Automatic Testing of Symbolic Execution Engines via Program Generation and Differential Testing

Timotej Kapus Cristian Cadar
Imperial College London
United Kingdom
{t.kapus, c.cadar}@imperial.ac.uk

***Abstract*—Symbolic execution has attracted significant attention in recent years, with applications in software testing, security, networking and more. Symbolic execution tools, like CREST, KLEE, FuzzBALL, and Symbolic PathFinder, have enabled researchers and practitioners to experiment with new ideas, scale the technique to larger applications and apply it to new application domains. Therefore, the correctness of these tools is of critical importance.**

In this paper, we present our experience extending compiler testing techniques to find errors in both the concrete and symbolic execution components of symbolic execution engines. The approach used relies on a novel way to create program versions, in three different testing modes—concrete, single-path and multi-path—each exercising different features of symbolic execution engines. When combined with existing program generation techniques and appropriate oracles, this approach enables differential testing within a single symbolic execution engine.

We have applied our approach to the KLEE, CREST and FuzzBALL symbolic execution engines, where it has discovered 20 different bugs exposing a variety of important errors having to do with the handling of structures, division, modulo, casting, vector instructions and more, as well as issues related to constraint solving, compiler optimisations and test input replay.

I. Introduction

Symbolic execution has established itself as an effective testing method, with many research groups working on improving various aspects of it. The technique has also started to be used in the industry, with several companies reporting successful adoption [8]. Key to this progress has been the availability of symbolic execution tools, which allows industrial users to apply and extend the technique, and researchers to experiment with new ideas. Notable examples include open-source tools CREST [15], KLEE [5], FuzzBALL [24] and Symbolic PathFinder [26], and closed-source tools PEX [31] and SAGE [19].

Therefore, the quality of these symbolic execution tools is essential for continuous progress in this area. In this paper, we present our experience adapting techniques from the compiler testing area [34]—which has seen tremendous success in revealing important bugs in popular compilers—for testing symbolic execution engines. More precisely, our techniques are based on program generation and differential testing, adapted to exercise several key inter-related aspects of symbolic execution tools: execution fidelity, accuracy of constraint solving, correct forking and faithful replay—that is, whether the symbolic execution tool correctly follows the paths it intends to follow, gathering precise constraints in the process and whether the generated inputs execute the same paths as the ones followed during symbolic execution.

Our method is effective for both symbolic execution tools which keep multiple path prefixes in memory, as in EXE [7], KLEE [5], Mayhem [10], Symbolic PathFinder [26] and S2E [12], as well as for those which implement the concolic variant of symbolic execution, in which paths are explored one at a time, as in DART [18], CREST [15] and CUTE [29].

One of the key ingredients of modern symbolic execution techniques is mixed concrete-symbolic execution [6], [18]. A reliable tool has to correctly implement both execution types. On the concrete side, symbolic execution engines either embed an interpreter for the language they analyse: Java bytecode in the case of Symbolic PathFinder, LLVM IR in the case of KLEE, and x86 code in the case of SAGE, or modify and instrument the code statically: e.g. both CREST and EXE first transform the program using CIL [25], and instrument it at that level. As we show in this paper, the execution fidelity of the interpretation or instrumentation can be effectively tested by adapting program generation and differential testing techniques employed in compiler testing [34].

On the symbolic side, the accuracy of the constraints gathered on each explored path is of critical importance if symbolic execution tools are to avoid exploring infeasible paths (and report spurious errors) and generate inputs that when run natively follow the same path as during symbolic execution. Our approach tests the symbolic execution component in two ways. First, by constraining the initial inputs to follow a single path. The key idea is simple, but effective: starting from an automatically-generated or real program, we create program versions which produce output for one path only (say when the integer input x has value 10) and then we check whether the symbolic execution correctly follows that path and outputs the expected results. Second, by running small programs symbolically, and verifying that the generated inputs are consistent with the paths followed.

In both the concrete and symbolic cases, our approach is based on differential testing, in which symbolic execution runs are crosschecked against native runs. An effective crosschecking needs to employ effective and inexpensive oracles, that is, oracles that find important bugs and do not add a significant runtime cost. We use four different oracles: (1) crashes of the symbolic execution tool, and whether the two executions produce (2) identical outputs, (3) generate the same sequence

Accepted for publication in Proc. of ASE 2017. Published version will be available at http://ieeexplore.ieee.org/

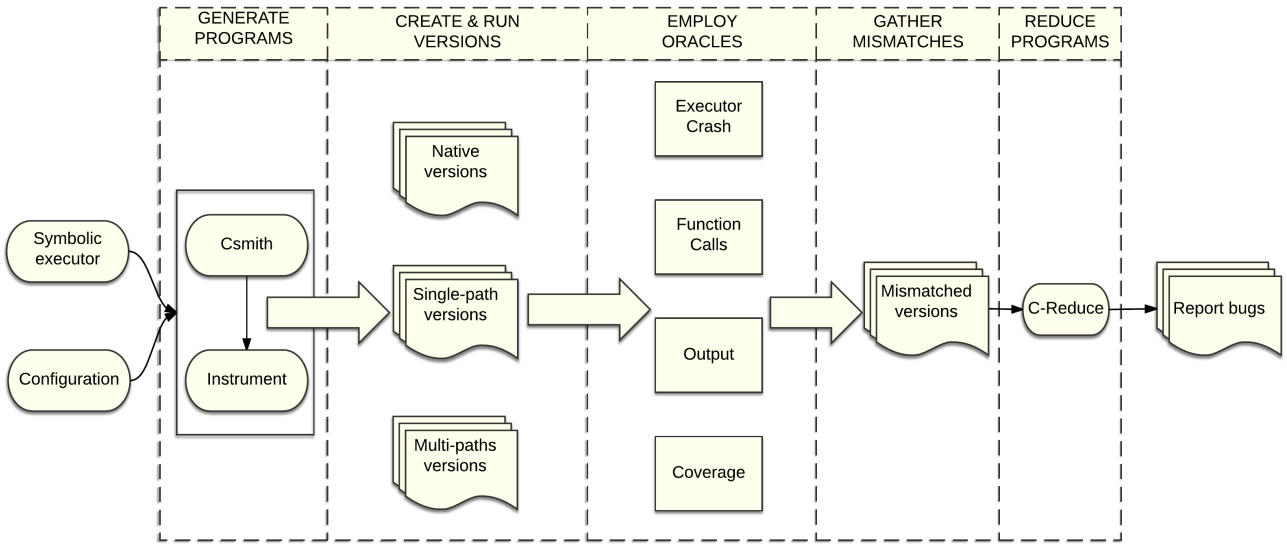

Fig. 1. The main stages of our testing approach.

of function calls, and (4) achieve the same coverage. In both the concrete and symbolic cases our approach also takes advantage of advances in automatic program generation [34], which allows us to quickly create small deterministic programs without undefined and unspecified behaviour, and therefore conduct a large number of experiments.

