Enhancing Symbolic Execution with Veritesting

By Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley

1. INTRODUCTION

Symbolic execution is a popular automatic approach for testing software and finding bugs. Over the past decade, numerous symbolic execution tools have appeared—both in academia and industry—demonstrating the effectiveness of the technique in finding crashing inputs, generating test cases with high coverage, and exposing software vulnerabilities.⁵ Microsoft's symbolic executor SAGE is responsible for finding *one-third* of all bugs discovered during the development of Windows 7.¹²

Symbolic execution is attractive because of two salient features. First, it generates *real* test cases; every bug report is accompanied by a concrete input that reproduces the problem (thus eliminating false reports). Second, symbolic execution systematically checks each program path exactly once—no work will be repeated as in other typical testing techniques (e.g., random fuzzing).

Symbolic execution works by automatically translating program fragments to logical formulas. The logical formulas are satisfied by inputs that have a desired property, for example, they execute a specific path or violate safety. Thus, with symbolic execution, finding crashing test cases effectively reduces to finding satisfying variable assignments in logical formulas, a process typically automated by Satisfiability Modulo Theories (SMT) solvers.⁹

At a high level, there are two main approaches for generating formulas: dynamic symbolic execution (DSE) and static symbolic execution (SSE). DSE executes the analyzed program fragment and generates formulas on a per-path basis. SSE translates program fragments into formulas, where each formula represents the desired property over *any* path within the selected fragment. The path-based nature of DSE introduces significant overhead when generating formulas, but the formulas themselves are easy to solve. The statement-based nature of SSE has less overhead and produces more succinct formulas that cover more paths, but the formulas are harder to solve. Is there a way to get the best of both worlds?

In this article, we present a new technique for generating formulas called *veritesting* that *alternates* between SSE and DSE. The alternation mitigates the difficulty of solving formulas, while alleviating the high overhead associated with a path-based DSE approach. In addition, DSE systems replicate the path-based nature of concrete execution, allowing them to handle cases such as system calls and indirect jumps where static approaches would need summaries or additional analysis. Alternating allows veritesting to switch to DSE-based methods when such cases are encountered.

We implemented veritesting in MergePoint, a system for automatically checking all programs in a Linux distribution.

MergePoint operates on 32-bit Linux binaries and does not require any source information (e.g., debugging symbols). We have systematically used MergePoint to test and evaluate veritesting on 33,248 binaries from Debian Linux. The binaries were collected by downloading and mining for executable programs all available packages from the Debian main repository. We did not pick particular binaries or a dataset that would highlight specific aspects of our system; instead we focus on our system as experienced in the general case. The large dataset allows us to explore questions with high fidelity and with a smaller chance of per-program sample bias. The binaries are exactly what runs on millions of systems throughout the world.

We demonstrate that MergePoint with veritesting beats previous techniques in the three main metrics: bugs found, node coverage, and path coverage. In particular, MergePoint has found 11,687 distinct bugs in 4379 different programs. Overall, MergePoint has generated over 15 billion SMT queries and created over 200 million test cases. Out of the 1043 bugs we have reported so far to the developers, 229 have been fixed.

Our main contributions are as follows. First, we propose a new technique for symbolic execution called veritesting. Second, we provide and study in depth the first system for testing every binary in an OS distribution using symbolic execution. Our experiments reduce the chance of per-program or per-dataset bias. We evaluate MergePoint with and without veritesting and show that veritesting outperforms previous work on all three major metrics. Finally, we improve open source software by finding over 10,000 bugs and generating millions of test cases. Debian maintainers have already incorporated 229 patches due to our bug reports. We have made our data available on our website.²⁰ For more experiments and details we refer the reader to the original paper.²

2. SYMBOLIC EXECUTION BACKGROUND

Symbolic execution¹⁴ is similar to normal program execution with one main twist: instead of using concrete input values, symbolic execution uses variables (*symbols*). During execution, all program values are expressed in terms of input variables. To keep track of the currently executing path, symbolic execution stores all conditions required to follow the same path (e.g., assertions, conditions on branch statements, etc.) in a logical formula called the path predicate.

The original version of this paper was published in the *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India, May 31–June 7, 2014). ACM, New York, NY, 1083–1094.

