Redundant State Detection for Dynamic Symbolic Execution

Suhabe Bugrara
Stanford University

Dawson Engler
Stanford University

Abstract

Many recent tools use dynamic symbolic execution to perform tasks ranging from automatic test generation, finding security flaws, equivalence verification, and exploit generation. However, while symbolic execution is promising, it perennially struggles with the fact that the number of paths in a program increases roughly exponentially with both code and input size. This paper presents a technique that attacks this problem by eliminating paths that cannot reach new code before they are executed and evaluates it on 66 system intensive, complicated, and widely-used programs. Our experiments demonstrate that the analysis speeds up dynamic symbolic execution by an average of 50.5 X, with a median of 10 X, and increases coverage by an average of 3.8 %.

1 Introduction

Dynamic symbolic execution has enabled many recent advances in program analysis such as high-coverage test input generation, patch checking, equivalence verification, malware signature generation, assertion checking, and debugging [7]. The technique's power comes from its ability to systematically and precisely enumerate program paths automatically. Further, in many cases it can eliminate false positives by producing a concrete test case to demonstrate a bug or specific path execution.

There are many variations in modern symbolic execution interpreters, but broadly speaking they work as follows. First, they start from some initial program state in which program inputs are represented by "unknowns" that take on any value. The interpreter symbolically executes the program by updating the state with the effect of each instruction. When it reaches a branch statement with condition C, the interpreter forks the state into two states, adding the constraint C to one and $\neg C$ to the other. This forking is skipped if one branch direction is infeasible. The interpreter repeatedly executes and forks states until either it generates every possible state of the program, thereby exploring every possible path with respect to the inputs or, more typically, it exhausts memory or exceeds a time limit.

While powerful, naive symbolic execution faces the significant challenge in practice that the number of paths (and thus states) increases roughly exponentially both with the size of the tested program and with the size of the program inputs. Programs with fewer than ten thousand lines of code routinely generate millions of states, each consisting of tens of thousands of memory locations. Thus, under realistic time and memory limits, state-of-the-art dynamic symbolic execution tools only explore a small percentage of paths.

As a result, while capable of deep reasoning, these tools have had limited applicability. They often fail when used to verify a property when doing so requires exploring every feasible path involving the property. Even when used purely for bug-finding, they often quickly get lost in an exponential number of superficially different but essentially identical states. Many tools [4, 5, 10, 12] counter this problem by using heuristic search strategies, but these have proven notoriously fragile.

This paper presents a novel, complementary approach that exploits a key observation: for many program analysis applications, most paths are redundant with respect to the goal of the symbolic execution and thus do not need to be explored. For example, if the goal is to generate a suite of program inputs that covers every line of code, then the symbolic execution only needs to explore paths that will reach lines which have not been covered by previously explored paths.

The contributions of this paper are (1) the design and implementation of a sound, redundant state detector capable of scaling up to handle real programs and (2) a thorough experimental evaluation on 66 system-intensive, complicated, and widely used programs which demonstrates that the detector yields dramatic performance improvements. Our technique speeds up dynamic

symbolic execution by an average of 50.5 X, with a median of 10 X, and increases coverage by an average of 3.8 %. On 12 of the benchmarks, the analysis reduces the state space sufficiently that the tool exhaustively explores all remaining states.

2 Overview

In this section, we give an overview of our analysis using the following program, which contains two variables: `w` and `m`.

```
1  if (w)
2    printf("X");
3  else
4    printf("Y");
5  if (m)
6    exit(0);
7  else
8    exit(1);
```

The program is symbolically executed from four initial states. States consists of a program counter and a set of constraints over program variables. Each of these four initial states starts at line 1. Suppose that state A has the constraints {`w = 0, m = 1`}, state B has constraints {`w = 1, m = 1`}, state C has constraints {`w = 2, m = 1`}, and state D has constraints {`w = 3, m = 0`}. When state A is symbolically executed through the program, it covers the lines 1, 3, 4, 5, and 6; state B covers 1, 2, 5 and 6; state C covers 1, 2, 5, and 6; and state D covers 1, 2, 5, 7, and 8. Figure 1 shows a diagram of the *state tree* produced by symbolically executing the program from these initial states.

As these four initial states are executed over the program, the same lines are covered over and over again. Since our goal is to produce high-coverage test suites and to explore different parts of the program to find bugs, we can avoid a significant amount of redundant work if we discard states that behave similarly to previously executed states. For example, after states A and B are executed to completion, state C will only cover a subset of the lines that A and B covered, so C is redundant and can be eliminated. The challenge is to detect that C is redundant without actually executing it. One simple approach is to compare the constraints of C to the constraints of A and B: if they have the same constraints, then they must follow the same paths, and thus will cover exactly the same lines. However, requiring that two states have identical constraints is unnecessarily conservative.

In this paper, we present an approach that precisely determines which constraints affect whether a state will cover the same lines as a previously executed state. We use *dynamic slicing* [1], a program analysis technique that finds all program statements that affected the value of a variable occurrence for given program inputs. Our

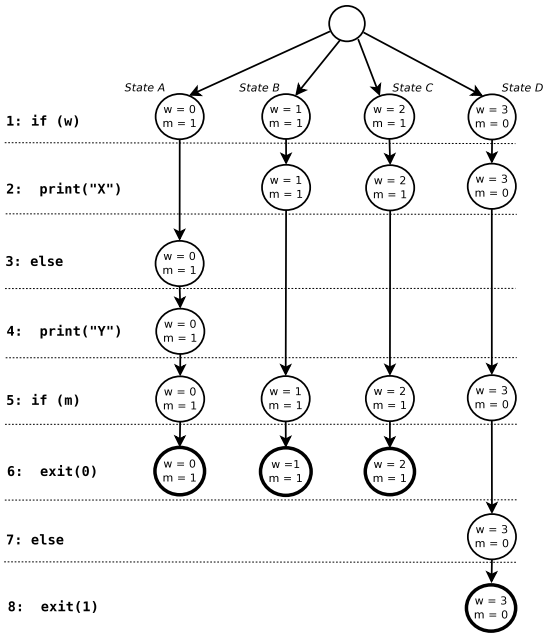

Figure 1: A diagram illustrating the *state tree* produced by symbolically executing the program in Section 2 from the initial states A, B, C, and D. Each state is represented by a circle that is labeled with the state's constraints. Each row of the diagram corresponds to a line of the program. States at exit instructions are denoted by thick borders. Let pc_1 be the program counter of state σ_1. An arrow from state σ_1 to state σ_2 means that σ_2 is the result of executing pc_1 on σ_1. The initial states A, B, C, and D appear in the top row, which corresponds to line 1. Note that the path starting from state A does not reach lines 2, 7, and 8, which is reflected in the diagram by missing circles in those rows along the path.

approach detects which variables affect branch instructions that control uncovered lines and restrict state comparison to constraints that involve these variables. By minimizing the set of constraints used to compare states, the analysis can find more opportunities to eliminate redundant states.

