Enhancing Symbolic Execution with Veritesting

Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley
Carnegie Mellon University
{thanassis, alexandre, sangkilc, dbrumley}@cmu.edu

ABSTRACT

We present MergePoint, a new binary-only symbolic execution system for large-scale testing of commodity off-the-shelf (COTS) software. MergePoint introduces *veritesting*, a new technique that employs static symbolic execution to amplify the effect of dynamic symbolic execution. Veritesting allows MergePoint to find twice as many bugs, explore orders of magnitude more paths, and achieve higher code coverage than previous dynamic symbolic execution systems. MergePoint is currently running daily on a 100 node cluster analyzing 33,248 Linux binaries; has generated more than 15 billion SMT queries, 200 million test cases, 2,347,420 crashes, and found 11,687 bugs in 4,379 distinct applications.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Symbolic execution*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program analysis*

General Terms

Algorithms, Security, Verification

Keywords

Veritesting, Symbolic Execution, Verification

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 [15, 26], generating test cases with high coverage [16], exposing software vulnerabilities [10], and generating exploits [18].

Symbolic execution is attractive because it systematically explores the program and produces real inputs. Symbolic execution works by automatically translating a program fragment to a logical formula. The logical formula is satisfied by inputs that have a desired property, e.g., they execute a specific path or violate safety.

At a high level, there are two main approaches for generating formulas. First, dynamic symbolic execution (DSE) explores programs and generates formulas on a per-path basis. Second, static symbolic execution (SSE) translates program statements into formulas, where the formulas represent the desired property over any path within the selected statements.

In this paper we describe MergePoint, a system for automatically checking all programs in a Linux distribution using a new technique called *veritesting*. 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. Veritesting *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 MergePoint with veritesting to switch to DSE-based methods when such cases are encountered.

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** (by stack hash) in 4,379 different programs. Overall, MergePoint has generated over 15 billion SMT queries and created over 200 million test cases. Out of the 11,687 bugs, 224 result in user input overwriting the instruction pointer, and we have confirmed shell-spawning exploits for 152.

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 162 patches due to our bug reports. We have made our data available on our website [41].

2. OVERVIEW

At a high level, symbolic execution can be partitioned into two main approaches: dynamic symbolic execution for testing, and static symbolic execution for verification. Dynamic approaches work by generating per-path formulas to test specific paths, while static-based approaches generate formulas over entire programs with the goal of verifying overall safety. Our main insight is to carefully *alternate* between the two schemes to harness the benefits of both while mitigating path explosion in dynamic approaches and solver blowup in static approaches. In particular, we start with dynamic symbolic execution, but switch to a static verification-based approach opportunistically. When we switch to static mode, we only check *program fragments* with the goal of testing, not verification. While we are not the first to suggest using static and dynamic techniques in combination, the careful application of alternation as proposed in veritesting reduces overall overhead, and results in improved performance along key metrics. Previous approaches typically lost on at least one metric, and sometimes several.

In this section we provide a high-level overview of standard metrics, the key parts of dynamic and static algorithms, as well as the tradeoffs between approaches.

2.1 Testing Metrics

Testing systems, including dynamic symbolic execution systems, are typically evaluated using three metrics: 1) number of real bugs found, 2) node coverage, and 3) path coverage.

Node (or code or line or statement) coverage measures the percentage of code covered by generated test cases with respect to the entire application. Node coverage is an effective way of measuring the performance of a test case generation system [49] and has been used repeatedly to measure symbolic execution systems' performance [16, 35].

Path coverage measures the percentage of program paths analyzed. Unlike node coverage, which has a finite domain (the total number of program statements), many programs have a potentially infinite number of paths (e.g., a server) and measuring the path coverage is not possible. In our evaluation, we use three distinct metrics for approximating path coverage §6.3.

The number of unique bugs is measured by counting the number of unique stack hashes [42] among crashes. We report bugs only when a generated test case can produce a core file during concrete execution.

All three metrics are important, and none dominates in all scenarios. For example, node coverage is useful, but even 100% node coverage may fail to find real bugs. Also, it may be possible to achieve 100% node coverage but never execute a loop more than once. Bugs that require several iterations to trigger, e.g., buffer overflows, will be missed. Testing more paths is better, but an analysis could game the metric by simply iterating over fast-to-execute loops more times and avoiding slow execution paths. Again, bugs may be missed and nodes may not be covered. One could find all bugs, but never know it because not all paths are exhausted.

Algorithm 1: Dynamic Symbolic Execution Algorithm with and without Veritesting

```
   Input: Initial location $\ell_0$, instruction decoder instrAt
   Data: Worklist $\mathcal{W}$, path predicate $\Pi$, symbolic store $\Delta$
1  $\mathcal{W} \leftarrow \{(\ell_0, \text{true}, \varnothing)\}$                    // initial worklist
2  while $\mathcal{W} \neq \varnothing$ do
3      $((\ell, \Pi, \Delta), \mathcal{W}) \leftarrow$ pickNext($\mathcal{W}$)
       // Symbolically execute the next instruction
4      switch instrAt($\ell$) do
5          case $v := e$                                  // assignment
6              $\mathcal{S} \leftarrow \{(\texttt{succ}(\ell), \Pi, \Delta[v \rightarrow \texttt{eval}(\Delta, e)])\}$
7          case if ($e$) goto $\ell'$                   // conditional jump
8              $e \leftarrow \texttt{eval}(\Delta, e)$
9              if (isSat($\Pi \wedge e$) $\wedge$ isSat($\Pi \wedge \neg e$)) then
10                 // DSE forks 2 states
11                 $\mathcal{S} \leftarrow \{(\ell', \Pi \wedge e, \Delta), (\texttt{succ}(\ell), \Pi \wedge \neg e, \Delta)\}$
10                 // Veritesting integration
11                 $\mathcal{S} \leftarrow \varnothing$
12                 $CFG \leftarrow$ CFGRecovery($\ell, \Pi$)
13                 $CFG_e, TransitionPoints \leftarrow$ CFGReduce($CFG$)
14                 $OUT \leftarrow$ StaticSymbolic($CFG_e, \Pi, \Delta$)
15                 for $Point \in TransitionPoints$ do
16                     if $OUT[Point] \neq \emptyset$ then
17                         $\mathcal{S} \leftarrow OUT[Point] \cup \mathcal{S}$
18                 $\mathcal{S} \leftarrow$ Finalize($\mathcal{S}$)
19             else if isSat($\Pi \wedge e$) then
20                 $\mathcal{S} \leftarrow \{(\ell', \Pi \wedge e, \Delta)\}$
21             else $\mathcal{S} \leftarrow \{(\texttt{succ}(\ell), \Pi \wedge \neg e, \Delta)\}$
22         case assert($e$)                                // assertion
23             $e \leftarrow \texttt{eval}(\Delta, e)$
24             if isSat($\Pi \wedge \neg e$) then reportBug($\Pi \wedge \neg e$)
25             $\mathcal{S} \leftarrow \{(\texttt{succ}(\ell), \Pi \wedge e, \Delta)\}$
26         case halt: continue ;                      // end of path
27     $\mathcal{W} \leftarrow \mathcal{W} \cup \mathcal{S}$
```