Finally, whenever a generated program finds a bug in the symbolic execution tool, we use existing program reduction techniques [27] which are combined with our oracles in order to obtain a small program (with fewer than 30 lines of code in our experiments) that forms an easy-to-understand, reproducible bug report.

We applied our approach to the KLEE, CREST and FuzzBALL symbolic execution engines, where it has found several serious functionality bugs, most of which have already been fixed or confirmed by the developers.

In summary, the main contributions of our paper are:

1) Our experience adapting compiler testing techniques to the novel problem of finding errors in both the concrete and symbolic execution components of symbolic execution engines;
2) A novel way to create program versions, in three different testing modes, which combined with existing program generation techniques and appropriate oracles, enables differential testing within a single symbolic execution engine;
3) A toolkit implementing our approach together with comprehensive case studies on three symbolic execution engines—KLEE, CREST and FuzzBALL—implementing different styles of symbolic execution (e.g. concolic vs. keeping all paths in memory, interpretation vs. instrumentation) and operating at different levels (source, LLVM bitcode and binary). Our approach found 20 important bugs in these engines.

The rest of the paper is structured as follows. §II gives an overview of our technique, showing how we generate random programs (§II-A) and create versions of these programs (§II-B) to be crosschecked using four different oracles (§II-C), and reduced to produce small bug reports (§II-D). §III presents our case studies on the KLEE, CREST and FuzzBALL systems, reporting the effectiveness and performance of our technique. The lessons learned from the case studies are then presented in §IV. Finally, §V discusses related work and §VI concludes.

II. Testing Approach

The main stages of our testing approach are shown in Figure 1. The inputs are a symbolic execution engine to be tested and a configuration file specifying the parameters of the testing process such as symbolic executor flags and timeouts.

In the first stage (*Generate programs* in the figure), we generate random, deterministic programs with the Csmith tool [34] and instrument them to support our oracles. In the second stage (*Create & run versions*), we create several different versions of a given generated program: a native version, designed to execute natively; single-path versions, designed to run a single path when executed symbolically; and multi-path versions, designed to run multiple paths when executed symbolically. These different versions are run and crosschecked using our four oracle types: crash detection, and output, function calls and coverage comparison (*Employ oracles*). Any programs exposing mismatches (as flagged by our oracles) between the native and symbolic execution runs (*Gather mismatches*) are then reduced using the C-Reduce tool [27] (*Reduce programs*) and reported to developers.

While our testing approach is general, our infrastructure is targeted toward symbolic execution of C code.

A. Generating random programs

The first step of our approach is to generate small programs using the Csmith [34] tool used in compiler testing. Csmith is a tool that can generate non-trivial C programs that leverage

many features of the C language and which has been used successfully to find many bugs in mature compilers [34].

Csmith generates programs in a top-down fashion. It starts by creating a single function, which is called from *main*. Csmith then randomly picks a structure from its grammar and checks if it is appropriate for the current context (e.g. *continue* can only appear in loops). Should the check fail, it makes a different choice until it succeeds. If the chosen structure needs a target (e.g. a variable to read or a function to call), it randomly chooses between using an existing construct and generating a new one. Care is taken not to generate constructs with undefined or unspecified behaviour, e.g. by guarding every division operation to ensure the divisor is not zero.

If the selected structure is a non-terminal, the process repeats. Finally, Csmith performs several safety checks to ensure there cannot be any undefined or unspecified behaviour. If that fails, the changes are rolled back and the process starts from the most recent successful stage.

The generated programs take no input, perform some deterministic computation and output the checksum of all global variables, which gives an indication of the state of the program upon termination. The length and complexity of the generated code is highly configurable. With the options we used, the generated programs are on average 1600 lines long, containing about 10 functions and 100 global variables. The global variables can have a wide range of types: signed and unsigned integers of standard widths, arrays, randomly generated structs and unions, pointers and nested pointers. The functions take varying number of arguments of different types and return a randomly-chosen type. Function bodies declare several local variables and include *if* and *for* statements, which in turn contain assignments to both local and global variables. The expressions assigned are deep and nested, reading from and writing to multiple global and local variables, performing pointer and arithmetic operations and calling other functions.

There are several reasons for using Csmith-generated programs as opposed to using real software:

1) Csmith programs are valid C programs without undefined or unspecified behaviour. This is important because the compiler used to generate the native version of the program and the engine used to symbolically execute the program might take advantage of undefined or unspecified behaviour in different ways, which might lead to spurious differences.
2) Csmith programs, by design, have a good coverage of C language features, which a limited collection of real programs might miss.
3) Most of the language features being used in Csmith programs can be enabled or disabled via command-line arguments. This is important because once the symbolic execution tool is found to mishandle a certain feature, we want to be able to continue testing without repeatedly hitting that same bug.
4) Csmith programs are deterministic and the input and output are easily identifiable: the input is represented by the set of global variables in the program, and the output consists of a checksum of the global variables, which is printed at the end of the execution.
5) Unlike real programs, Csmith programs are relatively small (or more exactly, Csmith can be configured to generate small programs), which allows us to perform a large number of runs.

Disadvantages of Csmith programs (and automatically generated programs more generally) are that they are artificial, hard to read by humans, and not guaranteed to terminate. We address the readability issue by automatically reducing the size of the program (§II-D), and the non-termination issue by using timeouts, as recommended by the Csmith authors [34] (§III-A2).

B. Creating and running versions

For each generated program, we first create and run an unmodified native binary version of the program. Then, for each of our three testing modes, we create a modified version of the program to be run by the symbolic execution engine under test.

1) C-Mode: Concrete mode: This mode is designed to test the concrete execution of the symbolic execution engine. For this mode, we run the program with the symbolic execution engine without marking any variable as symbolic. For example, we would compile the code to LLVM bitcode and then run it with KLEE directly without any symbolic input.

The symbolic execution run is then validated against the native one, using our oracles (§II-C). For example, the function call chain oracle would check that the native and symbolic runs generate the same sequence of function calls.

2) SP-Mode: Single-path mode: The aim of this mode is to test the accuracy of the constraints gathered by symbolic execution and its ability to correctly solve them. Essentially, this mode is checking the symbolic execution of individual paths in the program. For this mode, we modify the code to mark all the integer global variables of the generated program as symbolic and constraining them to have the unique value assigned to them in the original program. This essentially forces the symbolic execution engine to follow the same execution path as in the native version, but also collect and solve constraints on the way.