In the remainder of this section, we describe how our analysis works on the example program. As described above, symbolic execution of the program begins with four initial states A, B, C, and D whose program counters are set to line 1. First, state A is selected and a path through lines 1, 3, 4, 5, and 6 is explored until it terminates by executing the exit instruction on line 6. Figure 2 shows four different state trees at various stages of symbolically executing the program using our analysis. The leftmost diagram shows the state tree immediately after the path starting from A terminates. Note that the diagram indicates that the path starting from state B has not

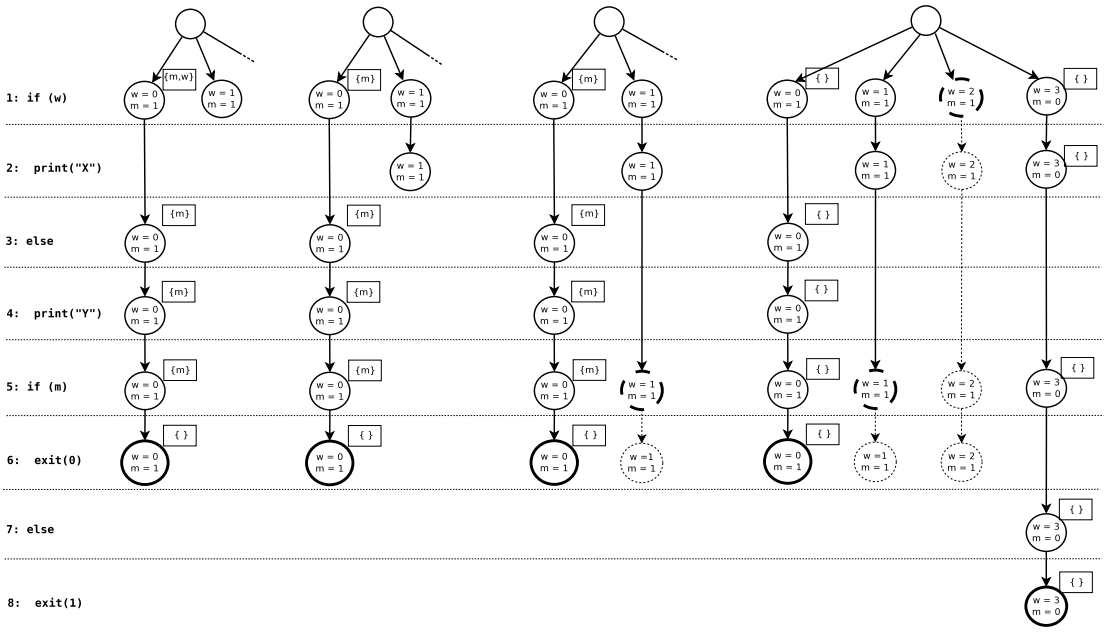

Figure 2: The state tree at four different stages of symbolic execution of the example program in Section 2. The box next to each state contains its *relevant location set*.

been explored yet since B has no successor states. At this stage, the uncovered lines are 2, 7 and 8.

As soon as the path starting from initial state A terminates, the analysis determines which variables affected the decisions of branch instructions along the path that controlled each uncovered line. We call these variables *relevant locations*. A set of relevant locations at each point along the terminated path is computed backwards starting from the termination point. For example, the set of relevant locations at line 1 along this path is {`m`,`w`} because, at this stage of symbolic execution, lines 2, 7, and 8 are uncovered, and both `m` and `w` affect the decisions of the branches that control these lines. Similarly, the relevant location set at line 5 is {`m`}, because, from line 5 onward, the only variable that affects the decision of a branch that controls an uncovered line is `m`.

After the analysis has computed a relevant location set for each of the previously executed states along this terminated path, it can now try to *match* any of the currently running states to the previously executed ones that have the same program counter. For example, our analysis tries to *match* the currently running state B to the previously executed state A. This matching is performed by looking up the relevant location set of state A, which is {`m`,`w`}, and checking whether the constraints of state A that involve `m` and `w` *imply* the constraints of state B that involve `m` and `w`. Because the constraints differ on variable `w`, the analysis determines that no match exists, and thus, it cannot eliminate B.

Next, the path starting at state B executes line 1 and moves to line 2. The second diagram from the left in Figure 2 shows the state tree at this stage of symbolic execution, which illustrates an important aspect of our analysis. Now, lines 1, 2, 3, 4, 5, and 6 are all covered, which means that the branch on line 1 no longer controls an uncovered line. Our analysis immediately detects this change and *refines* the relevant location sets along the path starting from state A by removing `w` because `w` no longer affects the decision of a branch that controls an uncovered line. Dynamically adjusting the relevant location sets as more lines become covered increases our analysis's ability to eliminate redundant states. As a result, in many cases, the analysis can eliminate all states, exhaustively exploring the state space and *soundly proving* that the remaining uncovered lines are dead code with respect to the modeled environment.

After the refinement of the relevant location sets, the analysis tries to match the currently running state on line 2 to other previously executed states along terminated paths. The analysis fails to find a match because, so far, no other states have reached line 2, so there's nothing to compare it to. Thus, the path continues executing until it reaches line 5 as illustrated in the third state tree from the left in Figure 2. Let this state be called $\sigma_{B,5}$ because it is generated on the path that started from state B and its program counter is line 5. Again, the analysis tries to find a match for $\sigma_{B,5}$ and sees that a previously executed state on the terminated path starting from A has the same program counter. Let this state be called $\sigma_{A,5}$.

Now, the analysis checks to see if $\sigma_{A,5}$'s constraint set implies $\sigma_{B,5}$'s constraint set with respect to the relevant location set associated with $\sigma_{A,5}$. In this case, a match does exist: $\sigma_{A,5}$'s constraint set is {w=0,m=1}, $\sigma_{B,5}$'s constraint set is {w=1,m=1}, and $\sigma_{A,5}$'s relevant location set is m. Note that even though their constraint sets differ on w, a match still exists because w is not relevant to covering the remaining lines 7 and 8. After finding the match, the analysis *prunes* $\sigma_{B,5}$ by eliminating it from symbolic execution and deallocating it. Figure 2 illustrates that $\sigma_{B,5}$ has been pruned by giving it a thick, dashed border.

Next, the analysis performs the same process on the paths starting from states C and D. State C is pruned immediately because it matches state A on m. State D is never pruned. The analysis tries to match states along the path starting from D to the previously executed states, but it finds that they always differ on m: in state D, m is 0 and in the other three states, it is 1. The path starting from D proceeds until it reaches the uncovered lines 7 and 8 and then terminates when it reaches the exit instruction on line 8.

The fourth state tree in Figure 2 illustrates the final state tree generated at the end of the symbolic execution. Note that because all lines are now covered, the relevant location sets for every state have been *refined* to the empty set because there are no longer any more branch instructions that control uncovered lines.

3 Redundant State Detector

This section explains how the redundant state detector dynamically monitors symbolic execution and eliminates redundant states. Algorithm 1 gives a high-level description of the different parts of the detector and how they interact with symbolic execution. The boxed lines are performed by the detector, and the non-boxed ones are performed by normal symbolic execution, which we discuss first.