2.2 Dynamic Symbolic Execution (DSE)

Algorithm 1 presents the core steps in dynamic symbolic execution. The algorithm operates on a representative imperative language with assignments, assertions and conditional jumps (adapted from previous work [35]). A symbolic executor maintains a state (ℓ, Π, Δ) where ℓ is the address of the current instruction, Π is the path predicate, and Δ is a symbolic store that maps each variable to either a concrete value or an expression over input variables. A satisfying assignment, typically checked by a SAT or SMT solver, is an assignment of values to symbolic input variables that will execute the same execution path. An unsatisfiable path predicate means the selected path is infeasible.

On line 1, the algorithm initializes the worklist with a state pointing to the start of the program. The `pickNext` function selects the next state to continue executing, and removes it from the worklist $\mathcal{S}$. There are a variety of search heuristics for selecting the next instruction to execute, including starting with a concrete trace [26, 46], generational search [28], DFS, and BFS. Symbolic execution switches over the instruction types in line 4. Safety checks are performed with assertions. For example, every memory dereference is preceded by an assertion that checks whether the pointer is in bounds. The semantics of assignment, assert, and halt are all straightforward. The central design point we focus on in this paper is handling a branch instruction, shown in line 7.

The two instances of line 11 contrast our approach with others'. In DSE, whenever both branches are feasible, two new states are added to the worklist (one for the true branch and one for the false), a process we refer to as "forking". Each one of the forked executors is later chosen from the worklist and explored independently.

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 SAGE [10] 99% of all queries takes less than 1 second) 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 the *path (or state) explosion*[1] problem: the number of executors can grow exponentially in the number of branches. The path explosion problem is the motivation for our veritesting algorithm §3.

2.3 Static Symbolic Execution (SSE)

Static Symbolic Execution (SSE) is a verification technique for representing a program as a logical formula. Potential vulnerabilities are encoded as logical assertions that will falsify the formula if safety is violated. Calysto [6] and Saturn [22, 48] are example SSE tools. 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 testing program fragments instead of verifying complete programs.

The main change is on line 11 of Algorithm 1. Modern SSE algorithms can 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 [6, 34, 48]. (§3 shows our SSE algorithm using a dataflow framework.)

Advantages/Disadvantages. Unlike DSE, SSE does not suffer from path explosion. All paths are encoded in a single formula that is then passed to the solver (note the solver may still have to reason internally about an exponential number of paths). For acyclic programs, existing techniques allow generating compact formulas of size $O\left(n^2\right)$ [24, 37], 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 [8, 30, 35]. 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? [22]), 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.

[1]Depending on the context, the two terms may be used interchangeably [14, 35]—an "execution state" corresponds to a program path to be explored.