Algorithm 1: Dynamic Symbolic Execution Algorithm

Input: Initial program counter (entry point): pc_0
 Instruction fetch & decode: `instrFetchDecode`
Data: State worklist: *Worklist*, path predicate: Π ,
 variable state: Δ

```

1 Function ExecuteInstruction(instr, pc,  $\Pi$ ,  $\Delta$ )
2   switch instr do
3     case var := exp // assignment
4        $\Delta[\text{var}] \leftarrow \text{exp}$ 
5       return [(succ(pc),  $\Pi$ ,  $\Delta$ )]
6     case assert (exp) // assertion
7       return [(succ(pc),  $\Pi \wedge \text{exp}$ ,  $\Delta$ )]
8     case if (exp) goto pc' // conditional jump
9       // Regular DSE forks 2 states
10      return [(pc',  $\Pi \wedge \text{exp}$ ,  $\Delta$ ), (succ(pc),  $\Pi \wedge \neg \text{exp}$ ,
11       $\Delta$ )]
12      // Veritesting integration
13      return Veritest(pc,  $\Pi$ ,  $\Delta$ )
14     case halt: return [] // terminate
15
16 Worklist = [(pc0, true, {})] // initial worklist
17 while Worklist ≠ [] do
18   pc,  $\Pi$ ,  $\Delta$  = removeOne(Worklist)
19   instr = instrFetchDecode(pc)
20   NewStates = ExecuteInstruction(instr, pc,  $\Pi$ ,  $\Delta$ )
21   Worklist = add(Worklist, NewStates)

```

Inputs that make the path predicate true are guaranteed to follow the exact same execution path. If there is no input satisfying the path predicate, the current execution path is infeasible.

In the following sections, we give a brief overview of the two main symbolic execution approaches: dynamic and SSE. We refer the reader to symbolic execution surveys for more details and examples.^{5, 21}

2.1. Dynamic symbolic execution (DSE)

Algorithm 1 presents the core steps in DSE. The algorithm operates on a representative low-level language with assignments, assertions and conditional jumps (simplified from the original Avgerinos et al.²).

Similar to an interpreter, a symbolic executor consists of a main instruction fetch-decode-execute loop (Lines 14–17). On each iteration, the `removeOne` function selects the next state to execute from *Worklist*, decodes the instruction, executes it and inserts the new execution states in *Worklist*. Each execution state is a triple (*pc*, Π , Δ) where *pc* is the current program counter, Π is the path predicate (the condition under which the current path will be executed), and Δ is a dictionary that maps each variable to its current value.

Unlike concrete interpreters, symbolic executors need to maintain a list of execution states (not only one). The reason is conditional branches. Line 10 (highlighted in red) demonstrates why: the executed branch condition could be true or false—depending on the program input—and the symbolic executor needs to execute *both* paths in order to check correctness. The process of

generating two new execution states out of a single state (one for the true branch and one for the false), is typically called “forking.” Due to forking, every branch encountered during execution, doubles the number of states that need to be analyzed, a problem known in DSE as *path (or state) explosion*.⁵

Example. We now give a short example demonstrating DSE in action. Consider the following program:

```

1   if (input_char == 'B') {
2       bug();
3   }

```

Before execution starts DSE initializes the worklist with a state pointing to the start of the program (Line 12): (1, *true*, {}). After it fetches the conditional branch instruction, DSE will have to fork two new states in `ExecuteInstruction`: (2, *input_char* = 'B', {}) for the taken branch and (3, *input_char* ≠ 'B', {}) for the non-taken. Generating a test case for each execution path is straightforward: we send each path predicate to an SMT solver and any satisfying assignment will execute the same path, for example, *input_char* → 'B' to reach the buggy line of code. An unsatisfiable path predicate means the selected path is infeasible.

Advantages/Disadvantages. Forking executors and analyzing a single path at a time has benefits: the analysis code is simple, solving the generated path predicates is typically fast (e.g., in SAGE⁴ 99% of all queries takes less than 1 s) since we only reason about a single path, and the concrete path-specific state resolves several practical problems. For example, executors can execute hard-to-model functionality concretely (e.g., system calls), side effects such as allocating memory in each DSE path are reasoned about independently without extra work, and loops are unrolled as the code executes. The disadvantage is path explosion: the number of executors can grow exponentially in the number of branches. The path explosion problem is the main motivation for our veritesting algorithm (see Section 3).

2.2. Static symbolic execution (SSE)

SSE is a verification technique for representing a program as a logical formula. Safety checks are encoded as logical assertions that will falsify the formula if safety is violated. Because SSE checks programs, not paths, it is typically employed to verify the absence of bugs. As we will see, veritesting repurposes SSE techniques for summarizing program fragments instead of verifying complete programs.

Modern SSE algorithms summarize the effects of both branches at path confluence points. In contrast, DSE traditionally forks off two executors at the same line, which remain subsequently forever independent. Due to space, we do not repeat complete SSE algorithms here, and refer the reader to previous work.^{3, 15, 23}

Advantages/Disadvantages. Unlike DSE, SSE does not suffer from path explosion. All paths are encoded in a

^a Note the solver may still have to reason internally about an exponential number of paths—finding a satisfying assignment to a logical formula is an NP-hard problem.

single formula that is then passed to the solver.^a For acyclic programs, existing techniques allow generating compact formulas of size $O(n^2)$,^{10, 18} where n is the number of program statements. Despite these advantages over DSE, state-of-the-art tools still have trouble scaling to very large programs.^{13, 16} Problems include the presence of loops (how many times should they be unrolled?), formula complexity (are the formulas solvable if we encode loops and recursion?), the absence of concrete state (what is the concrete environment the program is running in?), as well as unmodeled behavior (a kernel model is required to emulate system calls). Another hurdle is completeness: for the verifier to prove absence of bugs, *all* program paths must be checked.