Constraining a variable to have a unique value needs to be done in such a way that the symbolic execution engine does not infer it has a unique value (and reverts to concrete execution for that variable). In particular, assigning a symbolic variable to have a constant value (e.g. `x = 4`) or comparing it with a constant (e.g. `if (x == 5)` would typically make the engine treat that variable as concrete on that path.

We used four different ways of constraining a symbolic variable x to a given value v, which are listed in Table I. For example, the second method adds the constraint that x is less than or equal to v and greater than or equal to v, while the fourth method adds the constraint that x is divisible by all the prime divisors of v, is greater than 1 and less than or equal to v. At the implementation level, for each integer global variable initialization such as **int** x = 5, we add the following

TABLE I
FOUR WAYS OF CONSTRAINING A VARIABLE x TO A CONSTANT VALUE v. d_i IS A PRIME DIVISOR OF v.

Constrainer type	Constraint
$<$,$>$	$\neg(x < v) \wedge \neg(x > v)$
$\leq$, $\geq$	$x \leq v \wedge x \geq v$
range	$\neg(x \leq v-2) \wedge \neg(x \geq v+3) \wedge$ $\neg(x = v-1) \wedge \neg(x = v+1) \wedge \neg(x = v+2)$
divisors	$\wedge_i \neg(x \mod d_i \neq 0) \wedge x > 1 \wedge x \leq v$

code at the start of `main`, should we for example, follow the first constraining method:

```
make_symbolic(&x);
if (x < 5) silent_exit(0);
if (x > 5) silent_exit(0);
```

In this code, the `make_symbolic()` function is used to mark the given variable as symbolic, while the `silent_exit()` function terminates execution without generating a test input on that path.

Therefore, after executing the code fragment above, the symbolic execution engine will continue along a single path with the path condition $\neg(x < 5) \wedge \neg(x > 5)$, which effectively constrains x to value 5.

Once such a version of the program is constructed, its execution can be validated using our oracles, as for the previous C-Mode. Note that one oracle that is effective here, as we show in the evaluation, is to check that the symbolic execution engine executes a single path. However, we didn't add an explicit oracle for this, as other oracles, such as the function call oracle, would almost always catch such a bug.

3) MP-Mode: Multi-path mode: While the prior mode tested that the engine correctly performs symbolic execution of a given path, this final mode checks that symbolic execution explores multiple paths and generates inputs that exercise exactly those paths.

For this mode, we simply mark all integer global variables as symbolic, without constraining them to any value, and let the symbolic execution engine explore multiple execution paths. As a result, not all oracles are applicable to this mode. In particular, we could not use the output oracle for non-concolic execution engines, as the output could now be a function of some symbolic variables. Besides the crash oracle, we decided to solely use the function call chain oracle, which was the easiest to adapt for this scenario. Our approach was to record the sequence of function calls on each path explored during symbolic execution, and then, for each path, to run natively the generated test input and check whether it generates the same function call sequence.

C. Oracles

We next discuss in detail the four oracles that we used in our approach.

1) Executor crash oracle: The first basic oracle consists in detecting generic errors during symbolic execution runs, such as segmentation faults, assert violations and other abnormal terminations.

2) Output oracle: As discussed in §II-A, Csmith programs are designed to have no undefined or unspecified behaviour and produce deterministic output. More exactly, the programs print at the end a single value, the checksum of all global variables. For C-Mode, we simply compare the checksums printed out by the native and symbolic execution runs.

For SP-Mode, we found that computing checksums for symbolic variables is very expensive, resulting in many time-consuming solver queries. Our solution was to exclude the symbolic variables from the checksum computation, and instead simply print out their individual values. For non-concolic engines, we first ask the constraint solver for a solution (which in this case is unique) before printing out the symbolic value.

3) Function call chain oracle: The function call chain oracle compares the sequence of function calls executed by the native and symbolic execution versions. This oracle provides the ability to catch some bugs where symbolic execution follows the incorrect path, but without having any influence on the output. For C-Mode and SP-Mode, this oracle checks that the unique path followed by the symbolic execution engine produces the same sequence of function calls as the native execution. For MP-Mode, this oracle checks that when natively replaying a generated input, the same function call sequence is produced as in the corresponding path explored during symbolic execution. Because some execution paths may not be fully explored by non-concolic tools in MP-Mode (due to timeouts), we actually check that the function call chain generated during symbolic execution is a prefix of the corresponding native function call chain.

4) Coverage oracle: The coverage oracle was used in a similar way as the function call chain oracle, to ensure that the native and symbolic execution runs execute the same lines of code, the same number of times. While we could have used this oracle in MP-Mode as well, to check whether the natively replayed execution covers the same lines of code as during the corresponding path explored during symbolic execution, we found this more difficult to implement efficiently.

One interesting challenge we faced while implementing this oracle is that even when gathering coverage information on a single execution path, the performance overhead was extremely high. The problem was that the instrumentation would generate `select` instructions to index into an internal buffer used to track coverage, which would make that buffer symbolic, leading in turn to expensive constraint solving queries. Once we diagnosed the issue, the solution was simple: we modified the GCov instrumentation to generate explicit branches instead of `select` instructions. This made a huge impact on performance, making this oracle usable. More generally, this is an issue that one has to be aware of during symbolic execution when instrumenting programs with coverage information.

D. Reducing bug-inducing programs

As indicated before, our Csmith programs are on average 1600 lines long and hard to read by human developers. The code consists of huge nested expressions without any high-level meaning, referring to mechanically-named variables. Debugging such programs would be highly difficult. Therefore, for each generated program that exposes a bug, we used the C-Reduce tool [27] to reduce it to a manageable size.

At a high-level, C-Reduce tries various source-level transformations to reduce a C program (e.g. deleting a line) and then uses an oracle (in the form of a shell script) to decide if the transformation was successful. If so, C-reduce keeps the reduced program and attempts to reduce it further, otherwise it rolls back the change and tries other transformations. Integrating C-Reduce in our bug-finding process was easy, as we were able to reuse the same oracle as the one used to find the bug in the first place. Some manual effort was necessary, but it was modest overall, as we discuss in §IV. One challenge is that unlike Csmith, C-Reduce can introduce undefined behaviour. Luckily, compilers report most of those undefined behaviours as warnings. We integrated these warnings into our oracles, making them reject reduced programs that trigger these specific warnings.[1]

III. Case Studies

This section presents our experience applying our testing approach to find bugs in the KLEE, CREST and FuzzBALL symbolic execution engines. We provide an artifact with additional details about our case studies at https://srg.doc.ic.ac.uk/projects/symex-tester/.