3. VERITESTING

DSE has proven to be effective in analyzing real world programs [16, 27]. 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:

```
int counter = 0, values = 0;
for ( i = 0 ; i < 100 ; i ++ ) {
    if (input[i] == 'B') {
        counter ++;
        values += 2;
    } }
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 testcases 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[2]. We ran the above program with several state-of-the-art symbolic executors, including KLEE [16], S2E [19], Mayhem [18] and Cloud9 with state merging [35]. None of the above systems was able to find the bug within a 1-hour time limit (they ran out of memory or kept running). Veritesting allows us to find the bug and obtain full path coverage in 47 seconds 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 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.

Conceptually, the closest recent work to ours is dynamic state merging (DSM) by Kuznetsov *et al.* [35]. DSM maintains a history queue of DSE executors. Two DSEs may merge (depending on a separate and independent heuristic for SMT query difficulty) if they coincide in the history queue. Fundamentally, however, DSM still performs per-path execution, and only opportunistically merges. Veritesting always merges, using SSE (not DSE) on all statements within a fixed lookahead. The result is Veritesting formulas cover more paths than DSE (at the expense of longer SMT queries), but avoid the overhead of managing a queue and merging path-based executors.

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 concolic executor [46]. It starts exploration with a concrete

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

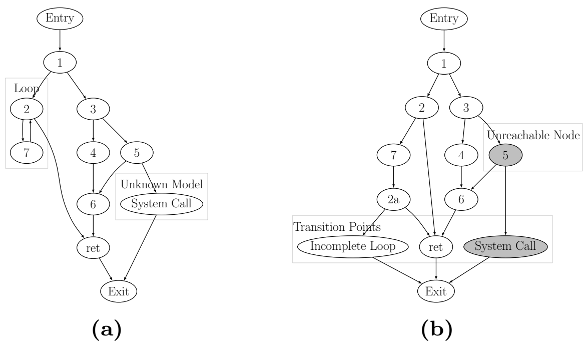


Figure 1: Veritesting on a program fragment with loops and system calls. (a) Recovered CFG. (b) CFG after transition point identification & loop unrolling. Unreachable nodes are shaded.

seed and explores paths in the neighborhood of the original seed following a generational search strategy [27]. MergePoint does not always fork when it encounters a symbolic branch. Instead, MergePoint intercepts the forking process—as shown in line 11 of algorithm 1—of DSE and performs veritesting.

Algorithm 1 presents the high-level process of veritesting. The algorithm augments DSE with 4 new steps:

1. `CFGRecovery`: recovers the CFG reachable from the address of the symbolic branch (§3.2).
2. `CFGReduce`: takes in a CFG, and outputs candidate transition points and a CFG$_e$, an acyclic CFG with edges annotated with the control flow conditions (§3.3). Transition points indicate program locations where DSE may continue.
3. `StaticSymbolic`: 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 (§3.4).
4. `Finalize`: given a list of transition points and SSE states, returns the DSE executors to be forked (§3.5).

3.2 CFG Recovery

The goal of the CFG recovery phase is to obtain a partial control flow graph 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 (non-approximate) CFG on binary code is known to be a hard problem and the subject of active research [7, 32]. 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 §3.4). MergePoint uses the CFG recovery mechanism from our Binary Analysis Platform (BAP) [12]. The algorithm is customized to stop recovery at function boundaries, system calls and unknown instructions.

The output of this step is a partial (possibly approximate) intra-procedural control flow graph. 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 & Unrolling

Once the CFG is obtained, MergePoint proceeds to identifying transition points. Transition points define the boundary of the SSE algorithm (where DSE will continue exploration). To calculate transition points, we require the notion of postdominators and immediate postdominators:

Definition 1 (Postdominator). *A node d postdominates a node n, denoted as* pdom *(d,n), iff every path from n to the exit of the graph goes through d.*

Definition 2 (Immediate Postdominator). *A node d immediately postdominates node n, denoted as* ipdom *(d,n), iff:* $\mathsf{pdom}(d,n) \land \neg\exists z \neq d : \mathsf{pdom}(d,z) \land \mathsf{pdom}(z,n)$.

Transition Points. For an entry node e ending in a symbolic branch, a transition point is defined as a node n such that $\mathsf{ipdom}(e,n)$. For a fully recovered CFG, a single transition point may be sufficient, e.g., 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 states 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, e.g., 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.

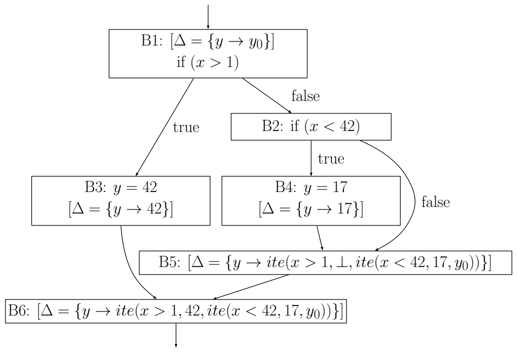
3.4 Static Symbolic Execution

Given the CFG$_e$, MergePoint applies SSE to summarize the execution of multiple paths. Previous work [5] first converted the program to Gated Single Assignment (GSA) [47] and then performed symbolic execution. In MergePoint, we encode SSE as a single pass dataflow analysis where GSA is computed on the fly. Table 1 presents the SSE algorithm, following standard notation [2, Section 9].

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

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

Figure 2 shows the progress of the symbolic store as SSE iterates through the blocks. SSE starts from the entry of the CFG$_e$ and executes basic blocks in topological order. Basic blocks contain straightline code and execution follows Algorithm 2, taking as input (from $IN[B]$) a path context Γ and a symbolic store Δ and outputting the updated versions (for $OUT[B]$). Γ enables multi-path SSE by encoding the conditionals required to follow an execution path using *ite*



B1: $[\Delta = \{y \to y_0\}]$
if $(x > 1)$
true
false
B2: if $(x < 42)$
true
false
B3: $y = 42$
$[\Delta = \{y \to 42\}]$
B4: $y = 17$
$[\Delta = \{y \to 17\}]$
B5: $[\Delta = \{y \to ite(x > 1, \bot, ite(x < 42, 17, y_0))\}]$
B6: $[\Delta = \{y \to ite(x > 1, 42, ite(x < 42, 17, y_0))\}]$

Figure 2: Symbolic store transformations during SSE.

Algorithm 2: Veritesting Transfer Function

```
Input: Basic block B, Path context $\Gamma$, Symbolic store $\Delta$
1 foreach $inst \in B$ do
2     switch $inst$ do
3         case $v := e$
4             $\Delta \leftarrow \Delta[v \to \texttt{eval}(\Delta, e)]$
5         case assert($e$)
6             $\Gamma \leftarrow \Gamma[\Lambda \to ite(\texttt{eval}(\Delta, e), \Lambda, \bot)]$
7 return $\Gamma$, $\Delta$
```

(if-then-else) expressions. For example, following the true branch after the condition $(x > 1)$ in Figure 2 gives: $\Gamma = ite(x > 1, \Lambda, \bot)$, where Λ denotes the taken path and $\bot$ the non-taken.

To compute the input set ($IN[B]$) for a basic block we apply a meet operation across all incoming states from predecessor blocks following Algorithm 3. The path context is obtained for each incoming edge and then applied to the symbolic store. For example, for the edge from B3 to B6 in Figure 2, Δ is updated to $\{y \to \Gamma_3[\Lambda \to \Delta[y]] = ite(x > 1, 42, \bot)\}$. To merge Δ's (or Γ's) from paths that merge to the same confluence point, we apply the following recursive merge operation $\mathcal{M}$ to each symbolic value:

$$\mathcal{M}(v_1, \bot) = v_1; \quad \mathcal{M}(\bot, v_2) = v_2;$$
$$\mathcal{M}(ite(e, v_1, v_2), ite(e, v_1', v_2')) = ite(e, \mathcal{M}(v_1, v_1'), \mathcal{M}(v_2, v_2'))$$

This way, at the last node of Figure 2, the value of y will be $\mathcal{M}(ite(x > 1, 42, \bot), ite(x > 1, \bot, ite(x < 42, 17, y_0)))$ which is merged to $ite(x > 1, 42, ite(x < 42, 17, y_0))$, capturing all possible paths. During SSE, MergePoint keeps a mapping from each traversed node to the corresponding state (OUT). Note that values from unmerged paths ($\bot$ values) can be immediately simplified, e.g., $ite(e, x, \bot) = x$.

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: $\Pi_{SSE} = \Gamma[\Lambda \to \mathsf{true}, \bot \to \mathsf{false}]$. MergePoint uses the resulting predicate to perform path pruning (lines 4 and 6 in Algorithm 3) offering two advantages: any infeasible edges introduced by CFG recovery are eliminated, and our formulas only consider feasible paths.

Algorithm 3: Veritesting Meet Function

