Verifying Systems Rules Using Rule-Directed Symbolic Execution

Heming Cui * Gang Hu * Jingyue Wu Junfeng Yang
{heming,ganghu,jingyue,junfeng}@cs.columbia.edu
Columbia University

Abstract

Systems code must obey many rules, such as "opened files must be closed." One approach to verifying rules is static analysis, but this technique cannot infer precise runtime effects of code, often emitting many false positives. An alternative is symbolic execution, a technique that verifies program paths over all inputs up to a bounded size. However, when applied to verify rules, existing symbolic execution systems often blindly explore many redundant program paths while missing relevant ones that may contain bugs.

Our key insight is that only a small portion of paths are relevant to rules, and the rest (majority) of paths are irrelevant and do not need to be verified. Based on this insight, we create WOODPECKER, a new symbolic execution system for effectively checking rules on systems programs. It provides a set of builtin checkers for common rules, and an interface for users to easily check new rules. It directs symbolic execution toward the program paths relevant to a checked rule, and soundly prunes redundant paths, exponentially speeding up symbolic execution. It is designed to be heuristic-agnostic, enabling users to leverage existing powerful search heuristics.

Evaluation on 136 systems programs totaling 545K lines of code, including some of the most widely used programs, shows that, with a time limit of typically just one hour for each verification run, WOODPECKER effectively verifies 28.7% of the program and rule combinations over bounded input, whereas an existing symbolic execution system KLEE verifies only 8.5%. For the remaining combinations, WOODPECKER verifies 4.6 times as many relevant paths as KLEE. With a longer time limit, WOODPECKER verifies much more paths than KLEE, *e.g.*, 17 times as many with a four-hour limit. WOODPECKER detects 113 rule violations, including 10 serious data loss errors with 2 most serious ones already confirmed by the corresponding developers.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; D.2.5 [*Software Engineering*]: Testing and Debugging; D.4.5 [*Operating Systems*]: Reliability

General Terms Algorithms, Experimentation, Reliability, Verification

* These authors contributed equally to this work.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ASPLOS'13, March 16–20, 2013, Houston, Texas, USA.
Copyright © 2013 ACM 978-1-4503-1870-9/13/03...$15.00

Keywords Symbolic execution, Path Slicing, Verification, Error Detection, Systems Rules

1. Introduction

Systems code must obey a variety of rules. For instance, assertions must succeed, allocated memory must be freed, opened files must be closed, file read or write must be done on opened files, and atomic `rename` must be used correctly [55, 63]. Violating these rules may easily lead to critical failures such as program crashes, resource leaks, data losses, and security breaches.

One approach to verifying rules is static analysis [25, 26, 61]. This technique has high coverage because it analyzes all code a compiler can see. However, without running code, it has difficulties inferring precise runtime effects of code, such as whether two pointers point to the same memory location or two branch conditions are equivalent. For soundness, it has to conservatively assume the worst, often emitting numerous false positives which require costly manual inspection. Although previous work [26] reduced false positives with heuristics, it aggressively traded off soundness. It may thus miss many errors and cannot be applied to verification.

An appealing alternative to verifying rules is *symbolic execution* [11, 12, 33, 42], a powerful technique that systematically explores program paths for errors. Instead of running a program on concrete input consisting of zeros and ones, it runs the program on *symbolic* input initially allowed to have any value. When the execution branches based on symbolic input, this technique explores both branches by conceptually forking two executions, each maintaining constraints on symbolic input required for the program to proceed down the given path. By systematically exploring paths, this technique achieves higher statement coverage than manual or random testing [12]. By actually running code, it sees the precise runtime effects of code and eliminates false positives (barring limitations of constraint solving).

Symbolic execution has been applied to detect serious errors in systems code [12, 13, 64] and generate high-coverage tests [12]. Moreover, it has been applied to *verify* program paths and code [12, 18, 39, 52]. At each dangerous operation such as an assertion on symbolic input, it checks if any input value may fail the operation. If it finds no such inputs for a path, it effectively verifies the path over the symbolic input. If it verifies all paths, it effectively verifies the entire program over the symbolic input. This verification approach is a form of *bounded verification* [17]. Researchers have argued and experimentally shown that checking all inputs within a small scope can effectively detect a high portion of bugs and achieve high statement and branch coverage [5, 12, 19, 38].

Although it is tempting to verify systems rules using symbolic execution, this technique faces another difficult challenge of *path explosion*: it can rarely explore all paths of a typical program simply because of the sheer number of paths.

Our key insight is that although the number of program paths is enormous, only a small portion of the paths are relevant to a checked rule, and the rest (majority) of the paths are irrelevant and do not need to be checked. Specifically, a typical rule involves only a small number of instructions, or *events*, during execution. For instance, the rule "opened files must be closed" involves only the events that open and close files, such as `fopen()` and `fclose()` calls. From a verification perspective, paths leading to the same event sequence are equivalent, and only one of them need be checked. (See §2 for an example.) Unfortunately, existing symbolic execution systems are not designed to check rules; they often blindly explore many redundant paths while missing relevant ones that may contain bugs.

Based on our key insight, we create WOODPECKER,[1] a symbolic execution system for effectively checking rules on systems programs. Similar to most previous symbolic execution systems, WOODPECKER emits no false positives, and can record and replay the program path leading to each error it detects. In addition, WOODPECKER allows users to easily check rules by writing custom checkers, and provides a set of builtin checkers to help users bootstrap the checking process. Given a rule, it performs *rule-directed symbolic execution*: instead of blindly exploring all paths, it first statically "peeks" into the paths and then symbolically executes only the paths relevant to the rule, greatly speeding up symbolic execution for both verification and bug detection.

To effectively direct symbolic execution toward a rule, WOODPECKER faces three key algorithmic challenges. First, given a rule, how can WOODPECKER determine what paths are redundant? Second, how can WOODPECKER work with different rules? It would be impractical if each rule requires a different symbolic execution algorithm. Third, how can WOODPECKER integrate with the clever search heuristics in existing symbolic execution systems? These heuristics absorb much dedicated research efforts and can steer checking toward interesting paths (*e.g.*, those more likely to have bugs) when verifying all is not feasible. They have been shown to effectively increase statement coverage and detect errors [12, 13].

WOODPECKER solves these challenges using two ideas: a simple but expressive checker interface and a sound, checker- and heuristic-agnostic search algorithm. A checker implementing the interface provides methods to inform WOODPECKER (1) which executed instructions are events and (2) which static instructions may be events regarding a rule. These methods abstract away the checker details, enabling WOODPECKER's search algorithm to be checker-agnostic.

Once WOODPECKER finishes exploring a path, it uses the checker-provided methods to determine which branches off the path should be pruned. Specifically, if an off-the-path branch cannot (1) affect any event executed in the path or (2) reach any new event not in the path, then it cannot lead to a different event sequence, so WOODPECKER prunes this branch without missing errors. This pruning is heuristic-agnostic because it is done only at the end of a path and does not interfere with the search heuristics otherwise. By pruning irrelevant branches, WOODPECKER can speed up symbolic execution *exponentially* because each pruned branch in principle halves the number of paths to explore.

We have implemented WOODPECKER on top of the LLVM compiler framework [1]. To leverage the powerful search heuristics in existing symbolic execution systems, we carefully integrate WOODPECKER with KLEE [12]. To prune branches, we leverage a modified version of *path slicing* [40] that computes a *slice* of a path containing instructions key for reaching an event sequence [23]. We also create five checkers, including the *assertion*, *memory leak*, *open-close*, *file access*, and *data loss* checkers, to check the five rules mentioned in the first paragraph of this paper.

We evaluated WOODPECKER on 136 systems programs, totaling 545K lines, including some of the most widely used programs such as file, shell, and text utilities in GNU `coreutils`, user and group management utilities in `shadow`, `tar`, `sed`, `CVS`, and `git`. Our results show that

1. With a time limit of typically just one hour for each verification run, WOODPECKER effectively verifies 111 out of 387 (28.7%) program and rule combinations over bounded input. These verification processes often finish within tens of minutes. In contrast, KLEE verifies only 33 out of 387 (8.5%) combinations.
2. For the remaining combinations, WOODPECKER verifies 4.6 times as many relevant paths as KLEE. This speedup is unsurprising as 86.8% of the paths KLEE explores are redundant.
3. With a longer time limit, WOODPECKER verifies even more relevant paths than KLEE. For instance, WOODPECKER verifies 17 times as many paths as KLEE for `coreutils` within 4 hours.
4. WOODPECKER detects total 113 rule violations. It detects 10 serious data loss errors. Nine of these errors may corrupt source code repositories and the other one may cause new user accounts to have wrong (security) settings. The 2 most serious ones have been confirmed by the corresponding developers.

