Combining Static Analysis and Constraint Solving for Automatic Test Case Generation

Kostyantyn Vorobyov  
Centre for Software Assurance  
Bond University  
Gold Coast, QLD, 4229, Australia  
kvorobyo@bond.edu.au

Padmanabhan Krishnan  
Centre for Software Assurance  
Bond University  
Gold Coast, QLD, 4229, Australia  
pkrishna@bond.edu.au

Abstract—In this paper we present an approach in automatic test generation that combines features of static analysis and bounded symbolic computation that is capable of producing a test suite that can be used to declare a program under test safe within bounds. We first use the results produced by static analysis which will identify a potential list of all errors in the program. We restrict our search to the locations where errors can exist and aim to find exactly one test case per real bug.

We have constructed a prototype tool (called Batg) that implements our approach. We report the results of its running on a number of benchmarks from well known benchmarking suites. We compare Batg to klee (an automatic test generation framework) and CBMC (a bounded model checker). This comparison is based on the time taken by the tools, the number of bugs found and the number of generated test cases. We analyse the results of our experiment, demonstrating the benefits of our approach.

Keywords—automatic test generation; bounded verification; constraint solving;

I. INTRODUCTION

Automatic test generation has been investigated for many years but there are still many open problems [1]. These include automation of testing and derivation of test cases. Additionally, one of the reasons why test input generation is difficult is that the techniques used (such as symbolic analysis) are often inexact. There are other approaches to finding potential bugs, e.g., static program analysis and model-checking.

Static analysis can check large code bases in a reasonable amount of time and identify all program errors [2]. However, results produced using this approach will usually contain a number of alarms that do not correspond to actual errors. In model checking all behaviours of the program are examined to determine if they satisfy a given property. Bounded model checking [3] is one way to handle infinite state space programs and explore only a finite number of states. Bounded model checking is still expensive as it has to exhaustively explore all states within a sufficient bound [2].

These approaches can be adapted for automatic test generation. Dynamic test generation (e.g., concolic testing) is a technique that combines concrete and symbolic execution. Examples of these include CUTE, DART, klee [4]–[7]. Concolic testing can be used to meet suitable coverage criteria. Results produced by testing will have no false positive as if a test passes there is no unexpected program behaviour. However in practice testing will miss bugs, since not all behaviours can be tested. Such automatic test generation techniques are often linked to specific properties.

To be useful in practice (e.g., for our industry partner), some of the key requirements on an automatic test generation scheme are:

- Be fast (faster than model-checking), but also be more reliable (i.e., reduce false positives) than static analysis
- Produce and execute actual test cases
- Reduce the number of test cases as otherwise the oracle problem could be time consuming
- Use existing tools that are freely available

In this paper we present a technique for automatically generating tests and measure its effectiveness. Our technique combines aspects of concolic execution and bounded model checking. We first utilise static analysis methods to generate a list of potential (including real) program errors. Owing to the approximations made by static analysis technique, this list will typically have potential bugs which are false alarms.

Given such a list of potential errors, we examine (up to a bound) only those paths that can lead to those bugs. This eliminates paths that never lead to program failures. We show that our technique is capable of producing a test suite sufficient for declaring a program under test safe (within the specified bounds). This is either when all the tests pass or no test to elicit incorrect behaviour can be derived. Ideally our technique will not generate a test for safe behaviours. The aim is also to reduce the number of test cases to a single test per bug.

We have implemented our technique in the prototype tool called Batg (bug-driven automatic test generator). The implementation focusses on unit testing (i.e., for single functions that could have parameters and global variables) for buffer overflows for a subclass of C programs. We have run Batg on the number of benchmarks and the results are reported in this paper.

The rest of the paper is organised as follows. In Section II we discuss prior achievements in the area and present an application of our approach on an example program in Section III, followed by its technical description along with the prototype implementation in Section IV. Our evaluation is presented in Section V. This uses programs from well known benchmarking repositories for C code [2], [8], [9]. We also

II. RELATED WORK

Directed Automated Random Testing (DART) [5] starts with random input. It then calculates new input vectors to generate new execution sequences. Such a vector contains solutions of symbolic constraints gathered from previous executions which tries to force the execution via a new path. This process attempts to drive a program through various feasible paths. DART can automate the process of unit testing. Scalability is a major issue due to the exploration of all available program paths. There is also the cost of running a large number of tests.