```
Input: Basic block $B$, Pred. blocks $B_1$, $B_2$, Store $\Pi_{DSE}$
1 function Context ($B$, $Parent$) begin
2     $\Gamma, \Delta \leftarrow \texttt{OUT}(Parent)$; $taken, e \leftarrow \texttt{edge}(Parent, B)$;
3     $e \leftarrow \texttt{eval}(\Delta, e)$; $\Pi \leftarrow \Pi_{DSE} \wedge \Gamma[\Lambda \to \mathsf{true}, \bot \to \mathsf{false}]$;
4     if $taken \wedge \texttt{isSat}(\Pi \wedge e)$ then
5         return $\Gamma[\Lambda \to ite(e, \Lambda, \bot)], \Delta$
6     else if $\neg taken \wedge \texttt{isSat}(\Pi \wedge \neg e)$ then
7         return $\Gamma[\Lambda \to ite(e, \bot, \Lambda)], \Delta$
8     else return $\bot, \emptyset$;            // infeasible edge
9 $\Gamma_1, \Delta_1 \leftarrow \texttt{Context}(B, B_1)$; $\Gamma_2, \Delta_2 \leftarrow \texttt{Context}(B, B_2)$;
10 $\Gamma \leftarrow \mathcal{M}(\Gamma_1, \Gamma_2)$; $\Delta \leftarrow \Delta_1$;
11 foreach $v \in \Delta_2$ do
12     $\Delta[v] = \mathcal{M}(\Gamma_1[\Lambda \to \Delta_1[v]], \Gamma_2[\Lambda \to \Delta_2[v]])$
13 return $\Gamma$, $\Delta$
```

3.5 Transition Point Finalization

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 (line 16 in Algorithm 1). For the set of distinct—based on their jump target address—transition points, MergePoint will fork a new symbolic state in a `Finalize` step, where a DSE executor is created (ℓ, Π, Δ) using the state (Γ, Δ) 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$_e$. By default, we only output one test per CFG node explored by static symbolic execution. (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.

Incremental Deployment. Veritesting is an online algorithm, it runs as the program executes. If any step of the veritesting algorithm fails, the system falls back to DSE until the next symbolic branch. An advantage of this approach

Table 1: SSE as a dataflow algorithm. $IN[B]$ and $OUT[B]$ denote the input and output sets of basic block B.

Domain	Symbolic execution state (Γ, Δ)
Direction	Forwards
Transfer Function	Algorithm 2
Boundary	Initial execution state (Λ, Δ_{init})
Initialize	$OUT[B] = (\bot, \emptyset)$
Dataflow Equations	$IN[B] = \bigwedge_{P, pred(B)} OUT[P]$ $OUT[B] = f_B(IN[B])$
Meet Function	Algorithm 3

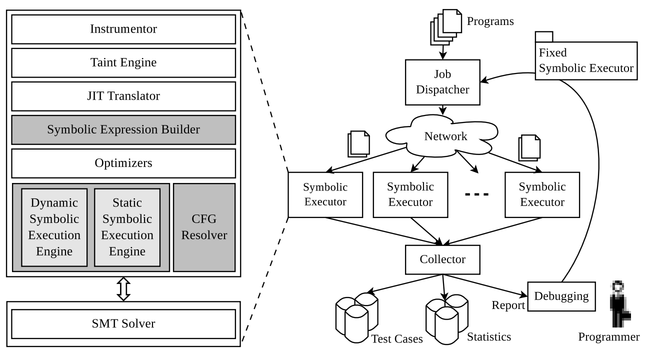
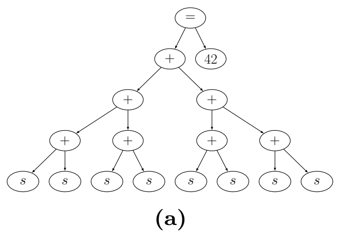


Figure 3: MergePoint Architecture.