3. VERITESTING

DSE has proven to be effective in analyzing real world programs.^{6, 12} However, the path explosion problem can severely reduce the effectiveness of the technique. For example, consider the following 7-line program that counts the occurrences of the character 'B' in an input string:

```
1  int  counter = 0,  values = 0;
2  for  ( i = 0 ; i < 100 ; i ++ )
3      if (input[i] == 'B') {
4          counter ++;
5          values += 2;
6      }
7  if (counter == 75)    bug  ( );
```

The program above has 2^{100} possible execution paths. Each path must be analyzed separately by DSE, thus making full path coverage unattainable for practical purposes. In contrast, two test cases suffice for obtaining full code coverage: a string of 75 'B's and a string with no 'B's. However, finding such test cases in the 2^{100} state space is challenging.^b We ran the above program with several state-of-the-art symbolic executors, including KLEE,⁶ S2E,⁸ Mayhem,⁷ and Cloud9 with state merging.¹⁶ None of the above systems was able to find the bug within a 1-h time limit (they ran out of memory or kept running). Veritesting allows us to find the bug and obtain full path coverage in 47 s on the same hardware.

Veritesting starts with DSE, but switches to an SSE-style approach when we encounter code that—similar to the example above—does not contain system calls, indirect jumps, or other statements that are difficult to precisely reason about statically. Once in SSE mode, veritesting performs analysis on a dynamically recovered control flow graph (CFG) and identifies a core of statements that are easy for SSE, and a frontier of hard-to-analyze statements. The SSE algorithm summarizes the effects of all paths through the easy nodes up to the hard frontier. Veritesting then switches back to DSE to handle the cases that are hard to treat statically.

In the rest of this section, we present the main algorithm and the details of the technique.

3.1. The algorithm

In default mode, MergePoint behaves as a typical dynamic symbolic executor. It starts exploration with a concrete seed and explores paths in the neighborhood of the original seed following a generational search strategy.¹² MergePoint does not always fork when it encounters a symbolic branch. Instead, MergePoint intercepts the forking process—as shown in Line 10 (highlighted in green) of algorithm 1—of DSE and performs veritesting.

Veritesting consists of four main steps:

1. *CFG Recovery*. Obtains the CFG reachable from the address of the symbolic branch (Section 3.2).
2. *Transition Point Identification & Unrolling*. Takes in a CFG, and outputs candidate transition points and a CFG_e , an acyclic CFG with edges annotated with the control flow conditions (Section 3.3). Transition points indicate CFG locations with hard-to-model constructs where DSE may continue.
3. *SSE*. Takes the acyclic CFG_e and current execution state, and uses SSE to build formulas that encompass all feasible paths in the CFG_e . The output is a mapping from CFG_e nodes to SSE states (Section 3.4).
4. *Switch to DSE*. Given the transition points and SSE states, returns the DSE executors to be forked (Section 3.5).

3.2. CFG recovery

The goal of the CFG recovery phase is to obtain a partial CFG of the program, where the entry point is the current symbolic branch. We now define the notion of underapproximate and overapproximate CFG recovery.

A recovered CFG is an underapproximation if all edges of the CFG represent feasible paths. A recovered CFG is an overapproximation if all feasible paths in the program are represented by edges in the CFG (statically recovering a perfect—that is, non-approximate—CFG on binary code can be non-trivial). A recovered CFG might be an underapproximation or an overapproximation, or even both in practice.

Veritesting was designed to handle both underapproximated and overapproximated CFGs without losing paths or precision (see Section 3.4). MergePoint uses a customized CFG recovery mechanism designed to stop recovery at function boundaries, system calls and unknown instructions.

The output of this step is a partial (possibly approximate) intraprocedural CFG. Unresolved jump targets (e.g., `ret`, `call`, etc.) are forwarded to a generic `Exit` node in the CFG. Figure 1a shows the form of an example CFG after the recovery phase.

3.3. Transition point identification and unrolling

Once the CFG is obtained, MergePoint proceeds to identifying a set of transition points. Transition points define the boundary of the SSE algorithm (where DSE will continue exploration). Note that every possible execution path from the entry of the CFG needs to end in a transition point (our

^b For example, $\binom{100}{75} \approx 2^{78}$ paths reach the buggy line of code. The probability of finding one of those paths by random selection is approximately $2^{78}/2^{100} = 2^{-22}$.

implementation uses domination analysis²).