In order to overcome such issues a number of techniques that use similar type of analysis have been presented. In SMART [6] all paths are enumerated by the search algorithm that explores possible branches of conditional statements. Paths for which path constraints are satisfiable are denoted as feasible and taken into account while executing the program. Cadar et al. [7] combine dynamic and symbolic execution in order to produce an appropriate test suite that covers feasible program paths that involve symbolic variables – input data that affects program behaviour. On each statement that includes such data, a constraint solver is called to detect possible directions based on given path conditions adding constraints to the path. Execution proceeds adding more constraints to the path until all feasible paths are explored. In CUTE [4] the execution is driven by depth first search by negating path constraints and solving the resulting constraint system that leads the execution through a different path.

For verification, tools such as CBMC [10] can be used. CBMC is an automatic model checker for ANSI C programs for verification of array bounds, pointer safety, exceptions and user-specified assertions. CBMC can check that no bugs exist beyond the bound provided sufficient unwinding is done. If the formula is verified but CBMC cannot prove that sufficient unwinding has been performed a claim (property) fails verification. But one can be sure that within the bounds, no bug exists. Such technique however suffers from extended execution time due to large number of states checked.

III. MOTIVATING EXAMPLE

The example is the modified program from the Verisec [9] benchmarking suite. Consider the illegal memory access bug at line 6.

```
1 void escape_absolute_uri (char *uri) {
2    int c = 0;
3    char *token = (char*)malloc(3);
4    while (uri[c] != 0 && c <= 3) {
5        if (uri[c] == '?') {
6            token[c] = c;
7            ++c;
8        }
9    }
```

Listing 1. Example Program

For this example, we assume the default unwinding bound is at least 4, since the bug is only exposed at the fourth iteration of the `while` loop. The syntactic path we need to consider is the loop involving lines 4, 5, 6 and 7. We make this path concrete by executing it, i.e., iterations 0 · · · 3. That is, every iteration will add the nodes corresponding to line numbers 4, 5, 6, 7. For each path the set of reachability constraints is extracted (constraints that restrict input that lead to node 6). Additionally the bug constraint $c \geq 3$ is generated. It can be seen that the first three paths generated (iterations 0, 1 and 2 where the constraints are $0 \geq 3, 1 \geq 3$ and $2 \geq 3$ respectively) the context is unsatisfiable. For the fourth path the context is satisfiable ($3 \geq 3$) and the solution to the constraints yields the length of the input buffer $uri$ to be 4, and each element in the buffer $uri$ to be equal to ‘?’.$ The value is calculated using the reachability constraint from `true` branch of the if statement at line 5. This value is passed to the test generator, which invokes function `escape_absolute_uri` with the calculated argument: `char *uri = "????"`. We use the memory analyser to confirm the illegal memory access at line 6.

IV. TECHNICAL DETAILS OF APPROACH

We now describe the details of our approach. Our technique depends on syntactic information on all potential errors of the program under test, including bug types and their locations. We use static analyser to generate the list of all program errors, restricted to the bug types that a particular analyser can find. Each potential bug is mapped to a node in the control flow graph. The high level steps of the technique, syntactic path generation, path processing and test generation and execution, are given in Algorithm 1 and explained below.

In the syntactic path generation phase we generate the set of all relevant simple paths using the control flow graph $CFG$ where $N$ is the set of nodes and $E$ is the set of edges). The set of paths generated relies on the set $B$ which has the nodes with the potential bugs. In this phase loops and recursive calls are unwound only to the first iteration. The overall set of processed paths (called $S$) is the set of paths for each buggy node. The set $S$ is said to provide bug coverage as it has all the simple syntactic paths that cover all the potential errors. This set $S$ is passed to the path processor which generates the concrete path.

