 'zoom' into potential bugs we call *bug candidates*. These bug candidates are identified by modern program analysis techniques including data flow, model checking and CEGAR-style trace refinement [14]. However, while this type of program analysis can be fast and scalable,

it is typically less precise than some actual execution or simulation. As such, there always remains a level of uncertainty regarding false positives. To counter this, we use the bug candidates determined by static analysis and pass them on to a symbolic execution execution pass, using those candidates as local reachability targets. This means, we attempt to confirm that a bug candidate is indeed a real bug. This allows us to boost the rate of true positives. The reachability targets assist additional heuristics that guide the search and are more efficient than classical *crawling* approaches [15].

Bug candidates that cannot be confirmed by symbolic execution remain potential bugs due to the under-approximating nature of symbolic execution unless, however, symbolic execution is able to explore all paths (symbolically) and as such can make a precise decision whether a bug candidate exists or not.

2.1 Other Related work

Most of the work in the symbolic execution and the static program analysis area has been focusing on improvements and heuristics in each individual field [16, 15, 17, 18]. Combining the different domains gained less attention and is mostly related to improving symbolic execution search strategies by adding static analysis information [19, 20]. In [21], Young and Taylor use static analysis to compute a concurrency graph and then to prune equivalent symbolic states by dynamic execution. Their ideas focus on concurrency errors for Ada programs and the goals are similar to symmetry and partial order reduction. Another combined approach is presented is [22]. However, the authors focus on obtaining maximal-coverage test cases for C programs. In contrast, our work focuses on the reachability of a set of error locations. Moreover, we use these locations to guide the symbolic execution search while [22] aims at heuristics for path coverage.

3 Our Approach to Static Analysis and Symbolic Execution

We deploy two complementary program analysis techniques: Static code analysis and symbolic execution. As the names suggest, the former is a static technique that takes the source code and builds an abstraction that is analyzed using a range of approaches including model checking and trace refinement. The latter is a dynamic technique that symbolically executes the program under test by building constraints over concrete execution paths and checking their validity using SMT solving.

3.1 Static Analysis using Model Checking and Trace Refinement

Static analysis comprises a number of techniques including data flow analysis, abstract interpretation and software model checking [23, 18]. The approach we use in this work is based on model checking and trace refinement as originated in the Goanna tool [24]. The core ideas are based on the observation that data flow analysis problems can be expressed in modal μ-calculus [25]. This has been developed further by Fehnker et al. in [26] and later expanded in [14].

```
void example() {
l0: int i, *q;
l1: int* p = malloc(sizeof(int));
    for(l2: i = 100000; l3: i >= 0; l7: i--) {
      l4: q = p;
      l5: if(i == 0)
       l6: free(p);
    }
}
```

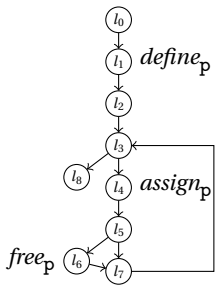

Fig. 2. Original program and automatically annotated CFG.

The main idea is to abstractly represent a program (or a single function) by its control flow graph (CFG) annotated with labels representing propositions of interest. Example propositions are whether memory is allocated or freed in a particular location, whether a pointer variable is assigned *null* or whether it is dereferenced. In this way the possibly infinite state space of a program is reduced to the finite set of locations and their propositions.

The annotated CFG consisting of the transition system and the (atomic) propositions can then be transformed into the input language of a model checker. Static analysis bug patterns can be expressed in temporal logic and evaluated automatically by a model checker. To illustrate the approach, we use a contrived function `example` shown in Fig. 2. It works as follows: First a pointer variable `p` is initialized and memory is allocated accordingly. Then, in a loop, a second pointer variable `q` is assigned the address saved in `p`. After hundred-thousand assignments `p` is freed and the loop is left.