```
x = s   (s is symbolic)
x = x + x
x = x + x
x = x + x
 assert (x == 42)
```

Figure 4: Hash consing example. Top-left: naïvely generated formula. Top-right: hash-consed formula.

is that the implementation can be gradually deployed; supporting all possible programming constructs is not necessary, since veritesting runs on a best-effort basis.

4. MERGEPOINT ARCHITECTURE

The ultimate goal of MergePoint is to perform effective testing on thousands of applications. In this section, we provide a high-level description of the system and key design choices.

4.1 Overview

MergePoint follows the design of a concolic executor. The symbolic execution engine runs on top of an instrumentation tool and x86 instructions are JITed to an intermediate representation before being symbolically executed. A taint analysis layer ensures that the symbolic executor is used only when necessary, i.e., only for instructions operating on input-derived data. The layers of the MergePoint executor are shown on the left of Figure 3.

To enable veritesting, the MergePoint executor is enhanced with two main modules (shaded): a static symbolic executor and a CFG recovery module. In the rest of this section, we discuss how the executor fits within the MergePoint distributed infrastructure (§4.2), and a key design decision in the handling of symbolic expressions (§4.3).

4.2 Distributed Infrastructure

As a stand-alone tool, the MergePoint executor takes in a program and a user configuration (including a time limit, inputs, etc.) and outputs test cases, bugs, and statistics. One goal of MergePoint is to test software en masse. However, a single 30-minute experiment on 1,000 programs requires almost 3 weeks of CPU time. To test our techniques, we developed a distributed infrastructure that utilizes multiple nodes to run programs in parallel. Figure 3 presents the end-to-end architecture of MergePoint.

MergePoint employs a first-come, first-served central queuing policy. The policy is simple, yet yields high-utilization: a program waiting at the top of the dispatcher queue is sent to the next available symbolic executor instance.

Data generated at every node are aggregated and stored in centralized storage. We use stored data as an immediate feedback mechanism about the performance behavior of the symbolic executor on a large number of programs. The feedback mechanism served as a guide on several design choices, e.g., using a hash-consed language (§4.3).

4.3 A Hash-Consed Expression Language

Whenever a program variable is used in an expression, `eval` in Algorithm 1 replaces it with its value in the symbolic store. A naïve substitution algorithm may introduce an exponential blowup, even for a straightline program. For example, the path predicate for Figure 4 is $s+s+s+s+s+s+s+s = 42$ (where there are $2^3 = 8$ uses of the s variable).

Hash consing [29] is a technique for avoiding duplication during substitution and reusing previously constructed expressions. Previous work in symbolic execution has made extensive use of hash-consing variants to avoid duplicate expressions. Examples include creating maximally-shared graphs [5], using expression caching [10], or ensuring that structurally equivalent expressions that are passed to the SMT solver are reused [16].

MergePoint goes one step further and builds hash-consing *into* the language. The constructor for every expression type is hash-consed by default, meaning that the implementor of the symbolic executor is incapable of creating duplicate expressions. Every previously computed expression is stored and will be reused. MergePoint also provides iterators over hash-consed expressions for standard operations (fold, map, map-reduce, etc.), to ensure all traversals are linear in the size of the expression.

Following the approach of [23], MergePoint stores hash-consed expressions in an array of weak references that can be efficiently garbage collected.

5. IMPLEMENTATION

MergePoint runs in a virtual machine cloud. Our architecture uses a central dispatcher to send individual programs to analysis nodes. The main MergePoint Veritesting implementation is built on top of Mayhem [18], and consists of an additional 17,000 lines of OCaml and 9,000 lines of C/C++. The communication between multiple nodes and the dispatcher is implemented in 3,500 lines of Erlang. MergePoint uses the BAP [12] platform for translating x86 code to an intermediate representation, CFG recovery and loop unrolling. We use PIN [38] for instrumentation and Z3 [21] for solving SMT queries.

6. 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 [35]. 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 [41].

Experiment Setup. We ran all distributed MergePoint experiments on a private cluster consisting of 100 virtual nodes running Debian Squeeze on a single Intel 2.68 GHz Xeon core with 1GB of RAM. All comparison tests against previous systems were run on a single node Intel Core i7 CPU and 16 GB 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)[3]. We use the coreutils benchmark to compare to previous work since: 1) the coreutils suite was originally used by KLEE [16] and other researchers [13, 16, 18, 35, 39] to evaluate their systems, and 2) configuration parameters for these programs used by other tools are publicly available [17]. 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 1 hour.

Benchmark 2: The BIN suite (1,023 programs). We obtained all the binaries located under the `/bin`, `/usr/bin`, and `/sbin` directories from a default Debian Squeeze installation[4]. 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 1,023 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 minutes.

[3]All generated test cases were executed natively to compute code coverage results. To avoid destructive side-effects we removed 3 coreutils (`rm`, `rmdir` and `kill`) from the original KLEE suite.

[4]What better source of benchmark programs than the ones you use everyday?

Table 2: Veritesting finds 2× more bugs.

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

Table 3: Veritesting improves node coverage.

	Veritesting	DSE	Difference
coreutils	75.27%	63.62%	**+11.65%**
BIN	40.02%	34.71%	**+5.31%**

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 minutes per experiment.

6.1 Bug Finding

Table 2 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 [13, 16, 18, 35, 39]. 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 several popular projects, and we have confirmed that the bug affects other programs, e.g. `find`, `patch`, `tar`.