Path processing derives a concrete path that demonstrates the buggy behaviour. This is achieved by symbolically executing the syntactic path up to the specified bound. Constraints that the path must satisfy are passed to a constraint solver to determine satisfiability. More specifically, we process every subset $Si$ from $S$. A bug-path (say $P_b$) is removed from $S_i$ (indicated by the function `removePath`) and processed as follows. As long as the unwinding limit ($BND$) is not reached and the bug is not found, $P_b$ is executed symbolically. During the symbolic execution we collect a set of path constraints into the set $C_b$ that corresponds to that specific path (this is indicated by the function `nextIterPath`, which returns a constraint system for every iteration).

In general $C_b$ contains constraints imposed by the semantics of the executable items such as assignment statements, function calls, loops etc. This collection of constraints is passed to the constraint solver which determines its satisfiability (indicated by the function `solve`).
Algorithm 1 Algorithmic description of our approach

Require: $\text{CFG} = (N, E)$ // control flow graph
Require: $B$ // potential list of program errors
Require: $\text{BND}$ // Explicit unwinding bound

Satisfiable Context Generation:

\[ S_n = \{ p | p \text{ is a simple path ending in node } n \} \]
\[ S = \{ S_n | n \in B \} \]

Path Processing:

for all $S_b \in S$ do
    while not $S_b$.empty() do
        $P_b = S_b$.removePath()
        unwind = 0, found = false
        while unwind < $\text{BND}$ and not found do
            $C_b = \{ \}
            $C_b$.insert(nextIterPath($P_b$))
            if $C_b$.solve() then
                found = true, $C_b$.state = solved, $S_b = \{ \}$
            else
                unwind++
            end if
        end while
    end while
end for

Test Generation and Execution:

for all $C_b$ corresponding to $S_b$ in $S$ do
    if $C_b$.state = solved then
        test_case = test_case.addtest($C_b$)
    else
        noBug($b$)
    end if
end for
execute(test_case)

The paths which have a satisfiable context (indicated by state being set to solved) are collected to be passed to the test-generator. Otherwise the path is unwound (indicated by statement unwind++) and the new set of constraints is generated. Each path is executed until the satisfiable context is found or the explicit unwinding bound (BND) is reached. Paths from each subset are processed until the constraint solver returns a satisfiable context (indicated by $C_b$.solve()). As we are interested only in one path per bug we set the set of paths to be explored to empty (i.e., $S_b = \{ \}$), which terminates the outer while-loop.

The satisfiable contexts ($C_b$) are passed to the test generator (indicated by the function addtest). For unsatisfiable contexts we indicate that there is no bug within the specified bound (indicated by the function noBug) and is thus a false positive. The test vector consists of key/value pairs denoting variable names and values. The program under test is then instrumented (indicated by the function execute) with those values, creating a separate test case.

We execute the generated test cases along with a suitable test oracle. For example, for buffer overflows we use a memory analyser, which can detect illegal memory accesses. Bugs that cannot be examined (say when a program crashes before the buggy statement was reached) are considered to be true positives.

A. Implementation Details

We have implemented our technique as a prototype tool called Batg, focusing only on buffer overflows. Batg is built on top of LLVM [11]. The input to Batg is a LLVM compiled program and the list of potential bugs obtained from the static analyser Parfait [2].

At this stage Batg supports only a subset of ANSI C. We do not handle struct, function, double pointers and multi-dimensional arrays in their full generality. This is because we analyse certain patterns of LLVM instructions used to implement these features. We analyse double pointers if they are implemented using the patterns we support. These patterns have been chosen based on the code base we wish to analyse. However we fully support single pointers and uni-dimensional arrays. The analysis in the current implementation is limited to intra-procedural. Constraints imposed by functions invoked by the function under analysis (e.g., standard library functions) are derived from manually specified function summaries.

Batg uses bug specifications from Parfait to generate the set of abstract paths for testing. It then generates paths complying to a full path coverage criteria, unwound to a user specified bound. We use the Yices constraint solver [12] to determine the satisfiability of the generated constraints. In cases when the context is satisfiable, we extract the input values and instrument the LLVM bitcode so that the main program calls the function under test using the calculated arguments. Finally Batg inserts an extra exit call after bug location, to avoid execution of the portion of the path irrelevant for testing. Once all bugs are processed we run the annotated executables through Valgrind [13] memory checker to check if any illegal memory access has occurred.