To automatically check for a use-after-free, i.e., whether the memory allocated for `p` is still accessed after it is freed, we define atomic propositions for allocating memory $\texttt{define}_\texttt{p}$, freeing memory $\texttt{free}_\texttt{p}$ and accessing memory $\texttt{assign}_\texttt{p}$, and we label the CFG accordingly. The above check can now be expressed in CTL as:

$$\forall p : \mathbf{AG}(\mathit{define}_\texttt{p} \Rightarrow \mathbf{AG}(\mathit{free}_\texttt{p} \Rightarrow \mathbf{AG}\neg \mathit{assign}_\texttt{p}))$$

This means, whenever memory is allocated, after $\texttt{free}_\texttt{p}$ there is no occurrence of a $\texttt{assign}_\texttt{p}$. Note that once a check has been expressed in CTL, the proposition can be generically pre-defined as a template of syntactic tree patterns on the abstract syntax tree of the code and determined automatically. Hence, it is possible to automatically check a wide range of programs for the same requirement.

Trace Refinement Loop Model checking the above property for the model depicted in Fig. 2 will find a violation and return a counter example. The following path denoted by the sequence of locations is such a counter example: $l_0, l_1, l_2, l_3, l_4, l_5, l_6, l_7, l_3, l_4, l_5$. However, if we match up the counter example in the abstraction with the

concrete program, we see that this path cannot possibly be executed, as the condition `i == 0` cannot be true in the first loop iteration and, therefore, l_5 to l_6 cannot be taken. This means, the counter example is spurious and should be discarded. We might get a different counter example in the last loop iteration $\ldots, l_5, l_6, l_7, l_3, l_4, l_5$. But again, such a counter example would be spurious, because once the condition `i == 0` holds, the loop condition prevents any further iteration.

To detect the validity of a counter example we subjected the path to a fine-grained simulation using an SMT solver. In essence, we perform a backward simulation of the path computing the *weakest precondition*. If the precondition for the initial state of the path is unsatisfiable, the path is infeasible and the counter example spurious. We use an efficient SMT encoding and a refinement loop by creating *observer automata* to successively eliminate sets of infeasible traces. For the example in Fig. 2 the approach is able to create two observer automata from minimal unsatisfiable cores of a single path leading to the elimination of all paths of the same nature, i.e., avoiding an unrolling of the loop. This approach is similar to interpolation-based solutions and more details can be found in Junker et al. [14].

False Positives and Tuning Even in this formal verification based framework of static program analysis there are possibilities for false positives (wrongly warned bugs) and false negatives (missed bugs). This is caused by the abstraction and encoding into the model checker, which is necessarily sound. For instance, certain semantic constructs such as function pointers are typically not modelled and their behaviour is optimistically assumed. And, finally, the false positive elimination itself might time out and a judgment call whether to report a potential bug or not is made.

Industrial static analysis tools regularly make the aforementioned trade-offs. In this work we scale back the potential false negatives and counter the increasing false positives with symbolic execution.

3.2 SMT Solving Based Symbolic Execution

Symbolic execution is a faithful technique to observe program behavior by evaluating it symbolically in an abstract or constraint-based domain [6]. This means, values, variables and expressions are encoded as constraints over program paths and solvers are used to determine the (symbolic) program state at each location. The most common use case is to determine test inputs and coverage criteria [27], which is generalized to the concept of concolic testing [7, 8].

These approaches basically divide executions into equivalence classes exhibiting the same behavior under a given symbolic value or constraint of input parameters and path conditions. The advantage of these approaches is the ability to take into account a wide range of (equivalent) inputs within one interpreted execution. However, besides building a faithful symbolic interpreter the challenges of semantic coverage and dealing with a potentially exponential set of execution paths with respect to the number of decision points remain.

A bug detected by symbolic execution or concolic testing is basically the same as if discovered by dynamic testing. Hence, it provides some concrete validation that

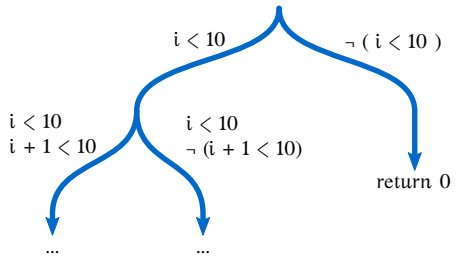

Fig. 3. Tree representation of the execution of program in Fig. 5

the program under test exposes some vulnerability. This is a clear advantage over static program analysis, where false positives are possible and further investigation of the results is often required. The downside is that both from a practical and theoretical point of view not all execution paths can typically be explored. As such symbolic execution helps for confirming bugs, but less so for ruling them out.

In the following we describe our approach to symbolic execution using multi-process execution and multi-theory SMT solving.