This paper makes three main contributions. Our first contribution is the rule-directed symbolic execution approach which checks systems rules and uses them to direct symbolic execution. Our second contribution is the WOODPECKER system with a simple yet expressive checker interface, a set of builtin checkers, and a sound, checker- and heuristic-agnostic search algorithm that leverages path slicing to prune redundant paths and speed up symbolic execution. Our last contribution is our experimental evaluation that verifies rules on real-world programs and detects serious errors.

The remainder of this paper is organized as follows. We first show an example (§2) and overview (§3) of WOODPECKER. We describe its checker interface (§4), search algorithm (§5), and builtin checkers (§6). We present its implementation (§7) and evaluation (§8). We finally discuss related work (§9) and conclude (§10).

2. An Example

To illustrate how WOODPECKER works and its key benefits, we show an example in Figure 1. Although simple, this program implements the core functionality of `cat` in GNU `coreutils`. It iterates through the command line arguments, opens each argument as a file, and prints the file to the standard output. If an argument is "`-`" or no argument is present, it reads from the standard input. It converts non-printable characters before printing them (lines 15–17), emulating what "`cat -v`" does.

2.1 Difficulties in Existing Approaches

Suppose we want to automatically verify the open-close rule on this program, *i.e.*, `fopen()` at line 9 and `fclose()` at line 22 always pair up. If this rule is violated, each iteration of the outer loop (lines 5–23) leaks a file pointer, so this program may exhaust all file descriptors on a long list of input files. Fortunately, this rule holds because the true branch of line 21 is taken if and only if the true branch of line 8 is taken. However, automatically verifying this fact using existing approaches can be quite challenging.

Static analysis is not good at computing precise constraints on values, so it may have difficulties computing that the conditions at lines 8 and 21 are, despite their syntactic differences, semantically equivalent. It is not good at precisely tracking where pointers point to either, so it may even have difficulties determining that variable `infile` must point to the same element of `argv` at lines 8 and 21 in the same iteration of the outer loop. Alias analysis is the standard

[1] We name our system after the woodpecker bird because this bird finds bugs similarly: it uses its acute hearing to locate wood-boring bugs before hammering into the wood [8].

```
1 : int main(int argc, char **argv) {
2 :   FILE *input_desc;
3 :   int argind = 1;
4 :   const char *infile = "-";
5 :   do { // iterate over input files and print one by one
6 :     if(argind < argc)
7 :       infile = argv[argind];
8 :     if(strcmp(infile, "-")) // input is a file
9 :       input_desc = fopen(infile, "r");
10:     else // input is stdin
11:       input_desc = stdin;
12:     if(!input_desc) continue;
13:     int c;
14:     while((c = fgetc(input_desc)) != EOF) {
15:       if(c < 32 && c != '\n') { // non-printable char
16:         putchar('^');
17:         putchar(c + 64);
18:       } else // printable char
19:         putchar(c);
20:     }
21:     if(infile[0] != '-' || infile[1] != 0)
22:       fclose(input_desc); // input is a file
23:   } while (++argind < argc);
24:   return 0;
25: }
```

Figure 1: *A simple program based on* `cat` *from* `coreutils`.

method to compute whether two pointers point to the same location, but it often collapses results from multiple loop iterations into one, especially when the loop bound cannot be statically determined, such as this outer loop. It may thus imprecisely compute that `infile` may point to different locations at lines 8 and 21. Due to these difficulties, static analysis may conservatively consider lines 8, 9, 21, and 23 a feasible path fragment and lines 8, 11, 21, and 22 another one, emitting false positives that require manual inspection.

Symbolic execution avoids false positives by exploring only feasible paths, but it faces another difficult challenge of path explosion. Consider the program in Figure 1. Although simple, it may still have too many paths for symbolic execution to completely explore, even if we bound the input to be small. In particular, the symbolic branch instruction at line 15 doubles the number of paths for each input byte.

We ran KLEE on this program with a small input bound: three command line arguments, each with two bytes, and two files, each with eight bytes. Although KLEE explored 698,116 paths[2] in an hour, it unsurprisingly did not explore all paths. Worse, of all the paths it explored, 99.84% are redundant regarding the rule. For instance, all paths forked at lines 14 and 15 are redundant to each other because they affect neither `fopen` nor `fclose` operations, so only one of them need be checked.

2.2 Bounded Verification with WOODPECKER

Since WOODPECKER already comes with an open-close checker, users simply specify this checker to check the rule. To check new rules, users can write their own checkers. To prepare the program in Figure 1, users compile it to *bitcode*, the well-defined, easy-to-analyze RISC-like LLVM assembly code [1]. Users then specify the symbolic input and start checking. For instance, they can mark command line arguments and input files symbolic, and specify their maximum sizes.

[2] Since KLEE works on LLVM assembly programs with `Libc` linked in, it sees much more paths than static analysis.

```
3:        argind = 1;
4:        infile = "-";
6:true    argind < argc
7:        infile = argv[argind];
8:true    strcmp(infile, "-")
9:        input_desc = fopen(infile);
12:       if(input_desk) continue;
14:true    (c=fgetc(input_desc)) != EOF
15:false   c<32 && c!= '\n'
19:       putchar(c);
14:false   (c=fgetc(input_desc)) != EOF
21:true    infile[0]!='-' || infile[1] != 0
22:       fclose(input_desc);
23:false   ++argind < argc
24:       return 0;
```

Figure 2: *A path explored by* WOODPECKER *and its pruning results.* Each executed instruction is tagged with its static line number. Branch instructions are also tagged with their outcome (true or false). Events (green) concerning the rule are the `fopen()` and `fclose()` calls tagged with 9 and 22. Crossed-out (gray) instructions are elided from the slice, so their off-the-path branches are not explored by WOODPECKER. Only the off-the-path branches of lines 6, 8, and 23 (italic, blue), are explored by WOODPECKER.

To explore paths, WOODPECKER conceptually forks executions upon a symbolic branch instruction with both branches allowed by the current constraints. For instance, it forks executions when it first executes lines 6, 8, 14, 15, and 23. It does not fork at line 21 because the constraints collected from line 8 makes only one branch of line 21 feasible. By exploring only feasible paths, WOODPECKER avoids emitting false positives.

Given all forked paths, WOODPECKER can prioritize how they are explored using various heuristics. To leverage existing heuristics, we have carefully integrated it with KLEE. We will describe this integration and WOODPECKER's search algorithm in §5; in the remainder of this section we illustrate how WOODPECKER prunes redundant paths using our example.[3]

WOODPECKER prunes redundant paths when it finishes exploring a path. Suppose it explores a path as shown in Figure 2. (The paths explored depend on the heuristic used.) As discussed in the previous subsection, without directing symbolic execution with the rule, KLEE would have to explore the off-the-path branches at lines 14 and 15. Fortunately, WOODPECKER prunes these branches because it determines that they have no effect on the event sequence (lines 9 and 22) in this path.

To do so, WOODPECKER leverages a previous *path slicing* algorithm that computes a *slice* that captures instructions key to reach the exact event sequence in a path [23]. The resultant slice contains the instructions not crossed out in Figure 2. It includes the events (9 and 22) and all instructions that the events transitively control- or data-depend upon. For instance, it includes 8:true because if the false branch is taken, the execution will not reach 9; it includes 7, 6:true, 4, and 3 because they set the `infile` used by 8:true. It includes 23:false because if the true branch is taken, the execution will reach new instances of `fopen()` and `fclose()` calls. It excludes all instructions in the inner loop (lines 14–20) and the return instruction (line 24) because they do not affect the event sequence. WOODPECKER then explores only the off-the-path branches included in the slice, and prunes the other ones, *i.e.*, those at lines 14 and 15.

[3] For clarity, this paper presents WOODPECKER's algorithm at the source level, instead of the bitcode level.

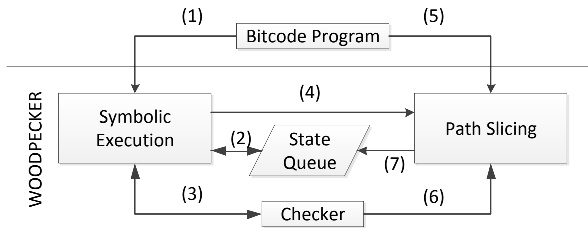

Figure 3: WOODPECKER *architecture.*

We ran WOODPECKER on this example with the same input bound as KLEE. WOODPECKER completely explored all 1,802 paths that may lead to different event sequences, verifying that our example has no open-close errors over the bounded input. This verification process took only 212.7 seconds. We observed similar benefits of WOODPECKER on the real programs evaluated (§8).