As a point of comparison, we ran Kuznetsov's DSM implementation [35], which missed the bugs. We also compared MergePoint with veritesting to S2E [19], a state-of-the-art binary-only symbolic execution system. S2E also missed the bugs. KLEE [16] argued that coreutils is one of the most well-tested suite of open-source applications. Since then, coreutils has become the de facto standard for evaluating bug-finding systems based on symbolic execution. Given the extensive subsequent testing of coreutils, finding two new crashes is evidence that veritesting extends the reach of symbolic execution.

6.2 Node Coverage

We evaluated MergePoint both with and without Veritesting on node coverage. Table 3 shows our overall results. Veritesting improves node coverage on average in all cases. MergePoint also achieved 27% more code coverage on average than S2E. Note that any positive increase in coverage is important. In particular, Kuznetsov *et al.* showed both dynamic state merging and static symbolic execution *reduced* node coverage when compared to vanilla DSE [35, Figure 8].

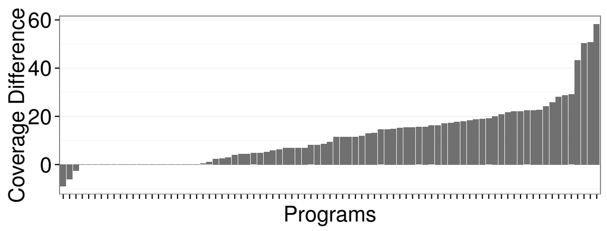


Figure 5: Code coverage difference on coreutils before and after veritesting.

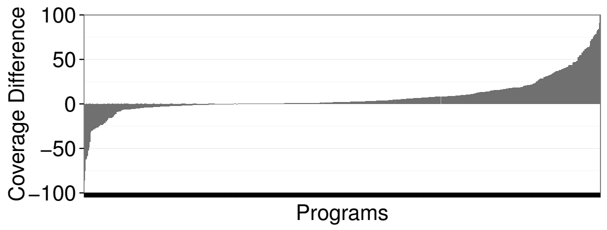


Figure 6: Code coverage difference on BIN before and after veritesting, where it made a difference.

Figures 5 and Figure 6 break down the improvement per program. For coreutils, enabling veritesting decreased coverage in only 3 programs (`md5sum`, `printf`, and `pr`). Manual investigation of these programs showed that veritesting generated much harder formulas, and spent more than 90% of its time in the SMT solver, resulting in timeouts. In Figure 6 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 7 shows the average coverage over time achieved by MergePoint with and without veritesting for the BIN suite. After 30 minutes, MergePoint without veritesting reached 34.45% code coverage. Veritesting achieved the same coverage in less than half the original time (12min 48s). Veritesting's coverage improvement becomes more substantial as analysis time goes on. Veritesting achieved higher coverage velocity, i.e., 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 veritesting becomes for high code coverage.

The above tests demonstrates the improvements of veritesting for MergePoint. We also ran both S2E and MergePoint (with veritesting) on coreutils using the same configuration for one hour on each utility in coreutils, excluding 11 programs where S2E emits assertion errors. Figure 8 compares the increase in coverage obtained by MergePoint with veritesting 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).

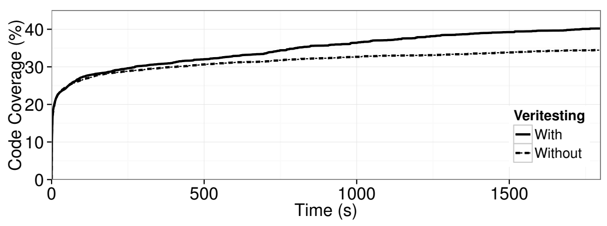


Figure 7: Coverage over time (BIN suite).

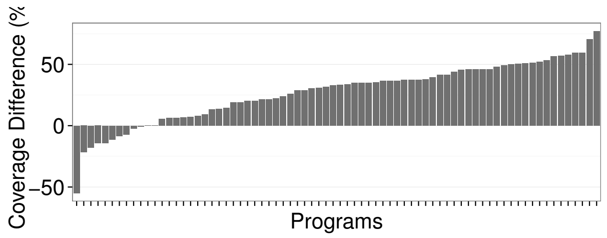


Figure 8: Code coverage difference on coreutils obtained by MergePoint vs. S2E

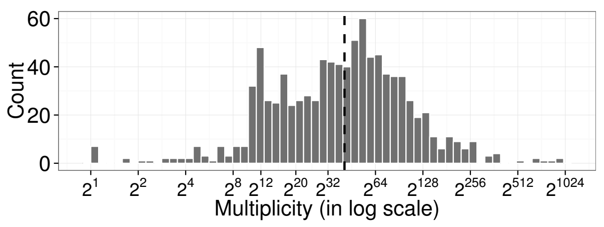


Figure 9: Multiplicity distribution (BIN suite).

6.3 Path Coverage

We evaluated the path coverage of MergePoint both with and without veritesting using three different metrics: time to complete exploration, multiplicity, and fork rate.

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 veritesting also completed, and b) if so, how long it took relative to DSE. MergePoint without veritesting was able to exhaust all paths for 46 programs. MergePoint with veritesting completes all paths 73% faster than without veritesting. This result shows that veritesting is faster when reaching the same end goal.

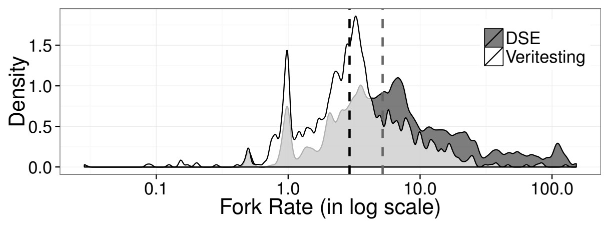


Figure 10: Fork rate distribution before and after veritesting with their respective medians (the vertical lines) for BIN.

Multiplicity. Multiplicity was proposed by Kuznetsov *et al.* [35] 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 9 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 an abnormal spike for programs with multiplicity of 4,096 (2^{12}). Further investigation showed that 72% of those programs came from the `netpbm` family of programs. Veritesting was unable to achieve very high multiplicity, due to the presence of unresolved calls in the recovered CFG. Improving the CFG recovery phase should further improve performance. Note that even with a multiplicity of 4,096, veritesting still improves coverage by 13.46% on the `netpbm` utilities. 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.

Fork rate. Another metric is the *fork rate* of an executor, which gives an indication of the size of the outstanding state space. If we represent the state space as a tree, where each node is a path, and its children are the paths that it forks, then the fork rate is the fanout factor of each node. Lower fork rate is better because it indicates a potentially exponentially-smaller state space. For instance, a tree of height n with a fanout factor of 5 has approximately 5^n nodes, while a tree with a fanout factor of 10 will have roughly 10^n nodes. Thus, a tree with a fanout factor of 5 is 2^n times smaller than a tree with a fanout factor of 10.

Figure 10 shows the fork rate distribution with and without veritesting of BIN. The graph shows that veritesting decreases the average fork rate by 65% (the median by 44%) from 13 to 4.6 (lower is better). In other words, without veritesting we used to have 13 new paths (forks) to explore for every analyzed path; with veritesting we have only 4.6. Thus, for the BIN programs, veritesting reduces the state space by a factor of $\left(\frac{13}{4.6}\right)^n \approx 3^n$, where n is the depth of the state space. This is an exponential reduction of the space, allowing symbolic execution to consider exponentially more paths during each analysis.

6.4 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 processing, the experiments took 18 CPU-months.

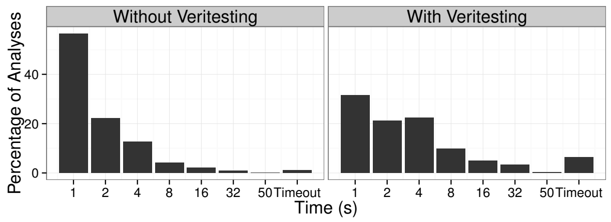
Our overall results are shown in Table 4. Veritesting found 11,687 distinct bugs (by stack hash) that crash programs.

Table 4: 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	235,623,757s
Symb exec time	125,412,247s
SAT time	40,411,781s
Model gen time	30,665,881s
# test cases	199,685,594
# crashes	2,365,154
# unique bugs	11,687
# fixed bugs	162
Confirmed control flow hijack	152

Component	DSE	Veritesting
Instrumentation	40.01%	16.95%
SMT Solver	19.23%	63.16%
Symbolic Execution	39.76%	19.89%

(a)



(b)

Figure 11: MergePoint performance before and after veritesting for BIN. The above figures show: (a) Performance breakdown for each component; (b) Analysis time distribution.