Our Approach to Symbolic Dynamic Execution Symbolic execution evaluates the code under test using symbolic variables. The symbolic variables can take any value of the replaced concrete variable range they substitute. This is in particular true for free (input) variables. Moreover, all operations on symbolic variables are recorded as a mathematical constraint over a pre-defined logic of an SMT solver as detailed below. In our approach, whenever the evaluator hits a conditional branch, the SMT solver is evaluating the two possible branching outcomes (true and false) to see if any or all of the branches are feasible. As a result, on each path the evaluator keeps a record of the set of constraints that must hold to follow this path and it only keeps the feasible paths in memory. This approach has proven to be a good compromise between generality, i.e., each path is represented as a formula that is valid for different input values, and specificity, i.e., only possible paths are considered, and they are considered explicitly.

To exemplify the approach, we can use the example shown in Figure 5 and the associated execution tree of Figure 3. When this example is run in the symbolic execution engine, it is emulated in a virtual environment that logs every access to a variable and every operation performed on a variable. In the example we start at the beginning of `main`. The first operation that takes place is the assignment of the constant 0 to the variable *i*. Then *i* is compared to 10. The framework keeps record of the fact that *i* comes from a constant, so there is no need to explore two branches in the condition, because only one can be executed. If *i* was not assigned to the constant 0 at the beginning, *i* would only have a symbolic assignment when reaching the condition. In that case, a SMT solver is called for the two possible outcomes of the branch, and the branch condition is added to a set of constraints that is independent for every path. If the solver gives a satisfying assignment for that branch output, we obtain

two outcomes: Firstly, we know that the path starting at the beginning of `main` is feasible. Secondly, we get a concrete input vector for all free variables demonstrating the reachability.

Multi-Process Execution As it can be intuitively seen in Figure 3, the number of paths grows exponentially with the number of branches in the code. To find non-trivial bugs and to scale to larger programs we use parallelization as one of the architectural solutions. This means for every decision point, i.e., every branching we spawn separate processes for the true and false branches. This enables us to parallelize the SMT solver computation as well as to independently follow different search strategies for different paths.

Multi-Theory Solving Another technique to speed up the process is to adjust the representation of the symbolic variables and their encoding in the SMT solver. For proving certain properties the sign of the variables might be enough or a representation as intervals is appropriate. If we want to account for overflows, or precisely capture sign-extension or bitwise operations, a bit-level representation for every variable is used. Some other representations we support are linear equations, where each variable is represented as a linear formula dependent of input variables, and polynomials, where sets of variables are represented as a polynomial equation. In our work we deploy heuristics to switch between different SMT solver theories dynamically based on the current context [5].

Execution Monitors While the main goal of symbolic execution is to generate input test vectors, it is possible to instrument the code on top of the symbolic execution framework to introduce *monitors* on it. Those monitors are observer code that check for errors during run-time. For instance, we add a monitor that on every pointer dereference checks that the value of the index to an array is in the range of allocated memory. Monitors can be expressed as SMT-formulas and their generation can be automated avoiding manual code annotations.

Although the semantics of symbolic execution precisely capture the set of program behaviors, the program is still under-approximated, since not all of paths can necessarily be explored, neither in theory nor in practice. This is caused by (non-regular) loops and recursion in programs leading to infinitely sized spaces. In order to maximize the set of explored states, different heuristics have been added to these frameworks [28, 15]. These heuristics do not solve, however, the fundamental problem of a potentially exponentially growing number of execution paths. Hence, our goal is to use static analysis for defining more constrained bug candidates and provide a guidance of the symbolic execution framework in the search strategy.

4 An Integrated Static-Dynamic Approach

Architecture We illustrate our approach in Figure 4: We start off with a static security analysis phase. If there is no vulnerability found the process stops. Otherwise,