3. WOODPECKER Overview

Figure 3 shows WOODPECKER's architecture. We explain this architecture as well as how the components interact; the numbered labels in the figure correspond to the numbers in the list below

(1) WOODPECKER checks programs in bitcode, the LLVM assembly language [1]. Given a bitcode program, WOODPECKER invokes KLEE to execute (more precisely, interpret) bitcode instructions.
(2) If executing one instruction (*e.g.*, a symbolic branch instruction) on a program state leads to new states, WOODPECKER adds them to the *state queue* for future exploration.
(3) It passes the executed instruction and its operands to the checker for errors.
(4) If the instruction executed is at the end of a path, WOODPECKER applies path slicing to compute redundant branches that do not affect the event sequence in the path.
(5) As part of the computation to determine if a branch is redundant, WOODPECKER statically visits the off-the-path instructions a branch may reach.
(6) For each instruction visited, it queries the checker to determine whether the instruction may be an event.
(7) Given the redundant branches, it searches a *branch tree* (§5) to find the corresponding states forked from the branches, and removes the states from the queue.

WOODPECKER then removes a state from the state queue and repeats these steps until it explores all paths or reaches a given time limit. This process essentially implements WOODPECKER's search algorithm (§5) with steps (3) and (6) invoking the checker through the checker interface (§4).

Assumptions. WOODPECKER often effectively prunes paths for rules involving a small number of events. Most systems rules, including all the rules we check, have this property. If a rule involves many instructions such as all load instructions, WOODPECKER may not be as effective.

WOODPECKER's effectiveness may also depend on specific search heuristics and programs. For instance, if all program paths are of the same length, depth-first search may work better because it reaches the end of a path and triggers pruning sooner. If some program paths are very short, breadth-first search may work better because it reaches the end of a short path sooner. We explicitly designed WOODPECKER to give users flexibility to choose heuristics.

Our current design of WOODPECKER checks one rule at a time because our anecdotal experience with statically checking system rules is that developers often prefer to focus on one type of bug before context-switching to a different type. WOODPECKER can be extended to check multiple rules simultaneously, to further amortize the cost of symbolic execution. However, WOODPECKER's path slicing needs to be modified so that events from different rules are analyzed separately. We leave this for future work.

At the implementation level, WOODPECKER requires bitcode or source (so we can compile it to bitcode) programs. WOODPECKER aborts a path upon inline x86 assembly or a call to an external function it does not know. For soundness, developers have to lift x86 assembly to bitcode and provide summaries for external functions. (The external function problem is alleviated because KLEE links in a `Libc` implementation.) Moreover, the underlying components that WOODPECKER leverages make assumptions as well. For instance, KLEE requires constraint solving, which can be quite expensive and sometimes intractable. Path slicing requires alias analysis, and the one used in WOODPECKER makes several assumptions [6]; a "sounder" alias analysis [35] would remove these assumptions. Our slicing is conservative: it never marks a relevant branch as redundant, but it may mark redundant branches as relevant [23].

Although WOODPECKER is similarly limited by its underlying components such as KLEE, the general ideas in WOODPECKER can benefit other symbolic execution systems. For instance, it can be combined with systems such as S2E [13] and its search techniques [9, 43] to soundly reduce the number of paths to explore.

```
class Checker {
  // Clones internal checker state.
  Checker *Clone();

  // Called by WOODPECKER after it executes instruction ki.
  // struct KInstruction contains instruction operands.
  // Returns false if an error is detected, and true otherwise.
  // s is program state after ki is executed.
  // evmask outputs whether the instruction is an event and,
  // if so, what operands matter.
  bool OnExecution(const ExecutionState *s,
                   const KInstruction *ki, Bitmask& evmask);

  // Called by WOODPECKER to check
  // whether a static instruction may be an event.
  bool MayBeEvent(const Instruction *i);
};
```

Figure 4: *The* WOODPECKER *checker interface.*

4. Checker Interface

Figure 4 shows WOODPECKER's checker interface. It consists of three key methods. The `Clone` method clones a checker's internal state. Whenever WOODPECKER clones a program state for later exploration, it also clones the checker state, so that the program and checker states are always consistent.

WOODPECKER calls `OnExecution` after executing each instruction. This method does three things. First, it updates the internal state of the checker. For instance, if the instruction is a call `fp = fopen()`, the open-close checker updates its internal state to map file pointer `fp` to the opened state. Second, this method detects whether the executed instruction causes an error. If so, it returns false, so WOODPECKER can invoke KLEE to generate a testcase that reproduces the error. Lastly, this method outputs variable `evmask`, which indicates whether the executed instruction is an event. This information helps WOODPECKER compute relevant branches (§5).

Variable `evmask` is a bitmask instead of a boolean flag because `OnExecution` also uses it to indicate which operands matter to the checked rule. If an operand does not matter, WOODPECKER does not include instructions affecting it in the slice. For instance,

Algorithm 1: WOODPECKER's search algorithm

```
  Input : program prog, initial state s_0, checker checker
  Global: state queue q, branch tree brtree
  RuleDirectedSE(prog, s_0, checker)
      q.add(⟨s_0, prog.entry⟩)
      while q not empty and time limit not reached do
          ⟨s, i⟩ ← q.remove()
          if i is not a symbolic branch statement or only one branch
          is feasible then
              ⟨s', i'⟩ ← run(s, i)
              HandleNewState(s, i, s', i', checker)
          else // fork states and add constraints
              s_true ← s + {i.cond = true}
              s_false ← s + {i.cond = false}
*             brtree.add(i, s_true, s_false)
              HandleNewState(s, i, s_true, i.true_br, checker)
              HandleNewState(s, i, s_false, i.false_br, checker)
  HandleNewState(s, i, s', i', checker) // Note: s' is updated
*     if not checker.OnExecution(s, i, evmask) then
*         return // error detected!
*     s'.path.push_back(i)
*     if evmask ≠ 0 then
*         s'.events.push_back(⟨i, evmask⟩)
      if s' is not end of path then q.add(⟨s', i'⟩)
*     else Prune(s', checker) // s' is end of path
  Prune(s, checker)
      slice ← Slice(s.path, s.events, checker)
      foreach symbolic branch i in s.path but not in slice do
          br ← branch of i not executed in s.path
          pruned ← brtree.find_states(br)
          q.remove(pruned)
      brtree.remove(s) // because s is end of path
  Slice(path, events, checker)
      slice ← empty sequence
      for i ∈ reverse(path) do
          if any e ∈ events transitively depends on i then
              slice.push_front(i)
          else if i is a branch instruction then
              for j ∈ instructions i reaches off path do
                  if checker.MayBeEvent(j) then
                      slice.push_front(i)
                      break
      return slice
```

consider the open-close checker. The arguments to `fopen()` do not matter, whereas the argument to `fclose()` does. Implementation-wise, the most significant bit of `evmask` indicates whether the instruction is an event, and the ith bit indicates whether the ith operand matters. (We show an example on how WOODPECKER uses `evmask` in §5.)

WOODPECKER calls `MayBeEvent` to check whether a static instruction, if executed, may be an event of the checked rule. WOODPECKER uses this information to compute which branches may lead to new events (§5).

5. Search Algorithm

This section describes WOODPECKER's checker- and heuristic-agnostic search algorithm (Algorithm 1) and how it integrates with existing search heuristics.

Functions `RuleDirectedSE` and `HandleNewState` together without the starred lines implement a typical search process in existing symbolic execution systems and model checkers. This search process maintains a queue of program state and instruction-to-execute pairs, initialized to contain the initial state and the program entry. It removes a state and instruction pair from the queue, executes the instruction on the state, adds the resultant states to the queue, and repeats until the state queue is empty or a time limit is reached. Different search heuristics can be implemented by varying the order in which states are added to or removed from the queue. For instance, depth-first search would remove the state last added, and breadth-first search would remove the state first added.

The starred lines and the additional functions `Prune` and `Slice` together give WOODPECKER the power of checking rules and pruning redundant paths without restricting search heuristics. This additional code does four things. First, in `HandleNewState`, it calls the checker's `OnExecution` to check for rule violations and set `evmask` (§4).

Second, it maintains a program path and an event sequence in the resultant state s', later used by `Prune` to prune redundant states. Specifically, it appends each executed instruction to the `path` field of s'. It appends the instruction to the `events` field of s' if `evmask` is nonzero, *i.e.*, the instruction is an event.

Third, whenever WOODPECKER reaches the end of a program path, it calls `Prune` to remove redundant states from the queue. `Prune` first calls `Slice` to compute from the current path a *slice* that, if preserved, guarantees the same event sequence. If a symbolic branch instruction is in the path but not the slice, taking either of its branches should lead to the same event sequence, so WOODPECKER prunes the states reached from the off-the-path branch.

Fourth, to find the states reached from a branch, WOODPECKER uses a *branch tree* to track forked branches and states on the queue. The leaf nodes of this tree are all states to explore, and the internal nodes are all branch instructions. At each fork, WOODPECKER replaces the node representing the current state with a node representing the branch instruction, and adds the two forked states as two child nodes. At the end of a path, WOODPECKER removes the node representing the current state from the tree, as well as its parents if they become childless. To find states given a branch *br*, function `find_states` simply returns the leaf nodes in the subtree rooted from *br*. The subtree rooted from *br* may have more than one states because the search heuristic may first probe *br* and its child states without reaching the end of a path, and then switch to explore *br*'s sibling branch to the end of a path.