Symbolic execution uses a worklist algorithm to generate and execute states until the entire state space is explored. A state consists of both a program counter and a constraint set that encodes the sequence of branch decisions made by the state thus far in terms of program memory locations. Line 2 initializes the worklist with special states in which global variables are initialized and program arguments have been assigned to arrays of "unknowns" which can take on any value. On each iteration of the worklist algorithm, line 4 selects and removes a state with constraint set C from the worklist. If the state is at a branch, lines 6 - 10 create two copies of the state and place them on the worklist if their paths are feasible. Otherwise, line 12 updates the state with the effects of the instruction specified by its program counter, which is then incremented on the following line. If the state has reached a program exit, line 21 generates concrete program inputs that drive program execution down the corresponding path, and then line 22 deletes the state. Otherwise, the state is inserted into the worklist on line 24.

```
   input: A program and a set of initial states
1  ConstructStaticControlDepGraph()
2  worklist ← InitialStates
3  while worklist ≠ ∅ do
4      state_C ← worklist.pop()
5      if state_C at branch condition B then
6          (state_B, state_¬B) ← Fork(state_C)
7          if C ∧ B is satisfiable then
8              worklist.insert(state_B)
9          if C ∧ ¬B is satisfiable then
10             worklist.insert(state_¬B);
11     else
12         Execute(state_C)
13         increment state_C's program counter
14         UpdateDynamicDepGraph(state_C)
15         if state_C.pc was previously uncovered then
16             UpdateRelBranchSet(state_C)
17             RefineRelLocSets()
18         FindMatch(state_C)
19         if state_C has match or is at program exit then
20             ConstructRelLocSets(state_C)
21             compute test inputs for state_C
22             delete state_C
23         else
24             worklist.insert(state_C)
```

Algorithm 1: High-level description of how the redundant state detector dynamically monitors symbolic execution to eliminate redundant states.

We now discuss the statements of Algorithm 1 performed by the redundant state detector. On line 1, the detector constructs a *static control dependence graph* to keep track of which static branches are *relevant* in the sense that they control a line that has not yet been covered by symbolic execution as described in Section 3.1. Line 14 updates the *dynamic dependence graph* immediately after every call to Execute with the effects of the recently executed instruction. This graph, which is described in Section 3.2.1, tracks the dynamic data dependencies between two writes as well as the dynamic control dependencies between a dynamic branch and the writes it controls.

On line 15, the algorithm checks whether the state has reached a previously uncovered instruction. If so, on line 16, the detector uses the static control dependence graph to update the current set of relevant static branches. Then, on line 17, it *refines* all relevant constraint sets constructed thus far because they may contain reads that are

no longer relevant as described in Section 3.4.

On line 18, as described in Section 3.3, the detector searches for a relevant constraint set constructed along a previously explored path that matches the state. A successful match implies that the state is redundant. As soon as a state reaches a program exit or is pruned, the detector dynamically slices the part of the dynamic dependence graph corresponding to the path followed by the state to determine which reads along the path were *relevant* in the sense that they potentially affected whether the state reached an uncovered instruction. Section 3.2 describes how the call on line 20 extracts the dynamic slice and how the slice is used to construct relevant constraint sets.

3.1 Relevant Static Branches

The most basic component of the redundant state detector is identifying which static branches of the program are *relevant*. A static branch is *relevant* if the outcome of its condition may affect whether an uncovered instruction is reachable. Consider the simplified code snippet from the Unix utility `chown` below. Suppose for now that the program never exits before line 11—we explain how we handle *embedded halts* in Section 3.2.3.

```
1  if (reference_file) {
2    if (stat (...))
3      error (...); //uncovered
4    ...;
5  } else {
6    if (parse_user_spec (...))
7      error (...);
8    ...;
9  }
10 if (chopt.recurse & preserve_root)
11   ...; //uncovered
```

Because lines 3 and 11 are uncovered, the branches `if(reference_file)` on line 1, `if(stat(...))` on line 2 and `if(chopt.recurse & preserve_root)` on line 10 are the relevant static branches since they affect whether an uncovered instruction is reachable.

The detector identifies the relevant static branches of a program by constructing a *static control dependence graph* [17] whose nodes are the static instructions of the program and whose edges (b,i) signify that branch b *statically controls* instruction i, so the outcome of b's condition affects whether i is executed. A static branch b is relevant if there exists a path in the static control dependence graph from b to some uncovered instruction i. For example, the static control dependence graph for this code snippet contains four nodes: one for each branch condition on lines 1, 2, 6, 10. The static branch `if(reference_file)` on line 1 is relevant because line 3 is uncovered, and the graph has a path from line 1 to line 3 via line 2.

Note that the set of relevant static branches gets smaller and smaller as more instructions are reached and become covered. Thus, every time a state reaches an uncovered instruction, the detector updates the set of relevant static branches as shown in Algorithm 1, line 16.

3.2 Relevant Locations

The detector determines that a state is redundant by comparing it to previously executed states. Conceptually, to perform this comparison, it records a complete history of the symbolic execution by taking a *snapshot* of each state every time it executes an instruction. The kth *snapshot* of a state is simply a copy of the state's constraint set immediately after it executes the kth instruction along its path.

If a snapshot's constraint set is equivalent to a state's constraint set, then the detector concludes that the state is redundant and eliminates it. However, this condition is unnecessarily conservative in the sense that a state may be redundant even if its constraint set is not entirely equivalent to a snapshot's. For example, if a snapshot's constraint set is a *subset* of a state's constraint set, then the detector can also conclude that the state is redundant because the state is "more constrained" than the snapshot and thus will not explore any new relevant behaviors of the program.

We say that this subset condition is more *precise* than the equivalence condition because its weaker and thus allows the detector to find more redundant states. We also say that it is *sound* because, even though it is weaker, it never concludes that a state is redundant when it is not.

This approach faces two practical challenges: how to incrementally encode every snapshot of the symbolic execution efficiently, and how to compare a state to a snapshot precisely. The detector addresses both challenges simultaneously using *dynamic slicing*, a program analysis technique that identifies which instructions along a path affect the value of a given memory location [1].

As soon as a state reaches a program exit, the detector dynamically slices the path taken by the state to identify, at every kth instruction along the path, which locations are *relevant* in the sense that they potentially affected the outcome of the decision of a relevant static branch further down the path. We refer to the relevant locations at the kth instruction along a path taken by a state as the state's kth *relevant location set*. Note that the relevant location sets for a state are not constructed until it has reached a program exit to ensure that no relevant locations are missed.

We use relevant location sets to devise an even more precise, yet sound, condition for detecting that a state is redundant: if the constraints of a snapshot that *depend on a relevant location* is a subset of the set of constraints of a state that *depend on a relevant location*, then the state is redundant. Conceptually, the constraints in constraint

set Δ that depend on a location l are those that limit the possible values that l can have. Formally, let L be a set of relevant locations. The subset $\Delta|_L$ of Δ that depends on locations in L is recursively defined as the constraints in Δ that use locations in L or locations that appear in some constraint in $\Delta|_L$. For example, the constraints in the set $\{a > b,\ b = 0,\ c = 1,\ d = 2\}$ that depend on the locations $\{a, c\}$ are $\{a > b,\ b = 0,\ c = 1\}$.