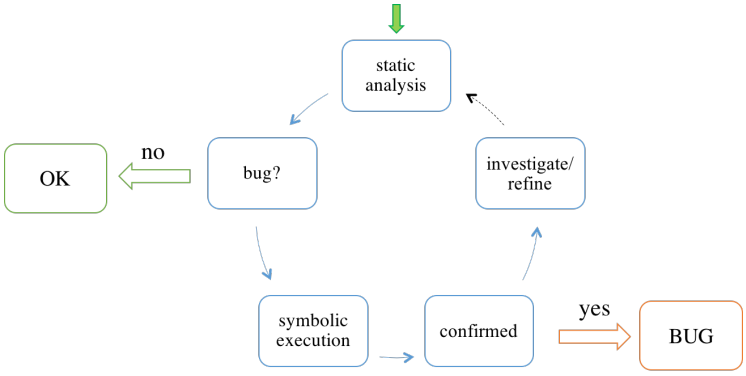

Fig. 4. SEEKFAULT Architecture: Static-Dynamic Integration.

we submit the bug candidate to the symbolic execution engine. If the symbolic execution engine is able to confirm the issue it generates a concrete trace and an input vector. Otherwise, the bug candidate is neither confirmed nor ruled out automatically and needs to be subjected to a manual investigation. Depending on that outcome either the issue will be manually confirmed or it proves to be a false positive that can be used to improve the static analysis checking algorithm or the exact CTL specification.

4.1 Implementation

We implement our approach in a new tool called SEEKFAULT. The SEEKFAULT engine makes use of two approaches: Static analysis based on model checking and SMT-based trace refinement as used in Goanna [29], and symbolic execution based on multi-theory SMT solving as used in Forest [5]. Z3 is used as the underlying SMT solver. SEEKFAULT itself is developed in a mixture of C/C++ and OCaml.

At the current stage of development the integrated SEEKFAULT tool first runs a static analysis pass to determine bug candidates and for each potential bug creates location information as well as a possible counter-example trace that is then passed on to the dynamic execution phase. Unlike traditional symbolic execution the combined approach in SEEKFAULT enables new search heuristics by applying the trace information as well as by using the bug locations as *reachability targets*. For instance, as one heuristics we calculate a distance measure from the last visited node in the program to the reachability target. This distance is computed statically over the control flow graph. The symbolic execution engine can then use that distance to sort the set of candidate paths during the guided search. To do so, we use the standard A* graph traversal algorithm. Finally, we use time outs on each branch of the symbolic execution if we are unable to reach a particular target.

```
int main(...) {
  int a[10];
  int i;
  for(i=0; i < 10; i++){
    a[i] = 10;
  }
  a[i] = 0;
  return 0;
}
```

Fig. 5. Overflow Detection Static Analysis

```
void main(...) {
  char password_buffer[10];
  int access = 0;
  strcpy(password_buffer, argv[1]);
  if(!strcmp(password_buffer, "
    passwd"))
    access = 1;
  printf("Access %d", access);
}
```

Fig. 6. Overflow Detection Symbolic Execution

5 Experiments

In this section we outline some of the experiments we performed and some of the experiences we gained so far. While implementation for larger projects is still underway it provides some valuable results.

5.1 Examples

We firstly demonstrate our idea by some examples from our internal test suite. The first example program is shown in Figure 5. An array with 10 elements ranging form 0 to 9 is initialized in a loop. However, in the last loop iteration the counter is increased to one beyond the array size and the subsequent access to that array would result in an out of bounds violation.

This error can be detected by our SEEKFAULT static analysis engine alone as the following command shows:

```
$ seekfault --static-only overflow.c
 SEEKFAULT - analyzing file overflow.c
 line 5: warning: Array ‘a’ subscript 10 is out of bounds [0,9].
```

For that example the analyzer is able to determine the array bounds as well as the number of loop iterations that are executed and, therefore, can derive the buffer overrun. However, in certain scenarios when the complexity of reasoning increasing by for instance copying memory around or reasoning about strings the analysis might lose precision. We do not warn in the latter cases. An example is shown in Figure 6. In the example, the buffer overflow introduces a real vulnerability, as it can be used to write in the memory occupied by the variable `access`, and grant the access to the application with an incorrect password.

This occurs when the size of the string passed as first parameter to the program is larger than 10 characters. In that case, the `strcpy` function writes in a space that was not allocated to store the variable `password_buffer`, but for `access`. Once access is overwritten with a different value than the initial 0, the access to the application is granted.