For a fully recovered CFG, a single transition point may be sufficient, for example, the bottom node in Figure 1a. However, for CFGs with unresolved jumps or system calls, any predecessor of the `Exit` node will be a possible transition point (e.g., the `ret` node in Figure 1b). Transition points represent the frontier of the visible CFG, which stops at unresolved jumps, function boundaries and system calls. The number of transition points gives an upper-bound on the number of executors that may be forked.

Unrolling Loops. Loop unrolling represents a challenge for static verification tools. However, MergePoint is dynamic and can concretely execute the CFG to identify how many times each loop will execute. The number of concrete loop iterations determines the number of loop unrolls. MergePoint also allows the user to extend loops beyond the concrete iteration limit, by providing a minimum number of unrolls.

To make the CFG acyclic, back edges are removed and forwarded to a newly created node for each loop, for example, the “Incomplete Loop” node in Figure 1b, which is a new transition point that will be explored if executing the loop more times is feasible. In a final pass, the edges of the CFG are annotated with the conditions required to follow the edge.

The end result of this step is a CFG_e and a set of transition points. Figure 1b shows an example CFG—without edge conditions—after transition point identification and loop unrolling.

3.4. Static symbolic execution

Given the CFG_e, MergePoint applies SSE to summarize the execution of multiple paths. Previous work,³ first converted the program to Gated Single Assignment (GSA)²² and then performed symbolic execution. In MergePoint, we encode SSE as a single pass dataflow analysis where GSA is computed on the fly—more details can be found in the full paper.²

To illustrate the algorithm, we run SSE on the following program:

```
if (x > 1) y = 1; else if (x < 42) y = 17;
```

Figure 1. Veritesting on a program fragment with loops and system calls. (a) Recovered CFG. (b) CFG after transition point identification & loop unrolling. Unreachable nodes are shaded.

Figure 2 shows the progress of the variable state as SSE iterates through the blocks. SSE starts from the entry of the CFG_e and executes basic blocks in topological order. SSE uses conditional *ite* (if-then-else) expressions—*ite* is a ternary operator similar to `?:` in C—to encode the behavior of multiple paths. For example, every variable assignment following the true branch after the condition ($x > 1$) in Figure 2 will be guarded as $ite(x > 1, value, \perp)$, where *value* denotes the assigned value and $\perp$ is a don't care term. Thus, for the edge from B3 to B6 in Figure 2, Δ is updated to $\{y \rightarrow ite(x > 1, 42, \perp)\}$.

When distinct paths (with distinct Δ 's) merge to the same confluence point on the CFG, a merge operator is needed to “combine” the side effects from all incoming edges. To do so, we apply the following recursive merge operation $\mathcal{M}$ to each symbolic value:

$$\begin{aligned} \mathcal{M}(v_1, \perp) &= v_1; & \mathcal{M}(\perp, v_2) &= v_2; \\ \mathcal{M}(ite(e, v_1, v_2), ite(e, v'_1, v'_2)) &= ite(e, \mathcal{M}(v_1, v'_1), \mathcal{M}(v_2, v'_2)) \end{aligned}$$

This way, at the last node of Figure 2, the value of y will be $\mathcal{M}(ite(x > 1, 42, \perp), ite(x > 1, \perp, ite(x < 42, 17, y_0)))$ which is merged to $ite(x > 1, 42, ite(x < 42, 17, y_0))$, capturing all possible paths.^c Note that this transformation is inlining multiple statements into a single one using *ite* operators. Also, note that values from unmerged paths ($\perp$ values) can be immediately simplified, for example, $ite(e, x, \perp) = x$. During SSE, MergePoint keeps a mapping from each traversed node to the corresponding variable state.

Handling Overapproximated CFGs. At any point during SSE, the path predicate is computed as the conjunction of the DSE predicate Π_{DSE} and the SSE predicate computed by substitution: Π_{SSE} . MergePoint uses the resulting predicate to perform path pruning offering two advantages: any infeasible edges introduced by CFG recovery are eliminated, and our formulas only consider feasible paths (e.g., the shaded

^c To efficiently handle deeply nested and potentially duplicated expressions, MergePoint utilizes hash-consing at the expression level.²

Figure 2. SSE running on an unrolled CFG—the variable state (Δ) is shown within brackets.

nodes in Figure 1b can be ignored).

3.5. Switch to DSE

After the SSE pass is complete, we check which states need to be forked. We first gather transition points and check whether they were reached by SSE. For the set of distinct, reachable transition points, MergePoint will fork a new symbolic state in a final step, where a DSE executor is created (pc, Π, Δ) using the state of each transition point.

Generating Test Cases. Though MergePoint can generate an input for each covered path, that would result in an exponential number of test cases in the size of the CFG. By default, we only output one test per CFG node explored by SSE. (Note that for branch coverage the algorithm can be modified to generate a test case for every edge of the CFG.) The number of test cases can alternatively be minimized by generating test cases only for nodes that have not been covered by previous test cases.