We use the following abridged code snippet from the Unix utility `chown` to demonstrate relevant location sets. The symbols `TRUE` and `FALSE` in the code below are `int` constant variables assigned the values 1 and 0, respectively. Suppose the initial state consists of three command line arguments: the program name in `argv[0]`, a three-byte array of "unknowns" in `argv[1]`, and a sixteen-byte array of "unknowns" in `argv[2]`. Further, suppose that lines 7 and 21 are the only uncovered lines.

```
1  chopt.recurse = FALSE;
2  preserve_root = FALSE;
3  ...
4  if (!strcmp(argv[1], "-R"))
5    chopt.recurse = TRUE;
6  if (!strcmp(argv[2], "--preserve-root"))
7    preserve_root = TRUE; //uncovered
8  ...
9  if (reference_file) {
10   ...
11   chopt.user_name = uid_to_name(...);
12   chopt.group_name = gid_to_name(...);
13 } else {
14   ...
15   if(!chopt.username && chopt.group_name)
16     chopt.username = "";
17   ...
18 }
19 if (chopt.recurse)
20   if (preserve_root)
21     ...; //uncovered
22 ok = chown_file(...,&chopt);
23 exit(ok);
```

The branches on lines 6, 19, 20 are the only relevant static branches because they control the only uncovered instructions as discussed in Section 3.1. Consider the state that takes the path through lines 1-6, 8-12, 19, 22, 23. As soon as it reaches the program exit on line 23, the detector constructs relevant location sets at every point along the path, which are shown in the table below.

Line	Relevant Location Set
1-3	{ `TRUE`, `argv[2][0]`, `argv[1][0..2]` }
4	{ `TRUE`, `argv[2][0]`, `argv[1][0..2]` }
5	{ `TRUE`, `argv[2][0]` }
6	{ `chopt.recurse`, `argv[2][0]` }
8-12	{ `chopt.recurse` }
19	{ `chopt.recurse` }
22	{}
23	{}

On line 23, the relevant location set is trivially empty because it is a program exit. On line 22, it is empty because no relevant static branches are reachable on the path from this point onward. Now on line 19, the detector determines that `chopt.recurse` is a relevant location at this point because it affects the decision of the relevant static branch on line 19. On lines 10-12, `chopt.recurse` is also relevant for the same reason. Note that `preserve_root` is not relevant on these lines because even though it *statically* controls an uncovered instruction, it does not *dynamically* control the uncovered instruction, since it is not read along the path taken by the state.

Now, on lines 8 and 9, the relevant location set is still {`chopt.recurse`} and does not include `reference_file`. Even though `reference_file` is read by a branch condition along the path, it does not affect the decision of a *relevant* branch. The power of the detector is demonstrated here: it is capable of reasoning that the entire `if-else` block on lines 9-18 does not affect whether the uncovered instruction on line 21 is reachable and can thus be ignored, keeping the number of relevant locations small, which is valuable when comparing a snapshot to a state: the fewer of a snapshot's constraints that need to appear in the state's constraints, the greater the chance of soundly concluding that the state is redundant.

On line 6, `argv[2][0]` is included in the relevant location set because this location is used by the condition of the relevant static branch on line 6 via the call to `strcmp`.

Now, on line 5, `chopt.recurse` is replaced by the constant `TRUE` in the relevant location set. Conceptually, the location `chopt.recurse` is not relevant at this point because the write on line 5 which affects the decision of the relevant static branch on line 19 has not occurred on the path yet. The location `TRUE`, however, is now relevant because it is used by the assignment to compute a value that is written to a relevant location. In Section 3.5 we describe a way to handle locations that have the same value across all paths, such as `TRUE`, in a special way that reduces the overhead of the detector.

Finally, on line 4, `argv[1][0..2]` is included in the relevant location set because it is used by the condition of the branch on line 4 that dynamically controls the write of the relevant location `chopt.recurse`. In the remainder of this section, we describe how the relevant location sets are constructed in the general case. First we describe how the *dynamic dependence graph* is constructed in Section 3.2.1 and then, in Section 3.2.2, how relevant locations are inferred by slicing the graph with respect to relevant static branches.

3.2.1 Dynamic Dependence Graph

Throughout symbolic execution, the detector records the dependencies between executed instructions along each path by incrementally updating the *dynamic dependence graph* whose nodes are byte-level writes and whose edges are either *data*, *control*, or *potential* dependencies [1, 2]. A write w_2 performed by instruction i is *data-dependent* on another write w_1 if i reads w_1. A write w_2 is *control-dependent* on a write w_1 if a branch that reads w_1 dynamically controls w_2. A write w_2 is *potentially-dependent* on a write w_1 if a branch that reads w_1 statically controls assignment $e_2 := e$ not along the path and e_2 aliases the static location of w_2.

Inferring data and control dependencies is straightforward because it requires reasoning about parts of the executed path only, and not other parts of the program. In contrast, inferring potential dependencies is challenging because it requires reasoning about both the executed path as well as static locations on non-executed paths. Consequently, potential dependencies require a sound, interprocedural aliasing analysis. Our implementation uses a sound, highly precise intraprocedural pointer analysis to resolve non-escaping aliases and a sound, scalable, yet coarse interprocedural alias analysis to resolve escaping aliases [13]. While state-of-the-art, sound aliasing analysis is notoriously imprecise, in Section 3.6, we describe a technique that allows the detector to recover from some of the precision loss. Note that this imprecision only affects the detector's ability to identify redundant states; it does not affect the completeness of the symbolic execution and does not cause it to report false positives.

3.2.2 Dynamic Slicing

This section describes how, in general, the dynamic dependence graph is sliced with respect to relevant static branches in order to infer relevant location sets. One slight complication, however, is that symbolic execution paths are not linear, but rather form trees, because a state may fork into several states. Thus, the detector cannot construct the kth relevant location set until *all* states generated by forks *after* the kth instruction have reached a program exit.

When a state reaches a program exit, the detector starts from the end of the path taken by the state and constructs the relevant location set L_k at the kth instruction along the path. Let C_k be the *relevant control set* used as an intermediate set for constructing relevant location sets that consist of the dynamic branches that control the kth instruction. The sets L_k and C_k are constructed as follows:

1. Compute $L_k^{\cup}$, which is the union of the relevant location sets at the immediate successor instructions of the kth instruction. Recall that an instruction along a path may have multiple immediate successor instructions if symbolic execution forked at that point.
2. Compute $C_k^{\cup}$, which is the union of the relevant control sets at the immediate successor instructions of the kth instruction.
3. If the kth instruction is a program exit, both L_k and C_k are empty.
4. If the kth instruction is a dynamic branch b, then add to C_k all of $C_k^{\cup} - \{b\}$, and also add to C_k the dynamic branch that controls b. If at least one of the following three situations hold, then add to L_k the locations used in the branch condition:
   (a) b corresponds to a relevant static branch,
   (b) b is in $C_k^{\cup}$,
   (c) some location in $L_k^{\cup}$ potentially depends on b.
5. If the kth instruction is a dynamic assignment $e_2 := e_1$ and the location of e_2 is in $L_k^{\cup}$, then add to C_k the dynamic branch that controls the assignment, and add to L_k all the locations read by the expression e_1.