A. KLEE

Our main case study uses KLEE, a popular symbolic execution engine for C code that operates at the level of LLVM bitcode. We chose KLEE for our main case study because we are familiar with it, it is actively maintained, highly-configurable and documented. As we discuss later, this is important in order to be able to iteratively find new bugs.

We start by describing our experimental setup (§III-A1) and methodology (§III-A2), and give an overview of our experimental runs (§III-A3). We then present a summary of the bugs found and discuss a few representative bugs (§III-A4). We finally discuss our experience applying our approach to a real application (§III-A5).

1) Experimental Setup: We used the KLEE commit `637e884bb` for all our experiments. KLEE was built using LLVM 3.4.2 and STP commit `a74241d5`. Initially, we used version 1.6 of STP in a small number of our experiments. We used Csmith 2.3.0, C-Reduce commit `49782e718`, and Clang 3.4. The experiments were run in parallel [30] on an 8-core 3.5GHz Intel Xeon E3-1280 machine with 16GB of RAM.

To automate the experiments, we have built a toolset consisting of a library and several scripts. It contains the implementation of the oracles, including the functions used to constrain symbolic variables to a single value (see Table I) and the necessary tools to deploy our technique, including generating, compiling, running and reducing Csmith programs.

2) Methodology: We conducted our experiments in *batches*, with essentially one batch for each bug found. In each batch, we performed the following steps:

1) Configure the experiment (what kind of programs to generate, mode to use, options to pass to KLEE). We started with the default configuration of Csmith and KLEE.
2) Run the experiment (typically overnight).
3) Reduce the first program exposing a bug, and sometimes further manually simplify it slightly to make it more readable.
4) Report the bug, attaching the reduced program.
5) Find a way to avoid the bug and reconfigure the experiment accordingly.

The reason for the last step is that we observed that certain bugs would reappear over and over again, making it difficult to identify new bugs. Therefore, we adopted an iterative approach in which once we identified a bug, we worked on either fixing it (or incorporating the developers' fix if available in a timely manner), or more often reconfiguring our experiment to avoid it. In the latter case, we either disabled some C features in Csmith so that the bug would not be triggered (for example, once a bug involving incorrect passing of structures by value was found, we disabled passing structures as arguments) or changed the KLEE options so that the affected code would not run (for example, by disabling the *counterexample cache* [5] which was involved in one of the bugs).

At the end of our experiments we were using the options *no-arg-structs*, *no-return-structs*, *no-arg-unions*, *no-divs* and *no-const* in Csmith, and *check-overshift=false* and *use-cex-cache=false* in KLEE. In MP-Mode we also used the *no-checksum* option in Csmith to disable the expensive checksum computation, since the output oracle was not used.

We used either the -O0 or -O1 optimisation levels to compile the generated programs, each with equal probability. We also attempted to use higher optimisation levels, however every Csmith-generated program compiled with optimisation level -O2 or higher exposes the *vector instruction unhandled* bug in KLEE (§III-A4), and therefore we only used these higher optimisation levels for a small number of runs.

We used a 1s timeout for the native execution of Csmith programs, as a longer runtime is a strong indication of non-termination. We did not use a timeout in C-Mode for KLEE, as we knew from the native run that the program should terminate. The timeouts for KLEE in SP-Mode and MP-Mode were set to 100s, as we accounted for constraint solving. In MP-Mode, we also set the maximum number of forks (i.e. paths to be explored) to 200.

3) Summary of Runs: In total we have generated and tested almost 700,000 programs. A summary of all the runs can be found in Table II. They are divided by the different modes they used. We performed most runs, around 520,000, in C-Mode,

[1] We haven't treated all warnings as errors, as Csmith programs already generate some warnings during compilation.

TABLE II
SUMMARY OF RUNS IN DIFFERENT MODES.

Mode	# runs	Avg input size (LOC)	Avg time per run (s)	
			Native	KLEE
C	520,930	1,622	0.0500	0.808
SP $<$,$>$	42,162	1,642	0.0581	1.69
SP $\leq$, $\geq$	42,162	1,642	0.0581	1.69
SP range	42,162	1,642	0.0581	1.68
SP divisors	42,162	1,642	0.0581	2.56
SP $<$, $>$ & coverage	1,992	1,637	0.0726	91.8
MP	6,625	1,640	15.82[1]	22.21

[1] Combined runtime of all replayed test cases.

which as expected have the shortest average running time. We conducted around 168,000 runs in SP-Mode, which on average took twice as long as those in C-Mode. Finally, we performed 6,625 runs in MP-Mode, which were around 44 times more expensive than those in C-Mode. Note that the average runtime for native runs in MP-Mode includes replaying all generated test cases. Overall, we spent around around 124 hours in C-Mode, around 140 hours in SP-Mode, and around 70 hours in MP-Mode. The technique found the bugs within the first 5000 runs of each batch. This means that we could have configured the batches to run for only ~2.5h, but we found it convenient to do longer overnight runs.

For SP-Mode, Table II also shows the number of runs performed with each way of constraining inputs to a single value. The runs involving inequalities and ranges took a similar amount of time, while those involving divisors took longer, as they involved more difficult constraints. We excluded the expensive coverage oracle in all runs, except 1,992 SP-Mode runs with $<$, $>$ constraints, which took around 92 seconds per run on average. We observed that using the coverage oracle involves about ten times more instructions (as the instrumentation also uses code from `libc`), and a significant number of extra I/O operations, all of which contribute to the significantly higher cost per run.

Finally, note that individual runs in SP-Mode and MP-Mode varied considerably, depending on the constraint solving queries generated in each run. For instance, SP-Mode runs ranged between 0.01s to 99.6s (remember our SP-Mode time-out was 100s). The reason some runs were very quick was that only a small part of the code was executed at runtime.

4) Bugs found: Table III summarises the 14 bugs we have found using our approach. We reported all bugs to the developers, except one which had already been reported and another three which had already been fixed before we managed to report them. We also reported a bug which we later discovered to have been reported before. At the time of writing, the bugs in bold had already been fixed.