To be able to detect these kind of errors we tune the static analysis engine of SEEKFAULT to always emit an error when it is not certain that a bug is absent. This

means it will generate a vulnerability candidate for the example in Figure 6. Moreover, using our symbolic execution engine on the target location of the static analysis candidate we get a concrete confirmation of that bug. The SEEKFAULT engine produces:

```
$ seekfault pwd.c
 SEEKFAULT - analyzing file pwd.c
 line 5: Array ‘password_buffer’ subscript 10 is out of bounds:
 Symbolic analysis:
 Testcase  12 : aaaaaaaaac\0
 Testcase  13 : aaaaaaaaba\0
 Testcase  14 : aaaaaaaaaba\0 < BufferOverflow
```

This shows this two-tiered approach where static analysis defined the bug candidate and symbolic execution is able to provide a real exploitable scenario in case the input is the `aaaaaaaaaba\0` string.

5.2 SV-COMP Benchmark Results

For the evaluation of our integrated solution we use the well known SV-COMP benchmark[3], in particular, the *loop* category. SV-COMP is a set of competition benchmarks used in the automated verification community to highlight complex verification problems and to test the strength of individual tools.

The loop category is comprised of 117 files. All of the test cases expose a potential error, but only a minority of 34 files exhibit a real bug. Hence, any brute force approach by warning at any uncertainty will overwhelmingly exhibit false positives.

We show the results of our integrated approach in Table 1. This table is broken down by the different analysis phases as well as the final verdict, where *SA* denotes static analysis, *SE* symbolic execution and *SF* SEEKFAULT. A tick means proven to be correct, a cross that a bug has been confirmed and a warning triangle means for static analysis that it flags a potential issue and for symbolic execution that is times out. The files names shaded in gray are those containing a bug.

We have broken the table in five groups, which are separated by horizontal lines.

1. In the first set of examples, the static analysis engine is able to conclude that the program is correct. This is because our static analysis phase over-approximates the possible behavior and the program does not contain any approximation breaking constructs such as function pointers.
2. In the second group, SEEKFAULT's static analysis engine produces some potential bug candidates that are passed to the symbolic analysis pass. However, the symbolic analysis engine was able to faithfully cover all the possible branches in the program and conclude that all of them are bug-free.