3.2.3 Irregular Control Dependence

The explanation of the detector thus far is not sound for programs that have irregular control flow introduced by *embedded halts* [17] which are instructions that terminate execution of the program, such as calls to `exit` and `abort`, that are distinct from the normal termination point. An embedded halt induces interprocedural control dependence from the branch that controls the halt to *any* statement statically reachable from that branch. These interprocedural control dependencies must be handled by the detector because, otherwise, it may not recognize a relevant location as relevant. By expanding the third step of the dynamic slicing algorithm in Section 3.2.2 as follows, the detector will soundly handle irregular control flow:

3. If the kth instruction is a program exit, then L_k is empty and C_k is the set of dynamic branches that control it.

3.2.4 Inter-Path Dynamic Slicing

So far in this paper, relevant location sets have been constructed along a path only when the state reaches a program exit. Because the detector eventually prunes almost every state before the state can reach a program exit, very few paths have relevant location sets constructed along them. Consequently, the precision of the detector is severely limited: in general, the more relevant location sets that are constructed, the more opportunities the detector has to find a match for state and thus to prune it.

We add an additional step to the dynamic slicing algorithm in Section 3.2.2 to give the detector the ability to construct relevant location sets along paths taken by states that are pruned before they reach a program exit:

6. If the kth instruction is the last instruction along the path of the state that was matched and pruned by the relevant location set L'_j, then add to L_k all of L'_j and add to C_k all the dynamic branches that control the kth instruction.

The effect of this additional step allows the dynamic slicing algorithm to incorporate the slicing result of previously explored paths to construct relevant location sets along new paths of pruned states.

3.3 State Matching

A state is redundant if it *matches* any snapshot, which occurs when the following conditions hold. Let L be the relevant location set that corresponds to the snapshot.

1. The state and the snapshot are at the same *context-sensitive static instruction*. A state's *context-sensitive static instruction* is a pair consisting of (1) the state's program counter and (2) the sequence of call instructions on its call stack.
2. The snapshot constraints that depend on L are a subset of the state constraints that depend on L.

Recall from Section 3.2 that the constraints of a constraint set that depend on a location l are those that limit the possible values that l can have in a satisfying assignment.

Throughout Section 3, we use the concept of snapshots to explain how the detector works. However, our implementation never explicitly constructs these snapshots because the time and space overhead of allocating millions of constraint sets, each of which contains tens of thousands of constraints is prohibitively expensive. Instead, the detector exploits the fact that the matching conditions above only need the constraints of a snapshot that *depend on relevant locations*, called the *relevant constraints* of the snapshot. The set of relevant constraints of a snapshot is 100X smaller than its entire set of constraints.

3.4 Dynamic Refinement

In this section we describe a technique that increases the detector's precision as symbolic execution covers more and more of the program. Recall from Section 3.1 that relevant static branches are those that control uncovered instructions, and a location is relevant if it affects the decision of a relevant static branch. Thus, as symbolic execution covers more uncovered instructions, the fewer the relevant static branches, and the fewer the relevant locations, and thus the more chances the detector has to find a match for a state.

As soon as symbolic execution reaches an uncovered instruction, the detector iterates over all the relevant location sets and removes any locations that were included because that line was previously uncovered.

This dynamic refinement of relevant location sets substantially improves the precision of the detector, making it possible to exhaustively explore the entire state space on some benchmarks, *proving* that any remaining uncovered instructions are dead code with respect to the environment without any false positives or false negatives.

3.5 Single-Valued Locations

In this section, we describe an optimization that reduces the overhead of the detector by exploiting the fact that, in practice, more than half of the locations that appear in relevant constraint sets are *single-valued*, meaning they have the same value written to them along every path explored thus far by symbolic execution. Note that a single-valued location is not necessarily constant; it may be that the symbolic execution will eventually explore a path that writes a different value to the location. Thus, a location that is *single-valued* for the first few minutes of symbolic execution may not be single-valued thereafter. These locations are prevalent in programs that extensively use libraries because the majority of locations are typically initialized to a single value and then mostly read and rarely overwritten with a different one.

The detector exploits this observation by removing single-valued locations from relevant constraint sets, thus reducing the sizes of the sets, and consequently, reducing overhead substantially. The optimization is sound and does not introduce a loss in precision.

One difficulty with implementing this optimization is that a location l may be single-valued for the first n symbolically executed instructions, but on the $n+1$ instruction, a different value may be written to it. At this point, the detector may become unsound and prune a non-redundant state because a relevant constraint set constructed before this point may have previously removed l. To ensure that soundness is maintained, as soon as l is written-to with a different value, the detector identifies which relevant constraint sets previously removed l and re-adds l to them before executing the $n+2$ instruction. This refinement step requires re-slicing the paths along which l was removed.

3.6 Relevant Search Heuristic

One source of imprecision in the detector is the use of a sound, coarse, interprocedural alias analysis to infer po-

tential dependencies as described in Section 3.2.1. Consequently, locations that are not actually relevant may be inferred as such. To recover from this loss of precision, the detector keeps *two* relevant location sets at each instruction along a path: one relevant location set is constructed *soundly* by slicing through data, control and potential dependencies in the dynamic dependence graph, and the other is constructed *unsoundly* by only slicing through data and control dependencies.

Throughout the entire symbolic execution, the detector uses only sound relevant location set to prune states. However, simultaneously, the detector incorporates the unsound relevant location sets into its search strategy by giving priority to states that do *not* match any unsound relevant location set. This technique substantially decreases the time it takes to reach the maximum coverage of a program. Intuitively, it is effective because it gives preference to states that are the most "different", in a relevant way, from any other previously explored state, and thus are more likely to reach an uncovered instruction.

3.7 Efficient State Matching

As discussed in Section 3.3, after each executed instruction, the detector searches for a match of a state by comparing it to all the relevant constraint sets at the same context-sensitive static instruction. A straightforward implementation will perform this search by comparing the state to each relevant constraint set individually.

Unfortunately, even programs with fewer than ten thousand lines of code will have thousands of unique relevant constraint sets at each program point, and each set may contain hundreds of constraints. Thus, comparing a state to each relevant constraint set individually after each executed instruction is prohibitively expensive. To address this challenge, the detector constructs, at each context-sensitive static instruction *pc*, a *decision tree* [16] that organizes relevant constraint sets at *pc* in a manner that makes searching for a match efficient. A decision tree is a highly effective data structure for this search problem because it can exploit the fact that the relevant constraint sets at a program point share many common locations.

Consider the following seven relevant constraint sets at context-sensitive static instruction *pc*.

1. { `x = 2, y = 4, u = 10` }
2. { `x = 2, y = 5, u = 10` }
3. { `x = 1, u = 10` }
4. { `x = 3, u = 11` }
5. { `x = 2, u = 11` }
6. { `x = 2, u = 10, w = 6` }
7. { `x = 2, u = 10, w = 7` }

Suppose that the current state is also at *pc* and has constraints {`x = 2, y = 4, z = 8, u = 11`}. If the state is compared to each relevant constraint set individually, the detector would need to perform eighteen lookups in the state's constraint set, one for each relevant constraint. However, by organizing these relevant constraints sets into a decision tree, only four lookups are required.