Function `Slice` implements an improved version of *path slicing* [40]. Given a trace of instructions and a target instruction in the trace, path slicing traverses the trace backwards and computes a slice capturing the instructions sufficient for the execution to reach the target. Intuitively, these instructions include "branches whose outcome matters" to reach the target and "mutations that affect the outcome of those branches" [21]. For instance, it takes a branch instruction into the slice if the instruction's off-the-path branch may cause the execution not to reach the target. The resultant slice includes all instructions that the target transitively control- and data-depends upon.

`Slice` improves the original version of path slicing in two ways. First, it slices toward a sequence of instructions, not just one instruction, to capture instructions that guarantee the same event sequence. Operationally, `Slice` captures a branch instruction into the slice if the off-the-path branch of this instruction may reach new events and thus alter the event sequence. (See our previous work [23] for more details.) Second, `Slice` uses the `evmask` values returned by `OnExecution` (§4) to compute a more precise slice. To illustrate, consider the code snippet below:

```
1: char *filename;
2: if(*)
3:     filename = "A";
4: else
5:     filename = "B";
6: FILE *fp = fopen(filename, "r");
```

Suppose `fopen` is an event and we are given a path 2:true, 3, and 6. If at line 6, the returned `evmask` indicates that no operands of `fopen` are important, then `Slice` does not take any instructions between lines 1–5. However, if `evmask` indicates that operand `filename` is also important, `Slice` takes 2:true and 3.

Discussion. Algorithm 1 is checker-agnostic because all checker details are abstracted behind the checker interface. It is also heuristic-agnostic because it prunes states only at the end of a program path, and does not dictate how states should be added to or removed from the queue. Thus, this algorithm enables WOODPECKER to work with any checker and any heuristic. Moreover, this algorithm is sound because it never prunes a branch that may lead to a different event sequence, guaranteed by `Slice`.

6. Checkers

WOODPECKER's checking interface enables users to write their own checkers. To bootstrap checking, WOODPECKER also provides a set of builtin checkers, described in this section.

The *assertion checker* is the simplest builtin checker. It checks that the `assert()` statements never fail. Although existing symbolic execution systems such as KLEE can already check assertions, WOODPECKER can use them to direct checking toward only the paths that may trigger the assertions, thus enjoying large speedup (§8). Implementing this checker is extremely easy because `assert` is macro-expanded down to an `if`-statement and a call to `assert_failed()`, so this checker considers only `assert_failed()` calls events.

The *memory leak checker* checks that (1) every heap-allocated object is freed and (2) freed memory is not freed again. It tracks memory allocators `malloc`, `calloc`, `realloc`, `strdup`, `strndup`, and `getline`, and deallocators `free` and `realloc`. Function `getline` allocates memory if its first argument points to NULL. Function `realloc` allocates memory if its pointer argument is NULL and its size argument is greater than 0; it frees memory if its pointer argument is not NULL and its size argument is 0; or it both allocates and frees memory otherwise from this checker's perspective. Tracking these nuances is made easy because WOODPECKER does run code.[4]

The *open-close checker* checks that (1) an opened file is always closed and (2) a closed file is not closed again. It tracks `fopen`, `fopen64`, and `fdopen` for opening and `fclose` for closing files.

The *file access checker* checks that (1) file read or write operations are done on opened files and (2) no file read or write operations are done on a file with errors (after `ferror()` returns nonzero). In addition to the open and close file operations, it also tracks nine file access operations, such as `fread`, `fwrite`, `fgets`, and `fputs` and two error-checking operations `ferror()` and `ferror_unlocked()`.

The *data loss checker* checks for problematic sequences of file operations that may cause data loss. Although a POSIX-conforming `rename` guarantees atomic replacement of the destination link despite failures, it does not guarantee anything about file data. Thus, the source must be flushed and synced before the rename so that a crash would not make the destination point to incomplete data [55, 63]. Moreover, prior to this rename, the destination should not be unlinked or renamed so that a crash would not lose the destination link. Figure 5 illustrates these rules. The rename checker tracks 19 file system operations, such as the ones shown in Figure 5. Specifically, it checks that (1) a file is properly flushed (via `fflush`) and synced (via `fsync`, `sync`, or `fdatasync`) before it is renamed (via `rename`) to a destination link; (2) the destination link is not unlinked (via `unlink`, `rmdir`, or `rename`) before the rename; and (3) a file descriptor is not closed via `close` if it is associated with a `FILE` object.[5]

```
fp = fopen("src", "w");
fwrite(fp, ...)
fflush(fp); // must flush data from Libc to OS
fd = fileno(fp);
fsync(fd); // must sync data from OS to disk
fclose(fd);
// unlink("dst"); ERROR! shouldn't remove dst
// rename("dst", "bak"); ERROR! shouldn't remove dst
rename("src", "dst");
```

Figure 5: *How to atomically rename a file to avoid data loss.*

Additional support for checkers. WOODPECKER provides three additional mechanisms to help user write checkers. First, it provides an alias analysis (§7) for checkers to resolve function pointers in `MayBeEvent`. If an instruction passed to `MayBeEvent` is an indirect call, `MayBeEvent` can use this analysis to resolve possible call targets. This mechanism is used in all our checkers.

Second, WOODPECKER provides a default mechanism to suppress redundant error reports. The same error may occur along multiple program paths, leading to redundant reports. Given a sequence of events in an error report, WOODPECKER hashes all these events and their call stacks, and filters future reports with an identical hash, improving diagnosis experience. This mechanism is used in our memory leak and open-close checkers because they often emit many reports.

Third, WOODPECKER allows a checker to concretize symbolic data. The instruction operands passed to `OnExecution` may be symbolic. For instance, the data loss checker may see symbolic file names in `rename`. Although a checker can track the checker-relevant status of symbolic data, the implementation would be quite complex because it has to track advanced constraints. Instead, WOODPECKER allows a checker to selectively concretize symbolic data into a constant allowed by the constraints. A checker should do so only on the operands it cares about. This mechanism is used only by the data loss checker on the `rename` operands.

7. Implementation

We have implemented WOODPECKER on top of the LLVM compiler [1] and the KLEE symbolic execution system [12]. We reimplemented a previous path slicing algorithm [23]. Our implementation consists of 10,328 lines of code, including 2,968 lines of modifications to KLEE, 5,771 lines of code for path slicing, and 1,589 lines of code for the builtin checkers. The lines of code for each individual checker are show in Table 1.

In the remainder of this section, we discuss several implementation issues, two on modeling the environment for symbolic execution and three on improving the precision and speed of path slicing.

7.1 Modeling the File System

Most of the systems programs we check need to interact with the file system. Although KLEE already comes with a symbolic file system implementation that supports fake in-memory files with symbolic data, this file system has three limitations, preventing us from checking some rules and programs. First, KLEE does not track

[4] An alternative leak checker implementation is to modify KLEE because it already tracks memory objects, but these modifications must (1) track whether each memory object is allocated by the application or library code because we want to report only application leaks; and (2) maintain diagnosis information such as the call stacks. We rejected this design because these modifications are too checker-specific.

[5] Libc function `fileno` allows developers to get the file descriptor from a `FILE` object as illustrated in Figure 5.

the modifications to the real files with concrete data on disk. Thus, when it restores a program state to explore, it may see the old on-disk file system state, inconsistent with the restored program state. Second, KLEE supports only a small set of file operations with its symbolic file system; operations such as `link`, `rename`, and `chdir` are not supported. Lastly, a relative minor issue is that KLEE restricts the names of symbolic files to be `A`, `B`, ..., and `Z`, and no other files can be symbolic.

WOODPECKER solves these issues with a new in-memory file system layer that implements copy-on-write on a real file system, similar to `unionfs` [51]. If a concrete file is never written along a program path, WOODPECKER directs all reads to the file on disk. If the file is ever written, WOODPECKER creates an in-memory copy of the file private to the current program state, and directs all future reads and writes to this private copy. This file system supports more operations than KLEE's, such as `link`, `rename`, and `chdir`. It also allows users to mark arbitrary files as symbolic if the file names match user-given patterns. For instance, users can mark all `git` internal files as symbolic with the "`.git/*`" pattern.

7.2 Modeling `mmap`

Several programs we check allocate memory or read file data by calling `mmap`, which KLEE does not handle: it simply returns `EPERM` on `mmap`. One solution is to pass `mmap` calls to the underlying operating system, but this solution is still problematic because the memory allocated by the operating system's `mmap` is not saved and restored with the program state. WOODPECKER solves this problem by re-implementing `mmap` to allocate memory using KLEE's `malloc`. Our implementation currently handles two `mmap` modes: `MAP_PRIVATE` and `MAP_ANONYMOUS` because the other two modes, `MAP_SHARED` and `MAP_FIXED`, are not used in the programs we check. (Adding them is easy.)