[3] `http://sv-comp.sosy-lab.org/`

Filename	SA	SE	SF	Filename	SA	SE	SF	Filename	SA	SE	SF
nested6_true-u…	✓	⚠	✓	simple_false-u…	⚠	×	×	functions_true…	⚠	⚠	⚠
nested9_true-u…	✓	⚠	✓	terminator_01_…	⚠	×	×	simple_true-un…	⚠	⚠	⚠
heapsort_true-…	✓	⚠	✓	underapprox_fa…	⚠	×	×	simple_true-un…	⚠	⚠	⚠
apache-escape-…	✓	⚠	✓	sum01_bug02_su…	⚠	×	×	simple_true-un…	⚠	⚠	⚠
apache-get-tag…	✓	⚠	✓	while_infinite…	⚠	×	×	SpamAssassin-l…	⚠	⚠	⚠
count_by_k_tru…	✓	⚠	✓	for_bounded_lo…	⚠	×	×	sum03_true-unr…	⚠	⚠	⚠
diamond_true-u…	✓	⚠	✓	count_up_down_…	⚠	×	×	trex03_true-un…	⚠	⚠	⚠
gj2007_true-un…	✓	⚠	✓	sum01_bug02_fa…	⚠	×	×	count_up_down_…	⚠	⚠	⚠
gr2006_true-un…	✓	⚠	✓	sum01_false-un…	⚠	×	×	ddlm2013_true-…	⚠	⚠	⚠
seq_true-unrea…	✓	⚠	✓	sum04_false-un…	⚠	×	×	jm2006_true-un…	⚠	⚠	⚠
down_true-unre…	✓	⚠	✓	terminator_02_…	⚠	×	×	jm2006_variant…	⚠	⚠	⚠
phases_true-un…	✓	⚠	✓	trex02_false-u…	⚠	×	×	overflow_true-…	⚠	⚠	⚠
up_true-unreac…	✓	⚠	✓	sum03_false-un…	⚠	×	×	half_true-unre…	⚠	⚠	⚠
bhmr2007_true-…	✓	⚠	✓	trex03_false-u…	⚠	×	×	nest-if3_true-…	⚠	⚠	⚠
hhk2008_true-u…	✓	⚠	✓	terminator_03_…	⚠	×	×	MADWiFi-encode…	⚠	⚠	⚠
half_2_true-un…	✓	⚠	✓	trex01_false-u…	⚠	×	×	trex04_true-un…	⚠	⚠	⚠
string_concat-…	✓	⚠	✓	simple_false-u…	⚠	×	×	trex01_true-un…	⚠	⚠	⚠
eureka_01_true…	✓	✓	✓	functions_fals…	⚠	×	×	sum01_true-unr…	⚠	⚠	⚠
n.c40_true-unr…	✓	✓	✓	simple_false-u…	⚠	×	×	string_true-un…	⚠	⚠	⚠
lu.cmp_true-un…	⚠	✓	✓	overflow_false…	⚠	×	×	vogal_true-unr…	⚠	⚠	⚠
veris.c_sendma…	⚠	✓	✓	phases_false-u…	⚠	×	×	afnp2014_true-…	⚠	⚠	⚠
eureka_05_true…	⚠	✓	✓	eureka_01_fals…	⚠	×	×	array_true-unr…	⚠	⚠	⚠
cggmp2005_true…	⚠	✓	✓	id_trans_false…	⚠	×	×	array_true-unr…	⚠	⚠	⚠
diamond_true-u…	⚠	✓	✓	string_false-u…	⚠	×	×	array_true-unr…	⚠	⚠	⚠
underapprox_tr…	⚠	✓	✓	vogal_false-un…	⚠	×	×	array_true-unr…	⚠	⚠	⚠
large_const_tr…	⚠	✓	✓	NetBSD_loop_tr…	⚠	⚠	⚠	cggmp2005b_tru…	⚠	⚠	⚠
nec40_true-unr…	⚠	✓	✓	sendmail-close…	⚠	⚠	⚠	const_true-unr…	⚠	⚠	⚠
sum04_true-unr…	⚠	✓	✓	simple_true-un…	⚠	⚠	⚠	count_by_1_tru…	⚠	⚠	⚠
terminator_02_…	⚠	✓	✓	terminator_03_…	⚠	⚠	⚠	count_by_1_var…	⚠	⚠	⚠
array_false-un…	⚠	×	×	trex02_true-un…	⚠	⚠	⚠	count_by_2_tru…	⚠	⚠	⚠
array_false-un…	⚠	×	×	css2003_true-u…	⚠	⚠	⚠	count_by_nonde…	⚠	⚠	⚠
const_false-un…	⚠	×	×	n.c11_true-unr…	⚠	⚠	⚠	gauss_sum_true…	⚠	⚠	⚠
diamond_false-…	⚠	×	×	while_infinite…	⚠	⚠	⚠	gj2007b_true-u…	⚠	⚠	⚠
diamond_false-…	⚠	×	×	while_infinite…	⚠	⚠	⚠	gsv2008_true-u…	⚠	⚠	⚠
ludcmp_false-u…	⚠	×	×	while_infinite…	⚠	⚠	⚠	id_build_true-…	⚠	⚠	⚠
multivar_false…	⚠	×	×	cggmp2005_vari…	⚠	⚠	⚠	multivar_true-…	⚠	⚠	⚠
nec11_false-un…	⚠	×	×	for_infinite_1…	⚠	⚠	⚠	nested_true-un…	⚠	⚠	⚠
phases_false-u…	⚠	×	×	for_infinite_1…	⚠	⚠	⚠	nec20_false-un…	⚠	⚠	⚠
simple_false-u…	⚠	×	×	fragtest_simpl…	⚠	⚠	⚠	verisec_NetBSD…	⚠	⚠	⚠

Table 1: Results of each engine and the integrated SEEKFAULT solution.
SA = static analysis, SE = symbolic execution, SF = SEEKFAULT, gray = bug

3. In the third group, the full potential of the SEEKFAULT approach is shown. In these cases static analysis concludes that there is a potential bug in the code and provides a set of candidate locations that exhibit the undesired behavior. This set of locations is used as target locations for the the symbolic execution heuristics.

In each case SEEKFAULT was able to find the bug and provide a test case that demonstrates this behavior.