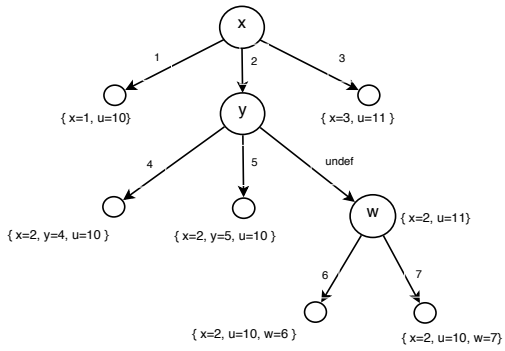

Figure 3: The decision tree for the seven relevant constraint sets in Section 3.7.

Figure 3 shows the decision tree for the seven relevant constraint sets above. Each of the tree's non-leaf nodes is labeled with a location l, and its outgoing edges are labeled with the possible values associated with l in the relevant constraint sets. If l does not appear in every relevant constraint set, then it may have an outgoing edge labeled `undef`. Each relevant constraint set is associated with exactly one of the nodes. In the figure, the node for location `x` has three outgoing edges, one for each of the possible values that `x` is constrained to. The node for location `y` has an outgoing edge labeled `undef` because it is not constrained in some relevant constraint sets.

The decision tree is used to search for a relevant constraint set that matches the state. The search process starts at root node `x`, so `x` is looked up in the state and is found to have the value 2. Thus, the search process moves to the child whose incoming edge has the value 2, which is node `y`. Then, `y` is looked up in the state and is found to have the value 4. Thus, the search process moves to the child whose incoming edge has the value 4 and contains the relevant constraint set {`x = 2, y = 4, u = 10`}. Now, the search process checks if this relevant constraint set matches the state entirely. The locations `x`, `y` and `u` are looked up in the state to see if they have the same values in the relevant constraint set, that is, 2 and 11, respectively. They do not match on the location u, so the state does not match this relevant constraint set. Because the current node is a leaf, the search process ends, which means that the state does not match any of these relevant constraint sets. Thus, by using a decision tree for the search process, only four lookups were necessary in this example, compared to eighteen lookups if the relevant constraint sets were compared individually.

In general, the search process starts at the root node of

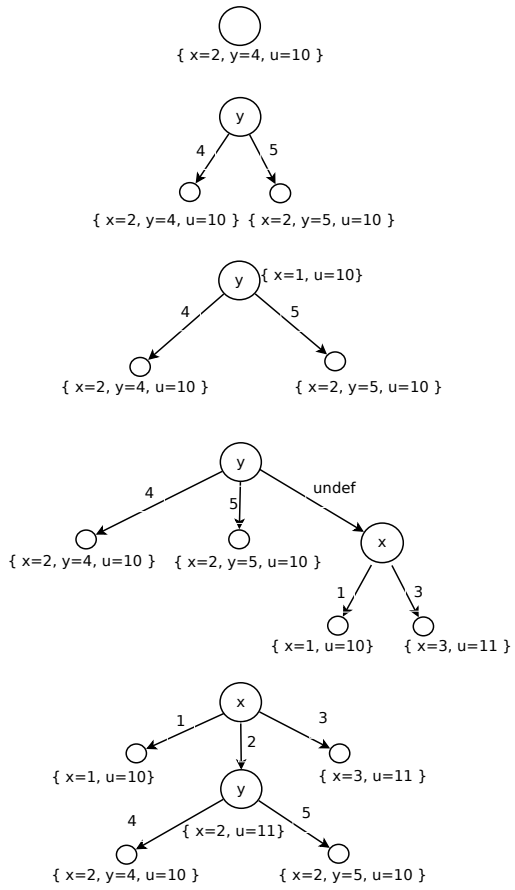

Figure 4: The different stages of the decision tree in Figure 3 as each of the first five relevant constraint sets in Section 3.7 is incorporated.

the decision tree and performs the following steps until either a match or a conflict is found. Let t be a non-leaf node representing a location l. If l is defined by the state, let v be its value. There are three possibilities:

1. If l is defined in the state and if t has an outgoing edge labeled v to a node s, then the search proceeds to s.

2. If l is defined in the state and if t has no outgoing edge labeled with v, then a conflict is found and the search process at this node ends without finding a match.

3. If l is undefined in the state, the search proceeds to each child node of t individually.

Construction. Even for programs with fewer than ten thousand lines of code, thousands of new, relevant constraint sets are generated during each minute of symbolic execution. Thus, the construction of these decision trees must be *incremental*: every time a relevant constraint set is generated, it must be efficiently incorporated into the already existing decision tree.

Figure 4 shows how the decision tree in Figure 3 is constructed incrementally as each of the first five relevant constraint sets above are added. Initially, the decision tree contains a single, empty root node. The first set { `x = 2, y = 4, u = 10` } is simply added to the root node. When the second relevant constraint set { `x = 2, y = 5, u = 10` } is added, the root node is labeled with the location that has most number of distinct values, which is `y` because it takes on the two values, 4 and 5, whereas locations `x` and `u` each only take on a single value. Then, two child nodes are added, one for each distinct value of `y`. The relevant constraint set { `x = 2, y = 4, u = 10` } is placed at the child node for value 4 and { `x = 2, y = 5, u = 10` } at the child node for value 5.

Conceptually, `y` is chosen because it splits the relevant constraint sets into as many partitions as possible, thus minimizing the number of lookups needed to find matching relevant constraint sets. If either `x` or `u` was chosen, no partitioning would have been possible. In general, the split heuristic used by the decision trees to minimize the number of lookups is to select locations that maximize the number of partitions at each level of the tree.

Next, the third set { `x = 1, u = 10` } is added to the root node because it does not have a constraint for `y`, and it is the only such set. Once the fourth set { `x = 3, u = 11` } is added, a new child node labeled `x` is created whose edge from the root node is labeled `undef`. Both the third and fourth sets are placed at separate child nodes of this `x` node.

Now, once the fifth set { `x = 2, u = 11` } is added, the location `x` takes on three distinct values whereas `y` only takes on two distinct values. Consequently, the split heuristic is violated at the first level of the tree. As a result, the tree is rebuilt from scratch to have location `x` at the root node and a node labeled `y` as one of its children.

We use two additional optimizations to further reduce construction overhead:

1. Relevant constraint sets are added to a decision tree *lazily*. Instead of incorporating the constraint set immediately after it is generated, the detector waits until a state actually needs to match itself against the corresponding decision tree.

2. Only the specific subtree of the decision tree that violates the split heuristic is rebuilt from scratch. The remaining portion of the tree remains intact.

4 Evaluation

We implemented our state space reduction analysis by augmenting a copy of the open source version of

KLEE [5], a symbolic virtual machine capable of automatically generating test inputs for complex programs such as device drivers, network drivers, and utility programs written in C. For the remainder of this paper, we refer to this unmodified open source version of KLEE as KLEE-BASE, and we refer to our augmentation of KLEE with our state space reduction analysis as KLEE-REDUCE. In this section, we discuss the details of our implementation that enabled us to achieve the orders of magnitude improvement in performance.