V. Evaluation

In this section we describe an experimental evaluation of the approach developed in this paper. We compare the results produced by Batg with the results produced by the tools klee and CBMC by measuring:

1) The time (in seconds) taken by each tool. For Batg and klee we separate the time to perform the analysis along with the test generation and the execution time of the produced test suite. We also measure the time used by Parfait to create the list of potential bugs to demonstrate the cost of the static analysis phase. For CBMC we measure the time to perform the verification only, as there is no testing involved.

2) The number of test cases generated by Batg that actually demonstrate the bug. Ideally this should be equal to the number of real errors in the code.

3) The number of test cases generated by klee. We expect Batg to produce one test case per bug while klee is likely to generate many more test cases. For klee we also measure the number of passed/failed tests.

4) We measure the number of bugs examined by Batg and the number of properties examined by CBMC.

The actual experiments was performed using a dual core Intel Pentium 3.0Gh with 2.0Gb of RAM, running 32-bit
Gentoo Linux. We restricted the use of the various tools as follows in order to ensure a fair comparison.

- Use the same unwinding bound for \textit{Batg} and \textit{CBMC}.
- We ignore the generation of unwinding assertions with \textit{CBMC} as they are not directly relevant to our measurements. But we enforce the analysis of each generated property in \textit{CBMC} to handle each potential bug.
- Perform manual code instrumentation using least possible input range of the benchmarks for the runs with \textit{klee}. Otherwise \textit{klee} would need to calculate these ranges increasing the time it spends on the analysis.

We conduct our evaluation in two stages. We first use a number of small benchmarks from well known code repositories of \textit{C} programs such as \textit{Samate} \cite{samate} and \textit{Verisec} \cite{verisec} and evaluate the results. We first choose benchmarks that have the maximum of one bug per file. These programs are provided by the suite to check if a particular tool can detect a specific type of a bug.

In the second part of the experiment we use four larger programs which have multiple bugs. The first program (called \texttt{strmod}) has been obtained from a buggy string manipulation library (copying, concatenation, upper and lower case conversion and sorting). The program contains five buffer-overflow vulnerabilities. The other programs are from various standard benchmarking suites, viz., \textit{Zitser} \cite{zitser} and \textit{Verisec} \cite{verisec}. These fragments have appeared in large programs such as sendmail. The selection of these programs has been influenced by the \textit{BegBunch suite} \cite{begbunch} and our previous experimentation \cite{preexp}.

As a first cut we choose programs that \textit{CBMC} is able to handle. In order to test buffer overflow, \textit{Samate} provides about 100 programs while \textit{Verisec} provides about 300 programs. Of these about 150 programs have bugs while the other programs do not have any bugs. We can handle about 50\% of the the programs in these benchmarks. This number could have been higher if function summaries were available for the various functions. We did write function summaries for some of the functions. For instance, the function \texttt{nondet_int()} occurs often in the \textit{Verisec} suite. We were able to add the constraint that the result belongs to the type \texttt{int}.

We also ignore programs whose bugs do not depend on the input. Such programs typically use constants to exhibit buggy behaviour. Their presence in the benchmarks is mainly aimed at static analysis tools. They are not relevant to us as no tests need to be generated.

In general we have chosen programs that provide a coverage of the various buffer overflow types (as defined by CWE) including range error, buffer copy without checking size of input stack-based and heap based buffer overflow.

Table I shows that \textit{Batg} finds all the bugs and all bugs reported by \textit{Batg} are real errors. Note that \textit{Batg} has to examine all the potential bugs reported by Parfait. On average \textit{Batg} and Parfait combined take less time than \textit{klee} and \textit{CBMC} to analyse the potential bugs. However in some cases (e.g., \texttt{full_bad}, \texttt{full_ok}) the time to analyse potential bugs exceeds the execution time of \textit{CBMC} and the time taken by \textit{klee} for the analysis. The exact cause of such \textit{Batg}’s behaviour needs further investigation.