Underapproximated CFGs. Last, before proceeding with DSE, veritesting checks whether we missed any paths due to the underapproximated CFG. To do so, veritesting queries the negation of the path predicate at the `Exit` node (the disjunction of the path predicates of forked states). If the query is satisfiable, an extra state is forked to explore missed paths.

4. EVALUATION

In this section we evaluate our techniques using multiple benchmarks with respect to three main questions:

1. Does Veritesting find more bugs than previous approaches? We show that MergePoint with veritesting finds twice as many bugs than without.
2. Does Veritesting improve node coverage? We show MergePoint with veritesting improves node coverage over DSE.
3. Does Veritesting improve path coverage? Previous work showed dynamic state merging outperforms vanilla DSE.¹⁶ We show MergePoint with veritesting improves path coverage and outperforms both approaches.

We detail our large-scale experiment on 33,248 programs from Debian Linux. MergePoint generated billions of SMT queries, hundreds of millions of test cases, millions of crashes, and found 11,687 distinct bugs.

Overall, our results show MergePoint with veritesting improves performance on all three metrics. We also show that MergePoint is effective at checking a large number of programs. Before proceeding to the evaluation, we present our setup and benchmarks sets. All experimental data from MergePoint are publicly available online.²⁰

Experiment Setup. We ran all distributed MergePoint experiments on a private cluster consisting of 100 virtual nodes running Debian Squeeze on a single Intel 2.68GHz Xeon core with 1GB of RAM. All comparison tests against previous systems were run on a single node Intel Core i7 CPU and 16GB of RAM since these systems could not run on our distributed infrastructure.

We created three benchmarks: coreutils, BIN, and Debian. Coreutils and BIN were compiled so that coverage information could be collected via `gcov`. The Debian benchmark consists of binaries used by millions of users worldwide.

Benchmark 1: GNU coreutils (86 programs) We use the

coreutils benchmark to compare to previous work since: (1) the coreutils suite was originally used by KLEE⁶ and other researchers^{6, 7, 16} to evaluate their systems, and (2) configuration parameters for these programs used by other tools are publicly available.⁶ Numbers reported with respect to coreutils do not include library code to remain consistent with compared work. Unless otherwise specified, we ran each program in this suite for 1h.

Benchmark 2: The BIN suite (1023 programs). We obtained all the binaries located under the `/bin`, `/usr/bin`, and `/sbin` directories from a default Debian Squeeze installation.^d We kept binaries reading from `/dev/stdin`, or from a file specified on the command line. In a final processing step, we filtered out programs that require user interaction (e.g., GUIs). BIN consists of 1023 binary programs, and comprises 2,181,735 executable lines of source code (as reported by `gcov`). The BIN benchmark *includes* library code packaged with the application in the dataset, making coverage measurements more conservative than coreutils. For example, an application may include an entire library, but only one function is reachable from the application. We nonetheless include all uncovered lines from the library source file in our coverage computation. Unless otherwise specified, we ran each program in this suite for 30 min.

Benchmark 3: Debian (33,248 programs). This benchmark consists of all binaries from Debian Wheezy and Sid. We extracted binaries and shared libraries from every package available from the main Debian repository. We downloaded 23,944 binaries from Debian Wheezy, and 27,564 binaries from Debian Sid. After discarding duplicate binaries in the two distributions, we are left with a benchmark comprising 33,248 binaries. This represents an order of magnitude more applications than have been tested by prior symbolic execution research. We analyzed each application for less than 15 min per experiment.

4.1. Bug finding

Table 1 shows the number of bugs found by MergePoint with and without veritesting. Overall, veritesting finds 2× more bugs than without for BIN. Veritesting finds 63 (83%) of the bugs found without veritesting, as well as 85 additional distinct bugs that traditional DSE could not detect.

Veritesting also found two previously unknown crashes in coreutils, even though these applications have been thoroughly tested with symbolic execution.^{6, 7, 16} Further investigation showed that the coreutils crashes originate from a library bug that had been undetected for 9 years. The bug is in the time zone parser of the GNU portability library `Gnulib`, which dynamically deallocates a statically allocated memory buffer. It can be triggered by running `touch -d 'TZ=""' ,` or `date -d 'TZ=""' .` Furthermore, `Gnulib` is used by

^d What better source of benchmark programs than the ones you use everyday?

Table 1. Veritesting finds 2× more bugs.

	Veritesting	DSE
Coreutils	2 bugs/2 progs	0/0
BIN	148 bugs/69 progs	76 bugs/49 progs

several popular projects, and we have confirmed that the bug affects other programs, for example, `find`, `patch`, `tar`.

4.2. Node coverage

We evaluated MergePoint both with and without Veritest-ing on node coverage. Table 2 shows our overall results. Veritest-ing improves node coverage on average in all cases. Note that any positive increase in coverage is important. In particular, Kuznetsov et al. showed both dynamic state merging and SSE *reduced* node coverage when compared to vanilla DSE (Figure 8 in Ref.¹⁶).