KLEE-BASE performs symbolic execution over LLVM bytecode instructions starting from several initial states, each with different numbers and sizes of symbolic objects representing the program's arguments. The initial states were generated using the following KLEE flags: `--sym-args 0 1 10 --sym-args 0 2 2 --sym-files 1 8 --sym-stdout`. We configured KLEE-BASE to use KLEE's best, built-in heuristic search strategy for line coverage, which is specified using the flags: `--use-random-path --use-interleaved-covnew-NURS --use-batching-search --batch-instructions=10000`.

This section measures how KLEE-REDUCE performs compared to KLEE-BASE on three metrics: (1) how much faster it reaches the same statement coverage, (2) how much more statement coverage it gets, and (3) how many more programs have their state spaces exhausted.

Our benchmarks consist of 66 programs from the GNU Coreutils utility suite, a diverse set of system-intensive, complicated programs that form the core user-level environment installed on millions of computer systems. They are the same benchmarks used in the original KLEE paper [5] and are among the largest and most complex benchmarks that constraint-based automatic input generation has been shown to run on for code coverage. The reader is referred to Figure 4 in [5] for the distribution of program sizes.

We view Coreutils as a fair test for KLEE-BASE since it was used in the original paper [5]. For similar reasons, our experiments follow the original paper's methodology: each program was checked for one hour each with the same flags and with the same number (and sizes) of symbolic inputs.

Our experiments have two changes from the originals, which we do not expect to have substantive impact. First, the open source version of KLEE eliminated some flags used by the original system, so we obviously could not use them. Second, we only checked 66 of the 89 possible utilities since the others either had errors when run on our 64-bit machine (the original KLEE-BASE results were on 32-bit) or were so small that they reached the maximum possible coverage in under ten seconds. The sizes of these 66 utilities varied from 6.6 K to 29 K instructions of *optimized* LLVM bytecode (using the `--optimize` KLEE flag) with a median size of 12.6 K. All together, they sum to 905 K instructions.

Speedups. We calculated speedup as follows:

1. Ran each program for one hour with KLEE-BASE and one hour with KLEE-REDUCE .
2. Recorded the maximum coverage C_{max} that both KLEE-BASE and KLEE-REDUCE were able to reach.
3. Recorded the times T_{base} and T_{red} at which KLEE-BASE and KLEE-REDUCE reached C_{max}, respectively.
4. Calculated the speedup as T_{base}/T_{red}.

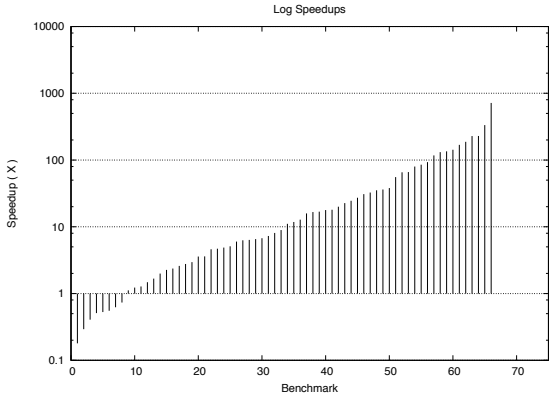

Figure 5: Log-scale speedups of how many times faster KLEE-REDUCE reaches the same statement coverage as KLEE-BASE.

Figure 5 uses a log scale to show the relative speedup, sorted from least to most. A bar below 1.0 means KLEE-REDUCE ran slower than KLEE-BASE and a bar above 1.0 means that it ran faster. As can be seen from the results, KLEE-REDUCE gives enormous speedups on the vast majority of benchmarks. KLEE-REDUCE 's average speedup is 50.5 X , that is, 50.5 times faster than KLEE-BASE. Its median speedup is 10 X , the maximum is 717 X and its maximum slow down is 0.2 X . And, 54 out of 66 (82 %) benchmarks had speedups greater than 1 X.

Table 1 shows the impact of KLEE-REDUCE on several metrics. Table 2 shows individual results from running KLEE-BASE and KLEE-REDUCE on the ten largest coreutils benchmarks.

Coverage. On 55 of the 66 (83 %) benchmarks, KLEE-REDUCE reaches at least the same coverage

Statistic	BASE	REDUCE	Reduction
Instructions	1.8 billion	222 million	8 X
Paths	5,285,596	391,057	14 X
Queries	266,286	77,614	3.4 X
Query Constructs	24.3 million	9.8 million	2.5 X
Solver Time	20.6 hours	4.1 hours	5 X
Solver Overhead	73 %	37 %	1.9 X

Table 1: Impact of KLEE-REDUCE on the number of instructions executed, paths generated, queries made, solver time, and solver overhead across all benchmarks.

	KLEE-BASE			KLEE-REDUCE				
	T	cov	T_c	T	cov	T_c	X	cov_{inc}
join	3,587	83.1	3,587	3,189	87.5	1,002	3.6	4.4
csplit	1,686	70.4	1,686	3,163	77.8	1,322	1.3	7.4
stty	2,340	58.2	2,340	694	86.5	66	35.2	28.3
dd	3,353	40.3	3,353	1,999	44.5	137	24.5	4.2
tail	3,230	70.1	3,230	2,305	76.8	143	22.7	6.7
od	3,599	74.7	3,599	1,250	84.9	200	18	10.2
tr	2,839	60.6	2,839	1,164	64	962	3	3.4
ptx	1,541	18.5	1,541	3,461	40.6	18	85.1	22.1
pr	234	57.2	29	54	39.4	54	0.5	-17.8
ls	298	34	92	82	29.5	82	1.1	-4.5

Table 2: Individual results from running KLEE-BASE and KLEE-REDUCE on the ten largest coreutils benchmarks. T is time to reach the maximum coverage, cov is maximum coverage, T_c is time to reach the same coverage, X is relative speedup, and cov_{inc} is coverage increase.

as KLEE-BASE, and on 30 (45 %) KLEE-REDUCE reaches higher coverage. The average coverage increase is 3.8 % with a median of 2.5 % on the 54 benchmarks on which either KLEE-BASE or KLEE-REDUCE did not reach the maximum coverage possible given the modeled environment. Figure 6 shows the coverage increases for each benchmark.

Exhaustion. Our analysis eliminates states to the extent that on 12 out of 66 benchmarks (18 %), KLEE-REDUCE explores every state in the state space exhaustively, *proving* that any remaining uncovered lines are dead code with respect to the modeled environment. KLEE-BASE , on the other hand, was not able to explore the state space exhaustively on any of these benchmarks.

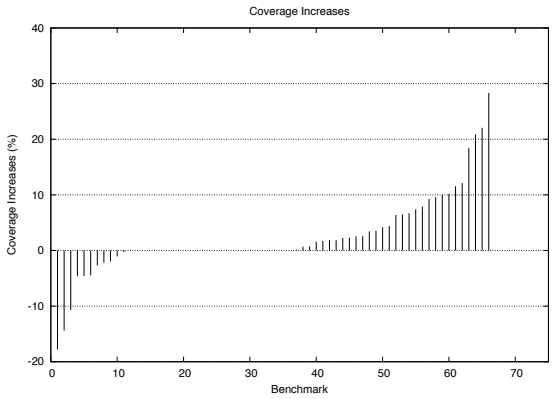

Figure 6: Coverage increases

Challenges. Now, we discuss the 14 out of 66 (21 %) benchmarks that were either slower than KLEE-BASE or had lower coverage. On eleven of these 14 benchmarks, KLEE-REDUCE did not reach the maximum coverage of KLEE-BASE, and on the remaining three, it reached the maximum coverage but was slower than KLEE-BASE.