The number of tests generated by \textit{klee} is significantly greater than the number of tests generated by \textit{Batg} and the majority of which do not result in bug discovery. Owing to the large number of tests generated by \textit{klee} their execution time is also large. The maximum execution time of \textit{Batg} does not exceed 0.3 seconds is less that minimum test execution time of \textit{klee} – 0.595 seconds. Although each program in the benchmarking suite has only one bug, the efficacy of \textit{Batg} should be measured against the number of potential bugs it has to explore (i.e., the list of bugs reported by Parfait).

Table II presents the results of running the tools on the larger programs with multiple bugs. For the function \texttt{strmod} \textit{Batg} takes approximately 4 seconds (preceded by 84 milliseconds taken by \textit{Parfait}) to analyse the 23 potential bugs returned by the Parfait static checker. During analysis \textit{Batg} processed 200 paths, finding satisfiable context for all 5 bugs in the code. Thus it generates 5 test cases that exhibit the bugs. \textit{CBMC} is also able to find all the five bugs. However \textit{CBMC} run takes over 22 seconds which is approximately 3 times longer that \textit{Batg}. The running time of \textit{CBMC} is related to the overall number of properties examined. In this case \textit{CBMC} generates 64 claims of which 44 fail verification. As there are only 5 real bugs, each bug has a number of verification conditions. As these verification conditions are not indexed by the bug \textit{CBMC} has to examine all of them. Such results demonstrates benefits of bug coverage. The unique bug reports reduces the state space explored as we use a single property to describe a single bug.

The running time of \textit{Batg} and \textit{CBMC} (approximately 4 and 22 seconds) is not high. This is due to the use of the small default bound that results in the detection of all five bugs. \textit{klee} generates 42 tests (24 of which force the program to exhibit the bugs). The running time for \textit{klee} is shown as infinite as \textit{klee} fails to terminate. However the results were obtained using \textit{klee}’s force termination functionality.

\textit{CBMC} is the ideal tool to find the bugs in the program \texttt{gecos_bad}. It is able to find all the bugs in about half the time taken by \textit{Batg}. Although \textit{klee} takes very little time it is able to find only 1 bug. For the program \texttt{util_bad} \textit{Batg} takes less time than \textit{CBMC}, \textit{klee} on the other hand takes a long time and also generates a large number of test cases.

The final set of programs were from the \textit{Verisec} suite. \textit{klee}’s performance on the program (\texttt{mime\_7to8\_bad}) is similar to \textit{Batg}; but \textit{klee} was able to find only one out of the four bugs. \textit{CBMC}’s performance on the last two programs was better than the total time taken by \textit{Batg}. Although the time spent by \textit{Batg} to actually generate the tests (i.e., the analysis time) was better than \textit{CBMC}, the time to execute the tests was relatively high. For instance, in the last two programs, \textit{CBMC} took 0.098s and 0.189s respectively. \textit{Batg} took only 0.030s and 0.046s to generate the test cases respectively. However, the execution took 0.468s and 0.701s. Our investigation shows that large execution time of a \textit{Batg} test can be attributed to the use of \textit{Valgrind} as the oracle. For the last two programs considered, an unobserved test run does not exceed 0.005 seconds. Thus the high testing time is predominantly due to \textit{Valgrind}. We argue that the time to execute the test with an oracle is necessary
as a buffer-overflow does not automatically cause the program to crash. However Valgrind will actually catch and report the bugs.

We now return to the behaviour of Batg on programs such as full_bad, full_ok and gecos_bad. The increase in time for analyses could be explained by the increase in the number of nodes that need to examined because of reachability constraints in loops. This increases the complexity of symbolic analysis and constraint solving. However not all loops seem to cause this behaviour. This is shown by the analysis time of the strmod benchmark, which has greater code complexity and the number of potential bugs.