Figures 3 and 4 break down the improvement per program. For coreutils, enabling veritest-ing decreased coverage in only three programs (`md5sum`, `printf`, and `pr`). Manual investigation of these programs showed that veritest-ing generated much harder formulas, and spent more than 90% of its time in the SMT solver, resulting in timeouts. In Figure 4 for BIN, we omit programs where node coverage was the same for readability. Overall, the BIN performance improved for 446 programs and decreased for 206.

Figure 5 shows the average coverage over time achieved by MergePoint with and without veritest-ing for the BIN suite. After 30 min, MergePoint without veritest-ing reached 34.45% code coverage. Veritest-ing achieved the same coverage in less than half the original time (12 min 48s). Veritest-ing's coverage improvement becomes more substantial as analysis time goes on. Veritest-ing achieved higher coverage

Table 2. Veritest-ing improves node coverage.

	Veritest-ing (%)	DSE (%)	Difference (%)
Coreutils	75.27	63.62	+11.65
BIN	40.02	34.71	+5.31

Figure 3. Code coverage difference on coreutils before and after veritest-ing.

Figure 4. Code coverage difference on BIN before and after veritest-ing, where it made a difference.

velocity, that is, the rate at which new coverage is obtained, than standard symbolic execution. Over a longer period of time, the difference in velocity means that the coverage difference between the two techniques is likely to increase further, showing that the longer MergePoint runs, the more essential veritest-ing becomes for high code coverage.

The above tests demonstrates the improvements of veritest-ing for MergePoint. We also ran both S2E and MergePoint (with veritest-ing) on coreutils using the same configuration for 1 h on each utility in coreutils, excluding 11 programs where S2E emits assertion errors. Figure 6 compares the increase in coverage obtained by MergePoint with veritest-ing over S2E. MergePoint achieved 27% more code coverage on average than S2E. We investigated programs where S2E outperforms MergePoint. For instance, on `pinky`—the main outlier in the distribution—S2E achieves 50% more coverage. The main reason for this difference is that `pinky` uses a system call not handled by the current MergePoint implementation (netlink socket).

4.3. Path coverage

We evaluated the path coverage of MergePoint both with and without veritest-ing using three different metrics: time to complete exploration, as well as multiplicity.

Time to complete exploration. The metric reports the amount of time required to completely explore a program, in those cases where exploration finished.

The number of paths checked by an exhaustive DSE run is also the total number of paths possible. In such cases we can measure (a) whether veritest-ing also completed, and (b) if so, how long it took relative to DSE. MergePoint without veritest-ing was able to exhaust all paths for 46 programs. MergePoint with veritest-ing completes all paths 73% faster than without veritest-ing. This result shows that veritest-ing

Figure 5. Coverage over time (BIN suite).

Figure 6. Code coverage difference on coreutils obtained by MergePoint versus S2E.

is faster when reaching the same end goal.

Multiplicity. Multiplicity was proposed by Kuznetsov et al.¹⁶ as a metric correlated with path coverage. The initial multiplicity of a state is 1. When a state forks, both children inherit the state multiplicity. When combining two states, the multiplicity of the resulting state is the sum of their multiplicities. A higher multiplicity indicates higher path coverage.

We also evaluated the multiplicity for veritesting. Figure 7 shows the state multiplicity probability distribution function for BIN. The average multiplicity over all programs was 1.4×10^{290} and the median was 1.8×10^{12} (recall, higher is better). The distribution resembles a lognormal with a spike for programs with multiplicity of 4096 (2^{12}). The multiplicity average and median for coreutils were 1.4×10^{199} and 4.4×10^{11} , respectively. Multiplicity had high variance; thus the median is likely a better performance estimator.

4.4. Checking Debian

In this section, we evaluate veritesting's bug finding ability on every program available in Debian Wheezy and Sid. We show that veritesting enables large-scale bug finding.

Since we test 33,248 binaries, any type of per-program manual labor is impractical. We used a single input specification for our experiments: `-sym-arg 1 10 -sym-arg 2 2 -sym-arg 3 2 -sym-anon-file 24 -sym-stdin 24` (3 symbolic arguments up to 10, 2, and 2 bytes, respectively, and symbolic files/stdin up to 24 bytes). MergePoint encountered at least one symbolic branch in 23,731 binaries. We analyzed Wheezy binaries once, and Sid binaries twice (one experiment with a 24-byte symbolic file, the other with 2100 bytes to find buffer overflows). Including data

Figure 7. Multiplicity distribution (BIN suite).

Table 3. Overall numbers for checking Debian.

Total programs	33,248
Total SMT queries	15,914,407,892
Queries hitting cache	12,307,311,404
Symbolic instrs	71,025,540,812
Run time	235,623,757 s
Symb exec time	125,412,247 s
SAT time	40,411,781 s
Model gen time	30,665,881 s
# test cases	199,685,594
# crashes	2,365,154
# unique bugs	11,687
# reported bugs	1043
# fixed bugs	229

processing, the experiments took 18 CPU-months.