On seven of these 14 benchmarks, KLEE-REDUCE did not outperform KLEE-BASE because these particular benchmarks had especially high solver overhead, as they spent more than 90% of their time solving queries. Thus, there was less opportunity for KLEE-REDUCE to improve performance. Furthermore, KLEE-REDUCE is biased, by design, to explore deeper parts of the program which causes it to encounter harder queries.

On the other seven of these 14 benchmarks, KLEE-REDUCE did not outperform KLEE-BASE because it does not reason precisely enough about constraints that are inconsistent in general, yet, with respect to reaching uncovered lines of code, they are equivalent. For example, consider the following common code pattern that uses the libc `read` function, which returns either -1 on error, 0 on reaching the end-of-file, or a positive value denoting the number of bytes read into `buf`.

```
1  while (1) {
2    bytes_read = read(fd, buf, sizeof buf);
3    if (bytes_read <= 0)
4      break; //uncovered
5    ...
6  }
```

Suppose a state reaches line 3 with the constraint `bytes_read = 5`, and the detector tries to match it against a relevant constraint set with `bytes_read = 7`. The detector will conclude that no match exists because it will see that these two constraints have different values for `bytes_read`. However, with respect to reaching the uncovered line, they essentially match and thus the state should be pruned.

5 Related Work

The closest antecedent to our work is Boonstoppel et al. [3]. They detect redundant states by comparing a state's *live* constraints against those of previous states that have reached the same context-sensitive program point, where a constraint is *live* if it involves a location that is read anywhere along the path taken by the previous state. They require that a depth-first search strategy is used for exploring states, which severely limits its ability to achieve high coverage. In contrast, our analysis works with any search strategy and only compares

constraints that are specifically *relevant* to uncovered instructions, thus making it significantly more precise. Inferring relevant locations is substantially more challenging because it requires slicing the dynamic data, control, and potential dependencies between locations (§ 3.2.2) and handling irregular control flow (§ 3.2.3). Furthermore, in our paper, we introduce the novel techniques of inter-path slicing (§ 3.2.4), dynamic refinement (§ 3.4), single-valued locations (§ 3.5) and using unsound relevant location sets to enhance search strategies (§ 3.6).

It is difficult to directly compare the two approaches. This previous system was built on EXE [6], which typically handles three to five orders of magnitude fewer states than KLEE, in large part because EXE created a new kernel process using (`fork`) at each branch point (e.g., each if-statement with two feasible branches). As a result, they did not solve many of the non-trivial engineering challenges we had to handle in order to successfully beat a much faster base system (KLEE). As a crude method to ignore this important engineering aspect and just compare the additional benefit of only the new refinement and propagation ideas in our approach, we disabled all techniques we added that were not in the original paper, and re-ran our system over the same 66 benchmarks in Section 4. On more than 90% of these benchmarks, the previous system either did not reach the same coverage as KLEE-BASE or was slower than it. On average, it had a coverage *decrease* of -15.5%.

In [15], the authors use dynamic slicing as part of a DART-based path exploration technique that helps avoid exploring redundant paths. They evaluate their approach on five very simple benchmarks. On the two smallest benchmarks each of which is less than 120 lines of code, they show that their approach saves a total of 60 seconds. On the other three benchmarks each of which is less than 260 lines, their approach either does not reach the same coverage as the base system or shows no improvement. We handle much larger programs, a much larger variety of them, and get much larger speedups.

Finally, there are a set of interesting state space reduction techniques for dynamic symbolic execution that improve scalability that are complementary to our work. Collingbourne et al. [8] use phi-node folding to replace control-flow forking with predicated select instructions in order to reduce the number of paths explored by symbolic execution. Kuznetsov et al. [14] propose a technique to merge states obtained on different paths to reduce the state space that a dynamic symbolic execution system needs to explore. The challenge they tackle is that merging states introduces disjunctions into the path condition and increases its complexity thereby stressing the underlying constraint solver. They demonstrate that their technique allows a symbolic execution system to explore substantially more paths. Godefroid et al. [9, 11] propose constructing function summaries for dynamic symbolic execution represented as input-output constraints. We believe using our techniques would allow this prior work to achieve even greater improvements (and vice versa).

6 Acknowledgments

We would like to thank Cristian Cadar, David Ramos, Philip Guo, and the anonymous reviewers for their insightful comments and feedback.

References

[1] AGRAWAL, H., AND HORGAN, J. R. Dynamic Program Slicing. In *PLDI* (1990).

[2] AGRAWAL, H., HORGAN, J. R., KRAUSER, E. W., AND LONDON, S. Incremental Regression Testing. In *CSM* (1993).

[3] BOONSTOPPEL, P., CADAR, C., AND ENGLER, D. RWset: Attacking Path Explosion in Constraint-based Test Generation. In *TACAS* (2008).

[4] BURNIM, J., AND SEN, K. Heuristics for Scalable Dynamic Test Generation. In *ASE* (2008).

[5] CADAR, C., DUNBAR, D., AND ENGLER, D. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *OSDI* (2008).

[6] CADAR, C., GANESH, V., PAWLOWSKI, P. M., DILL, D. L., AND ENGLER, D. R. EXE: Automatically Generating Inputs of Death. In *CCS* (2006).

[7] CADAR, C., GODEFROID, P., KHURSHID, S., PĂSĂREANU, C. S., SEN, K., TILLMANN, N., AND VISSER, W. Symbolic Execution for Software Testing in Practice: A Preliminary Assessment. In *ICSE* (2011).

[8] COLLINGBOURNE, P., CADAR, C., AND KELLY, P. H. Symbolic Crosschecking of Floating-point and SIMD Code. In *Eurosys* (2011).

[9] GODEFROID, P. Compositional Dynamic Test Generation. In *POPL* (2007).

[10] GODEFROID, P., KLARLUND, N., AND SEN, K. DART: Directed Automated Random Testing. In *PLDI* (2005), V. Sarkar and M. W. Hall, Eds.

[11] GODEFROID, P., NORI, A. V., RAJAMANI, S. K., AND TETALI, S. D. Compositional May-Must Program Analysis: Unleashing the Power of Alternation. In *POPL* (2010).

[12] GROCE, A., AND VISSER, W. Heuristic Model Checking for Java Programs. In *STTT* (2004).

[13] HACKETT, B., AND AIKEN, A. How Is Aliasing Used in Systems Software? In *FSE* (2006).

[14] KUZNETSOV, V., KINDER, J., BUCUR, S., AND CANDEA, G. Efficient State Merging in Symbolic Execution. In *PLDI* (2012).

[15] QI, D., NGUYEN, H. D., AND ROYCHOUDHURY, A. Path Exploration Based on Symbolic Output. In *FSE* (2011).

[16] QUINLAN, J. R. Induction of Decision Trees. *Machine Learning 1*, 1 (Mar. 1986).

[17] SINHA, S., HARROLD, M. J., AND ROTHERMEL, G. Computation of Interprocedural Control Dependence. In *ISSTA* (1998).