The bugs appear in 4,379 of the 33,248 programs. Veritesting also finds bugs that are potential security threats. 224 crashes have a corrupt stack, i.e. a saved instruction pointer has been overwritten by user input. Those crashes are most likely exploitable, and we have already confirmed exploitability of 152 programs. 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 1,043 crashes in total [40]. Not a single report was marked as unreproducible on the Debian bug tracking system. 162 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.

6.5 Discussion

Our experiments so far show that veritesting can effectively reduce the fork rate, 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 11b) shows that the vast majority (56%) of paths explored take less than 1 second for standard symbolic execution. With veritesting, the fast paths are fewer (21%), and we get more timeouts (6.4% vs. 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. In order to better understand cases where merging is worse than per-path DSE, we performed a targeted experiment to identify the characteristics that make a program amenable to veritesting. To do so, we gathered the programs where veritesting impacts the coverage by more than 5%, positively or negatively, and generated a performance breakdown similar to Figure 11a:

Improvement	SMT	Timeout	Coverage
Largest	30%	1%	82.14%
Smallest	73%	6%	34.31%

We observe again that veritesting performs best when the solver is not dominating the cost of symbolic execution or causing timeouts (recall from §6.2). One possible future direction is to employ a heuristic to decide transition points, e.g., combining veritesting with QCE [35].

We also measured the effect of our hash-consed based language on veritesting. We performed 4 experiments on our BIN suite (5 min/prog) and measured performance across two axes: veritesting vs. DSE and hash-consing vs. no hash-consing. The table below summarizes our results in terms of average coverage and generated test cases:

Technique	No hash-consing	Hash-consing	Difference
Veritesting	24.24%	29.64%	**+5.40%**
DSE	26.82%	28.98%	**+2.16%**

We see that hash-consing affects performance dramatically: disabling it would make veritesting worse than DSE (within the 5 minute interval). In fact, our initial veritesting implementation did not include hash-consing, and did not improve coverage. The cost of duplicating or naïvely recursing over expressions is prohibitive for expressions encompassing multiple paths (note that DSE is not as strongly affected).

7. RELATED WORK

Symbolic execution was discovered in 1975 [11, 31, 33], with the volume of academic research and commercial systems exploding in the last decade. An online bibliography [1] currently lists over 150 papers on symbolic execution techniques and applications. Notable symbolic executors include SAGE and KLEE. SAGE [10] is responsible for finding one third of all bugs discovered by file fuzzing during the development of Windows 7 [10]. KLEE [16] 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 [14, 43, 45].

Merging execution paths is not new. Koelbl *et al.* [34] pioneered path merging in SSE. Concurrently and independently, Xie *et al.* [48] developed Saturn, a verification tool capable of encoding of multiple paths before converting the problem to SAT. Hansen *et al.* [30] follow an approach similar to Koelbl *et al.* at the binary level. Babic [5] improved their static algorithm to produce smaller and faster to solve formulas by leveraging Gated Single Assignment (GSA) [47] and maximally-shared graphs (similar to hash-consing [29]). 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* [3], a technique for converting code with branches into predicated straightline statements. The technique is also known as ϕ-folding [36], a compiler optimization technique that collapses simple diamond-shaped structures in the CFG. Collingbourne *et al.* [20] used ϕ-folding to verify semantic equivalence of SIMD instructions.

Boonstoppel *et al.* proposed RWSet [9], a state pruning technique identifying redundant states based on similarity of their live variables. If live variables of a state are equivalent to a previously explored path, RWSet will stop exploring the states, as it will not offer additional coverage.

Godefroid *et al.* [25] introduced function summaries to test code compositionally. The main idea is to record the output of an analyzed function, and to reuse it if it is called again with similar arguments. The work was later expended to generate such summaries on demand [4]. Compositional analysis and generalizations over loops are orthogonal and complementary to veritesting.

Shortly after releasing our bug reports, Romano [44] reported bugs on at least 30,000 distinct binaries. He reports 5 minutes of symbolic execution per binary. Romano's result is impressive. Unfortunately, Romano has not provided a report detailing his methods, benchmarks, or methods, leaving comparison to his work impossible.

8. CONCLUSION

In this paper we proposed MergePoint and veritesting, a new technique to enhance symbolic execution with verification-based algorithms. We evaluated MergePoint on 1,023 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 162 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 is supported in part by grants from DARPA and the NSF, as well as the Prabhu and Poonam Goel Fellowship.

9. REFERENCES

[1] Online Bibliography for Symbolic Execution. http://sites.google.com/site/symexbib.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[3] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of Control Dependence to Data Dependence. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 177–189, New York, NY, USA, 1983. ACM Press.

[4] S. Anand, P. Godefroid, and N. Tillmann. Demand-Driven Compositional Symbolic Execution. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 367–381, Berlin, Heidelberg, 2008. Springer-Verlag.

[5] D. Babic. *Exploiting structure for scalable software verification*. PhD thesis, University of British Columbia, Vancouver, Canada, 2008.

[6] D. Babic and A. J. Hu. Calysto: Scalable and Precise Extended Static Checking. In *Proceedings of the 30th International Conference on Software Engineering*, pages 211–220, New York, NY, USA, 2008. ACM.

[7] S. Bardin, P. Herrmann, J. Leroux, O. Ly, R. Tabary, and A. Vincent. The BINCOA Framework for Binary Code Analysis. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, pages 165–170, Berlin, Heidelberg, 2011. Springer-Verlag.

[8] D. Beyer, T. A. Henzinger, and G. Theoduloz. Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis. In *Proceedings of the 19th International Conference on Computer Aided Verification*, pages 504–518, Berlin, Heidelberg, 2007. Springer-Verlag.