4. In the next two groups, the relaxation of the rules in the static analysis tool makes the analysis to produce error candidates in programs that however do not exhibit undesired behavior under the fully-accurate semantics of the operations of the program. The set of feasible paths, however, is too big to be fully exercised by symbolic execution, so under the requirements of a sound analysis, the algorithm has to output an inconclusive output. We observe, however that the fact of having a concrete goal to reach helps a lot in the symbolic execution framework so most of these cases (41 over 43) are actually correct. Considering the two remaining cases as correct would break the soundness of the approach but would leave us with an error rate of only 2/117.

In summary, the combined approach has a detection rate (number of detected errors over files with an error) of 98%. The true negative rate of the combined approach (number of files "proven" as correct when they are correct) is 35%, which is approximately 50% above the rate obtained by only using a static analysis approach.

5.3 Observations and Limitations

It is worth noting some observations: Firstly, our SEEKFAULT solution is quite capable of detecting bugs. All bugs have been identified by SEEKFAULT and all apart from two have been confirmed with symbolic execution inputs and traces. Secondly, the SEEKFAULT approach gives a slightly better coverage to demonstrate the absence of bugs compared to single static analysis approach. However, the SEEKFAULT solution is not yet very capable to prove the absence of bugs in general.

Having said that, the SV-COMP results need to be taken with a grain of salt: Many of the competition tools are variants of bounded model checking tools that declare a program bug free if no violation up to a certain bound can be found. In our case, if we declared a program bug free when both SEEKFAULT phases cannot come to a combined negative conclusion, we would correctly identify all benchmark cases apart from two, keeping the overall error rate at around 1%. This is better than the rate exhibited by more mature state-of-the-art tools in this set of programs.

Finally, we expect SEEKFAULT to shine outside the small but very complex SV-COMP cases. The main reason is that symbolic execution adds a lot of precision to static analysis, but is typically hampered by scalability. In the SEEKFAULT approach, however, static analysis takes care of scalability and provides local bug candidates that should be easier identifiable. Implementation work for those additional experiments is underway.

6 Conclusions

In this work we presented an integrated approach of static program analysis and symbolic execution. In this new two-phased solution static analysis is tuned to not miss bugs at the expense of higher false positives, which are filtered in the second

phase using symbolic execution. We implemented the solution in the tool SEEKFAULT.

Our experiments on the challenging SV-COMP benchmark shows a 98% vulnerability detection rate with a 50% reduced false positive rate compared to a singular static analysis solution. Moreover, the overall true negative rate remains at around 35%, which is quite reasonable for this set of benchmarks. However, overall the false positive rate it still too high, unless we add the soundness breaking assumption that inconclusive symbolic execution results indicate the absence of a bug.

Future work is to experiment on larger open source projects. Our conjecture is that most detectable bugs are less complex than the SV-COMP ones and we should see lower false positive rates. However, this will largely depend on the scalability results for the symbolic execution phase. Earlier experiments with the use of reachability targets, however, showed that our symbolic analysis scales well to around several hundred to thousand lines of code.

Moreover, right now we still manually adjust the static analysis engine whenever possible to feedback the new information we gained from the symbolic execution phase. Another line of future work is to investigate a learning mechanism to at least partially automate that process.

References

1. MIRA Ltd: MISRA-C:2004 Guidelines for the use of the C language in critical systems (October 2004)
2. Miller, C., Valasek, C.: A survey of remote automotive attack surfaces. Black Hat USA (2014)
3. Checkoway, S., McCoy, D., Kantor, B., Anderson, D., Shacham, H., Savage, S., Koscher, K., Czeskis, A., Roesner, F., Kohno, T., et al.: Comprehensive experimental analyses of automotive attack surfaces. In: USENIX Security Symposium, San Francisco (2011)
4. Huuck, R.: Technology transfer: Formal analysis, engineering, and business value. Sci. Comput. Program. **103** (2015) 3–12
5. Gonzalez-de Aledo, P., Sanchez, P.: Framework for embedded system verification. In: Tools and Algorithms for the Construction and Analysis of Systems. Volume 9035 of LNCS. (2015) 429–431
6. Clarke, L.A.: A system to generate test data and symbolically execute programs. IEEE Trans. Softw. Eng. **2**(3) (May 1976) 215–222
7. Sen, K., Marinov, D., Agha, G.: CUTE: A concolic unit testing engine for C. In: Proceedings of the 10th European Software Engineering Conference. ESEC/FSE-13, New York, NY, USA, ACM (2005) 263–272
8. Godefroid, P., Klarlund, N., Sen, K.: Dart: Directed automated random testing. In: In Programming Language Design and Implementation (PLDI. (2005)
9. Qu, X., Robinson, B.: A case study of concolic testing tools and their limitations. In: Empirical Software Engineering and Measurement (ESEM), 2011 International Symposium on. (Sept 2011) 117–126
10. Cadar, C., Godefroid, P., Khurshid, S., Păsăreanu, C.S., Sen, K., Tillmann, N., Visser, W.: Symbolic execution for software testing in practice: Preliminary assessment. In: Proceedings of the 33rd International Conference on Software Engineering. ICSE '11, New York, NY, USA, ACM (2011) 1066–1071
11. Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Henri-Gros, C., Kamsky, A., McPeak, S., Engler, D.: A few billion lines of code later: using static analysis to find bugs in the real world. Commun. ACM **53**(2) (February 2010) 66–75