There are some obvious threats to the validity of our claims. The first is that the choice of benchmarks could have skewed the results. We are planning to conduct experiments on larger code bases to check if Batg continues to exhibit superior performance. We also need to analyse in more detail the benchmarks (e.g., full_bad) where Batg performed worse than klee and CBMC. The second is that there may be tools that are better than klee and CBMC. Tools such as Batg and klee are research prototypes while CBMC is more robust. A better implementation of Batg and klee could change the results. While the actual numbers could change, we believe the relative advantages would remain. Thirdly we have focussed only on buffer overflows. More experimentation with other bug types is needed before we can evaluate the strength of Batg.

Finally we have used Valgrind as a convenient oracle to demonstrate buggy behaviour. A different oracle engine could yield different results. While this can affect the overall running time, the time spent in the analysis phase (i.e., identifying a concrete path) is better than the time spent by the other tools we have considered.

**Summary**

We have presented an automatic test generation approach for unit testing which combines static analysis and constraint solving. Our initial experiments using benchmarks from well known repositories is positive and meets the goals set for industrial adoption.

**Acknowledgment**

We thank Cristina Cifuentes and the Parfait team at Oracle Labs for their sustained support, continuous feedback during this project and access to the Parfait static analyser and BمبBunch benchmarking repository used in Batg design and testing. This first author is supported by a research grant from Oracle Labs.

<table>
<thead>
<tr>
<th>Name</th>
<th>Parfait</th>
<th>Batg</th>
<th>Klee</th>
<th>CBMC</th>
</tr>
</thead>
<tbody>
<tr>
<td>scpy1-bad</td>
<td>0.018</td>
<td>0.002</td>
<td>0.297</td>
<td>1.702</td>
</tr>
<tr>
<td>scpy2-bad</td>
<td>0.019</td>
<td>0.002</td>
<td>0.259</td>
<td>1.793</td>
</tr>
<tr>
<td>scpy3-bad</td>
<td>0.026</td>
<td>0.002</td>
<td>0.256</td>
<td>1.704</td>
</tr>
<tr>
<td>scpy4-bad</td>
<td>0.009</td>
<td>0.001</td>
<td>0.260</td>
<td>1.785</td>
</tr>
<tr>
<td>scpy5-bad</td>
<td>0.060</td>
<td>0.002</td>
<td>0.254</td>
<td>1.704</td>
</tr>
<tr>
<td>memmove1-bad</td>
<td>0.018</td>
<td>0.002</td>
<td>0.258</td>
<td>1.704</td>
</tr>
<tr>
<td>offby1_bad</td>
<td>0.020</td>
<td>0.002</td>
<td>0.257</td>
<td>1.704</td>
</tr>
<tr>
<td>offby2_gk</td>
<td>0.026</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>simp1_bad</td>
<td>0.023</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>simp2_gk</td>
<td>0.026</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>simp3_bad</td>
<td>0.033</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>simp4_gk</td>
<td>0.033</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>full_bad</td>
<td>0.155</td>
<td>0.237</td>
<td>0.755</td>
<td>0.000</td>
</tr>
<tr>
<td>full_ok</td>
<td>0.115</td>
<td>0.194</td>
<td>7.562</td>
<td>0.000</td>
</tr>
<tr>
<td>no_strmp_bad</td>
<td>0.005</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>no_strmp_gk</td>
<td>0.307</td>
<td>2.231</td>
<td>0.000</td>
<td>0.000</td>
</tr>
</tbody>
</table>

**Table II: Multiple Bug Benchmark**

<table>
<thead>
<tr>
<th>Name</th>
<th>Parfait</th>
<th>Batg</th>
<th>Klee</th>
<th>CBMC</th>
</tr>
</thead>
<tbody>
<tr>
<td>strmod</td>
<td>0.084</td>
<td>1.780</td>
<td>0.000</td>
<td>1.704</td>
</tr>
<tr>
<td>gecos_bad</td>
<td>0.117</td>
<td>2.640</td>
<td>0.000</td>
<td>2.237</td>
</tr>
<tr>
<td>unit_bad</td>
<td>0.024</td>
<td>0.468</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>memmove1-arr_one_char</td>
<td>0.024</td>
<td>0.468</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>memmove1-arr_two_char</td>
<td>0.024</td>
<td>0.468</td>
<td>0.000</td>
<td>0.000</td>
</tr>
</tbody>
</table>

**References**