7.3 Summarizing Functions

Users can improve WOODPECKER by providing two types of function summaries. First, to track data dependencies, path slicing needs to know what memory locations an external function may read or write. For instance, given a branch instruction that matters to reach an event, if its outcome depends on a value written by an external function, then the corresponding call must be included in the slice. For speed, we also summarize several popular `Libc` functions already linked in by KLEE, such as `memcpy`. These functions are frequently called, and their execution may contain a large number of instructions, so iterating over all these instructions for slicing may take long. In total, we summarize 22 `Libc` or external functions.

Second, to track dependencies between instructions, our slicing algorithm frequently queries alias analysis. The particular one we used is `bddbddb` [6, 58, 59]. For better alias results, `bddbddb` needs to know custom memory allocators, deallocators, and copiers such as `memcpy`. In total, we summarize 43 such functions.

7.4 Limiting Context Sensitivity

For precision, `bddbddb`'s alias analysis distinguishes alias results for different call graph paths, or *contexts*. In programming language terms, this analysis is *context-sensitive*. To gain context sensitivity, `bddbddb` conceptually creates a unique function clone for each possible context to the function. To efficiently store these function clones and contexts, `bddbddb` uses an advanced data structure called the binary decision diagrams [46], which often store large data sets compactly. Despite so, this cloning may explode for large programs with complex call graphs, such as `git`. Moreover, when there are many contexts, computing the alias results also takes long.

We worked around context explosion using two techniques. First, we modified `bddbddb` to limit the maximum number of function clones. If this limit is ever reached, our modified `bddbddb` stops creating function clones. Although in theory this approach may cause some precision loss because results of some contexts are merged together, in practice the maximum number of function clones is already large enough to yield precise results. We used this workaround only for `git`. Second, we replaced a few indirect calls in the evaluated programs with direct calls. If a function pointer has many call sites and may point to several functions, it may explode the contexts because each call site and possible function target combination leads to a function clone. In many cases, though, the function pointer has only one target, but `bddbddb` cannot infer this fact. We thus modified 23 lines in `shadow`, 10 in `tar`, and 17 in `git` to replace frequent indirect calls with direct calls.

7.5 Caching Analysis Results

For speed, WOODPECKER caches static analysis results extensively. For instance, it caches alias results. It also caches the results from a checker's `MayBeEvent` because whether a static instruction is an event does not change during symbolic execution.

8. Evaluation

We evaluated WOODPECKER on total 136 widely used systems programs, including 96 programs in GNU `coreutils` version 8.12, a basic file, shell, and text manipulation utilities suite [20]; 30 programs in `shadow` 4.1.5, a user and group management utilities suite [54]; `tar` 1.26, an archival program [56]; `sed` 4.2.1, a text stream transformation program [31]; `CVS` 1.11.23, a version control program [24]; and seven programs from `git` 1.7.9.4, another version control program [30]. We chose the latest stable version of each program at the time of experiment. We excluded `printf`, `md5sum`, and `date` from `coreutils` and `sulogin`, `chfn`, and `gpasswd` from `shadow` because they caused aborts either in WOODPECKER, KLEE, or `STP` (KLEE's underlying constraint solver [2, 28]). For example, a floating point constraint generated by `printf` likely caused `STP` to crash.

We compiled all programs to LLVM bitcode using LLVM `gcc`-2.7 with `-O2` or `-O3`. We ran all experiments on four 2.8 Ghz dual-socket hexa-core Intel Xeon X5660 machines with 64 GB memory and Linux 2.6.38. To fully use the available cores, we ran roughly ten experiments concurrently on each machine. We observed little interference between concurrent experiments because symbolic execution is mostly CPU bound.

In addition to measuring WOODPECKER's verification results, we also compared it with KLEE, a state of the art symbolic execution system. For both WOODPECKER and KLEE experiments, we used the same settings as the ones used by the KLEE authors [12] whenever applicable. Some settings were removed from the open source version of KLEE, so we reimplemented them with the help from a KLEE author [10]. We adjusted some settings because some programs we evaluated are different, sometimes much larger.

Specifically, we used up to three two-byte symbolic command line arguments for `coreutils` and `shadow`, and four eight-byte arguments for other programs. We added a concrete username argument for most programs in `shadow` to match their expectations. We used five 200-byte symbolic files for `tar` because it requires large file headers, and two eight-byte files for all the other programs. We also marked the configuration files symbolic for `CVS` and `git` because these version control programs run different paths based on different configurations. Since exploring all paths may take forever, we bounded each experiment for one hour for `coreutils` and `shadow`, three hours for `git`, and twelve hours for all other programs. In many experiments, WOODPECKER verified all paths within minutes; see §8.1. Since the results may depend on search heuristics, we chose the same heuristics that yielded the best results for KLEE [12]. (We also experimented with depth-first search but it

Checkers	Lines of Code	Programs Checked	Programs Verified		Relevant Paths Verified		Redundant Paths	
			WOODPECKER	KLEE	WOODPECKER	KLEE	WOODPECKER	KLEE
Assertion	102	57	13	3	195,268	45,763	69,795	195,178
Memory leak	399	103	32	7	1,024,676	176,475	451,836	1,657,721
Open-close	211	72	19	4	528,676	82,883	203,407	512,439
File access	344	120	40	12	1,694,393	377,181	496,651	2,141,111
Data loss	533	35	7	7	132,136	89,779	22,996	117,225
Total	1,589	387	111	33	3,575,149	772,081	1,244,685	4,623,674

Table 1: *Summary of verification results.*

caused KLEE to cover much fewer statements. Thus, we exclude depth-first search results from this section.)

The following subsections focus on

§8.1: Can WOODPECKER effectively verify many or all relevant paths over bounded input? Can it outperform KLEE significantly in this regard?

§8.2: Can WOODPECKER effectively detect rule violations? WOODPECKER can also be used as an error detector in addition to a verifier.

§8.3: Is it costly to compute what branches to prune? This cost must be smaller than the time it takes to actually explore the branches.

8.1 Verification Results

Table 1 summarizes the verification results. The number of programs checked by each checker is smaller than the total number of programs evaluated because we excluded trivially verifiable programs, *i.e.*, those with no events regarding a rule. The table shows the number of programs verified over bounded input for both WOODPECKER and KLEE. WOODPECKER verifies 111 out of 387 (28.7%) program and rule combinations over bounded input. KLEE verifies only 33 (8.5%) combinations. WOODPECKER's verification experiments often finished quickly, within tens of minutes (not shown in the table).

For the other 276 programs, Table 1 shows the number of relevant paths verified. Overall, WOODPECKER verifies 4.6 times as many relevant paths as KLEE. This speedup is unsurprising because 86.8% of the paths KLEE explores are redundant (see detailed results below). The table also shows the number of redundant paths pruned by WOODPECKER or explored by KLEE. Although WOODPECKER prunes redundant paths, they are already forked, which means that WOODPECKER has already queried the constraint solver to determine whether the corresponding branches are feasible. Since constraint solving is the most costly for WOODPECKER (and likely any symbolic execution system), we count the number of pruned paths as wasted work done by WOODPECKER. The real benefit of pruning is that it prevents future forks from the pruned paths. As shown in the table, KLEE explores 3.7 times as many redundant paths as WOODPECKER prunes. Note that it hugely favors KLEE to compare the amount of wasted work by the number of redundant paths because KLEE fully explores them whereas WOODPECKER does not.

We observed that, with a longer time limit, WOODPECKER verified much more paths than KLEE's. The reason is that initially KLEE may hit some relevant paths by chance, but over time it gets stuck exploring redundant paths nearby. Table 2 shows the results with the time limit set to one, two, and four hours for `coreutils`. With a one-hour limit, WOODPECKER verified 5.2 times as many paths as KLEE. With a four-hour limit, WOODPECKER verified 17.0 times as many. This big increase is unsurprising because WOODPECKER exponentially speeds up symbolic execution. We observed similar results (not shown) with a larger input bound.

To evaluate how directed WOODPECKER is, we define *search efficiency* as the percentage of relevant paths explored over all paths ever forked. We use this metric only on the programs with a subset of paths verified because WOODPECKER finished quickly on the verified programs. From the data in Table 1, we can compute that WOODPECKER's average search efficiency is 64.9% whereas KLEE's is 29.2%, a 35.7% difference.

This difference is even larger on each individual program. Figure 6–10 show the search efficiency for each program and checker. The top subfigure show results for WOODPECKER, and the bottom KLEE. The solid or hatched portion of each bar shows the search efficiency, with solid meaning all relevant paths verified. The white portion shows roughly the amount of wasted work. The more solid bars in a figure, the more programs are verified. The larger the solid or hatched portion and the smaller the white portion in a bar, the less work is wasted. We identified whether a path KLEE explored was relevant or redundant by running WOODPECKER's search algorithm together with KLEE without actually pruning paths. Over all programs and checkers, WOODPECKER's median search efficiency is 62.9% whereas KLEE's is 16.7%, a 46.2% difference.