As can be seen from Table III, we have found a variety of bugs, involving the handling of structures, division, modulo,

```
1 union U0 {
2   signed f3 :18;
3 };
4
5 static union U0 g_988 = { 0UL };
6
7 int main(int argc, char* argv[]) {
8   g_988.f3 = 534;
9   printf("f3␣%d␣\n", g_988.f3);
10  return 0;
11 }
```

Listing 1. Reduced program exposing a bug where union fields are not updated correctly. The native run correctly prints *f3 534*, while the KLEE run prints *f3 22*.

```
1 #include <stdint.h>
2 static int32_t g_976;
3 int32_t func_46() {
4   printf("function␣call\n");
5   return 0;
6 }
7
8 void main() {
9   klee_make_symbolic(&g_976, sizeof g_976);
10  int32_t *l_1985 = &g_976;
11 lbl_2550:
12  func_46();
13  *l_1985 &= 2;
14  if ((3 ^ *l_1985) / 1)
15    goto lbl_2550;
16 }
```

Listing 2. Program exposing division by 1 bug in KLEE.

casting, vector instructions and more, as well as issues having to do with constraint solving, compiler optimisations, and test input replay. These bugs were revealed by different modes and oracles. We found 5 bugs in C-Mode, 6 bugs in SP-Mode and 5 bugs in MP-Mode, with 2 bugs found in both SP-Mode and MP-Mode. In terms of oracles, the crash oracle found 5 bugs, the output oracle 6 and the function call chain oracle 4, with 1 bug found by both the output and the function call chain oracles.

As mentioned before, the size of the Csmith programs we generated is on average 1600 lines of code. The last column of Table III shows the size of the reduced programs. In all cases, C-Reduce managed to reduce the programs substantially, to fewer than 30 lines of code, with most at under 15 lines.

Below we give some examples of the bugs found by our approach, including the reduced programs that were reported to developers.

Some unions not retaining values. Listing 1 shows an example of a bug found in C-Mode. The program initialises a union containing a signed field of non-standard length, and then writes 534 to that field and prints it. Running the program natively correctly prints out 534, while running it with KLEE prints out 22 (which represents the lower 9 bits of 534).

The root cause of this bug is an optimisation in KLEE which uses faster functions for memory writes of sizes 1, 8, 16, 32 and 64 bits. The code contained a check which enabled the optimisation only if the write in question was less than or equal to 64 bits. If this was not the case, the slower general approach was used. This incomplete check caused the program

TABLE III
SUMMARY OF BUGS FOUND IN KLEE, INCLUDING THE MODE USED, THE ORACLE(S) THAT DETECTED THEM, AND THE SIZE OF THE REDUCED PROGRAM USED IN THE BUG REPORT. ISSUES IN BOLD HAVE BEEN FIXED. MORE DETAILS ON THESE BUGS CAN BE FOUND AT HTTPS://GITHUB.COM/KLEE/KLEE/ISSUES/<ISSUE#>.

Issue#	Bug description	Mode	Oracle	Reduced size (LOC)
246	**Some unions not retaining values**	C	output	11
247	Incorrect by value structure passing	C	output	18
747	Invalid overshift error triggered by optimisation bug in LLVM	C	output	5
163	**Vector instructions unhandled, caused by -O2 optimisations**	C	output	6
268	**Floating point exception**	C	crash	14
266	**Incorrect handling of division by 1**	SP	function calls & output	17
262	**Execution forks unexpectedly**	SP	function calls	14
261	**Segmentation fault due to % operator**	SP	crash	12
n/a[2]	**Incorrect casting from signed to unsigned short**	SP	output	27
308	Abnormal termination in STP solver	SP, MP	crash	10
n/a[1]	**Assertion failure in STP solver 1.6**	SP, MP	crash	–
264	Replaying read-only variables not handled	MP	crash	8
331	File system model and replay library interplay	MP	function calls	9
n/a[2]	**Divergence b/w test generating path and test replay path**	MP	function calls	21

[1] Not explored further as the bug seems to have been fixed in the newest release of STP.
[2] Fixed prior to reporting as the side effect of what looks to be an unrelated patch.

```
1  int a, b;
2  safe_lshift_func_int16_t_s_u(short p1, p2) {
3      p1 < 0 || p1 ? p1 : p2;
4  }
5
6  main() {
7    klee_make_symbolic(&a, sizeof a);
8    if (a > (int)2453014441)
9      klee_silent_exit();
10   int i = a % (1 % a);
11   safe_lshift_func_int16_t_s_u(i, i || b);
12 }
```

Listing 3. Program triggering a segmentation fault in KLEE due to incorrect handling of some modulo expressions.

```
1  static int a;
2  static int b[1] = {1};
3  void fn1(short p1) { p1 - 0; }
4
5  static long fn2(p1) {
6    return 2036854775807 / p1 ? 1 : 0;
7  }
8
9  int main() {
10   long c;
11   int *d = &a;
12   c = fn2(b == d);
13   fn1(c);
14 }
```

Listing 4. Program that triggers a floating point exception in KLEE due to a missing division-by-zero check.

in Listing 1 (for which LLVM 3.4 generates a memory access of size 24) to run the optimisation path, and thus behave incorrectly. The bug has now been fixed.

Incorrect handling of division by 1. When executed natively, the code in Listing 2 loops indefinitely. The `if` statement at line 14 keeps evaluating to true and therefore the execution jumps back to line 11. In KLEE, the `if` statement evaluates to false, so KLEE terminates after a single iteration. This bug was caught by both the output and function call chain oracles. The bug was found in SP-Mode, but the reported program does not constrain the symbolic variable to have a single value, as we realised this is not needed to expose the bug (so the automatically reduced program was several lines longer). We also note that prior to running C-Reduce, the Csmith program exposed the bug without containing an infinite loop.

We initially managed to avoid this bug by disabling division expression generation in Csmith, but the bug was later debugged and fixed by the developers. The problem was that division by a constant is optimised prior to invoking the solver using multiplication and shift operations. However, the optimisation is incorrect for constants 1 and -1. The fix was to disable the optimisation for these special cases.

Segmentation fault due to % operator. The code in Listing 3 causes a segmentation fault in KLEE. The bug was found in SP-Mode and diagnosed by the developers to be caused by an incorrect semantics assigned to the % operator when negative numbers were used as divisors. The second part of the code that constrained variable `a` to have a single value was manually removed by us prior to reporting the bug, as it was not needed to expose this bug.

Floating point exception. The code in Listing 4 triggers a floating-point exception in KLEE. The bug, now fixed, is due to a missing division-by-zero check when processing constant expressions that are not folded in the LLVM bitcode.

TABLE IV
FUNCTION CALL DIVERGENCE IN GREP.

Native	fillbuf				close_stdout
KLEE	fillbuf	grepbuf	EGexecute	kwsexec	close_stdout

5) Grep case study: We have also considered applying the techniques presented in this paper to real programs. For this purpose, we used the popular UNIX utility grep which finds lines of text matching a certain string pattern.