Our overall results are shown in Table 3. Veritesting found 11,687 distinct bugs that crash programs. The bugs appear in 4379 of the 33,248 programs. Veritesting also finds bugs that are potential security threats. Two hundred and twenty-four crashes have a corrupt stack, that is, a saved instruction pointer has been overwritten by user input. As an interesting data point, it would have cost \$0.28 per unique crash had we run our experiments on the Amazon Elastic Compute Cloud, assuming that our cluster nodes are equivalent to large instances.

The volume of bugs makes it difficult to report all bugs in a usable manner. Note that each bug report includes a crashing test case, thus reproducing the bug is easy. Instead, practical problems such as identifying the correct developer and ensuring responsible disclosure of potential vulnerabilities dominate our time. As of this writing, we have reported 1043 crashes in total.¹⁹ Not a single report was marked as unreproducible on the Debian bug tracking system. Two hundred and twenty-nine bugs have already been fixed in the Debian repositories, demonstrating the real-world impact of our work. Additionally, the patches gave an opportunity to the package maintainers to harden at least 29 programs, enabling modern defenses like stack canaries and DEP.

4.5. Discussion

Our experiments so far show that veritesting can effectively increase multiplicity, achieve higher code coverage, and find more bugs. In this section, we discuss why it works well according to our collected data.

Each run takes longer with veritesting because multi-path SMT formulas tend to be harder. The coverage improvement demonstrates that additional SMT cost is amortized over the increased number of paths represented in each run. At its core, veritesting is pushing the SMT engine harder instead of brute-forcing paths by forking new DSE executors. This result confirms that the benefits of veritesting outweigh its cost. The distribution of path times (Figure 8b) shows that the vast majority (56%) of paths explored take less than 1 s for standard symbolic execution. With veritesting, the fast paths are fewer (21%), and we get more timeouts (6.4% vs.

Figure 8. MergePoint performance before and after veritesting for BIN. The above figures show: (a) Performance breakdown for each component; (b) Analysis time distribution.

Component	DSE (%)	Veritesting (%)
Instrumentation	40.01	16.95
SMT solver	19.23	63.16
Symbolic execution	39.76	19.89

(a)

1.2%). The same differences are also reflected in the component breakdown. With veritesting, most of the time (63%) is spent in the solver, while with standard DSE most of the time (60%) is spent re-executing similar paths that could be merged and explored in a single execution.

Of course there is no free lunch, and some programs do perform worse. We emphasize that on average over a fairly large dataset our results indicate the tradeoff is beneficial.

5. RELATED WORK

Symbolic execution was discovered in 1975,¹⁴ with the volume of academic research and commercial systems exploding in the last decade. Notable symbolic executors include SAGE and KLEE. SAGE⁴ is responsible for finding one third of all bugs discovered by file fuzzing during the development of Windows 7.⁴ KLEE⁶ was the first tool to show that symbolic execution can generate test cases that achieve high coverage on real programs by demonstrating it on the UNIX utilities. There is a multitude of symbolic execution systems—for more details, we refer the reader to recent surveys.^{5,21}

Merging execution paths is not new. Koelbl and Pixley¹⁵ pioneered path merging in SSE. Concurrently and independently, Xie and Aiken²³ developed Saturn, a verification tool capable of encoding of multiple paths before converting the problem to SAT. Hansen et al.¹³ follow an approach similar to Koelbl et al. at the binary level. Babic and Hu³ improved their static algorithm to produce smaller and faster to solve formulas by leveraging GSA.²² The static portion of our veritesting algorithm is built on top of their ideas. In our approach, we alternate between SSE and DSE. Our approach amplifies the effect of DSE and takes advantage of the strengths of both techniques.

The efficiency of the static algorithms mentioned above typically stems from various types of *if-conversion*,¹ a technique for converting code with branches into predicated straightline statements. The technique is also known as ϕ -folding,¹⁷ a compiler optimization technique that collapses simple diamond-shaped structures in the CFG. Godefroid¹¹ introduced function summaries to test code compositionally. The main idea is to record the output of an analyzed function, and reuse it whenever the function is called again. Veritesting generates context-sensitive on-demand summaries of code fragments as the program executes—extending to compositional summaries is possible future work.

6. CONCLUSION

In this article we proposed MergePoint and veritesting, a new technique to enhance symbolic execution with verification-based algorithms. We evaluated MergePoint on 1023 programs and showed that veritesting increases the number of bugs found, node coverage, and path coverage. We showed that veritesting enables large-scale bug finding by testing 33,248 Debian binaries, and finding 11,687 bugs. Our results have had real world impact with 229 bug fixes already present in the latest version of Debian.