Time Limit	Programs Verified			Paths Verified		
	W	K	W- K	W	K	W/K
1 hour	73	7	67	2,776,499	532,222	5.2
2 hours	104	31	73	6,933,817	662,558	10.5
4 hours	112	39	73	14,437,294	847,621	17.0

Table 2: *With a longer time limit,* WOODPECKER *verified much more paths than* KLEE. W represents WOODPECKER's results and K KLEE's, both obtained from `coreutils`.

8.2 Rule Violations Detected

Table 3 shows the rule violations detected, broken down by checkers and programs. WOODPECKER detected many memory leaks and unclosed file pointers in the checked programs. A common pattern is that these programs allocate memory or open files, but exit without freeing these resources if some failure occurs. This pattern is considered bad programming practice at least. Worse, they may also have security implications because previous work [15, 16] has shown that leaked memory or file pointers may needlessly extend sensitive data lifetime, posing privacy risks. Some of the detected errors are already fixed in the latest version of the program, illustrating developers' concerns over these errors. (To avoid inflating our bug counts, we did not include these fixed errors in the table.)

Programs	mem leak	open-close	data loss
`coreutils`	40	13	0
`shadow`	11	5	1
`tar`	4	0	0
`sed`	3	0	0
`CVS`	3	1	2
`git`	19	4	7
Total	80	23	10

Table 3: *Number of rule violations detected.* The assertion and file access checkers are not listed because they detected no violations.

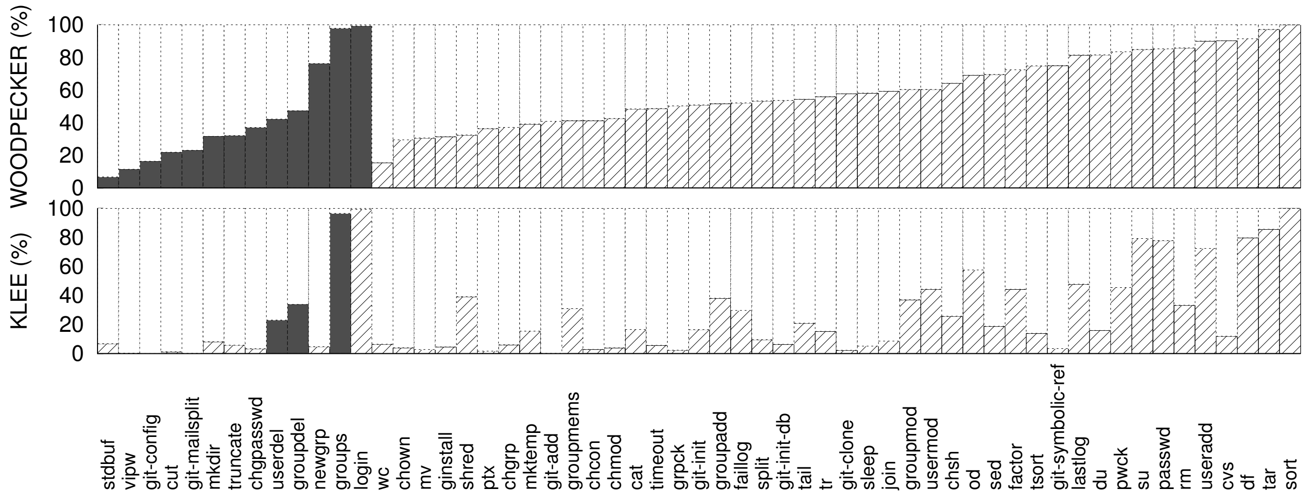

Figure 6: *Search efficiency with the assertion checker.* The top subfigure show results for WOODPECKER, and the bottom KLEE. The solid or hatched portion of each bar shows the search efficiency, with solid meaning all relevant paths verified. The white portion shows the amount of wasted work. The more solid bars in a figure, the more programs are verified. The larger the solid or hatched portion and the smaller the white portion in a bar, the less work is wasted. WOODPECKER's median search efficiency of the hatched bars is 56.8%, whereas KLEE's is only 16.2%.

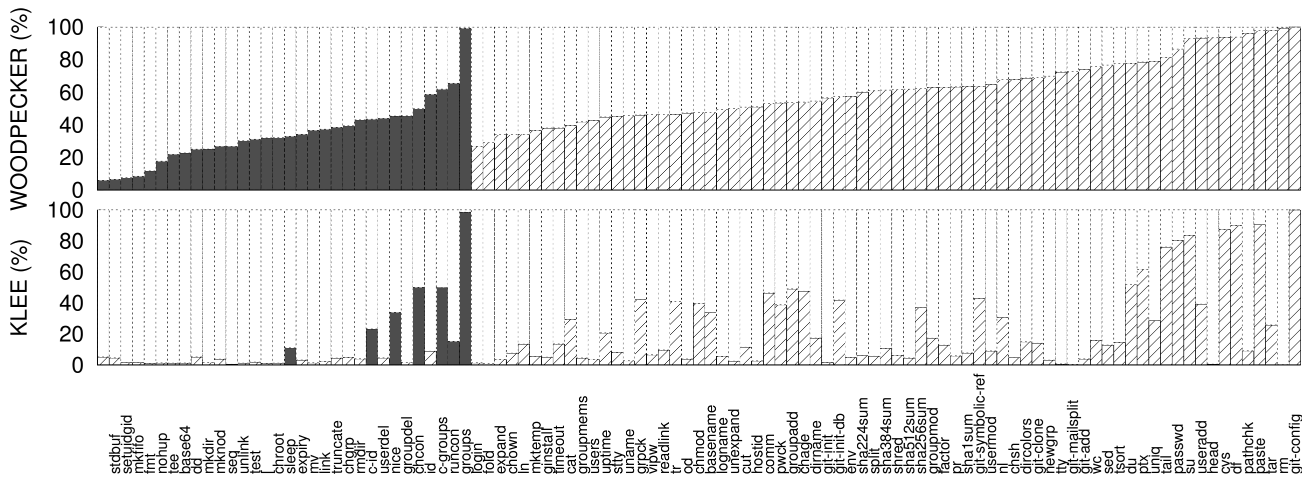

Figure 7: *Search efficiency with the memory leak checker.* WOODPECKER's median search efficiency of the hatched bars is 61.3%, whereas KLEE's is 12.7%.

There are two more-serious open-close errors detected in `git` and `CVS`. The `git mailsplit` command iterates through a list of mbox files and splits each into individual email files. Its `split_mbox` function opens a mbox file, but does not close the file if it reaches the end of the file before a newline. Thus, file descriptors may be exhausted with a long list of such files. The `CVS` function `parse_config` returns without closing a configuration file if there is any syntax error. This function is also called for each authentication or `Root` request in `CVS` server mode, potentially exhausting file descriptors.

WOODPECKER also detected ten serious data loss errors, seven in `git`, two in `CVS`, and one in `shadow`. These programs often deal with crucial files such as source code or `/etc` configurations, so they are extremely careful with file writes. They do not update files in place because a crash may leave a partially updated file. Instead, they create temporary files, and then sync and atomically rename them. Despite so, WOODPECKER still finds serious errors in them.

The seven `git` errors may all corrupt source repositories. They share the same pattern. To commit changes, users first run a series of "`git add`" to add the new or modified files to an *index*, and then commit the index. This index is stored in file `.git/index`. When "`git add`" is called, `git` carefully creates a temporary file `.git/index.lock` to store the new index, and then renames it to `.git/index`. Unfortunately, `git` does not call `fsync` on the temporary file, so a crash can still corrupt the index, which further corrupts the repository if committed. This error pattern has been confirmed by the `git` developers. The two `CVS` errors are similar.

The `shadow` data loss error WOODPECKER detected may cause new user accounts to have wrong settings such as wrong home directory locations, account expiration dates, and maximum numbers of inactive days. The defaults of these settings are stored in `/etc/default/useradd`, which `useradd` in `shadow` updates. However, when updating this file, `useradd` first renames `/etc/default/useradd` to a backup file, before renaming the

Figure 8: *Search efficiency with the open-close checker.* WOODPECKER's median search efficiency of the hatched bars is 59.2%, whereas KLEE's is 15.3%.

Figure 9: *Search efficiency with the file access checker.* WOODPECKER's median search efficiency of the hatched bars is 65.4%, whereas KLEE's is 20.9%.

Figure 10: *Search efficiency with the data loss checker.* WOODPECKER's median search efficiency of the hatched bars is 89.4%; KLEE's is 71.1%.

temporary file to `/etc/default/useradd`. If a crash occurs between the two rename operations, the `/etc/default/useradd` link would be lost. When new users are created, `useradd` would use wrong defaults, potentially causing security issues. This error has been confirmed and fixed by the `shadow` developers.