We found several mismatches in MP-Mode, which were caught by the function call chain oracle. An example of the difference between the function call chains executed on one path explored by KLEE and the corresponding native run can be seen in Table IV. We did not report this bug yet, as we found it difficult to reduce (C-Reduce works on a single C file) and debug (given the much larger size of grep). Overall, this experience has reinforced our initial preference for using generated programs, which present the advantages discussed in §II-A. However, with more engineering work, our approach could be applicable to real programs too.

B. CREST and FUZZBALL

To show the generality of our technique, we also applied it to two other symbolic execution engines. We chose CREST [15] and FuzzBALL [24], because they are different from KLEE in important ways: CREST is a concolic execution tool [18], [29], a variant of symbolic execution which differs significantly at the implementation level from the one used by KLEE, while FuzzBALL is a symbolic execution for binary code, which again results in significant differences in the way the tool is implemented.

At the implementation level, to apply our framework to a new tool, one obviously has to be aware of the way the code is compiled and run with each new symbolic executor. Also, one needs to know the API the tool uses to mark inputs as symbolic. We have defined a general interface for creating and constraining variables, which enables us to use the same transformed program with multiple symbolic execution engines by simply changing the library we link with appropriately.

CREST implements the concolic form of symbolic execution [18], [29], in which the code is executed on concrete values and constraints are gathered on the side. To generate a new path, one constraint is negated, a new concrete input is generated and the process is repeated. Therefore, one important difference with KLEE is that paths are explored one at a time. A second important difference (but orthogonal to the first) is that CREST instruments programs for symbolic execution (using CIL) as opposed to interpreting them like KLEE.

We faced several practical difficulties when applying our approach to CREST. First, CREST is less feature-complete than KLEE. For example, it does not support symbolic 64-bit integers and its solver does not support some arithmetic operations such as modulo. However, we were able to work around these limitations by slightly tweaking our instrumentation and Csmith configuration.

Second and more importantly, CREST is not an actively developed project, and the tool does not seem to expose many options to enable or disable various sub-components, like KLEE does. Therefore we found it difficult to find ways around the bugs we discovered, in order to find new bugs.

In spite of these difficulties, our approach found three bugs in CREST within the first 1000 runs or about 2 hours worth of computation time. Further experiments were not run with CREST, due to the pervasiveness of the bugs already found. Note that CREST runs program paths significantly faster than KLEE or FuzzBALL, which accounts for the short total computation time.

A summary of the bugs found in CREST is shown in Table V. The first bug is exposed in C-Mode by a program with functions that return structs or unions. Here the CREST compiler throws an error when given such programs as input. Interestingly, KLEE had a similar problem with structs and function calls.

The other two bugs are exposed in SP-Mode. For instance, the code in Listing 5 makes CREST explore two branches, with both having the same constraints. The bug was caused by an incorrect use of the API of the constraint solver Yices, and has now been fixed.

FuzzBALL is similar to KLEE in that it implements the non-concolic style of symbolic execution, where execution starts with unconstrained symbolic variables. On the other hand, like

TABLE V
SUMMARY OF BUGS FOUND IN CREST AND FUZZBALL, INCLUDING THE MODE USED AND THE ORACLE THAT DETECTED THEM. THE SIZES OF THE REDUCED PROGRAMS VARY BETWEEN 8 AND 15 LINES OF CODE. ISSUES IN BOLD HAVE BEEN FIXED.

Issue#	Bug description	Mode	Oracle
CREST `github.com/jburnim/crest/issues/<Issue#>`			
7	Return struct errror	C	crash
6	**Big integer in expression**	SP	output
9	Non 32-bit wide bitfields	SP	output
FuzzBALL `github.com/bitblaze-fuzzball/fuzzball/issues/<Issue#>`			
21	**STP div by zero failure**[1]	SP	crash
20	**Strange term failure**	SP	crash
22	**Wrong behaviour**	SP	output

[1] Fixed in the upstream version of STP.

```
1 unsigned int a;
2 int main() {
3   __CrestUInt(&a);
4   printf("a: %d\n",a);
5   if( a < 2294967295) {
6     exit(0);
7   }
8 }
```

Listing 5. CREST explores two branches in both of which a is smaller than 2294967295.

```
1  unsigned int g_893 = 124;
2  int safe_sub(long long p1, int p2) {
3    return (p1 ^ ~9223372036854775807LL) - p2 < 0 ? 0 :
         p2;
4  }
5  static unsigned int magic_symbols[1] = {0};
6
7  int main() {
8    g_893 = *magic_symbols;
9    if(g_893 > 124) exit(0);
10   if(g_893 < 124) exit(0);
11   printf("%u\n", safe_sub(1UL ^ g_893, 1));
12 }
```

Listing 6. Reduced program for which the native run correctly prints *0*, where the FuzzBALL run prints *1*.

CREST, FuzzBALL executes paths one at a time, keeping only a lightweight execution tree in memory. Finally, like KLEE, FuzzBALL interprets the code rather than instrument it for symbolic execution, but does this at the binary rather than LLVM bitcode level. These design decisions make FuzzBALL an interesting complement to KLEE and CREST for our technique.

The only engineering challenge we had to address to use our technique on FuzzBALL was related to the fact that unlike KLEE and CREST, FuzzBALL does not provide an API for marking variables as symbolic. Instead, one has to specify on the command line the address range(s) that the tool should mark as symbolic (e.g. 16 bytes starting with `0xdeadbeef`). Therefore, the library we created for FuzzBALL defines a large static array which we mark as symbolic from the command line. At runtime, when a variable is supposed to be marked as symbolic, we get unused bytes from the static array and copy them to the variable to emulate the behaviour of `make_symbolic` functions that symbolic execution engines like CREST and KLEE provide.

With 2000 runs or about a day worth of computation time our approach has found three bugs in FuzzBALL, all of which have been fixed. Further experiments were not run with FuzzBALL as the bugs have not been fixed promptly enough to run more batches.

A summary of the bugs found in FuzzBALL is shown in Table V. Like KLEE, FuzzBALL uses STP as its main constraint solver, and we managed to trigger the same STP bug while testing FuzzBALL. The crash bug that we found stems from the fact that FuzzBALL is unable to distinguish between pointers and integers well, due to the nature of machine code at which it operates. However, it still finds this distinction useful for various reasons, therefore it employs some heuristics to classify words either into integers or pointers. The program we generated causes this heuristic to fail. The developers added further simplification rules and another command line option to help mitigate this issue. The last bug causes FuzzBALL to compute the wrong results and is illustrated in Listing 6. The bug was debugged to a formula simplification rule that was incorrect when signed overflow occurred. Developers removed the rule to fix the bug.