Acknowledgments

We would like to thank Samantha Gottlieb, Tiffany Bao, and our anonymous reviewers for their comments and suggestions. We also thank Mitch Franzos and PDL for the support

they provided during our experiments. This research was supported in part by grants from DARPA and the NSF, as well as the Prabhu and Poonam Goel Fellowship.

References

- Allen, J.R., Kennedy, K., Porterfield, C., Warren, J. Conversion of control dependence to data dependence. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Austin, Texas, 1983). ACM Press, New York, NY, 177–189.
- Avgerinos, T., Rebert, A., Cha, S.K., Brumley, D. Enhancing symbolic execution with veritesting. In *Proceedings of the 36th International Conference on Software Engineering, ICSE 2014* (Hyderabad, India, 2014). ACM, New York, NY, 1083–1094. DOI: 10.1145/2568225.2568293. URL <http://doi.acm.org/10.1145/2568225.2568293>.
- Babic, D., Hu, A.J. Calysto: Scalable and precise extended static checking. In *Proceedings of the 30th International Conference on Software Engineering* (Leipzig, Germany, 2008). ACM, New York, NY, 211–220.
- Bounimova, E., Godefroid, P., Molnar, D. Billions and billions of constraints: Whitebox Fuzz testing in production. In *Proceedings of the 35th IEEE International Conference on Software Engineering* (San Francisco, CA, 2013). IEEE Press, Piscataway, NJ, 122–131.
- Cadar, C., Sen, K. Symbolic execution for software testing: three decades later. *Commun. ACM* 56, 2 (2013), 82–90.
- 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 Symposium on Operating System Design and Implementation* (San Diego, CA, 2008). USENIX Association, Berkeley, CA, 209–224.
- Cha, S.K., Avgerinos, T., Rebert, A., Brumley, D. Unleashing mayhem on binary code. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy* (2012). IEEE Computer Society, Washington, DC, 380–394.
- Chipounov, V., Kuznetsov, V., Candea, G. S2E: A platform for in vivo multi-path analysis of software systems. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems* (Newport Beach, CA, 2011). ACM, New York, NY, 265–278.
- de Moura, L., Bjørner, N. Satisfiability modulo theories: Introduction and applications. *Commun. ACM* 54, 9 (Sept. 2011), 69. ISSN 00010782. doi: 10.1145/1995376.1995394. URL <http://dl.acm.org/citation.cfm?doid=1995376.1995394>.
- Flanagan, C., Saxe, J. Avoiding exponential explosion: Generating compact verification conditions. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (London, United Kingdom, 2001). ACM, New York, NY, 193–205.
- Godefroid, P. Compositional dynamic test generation. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Nice, France, 2007). ACM, New York, NY, 47–54.
- Godefroid, P., Levin, M.Y., Molnar, D. SAGE: Whitebox fuzzing for security testing. *Commun. ACM* 55, 3 (2012), 40–44.
- Hansen, T., Schachte, P., Søndergaard, H. State joining and splitting for the symbolic execution of binaries. *Runtime Verif.* (2009), 76–92.
- King, J.C. Symbolic execution and program testing. *Commun. ACM* 19, 7 (1976), 385–394.
- Koelbl, A., Pixley, C. Constructing efficient formal models from high-level descriptions using symbolic simulation. *Int. J. Parallel Program.* 33, 6 (Dec. 2005), 645–666.
- Kuznetsov, V., Kinder, J., Bucur, S., Candea, G. Efficient state merging in symbolic execution. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China, 2012). ACM, New York, NY, 193–204.
- Lattner, C., Adve, V. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization* (Palo Alto, CA, 2004). IEEE Computer Society, Washington, DC, 75–86.
- Leino, K.R.M. Efficient weakest preconditions. *Inform. Process. Lett.* 93, 6 (2005), 281–288.
- Mayhem. 1.2K Crashes in Debian, 2013. URL <http://lists.debian.org/debian-devel/2013/06/msg00720.html>.
- Mayhem. Open Source Statistics & Analysis, 2013. URL <http://www.forallsecure.com/summaries>.
- Schwartz, E.J., Avgerinos, T., Brumley, D. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the 31st IEEE Symposium on Security and Privacy* (2010). IEEE Computer Society, Washington, DC, 317–331.
- Tu, P., Padua, D. Efficient building and placing of gating functions. In *Proceedings of the 16th ACM Conference on Programming Language Design and Implementation* (La Jolla, CA, 1995). ACM, New York, NY, 47–55.
- Xie, Y., Aiken, A. Scalable error detection using boolean satisfiability. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Long Beach, CA, 2005). ACM, New York, NY, 351–363.

Thanassis Avgerinos, Alexandre Rebert, and David Brumley ([thanassis, alex]@forallsecure.com), For AllSecure, Inc., Pittsburgh, PA.

Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley ([sangkilc, dbrumley]@cmu.edu), Carnegie Mellon University, Pittsburgh, PA.