Detecting these data loss errors requires much precision. For instance, one of the `git` errors involves file operations spanning five levels of call stack with hundreds of function calls between them. Static analysis would likely get lost facing this complexity.

Programs	**Pruning (%)**	**Alias (%)**
`coreutils`	0.82	0.49
`shadow`	1.98	0.11
`tar`	4.59	5.87
`sed`	0.69	0.50
`CVS`	4.36	4.64
`git`	8.99	11.71

Table 4: *Analysis overhead over total verification time.*

8.3 Overhead of Pruning

When WOODPECKER prunes branches, it needs to slice out irrelevant instructions from a path. In addition, it also needs to compute alias results needed by path slicing. Both pruning and alias analysis

may take time. If the combined overhead is larger than the time it takes to actually explore the pruned paths, pruning would not be worthwhile. Table 4 shows the relative overhead of pruning and analysis analysis. Both analyses have low overhead for simple programs, and moderately increased overhead for complex programs such as `git`. Note that `bddbddb` saves alias results on disk, so we can run it only once for a program and reuse the results over all checking sessions. We plan to implement this optimization in future work.

9. Related Work

This section compares WOODPECKER to automated checking or verification techniques.

Static analysis. Several static analysis systems can also check rules. Meta Compilation [26, 27, 36, 61] provides a simple state machine language with pattern matching support for describing rules. It checks the rules on (potentially infeasible) program paths. It aggressively traded off soundness for low false positive rates, so while it can effectively detect thousands of errors [14], it cannot be used for verification. ESP is a static verifier [25] that also abstracts rules into state machines and verifies them on program paths. Although it has been applied to verify that the `gcc` from SPEC95 benchmark does not print to unopened files, this version of `gcc` uses only simple boolean flags to control whether to open files, and opens only up to a statically determined number (15) of files in a compilation pass [25]. Without inferring precise runtime effects of code, ESP is likely to emit false positives on many programs such as the simple one in Figure 1.

Symbolic execution. Symbolic execution, or more precisely the recent development of this technique characterized by the mixture of concrete and symbolic executions [11, 12, 33, 42, 45, 48, 49, 53], has gained much traction. Researchers have used this technique to detect serious errors [12, 13, 64], generate high-coverage tests [12], reuse thread schedules on different inputs [22, 23], (re)produce a buggy execution [4, 44, 66], verify paths or programs over bounded input [52], *etc*. WOODPECKER is complementary to much of the previous work. Its rule-directed approach may benefit many of these systems and, as discussed earlier, its carefully designed search algorithm can leverage the powerful search heuristics in existing systems. Below we discuss five related systems.

The S2E system [13] symbolically executes binary programs by dynamically translating x86 assembly to LLVM bitcode using QEMU [50]. It allows users to specify what code to symbolically explore paths for, and treats other code as the "environment" that runs concretely. It categorizes a set of heuristics with variable soundness and performance to handle the transitions between symbolic and concrete executions. It can also use a cluster to provide linear speedup of symbolic execution [9]. Recently, a state merging technique [43] has been proposed within S2E. This technique reduces the number of paths to explore by merging several into one. Doing so may actually slow down symbolic execution, so this technique employs heuristics to estimate the cost and benefit of state merging. S2E's goals are very different from WOODPECKER's. S2E is not designed to check rules. Nor does it automatically direct symbolic execution given a rule implemented using its interface. Its heuristics may remove paths unsoundly, inappropriate for verification. Its state merging technique is purely dynamic, whereas WOODPECKER leverages static analysis to prune paths before they are executed. Nonetheless, many ideas in WOODPECKER are orthogonal to those in S2E. For instance, WOODPECKER may use S2E to merge paths and provide linear speedup of symbolic execution on a cluster; S2E may use WOODPECKER to guide symbolic execution toward the rules users want to check and soundly prune irrelevant paths.

To reproduce a buggy execution, two systems attempt to compute a feasible program path reaching a given line of code using symbolic execution [44, 66]. They do so using a combination of symbolic execution and static analysis. DyTa [29] appears to share the same goal except it starts from statically detected errors. DyTa was briefly described in a 3-page paper [29] in the demonstration track of ICSE '11, and we could not find a full paper describing its implementation. Another system focuses symbolic execution on the patch between two versions of a program [47]. Again, these systems do not check rules. Nor do they attempt to explore all paths relevant to the rules for verification. At a technical level, these systems separate their static analysis and symbolic execution: they statically analyze a program's data- and control-dependencies to compute a potentially feasible path, then use symbolic execution to validate whether the path is feasible. This clear separation is different from how WOODPECKER combines static analysis and symbolic execution: WOODPECKER uses symbolic execution to explore paths and, for each explored path, it applies static analysis on the off-the-path branches to compute whether they are relevant. Previous work has shown that the information provided by a path significantly improves static analysis precision [40], a key advantage of path slicing over program slicing.

Bouncer [21] computes filters to block malicious input by combining static analysis and symbolic execution. Bouncer can be viewed as an application of path slicing [40], with precision improvements to the original path slicing algorithm in [40]. WOODPECKER applies path slicing for a different goal: speeding up symbolic exploration of paths. One path slicing improvement in WOODPECKER is that WOODPECKER slices toward an event sequence, instead of a single event.

Program Slicing. Program slicing [57] can remove irrelevant static statements from a program. It can aid debugging and optimization. Dynamic program slicing [3, 67] produces more accurate slicing results than static program slicing, but may unsoundly remove relevant branches. Path slicing [40] soundly removes irrelevant statements from a program path, and is more precise than program slicing. WOODPECKER greatly benefits from path slicing and can be viewed as an application of path slicing to speeding up symbolic execution.

Other techniques. Testing is lightweight and can address some limitations of static analysis, but previous work shows that manual or random testing tends to have low coverage [12]. As Jhala and Majumdar [40] pointed out in their path slicing paper, counterexample-guided program analyses [7, 37**? ?**] can leverage path slicing to compute shorter counterexamples and better refine their abstractions. These analyses are purely static, unlike the symbolic execution WOODPECKER targets. Recent work on explicit-state software model checking [32, 34, 41, 60, 62, 63, 65] has yielded many serious errors in real systems. This technique excels at exploring nondeterministic choices in the environment such as whether a disk `read` fails, thus it is complementary to symbolic execution that explores feasible program paths over unconstrained input, such as the input data `read` reads. Some of these systems can also soundly reduce the number of executions to explore. However, their reduction is purely dynamic and they do not leverage static analysis to guide the exploration.

10. Conclusion

We have presented WOODPECKER, a symbolic execution system designed to verify systems rules over bounded input. Leveraging the insight that only a small portion of paths are relevant to a checked rule, WOODPECKER soundly removes redundant paths and drives symbolic execution to effectively verify rules. It comes with a set of builtin checkers for common rules, and an interface

for users to check custom rules. Given a rule, WOODPECKER uses a sound, checker- and heuristic-agnostic search algorithm to direct symbolic execution toward the paths relevant to the rule. Evaluation on 136 widely used programs totaling 545K lines of code shows that WOODPECKER can effectively verify rules on programs and paths over bounded input and detect serious errors.

Acknowledgments

Roxana Geambasu, Ying Xu, and the anonymous reviewers provided many helpful comments, which have substantially improved the content and presentation of this paper. This work was supported in part by AFRL FA8650-11-C-7190, FA8650-10-C-7024, and FA8750-10-2-0253; ONR N00014-12-1-0166; NSF CCF-1162021, CNS-1117805, CNS-1054906 (CAREER award), and CNS-0905246; an AFOSR YIP award; and a Sloan Research Fellowship.

References

[1] The LLVM compiler framework. `http://llvm.org`.

[2] STP Constraint Solver. `https://sites.google.com/site/stpfastprover/`.

[3] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation (PLDI '90)*, pages 246–256, 1990.

[4] G. Altekar and I. Stoica. ODR: output-deterministic replay for multicore debugging. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, pages 193–206, Oct. 2009.

[5] A. Andoni, D. Daniliuc, S. Khurshid, and D. Marinov. Evaluating the "small scope hypothesis". Technical report, MIT CSAIL, 2002.

[6] D. Avots, M. Dalton, V. B. Livshits, and M. S. Lam. Improving software security with a C pointer analysis. In *Proceedings of the 27th International Conference on Software Engineering (ICSE '05)*, pages 332–341, May 2005.

[7] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proceedings of the Eighth International SPIN Workshop on Model Checking of Software (SPIN '01)*, pages 103–122, May 2001.

[8] BBC. The life of birds documentary.

[9] S. Bucur, V. Ureche, C. Zamfir, and G. Candea. Parallel symbolic execution for automated real-world software testing. In *Proceedings of the 6th ACM European Conference on Computer Systems (EUROSYS '11)*, pages 183–198, 2011.