IV. Discussion and Lessons Learned

The approach of combining program generation with our novel way of creating program versions in three different modes to be crosschecked by appropriate oracles was successful in finding important bugs in three different engines operating in different ways (e.g. concolic variant vs. keeping all paths in memory, interpretation vs. instrumentation, etc.) and at different levels (source, LLVM bitcode and binary).

Adapting compiler testing techniques. Overall, we found existing compiler techniques to be a great match for testing symbolic execution engines. In many ways, both compilers and program analysis techniques like symbolic execution take as input programs, so program generation techniques for compiler testing such as Csmith can be easily reused in this new context.

On the other hand, we found that the differential testing part—where Csmith programs compiled by different compilers are checked to ensure that they produce identical results—is not easily translatable for checking symbolic executors. The main problem is that different symbolic executors may explore different paths in a given time budget, and also that the same code may have different number of paths at different levels (e.g., source, binary and LLVM) [3]. Therefore, instead of performing differential testing between different symbolic execution engines, we crosschecked native and symbolic execution versions of the same program, with the symbolic execution versions carefully constructed in three different modes (§II-B). We believe this approach could be applicable for testing other types of program analysers, but the way in which different versions are generated would have to be guided by the specifics of each program analysis.

Generated programs vs. real ones. We noticed some of the bugs we found were also reported by users while running real programs, but debugging large programs under symbolic execution is often incredibly difficult due to the many different variables involved. For instance, if symbolic execution fails to cover a part of the program, this could be due to a bug in the constraint solver, or a bug in the interpretation/instrumentation, or a limitation of the search heuristic used. Instead, the programs generated by Csmith and C-Reduce are small, and the way we generate program versions for differential testing makes it easy to debug the root cause of the problem—for instance, program versions created in SP-Mode must follow a single path when executed symbolically, and the path should be the same as the one executed by running the program natively on corresponding program inputs.

However, as we discuss in Section III-A5, our approach could be used to test symbolic execution engines with real programs too. While we found it difficult to do so with `grep`, developers familiar with the program might be able to more easily diagnose issues such as the function call divergence that we report in Table IV.

Finally, the fact that some of the bugs we found were also reported by users suggests that our approach finds bugs that matter, while also having the advantage that the bug reports are more easy to understand and debug.

Oracles. Overall, the output oracle found almost half of the bugs, followed closely by the crash oracle, and at some distance by the function call oracle. As explained in the paper, we ended up using the coverage oracle only for a small number of runs—despite optimising its performance as described in §II-C4, it is still very expensive, often involving more than 10x instructions when used (see §III-A3).

We also experimented with a performance oracle which flagged generated programs on which the symbolic execution engine spent disproportionately more time. For instance, the 520K C-Mode runs for KLEE have a mean performance slowdown compared to native execution of 120x with a standard deviation of 228. However, we had 28 programs on which the slowdown was over 8000x. Such outliers might point to program bugs, or at least highlight features which result in high overhead in symbolic execution engines. However, after reporting two such anomalies to KLEE developers without receiving a response, we realised that such performance reports are not too actionable, and decided not to include the performance oracle in the experiments we describe.

False positives. By design, our approach has no false positives—any program flagged by our oracles is a real bug. One exception is the function call oracle, which assumes that the order in which the arguments to a function are evaluated—which could be functions themselves—is the same across versions. Only the CREST experiments generated such false positives, as CREST is based on the CIL compiler infrastructure, which evaluates arguments in a different order from the GCC compiler with which the native versions were created. However, we found it easy to filter out such false positives. Another solution would be to force Csmith not to generate programs with function calls as function arguments.

Manual effort. Our approach involves manual work in only two cases. First, to reconfigure the experiments before running a new batch so that the previously found bugs are not triggered again. Second, to configure C-Reduce to shorten the program while preserving the essence of the bug (e.g. that it prints a certain wrong value or stack trace). Fortunately, such manual effort is only needed whenever we discover a bug, and we found the overall effort to be relatively modest.

V. Related Work

As far as we know, this is the first approach specifically targeted toward testing symbolic execution tools, and the first paper to present the experience of adapting compiler testing techniques to check mature symbolic execution engines.

More generally however, the research community has started to invest effort into ensuring the correctness and reliability of program analysis tools [4]. One example is represented by competitions among tools, which have the important side effect of finding bugs in participating tools: the annual SV-COMP competition for software verification tools is a prime example of such competitions [2].

Roy and Cordy's work on evaluating clone detection tools is one example of work that uses program generation to evaluate program analysis tools [28]. Their approach starts from real programs, which are mutated to create code clones.

Wu et al. [33] present a system for checking pointer alias implementations, which validates the results of pointer alias analysis tools against the pointer values observed at runtime. This is a form of differential testing, between dynamic and static information.

A technique similar in spirit to ours, although different in terms of application domain and techniques used, is Daniel et al.'s work on testing refactoring engines [16]. They combined program generation, by providing developers with a declarative way of constructing abstract syntax trees for Java programs using a bounded-exhaustive approach, with a crash oracle, oracles that take into account the semantics of the refactoring, and differential testing between refactoring engines.

Our approach takes advantage of the recent work on testing compilers [22], [23], [34], especially the work of Regehr et al. on program generation [34] and reduction [27]. Combined with our technique for generating program variants in three different modes and a set of appropriate oracles, we show that these techniques are effective at finding bugs in symbolic execution engines.

With respect to creating oracles based on differential testing, Weyuker proposed the use of *pseudo-oracles*, in which one creates an independently-written program that meets the same specification as the program under testing [32]. A similar approach is used for fault tolerance and reliability by the *N-version programming* approach in which multiple versions of the same program are run in parallel and their behaviour compared at runtime [1], [13], [14], [21]. Other types of cross-checking oracles exploit equivalences in the specification [11], [17], [20] or redundancies in the code [9].

VI. Conclusion

Symbolic execution has seen significant interest in the last few years, across a large number of computer science areas, such as software engineering, systems and security, among many others. As a result, the availability and correctness of symbolic execution tools is of critical importance for both researchers and practitioners. In this paper, we have described our experience extending compiler testing techniques to checking the correctness of symbolic execution tools. We have evaluated our technique via case studies on the KLEE, CREST and FuzzBALL symbolic execution engines, where it has found subtle errors involving structures, division, modulo, casting, vector instructions and more, as well as issues having to do with constraint solving, compiler optimisations and test input replay.