12. GrammaTech: CodeSurfer. http://www.grammatech.com/
13. Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.: Scalable shape analysis for systems code. In: Proceedings of the 20th International Conference on Computer Aided Verification. CAV '08, Berlin, Heidelberg, Springer-Verlag (2008) 385–398
14. Junker, M., Huuck, R., Fehnker, A., Knapp, A.: SMT-based false positive elimination in static program analysis. In: 14th International Conference on Formal Engineering Methods, Japan. Volume 7635 of LNCS. Springer (2012) 316–331
15. Williams, N., Marre, B., Mouy, P., Roger, M.: Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis. In: Dependable Computing - EDCC 5. Volume 3463 of LNCS. Springer (2005) 281–292
16. Cadar, C., Sen, K.: Symbolic execution for software testing: Three decades later. Commun. ACM **56**(2) (February 2013) 82–90
17. Escalona, M.J., Gutierrez, J.J., Mejías, M., Aragón, G., Ramos, I., Torres, J., Domínguez, F.J.: An overview on test generation from functional requirements. J. Syst. Softw. **84**(8) (August 2011) 1379–1393
18. D'Silva, V., Kroening, D., Weissenbacher, G.: A survey of automated techniques for formal software verification. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD) **27**(7) (July 2008) 1165–1178
19. Pasareanu, C.S., Visser, W.: A survey of new trends in symbolic execution for software testing and analysis. Int. J. Softw. Tools Technol. Transf. **11**(4) (October 2009) 339–353
20. Qu, X., Robinson, B.: A case study of concolic testing tools and their limitations. In: Empirical Software Engineering and Measurement (ESEM), 2011 International Symposium on. (Sept 2011) 117–126
21. Young, M., Taylor, R.N.: Combining static concurrency analysis with symbolic execution. Software Engineering, IEEE Transactions on **14**(10) (1988) 1499–1511
22. Williams, N., Marre, B., Mouy, P., Roger, M.: Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis. In: Dependable Computing-EDCC 5. Springer (2005) 281–292
23. Nielson, F., Nielson, H.R., Hankin, C.L.: Principles of Program Analysis. Springer (1999)
24. Fehnker, A., Huuck, R., Seefried, S.: Counterexample guided path reduction for static program analysis. In: Concurrency, Compositionality, and Correctness. Volume 5930 of Lecture Notes in Computer Science., Springer (2010) 322–341
25. Schmidt, D.A., Steffen, B.: Program analysis as model checking of abstract interpretations. In: Proc. SAS '98, Springer-Verlag (1998) 351–380
26. Fehnker, A., Huuck, R., Jayet, P., Lussenburg, M., Rauch, F.: Model checking software at compile time. In: Proceedings of the First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering. TASE '07, Washington, DC, USA, IEEE Computer Society (2007) 45–56
27. Cadar, C., Dunbar, D., Engler, D.: Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation. OSDI'08, Berkeley, CA, USA, USENIX Association (2008) 209–224
28. Burnim, J., Sen, K.: Heuristics for scalable dynamic test generation. In: Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering. ASE '08, Washington, DC, USA, IEEE Computer Society (2008) 443–446
29. Bradley, M., Cassez, F., Fehnker, A., Given-Wilson, T., Huuck, R.: High performance static analysis for industry. ENTCS **289**(0) (2012) 3 – 14 Third Workshop on Tools for Automatic Program Analysis (TAPAS' 2012).