[10] C. Cadar. Private email communication, Mar. 2012.

[11] 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 (CCS '06)*, pages 322–335, Oct.–Nov. 2006.

[12] C. Cadar, D. Dunbar, and D. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the Eighth Symposium on Operating Systems Design and Implementation (OSDI '08)*, pages 209–224, Dec. 2008.

[13] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: a platform for in-vivo multi-path analysis of software systems. In *Sixteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '11)*, pages 265–278, 2011.

[14] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 73–88, Nov. 2001.

[15] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding Data Lifetime via Whole System Simulation. In *Proceedings of the 13th USENIX Security Symposium*, 2004.

[16] J. Chow, B. Pfaff, T. Garfinkel, and M. Rosenblum. Shredding Your Garbage: Reducing Data Lifetime Through Secure Deallocation. In *Proceedings of the 14th USENIX Security Symposium*, 2005.

[17] E. Clarke and D. Kroening. Hardware verification using ANSI-C programs as a reference. In *Proceedings of ASP-DAC 2003*, pages 308–311, January 2003.

[18] A. Coen-Porisini, G. Denaro, C. Ghezzi, and M. Pezzé. Using symbolic execution for verifying safety-critical systems. In *Proceedings of the Eighth European Software Engineering Conference held jointly with the Ninth ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-9)*, pages 142–151, 2001.

[19] P. Collingbourne, C. Cadar, and P. H. Kelly. Symbolic crosschecking of floating-point and SIMD code. In *Proceedings of the 6th ACM European Conference on Computer Systems (EUROSYS '11)*, pages 315–328, Apr. 2011.

[20] Coreutils - GNU core utilities. `http://www.gnu.org/software/coreutils`.

[21] M. Costa, M. Castro, L. Zhou, L. Zhang, and M. Peinado. Bouncer: securing software by blocking bad input. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP '07)*, pages 117–130, Oct. 2007.

[22] H. Cui, J. Wu, C.-C. Tsai, and J. Yang. Stable deterministic multithreading through schedule memoization. In *Proceedings of the Ninth Symposium on Operating Systems Design and Implementation (OSDI '10)*, Oct. 2010.

[23] H. Cui, J. Wu, J. Gallagher, H. Guo, and J. Yang. Efficient deterministic multithreading through schedule relaxation. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, Oct. 2011.

[24] CVS. `http://www.cvshome.org`.

[25] M. Das, S. Lerner, and M. Seigle. Esp: path-sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI '02)*, pages 57–68, June 2002.

[26] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI '00)*, Sept. 2000.

[27] D. Engler, D. Yu Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, 2001.

[28] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *Proceedings of the 19th International Conference On Computer Aided Verification (CAV' 07)*, pages 519–531, 2007.

[29] X. Ge, K. Taneja, T. Xie, and N. Tillmann. Dyta: dynamic symbolic execution guided with static verification results. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 992–994, 2011.

[30] Git. `http://git-scm.com/`.

[31] GNU sed. `http://www.gnu.org/software/sed`.

[32] P. Godefroid. Model Checking for Programming Languages using VeriSoft. In *Proceedings of the 24th Annual Symposium on Principles of Programming Languages (POPL '97)*, pages 174–186, Jan. 1997.

[33] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI '05)*, pages 213–223, June 2005.

[34] H. Guo, M. Wu, L. Zhou, G. Hu, J. Yang, and L. Zhang. Practical software model checking via dynamic interface reduction. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, Oct. 2011.

[35] B. Hackett and A. Aiken. How is aliasing used in systems software? In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '06/FSE-14)*, pages 69–80, Nov. 2006.

[36] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *Proceedings of the ACM*

SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI '02), 2002.

[37] T. A. Henzinger, R. Jhala, R. Majumdar, , and G. Sutre. Lazy abstraction. In *Proceedings of the 29th Annual Symposium on Principles of Programming Languages*, pages pp. 58–70, 2002.

[38] D. Jackson and C. A. Damon. Elements of style: Analyzing a software design feature with a counterexample detector. *IEEE Trans. Softw. Eng.*, 22(7):484–495, July 1996.

[39] J. Jaffar, V. Murali, J. A. Navas, and A. E. Santosa. Tracer: A symbolic execution tool for verification. In *Proceedings of the 24th international conference on Computer aided verification*, CAV'12, July 2012.

[40] R. Jhala and R. Majumdar. Path slicing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI '05)*, pages 38–47, 2005.

[41] C. Killian, J. W. Anderson, R. Jhala, and A. Vahdat. Life, death, and the critical transition: Finding liveness bugs in systems code. In *Proceedings of the Fourth Symposium on Networked Systems Design and Implementation (NSDI '07)*, pages 243–256, April 2007.

[42] J. C. King. A new approach to program testing. In *Proceedings of the international conference on Reliable software*, pages 228–233, 1975.

[43] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea. Efficient state merging in symbolic execution. In *Proceedings of the ACM SIGPLAN 2012 Conference on Programming Language Design and Implementation (PLDI '12)*, pages 193–204, 2012.

[44] K.-K. Ma, Y. P. Khoo, J. S. Foster, and M. Hicks. Directed symbolic execution. In *The 18th International Static Analysis Symposium*, SAS '11, Sept. 2011.

[45] R. Majumdar and R.-G. Xu. Directed test generation using symbolic grammars. In *Proceedings of the Seventh European Software Engineering Conference held jointly with the Seventh ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-7)*, pages 553–556, 2007.

[46] J. S. Metos and J. V. Oldfield. Binary decision diagrams: From abstract representations to physical implementations. In *DAC '83: Proceedings of the 20th conference on Design automation*, pages 567–570, 1983.

[47] S. Person, G. Yang, N. Rungta, and S. Khurshid. Directed incremental symbolic execution. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 504–515, 2011.

[48] C. S. Păsăreanu and N. Rungta. Symbolic pathfinder: symbolic execution of java bytecode. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE '10, pages 179–180, 2010.

[49] C. S. Păsăreanu, N. Rungta, and W. Visser. Symbolic execution with mixed concrete-symbolic solving. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 34–44, 2011.

[50] QEMU. `http://www.qemu.org`.

[51] D. P. Quigley, J. Sipek, C. P. Wright, and E. Zadok. UnionFS: User- and Community-oriented Development of a Unification Filesystem. In *Proceedings of the 2006 Linux Symposium*, volume 2, pages 349–362, Ottawa, Canada, July 2006.

[52] D. A. Ramos and D. R. Engler. Practical, low-effort equivalence verification of real code. In *Proceedings of the 23rd international conference on Computer aided verification*, CAV'11, pages 669–685, 2011.

[53] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Proceedings of the 10th European Software Engineering Conference held jointly with the 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-13)*, pages 263–272, Sept. 2005.

[54] shadow. `http://pkg-shadow.alioth.debian.org/`.

[55] S. Smith. Eat my data: How everybody gets file io wrong. In *OSCON 2008*, July 2008.

[56] tar. `http://www.gnu.org/software/tar/`.

[57] M. Weiser. Program slicing. In *Fifth International Conference on Software Engineering*, pages 439–449, 1981.

[58] J. Whaley. bddbddb Project. `http://bddbddb.sourceforge.net`.

[59] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI '04)*, pages 131–144, June 2004.

[60] M. Yabandeh, N. Knezevic, D. Kostic, and V. Kuncak. CrystalBall: Predicting and preventing inconsistencies in deployed distributed systems. In *Proceedings of the Sixth Symposium on Networked Systems Design and Implementation (NSDI '09)*, Apr. 2009.

[61] J. Yang, T. Kremenek, Y. Xie, and D. Engler. MECA: an extensible, expressive system and language for statically checking security properties. In *Proceedings of the 10th ACM conference on Computer and communications security (CCS '03)*, pages 321–334, Oct. 2003.

[62] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using model checking to find serious file system errors. In *Proceedings of the Sixth Symposium on Operating Systems Design and Implementation (OSDI '04)*, pages 273–288, Dec. 2004.

[63] J. Yang, C. Sar, and D. Engler. Explode: a lightweight, general system for finding serious storage system errors. In *Proceedings of the Seventh Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 131–146, Nov. 2006.

[64] J. Yang, C. Sar, P. Twohey, C. Cadar, and D. Engler. Automatically generating malicious disks using symbolic execution. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy (SP '06)*, pages 243–257, May 2006.

[65] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou. MODIST: Transparent model checking of unmodified distributed systems. In *Proceedings of the Sixth Symposium on Networked Systems Design and Implementation (NSDI '09)*, pages 213–228, Apr. 2009.

[66] C. Zamfir and G. Candea. Execution synthesis: a technique for automated software debugging. In *Proceedings of the 5th ACM European Conference on Computer Systems (EUROSYS '10)*, pages 321–334, Apr. 2010.

[67] X. Zhang and R. Gupta. Cost effective dynamic program slicing. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI '04)*, pages 94–106, 2004.