VII. Acknowledgements

We would like to thank the authors of Csmith for their assistance using the tool, the developers of CREST, FuzzBALL and KLEE for their feedback on our bug reports, and Frank Busse, Andrea Mattavelli, Stefan Winter and the anonymous reviewers for their valuable comments on our paper and artefact. This research has been generously supported by the EPSRC through an Early-Career Fellowship.

REFERENCES

[1] A. Avizienis, "The N-version approach to fault-tolerant software," *IEEE Transactions on Software Engineering (TSE)*, vol. 11, pp. 1491–1501, December 1985.

[2] D. Beyer, "Software verification and verifiable witnesses (Report on SV-COMP 2015)," in *Proc. of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'15)*, Apr. 2015.

[3] C. Cadar, "Targeted program transformations for symbolic execution," in *Proc. of the joint meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering, New Ideas Track (ESEC/FSE NI'15)*, Aug.-Sep. 2015.

[4] C. Cadar and A. Donaldson, "Analysing the program analyser," in *Proc. of the 38th International Conference on Software Engineering, New Ideas and Emerging Results (ICSE NIER'16)*, May 2016.

[5] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," in *Proc. of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*, Dec. 2008.

[6] C. Cadar and D. Engler, "Execution generated test cases: How to make systems code crash itself," in *Proc. of the 12th International SPIN Workshop on Model Checking of Software (SPIN'05)*, Aug. 2005.

[7] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler, "EXE: Automatically Generating Inputs of Death," in *Proc. of the 13th ACM Conference on Computer and Communications Security (CCS'06)*, Oct.-Nov. 2006.

[8] C. Cadar, P. Godefroid, S. Khurshid, C. Pasareanu, K. Sen, N. Tillmann, and W. Visser, "Symbolic Execution for Software Testing in Practice—Preliminary Assessment," in *Proc. of the 33rd International Conference on Software Engineering, Impact Track (ICSE Impact'11)*, May 2011.

[9] A. Carzaniga, A. Goffi, A. Gorla, A. Mattavelli, and M. Pezzè, "Cross-checking oracles from intrinsic software redundancy," in *Proc. of the 36th International Conference on Software Engineering (ICSE'14)*, May 2014.

[10] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing Mayhem on binary code," in *Proc. of the IEEE Symposium on Security and Privacy (IEEE S&P'12)*, May 2012.

[11] T. Chen, S. Cheung, and S. Yiu, "Metamorphic testing: a new approach for generating next test cases," Hong Kong University of Science and Technology, Tech. Rep. HKUST-CS98-01.

[12] V. Chipounov, V. Georgescu, C. Zamfir, and G. Candea, "Selective symbolic execution," in *Proc. of the 5th Workshop on Hot Topics in System Dependability (HotDep'09)*, Jun. 2009.

[13] J. E. Cook and J. A. Dage, "Highly reliable upgrading of components," in *Proc. of the 21st International Conference on Software Engineering (ICSE'99)*, May 1999.

[14] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser, "N-variant systems: A secretless framework for security through diversity," in *Proc. of the 15th USENIX Security Symposium (USENIX Security'06)*, Jul.-Aug. 2006.

[15] "CREST: Automatic Test Generation Tool for C," http://code.google.com/p/crest/.

[16] B. Daniel, D. Dig, K. Garcia, and D. Marinov, "Automated testing of refactoring engines," in *Proc. of the joint meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE'07)*, Sep. 2007.

[17] R.-K. Doong and P. G. Frankl, "The ASTOOT approach to testing object-oriented programs," *ACM Transactions on Software Engineering Methodology (TOSEM)*, vol. 3, no. 2, pp. 101–130, 1994.

[18] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in *Proc. of the Conference on Programing Language Design and Implementation (PLDI'05)*, Jun. 2005.

[19] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated whitebox fuzz testing," in *Proc. of the 15th Network and Distributed System Security Symposium (NDSS'08)*, Feb. 2008.

[20] A. Gotlieb, "Exploiting symmetries to test programs," in *Proc. of the 14th International Symposium on Software Reliability Engineering (ISSRE'03)*, Nov. 2003.

[21] P. Hosek and C. Cadar, "Varan the Unbelievable: An efficient N-version execution framework," in *Proc. of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)*, Mar. 2015.

[22] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," in *Proc. of the Conference on Programing Language Design and Implementation (PLDI'14)*, Jun. 2014.

[23] C. Lidbury, A. Lascu, N. Chong, and A. F. Donaldson, "Many-core compiler fuzzing," in *Proc. of the Conference on Programing Language Design and Implementation (PLDI'15)*, Jun. 2015.

[24] L. Martignoni, S. McCamant, P. Poosankam, D. Song, and P. Maniatis, "Path-exploration lifting: Hi-fi tests for lo-fi emulators," in *Proc. of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'12)*, Mar. 2012.

[25] G. C. Necula, S. McPeak, S. Rahul, and W. Weimer, "CIL: Intermediate language and tools for analysis and transformation of C programs," in *Proc. of the 11th International Conference on Compiler Construction (CC'02)*, Mar. 2002.

[26] C. S. Păsăreanu, W. Visser, D. Bushnell, J. Geldenhuys, P. Mehlitz, and N. Rungta, "Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis," *Automated Software Engineering*, vol. 20, no. 3, pp. 391–425, Sep 2013.

[27] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, "Test-case reduction for C compiler bugs," in *Proc. of the Conference on Programing Language Design and Implementation (PLDI'12)*, Jun. 2012.

[28] C. Roy and J. Cordy, "A mutation/injection-based automatic framework for evaluating code clone detection tools," in *Proc. of the 4th International Workshop on Mutation Analysis (Mutation'09)*.

[29] K. Sen, D. Marinov, and G. Agha, "CUTE: A concolic unit testing engine for C," in *Proc. of the joint meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE'05)*, Sep. 2005.

[30] O. Tange, "GNU parallel: The command-line power tool," *;login: The USENIX Magazine*, vol. 36, no. 1, pp. 42–47.

[31] N. Tillmann and J. De Halleux, "Pex: white box test generation for .NET," in *Proc. of the 2nd International Conference on Tests and Proofs (TAP'08)*, Apr. 2008.

[32] E. J. Weyuker, "Pseudo-oracles for non-testable programs," *The Computer Journal*, vol. 25, no. 4, pp. 465–470, 1982.

[33] J. Wu, G. Hu, Y. Tang, and J. Yang, "Effective dynamic detection of alias analysis errors," in *Proc. of the joint meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE'13)*, Aug. 2013.

[34] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *Proc. of the Conference on Programing Language Design and Implementation (PLDI'11)*, Jun. 2011.