[9] P. Boonstoppel, C. Cadar, and D. Engler. RWset: Attacking Path Explosion in Constraint-Based Test Generation. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 351–366, Berlin, Heidelberg, 2008. Springer-Verlag.

[10] E. Bounimova, P. Godefroid, and D. Molnar. Billions and Billions of Constraints: Whitebox Fuzz Testing in Production. In *Proceedings of the 35th IEEE International Conference on Software Engineering*, pages 122–131, Piscataway, NJ, USA, 2013. IEEE Press.

[11] R. S. Boyer, B. Elspas, and K. N. Levitt. SELECT—a formal system for testing and debugging programs by symbolic execution. *ACM SIGPLAN Notices*, 10(6):234–245, 1975.

[12] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz. BAP: A Binary Analysis Platform. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, pages 463–469. Springer-Verlag, 2011.

[13] S. Bucur, V. Ureche, C. Zamfir, and G. Candea. Parallel symbolic execution for automated real-world software testing. In *Proceedings of the 6th ACM SIGOPS European Conference on Computer Systems*, pages 183–198. ACM Press, 2011.

[14] C. Cadar and K. Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90, 2013.

[15] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE : Automatically Generating Inputs of Death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, New York, NY, USA, 2006. ACM.

[16] C. Cadar, D. Dunbar, and D. Engler. 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*, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[17] C. Cadar, D. Dunbar, and D. R. Engler. KLEE Coreutils Experiment. http://klee.github.io/klee/CoreutilsExperiments.html, 2008.

[18] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing Mayhem on Binary Code. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, pages 380–394, Washington, DC, USA, 2012. IEEE Computer Society.

[19] V. Chipounov, V. Kuznetsov, and G. Candea. 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*, pages 265–278, New York, NY, USA, 2011. ACM.

[20] P. Collingbourne, C. Cadar, and P. H. Kelly. Symbolic crosschecking of floating-point and SIMD code. *Proceedings of the 6th ACM SIGOPS European conference on Computer Systems*, pages 315–328, 2011.

[21] L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[22] I. Dillig, T. Dillig, and A. Aiken. Sound, Complete and Scalable Path-Sensitive Analysis. In *Proceedings of the 29th ACM Conference on Programming Language Design and Implementation*, pages 270–280, New York, NY, USA, 2008. ACM.

[23] J. Filliâtre and S. Conchon. Type-safe modular hash-consing. In *Proceedings of the Workshop on ML*, pages 12–19, New York, NY, USA, 2006. ACM.

[24] C. Flanagan and J. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 193–205, New York, NY, USA, 2001. ACM.

[25] P. Godefroid. Compositional Dynamic Test Generation. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 47–54, New York, NY, USA, 2007. ACM.

[26] P. Godefroid, N. Klarlund, and K. Sen. DART : Directed Automated Random Testing. In *Proceedings of the 26th ACM Conference on Programming Language Design and Implementation*, New York, NY, USA, 2005. ACM.

[27] P. Godefroid, M. Y. Levin, and D. Molnar. Automated Whitebox Fuzz Testing. In *Proceedings of the 15th Network and Distributed System Security Symposium*. The Internet Society, 2008.

[28] P. Godefroid, M. Y. Levin, and D. Molnar. SAGE: Whitebox Fuzzing for Security Testing. *Communications of the ACM*, 55(3):40–44, 2012.

[29] E. Goto. Monocopy and Associative Algorithms in Extended Lisp. Technical Report TR-74-03, University of Tokyo, 1974.

[30] T. Hansen, P. Schachte, and H. Søndergaard. State Joining and Splitting for the Symbolic Execution of Binaries. *Runtime Verification*, pages 76–92, 2009.

[31] W. Howden. Methodology for the Generation of Program Test Data. *IEEE Transactions on Computers*, C-24(5):554–560, 1975.

[32] J. Kinder and H. Veith. Jakstab: A Static Analysis Platform for Binaries. In *Proceedings of the 20th International Conference on Computer Aided Verification*, pages 423–427, Berlin, Heidelberg, 2008. Springer-Verlag.

[33] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[34] A. Koelbl and C. Pixley. Constructing Efficient Formal Models from High-Level Descriptions Using Symbolic Simulation. *International Journal of Parallel Programming*, 33(6):645–666, Dec. 2005.

[35] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea. Efficient state merging in symbolic execution. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 193–204, New York, NY, USA, 2012. ACM.

[36] C. Lattner and V. Adve. 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*, pages 75–86, Washington, DC, USA, 2004. IEEE Computer Society.

[37] K. R. M. Leino. Efficient weakest preconditions. *Information Processing Letters*, 93(6):281–288, 2005.

[38] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tols with Dynamic Instrumentation. In *Proceedings of the 26th ACM Conference on Programming Language Design and Implementation*, pages 190–200, New York, NY, USA, 2005. ACM.

[39] P. D. Marinescu and C. Cadar. Make test-zesti: A symbolic execution solution for improving regression testing. In *Proceedings of the 34th International Conference on Software Engineering*, pages 716–726, Piscataway, NJ, USA, 2012. IEEE Press.

[40] Mayhem. 1.2K Crashes in Debian, 2013. URL http://lists.debian.org/debian-devel/2013/06/msg00720.html.

[41] Mayhem. Open Source Statistics & Analysis, 2013. URL http://www.forallsecure.com/summaries.

[42] D. Molnar, X. Li, and D. Wagner. Dynamic test generation to find integer bugs in x86 binary linux programs. In *Proceedings of the USENIX Security Symposium*, pages 67–82, 2009.

[43] C. S. Păsăreanu and W. Visser. A survey of new trends in symbolic execution for software testing and analysis. *International Journal on Software Tools for Technology Transfer*, 11(4):339–353, Aug. 2009.

[44] A. J. Romano. Linux Bug Release, July 2013. URL http://www.bugsdujour.com/release/.

[45] E. J. Schwartz, T. Avgerinos, and D. Brumley. 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*, pages 317–331, Washington, DC, USA, 2010. IEEE Computer Society.

[46] K. Sen, D. Marinov, and G. Agha. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of the 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 263–272, New York, NY, USA, 2005. ACM.

[47] P. Tu and D. Padua. Efficient building and placing of gating functions. In *Proceedings of the 16th ACM Conference on Programming Language Design and Implementation*, pages 47–55, New York, NY, USA, 1995. ACM.

[48] Y. Xie and A. Aiken. Scalable error detection using boolean satisfiability. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 351–363, New York, NY, USA, 2005. ACM.

[49] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, 1997.