RWset: Attacking Path Explosion in Constraint-Based Test Generation

Peter Boonstoppel, Cristian Cadar, Dawson Engler

Computer Systems Laboratory, Stanford University

Abstract. Recent work has used variations of symbolic execution to automatically generate high-coverage test inputs [3, 4, 7, 8, 14]. Such tools have demonstrated their ability to find very subtle errors. However, one challenge they all face is how to effectively handle the exponential number of paths in checked code. This paper presents a new technique for reducing the number of traversed code paths by discarding those that must have side-effects identical to some previously explored path. Our results on a mix of open source applications and device drivers show that this (sound) optimization reduces the numbers of paths traversed by several orders of magnitude, often achieving program coverage far out of reach for a standard constraint-based execution system.

1 Introduction

Software testing is well-recognized as both a crucial part of software development and, because of the weakness of current testing techniques, a perennial problem as well. Manual testing is labor intensive and its results often closer to embarrassing than impressive. Random testing is easily applied, but also often gets poor coverage. Even a single equality conditional can derail it: satisfying a 32-bit equality in a branch condition requires correctly guessing one value out of four billion possibilities. Correctly getting a sequence of such conditions is hopeless. Recent work has attacked these problems using *constraint-based execution* (a variant of *symbolic execution*) to automatically generate high-coverage test inputs [3, 4, 7, 8, 14].

At a high-level, these tools use variations on the following idea. Instead of running code on manually or randomly constructed input, they run it on symbolic input that is initially allowed to take any value. They substitute program variables with symbolic values and replace concrete program operations with ones that manipulate symbolic values. When program execution branches based on a symbolic value, the system (conceptually) follows both branches at once, maintaining a set of constraints called the *path constraint* which must hold on execution of that path. When a path terminates or hits a bug, a test case can be generated by solving the current path constraint to obtain concrete input values. Assuming deterministic code, feeding this concrete input to an uninstrumented version of the checked code will cause it to follow the same path and hit the same bug.

A significant scalability challenge for these tools is how to handle the exponential number of paths in the code. Recent work has tried to address this scalability challenge in a variety of ways: by using heuristics to guide path exploration [4]; caching function summaries for later use by higher-level functions [7]; or combining symbolic execution with random testing [13].

This paper presents a largely complementary technique that prunes redundant paths by tracking the memory locations read and written by the checked code, in order to determine when the remainder of a particular execution is capable of exploring new behaviors. This technique, which we call *read-write set* (RWset) analysis, dramatically reduces the number of paths explored by discarding those that will produce the same effects as some previously explored path.

RWset analysis employs two main ideas. First, an execution that reaches a program point in the same state as some previous execution will produce the same subsequent effects and can be pruned. Second, this idea can be greatly amplified by exploiting the fact that two states that only differ in program values that are not subsequently read will produce the same subsequent effects and can be treated as being identical. Consequently, the second execution is redundant and can also be pruned.

We measure the effectiveness of our RWset implementation by applying it to server, library, and device driver code. Our results show that RWset analysis is effective in discarding redundant paths, often reducing the number of paths traversed by several orders of magnitude and achieving coverage out-of-reach for the base version of the system, which easily gets stuck continuously revisiting provably redundant states.

The paper is structured as follows. Section 2 gives an overview of RWset analysis using several small examples, while Section 3 discusses the implementation more thoroughly. Section 4 measures the efficiency of our implementation. Finally, Section 5 discusses related work and Section 6 concludes.

2 Overview

This section gives a general overview of RWset analysis. To make the paper self-contained, we first briefly describe how constraint-based execution works in the context of our tool, EXE. (For the purpose of this paper, one can view other constraint-based execution tools as roughly equivalent; RWset analysis can be implemented in any of them.) We subsequently describe the main idea behind RWset analysis and then discuss some of the refinements employed to maximize the number of redundant paths detected.

2.1 Constraint-based execution

EXE lets users explicitly mark which memory locations should be treated as holding *symbolic data*, whose values are initially entirely unconstrained. EXE instruments the user program so that it can track these symbolic values. When

	Path Constraints	
	`Path 1`	`Path 2`
`1: x = read_sym_input();`	$\{x = *\}$	
`2: if(x == 1234)`	$\swarrow$ *fork*	$\searrow$
`3:   printf("foo");`	$\{x = 1234\}$	
`4: else printf("bar");`		$\{x \neq 1234\}$
`5: ...`	$\{x = 1234\}$	$\{x \neq 1234\}$

Fig. 1: Contrived example to illustrate constraint-based execution. The code has two paths, both of which will be followed. The first path (lines 1,2,3,5) ends with the constraint that $x = 1234$. The second (lines 1,2,3,4) with the constraint that $x \neq 1234$.

the program runs, at each statement EXE checks if all inputs to that statement have exactly one value, i.e. they are *concrete* rather than symbolic. In such cases, the statement executes exactly as it would in the uninstrumented code. Otherwise, EXE adds the effects of the statement as a constraint to the current path. For example, given the statement $i = x + y$, if x and y have the values $x = 4$ and $y = 5$, EXE executes the statement and assigns the 9 to i. If not, it adds the path constraint that $i = x + y$.

When execution reaches a symbolic branch condition, EXE uses the STP constraint solver [5] to check if the current path constraints make either branch direction infeasible, and, if so, follows the other. If it cannot prove that one direction is infeasible, EXE (conceptually) forks execution and follows both paths, adding the appropriate branch constraint on each path.

To illustrate these points, consider the contrived code in Figure 1, where the call to `read_sym_input()` marks `x` as an unconstrained symbolic value. This code has two feasible paths, both of which EXE will explore, and generates two concrete test cases to exercise each path. The steps followed by EXE are as follows:

Line 1: EXE adds the path constraint $x = *$, i.e. x is unconstrained.

Line 2: Since both branches are possible, EXE forks execution: on the true path it adds the constraint $x = 1234$ and on the false path the constraint $x \neq 1234$.

Line 3, 5: Assume that EXE follows the true path first. When it terminates or hits an error, EXE solves the path constraints for concrete values. In this case it will generate $x = 1234$. If the code is deterministic, rerunning the program on this value will cause the same path (lines 1,2,3,5,...) to execute.

Line 4, 5: Similarly, the false path is followed and generates more test cases.

In order to handle real code, EXE tracks all constraints with bit-level accuracy. EXE supports pointers, arrays, unions, bit-fields, casts, and aggressive bit-operations such as shifting, masking, and byte swapping. The interested reader can refer to [4] for details.

2.2 Scalability challenge: discarding redundant paths

While constraint-based execution can automatically explore program paths, the number of distinct paths increases exponentially with the number of conditional statements traversed. In all but the smallest programs, this typically leads to an essentially inexhaustible set of paths to explore. However, not all paths are equal; very often multiple paths produce the same *effects*, and there is no reason to explore more than one such path. The effects of an execution path can be defined in any way desired, depending on the needs of the testing process. One common definition, which we also use in this paper, defines the effects of an execution path to be the basic blocks it executes.

The basic idea behind RWset analysis is to truncate exploration of a path as soon as we can determine that its continued execution will produce effects that we have seen before. In particular, we stop exploring a path as soon as we can determine that its suffix will execute exactly the same as the suffix of a previously explored path. Note that truncating a path explored by EXE results in a large gain, as the number of new paths spawned from a path can be exponential in the number of symbolic branch conditions encountered in its suffix. In real code, this truncation can easily be the difference between doing useful work and getting uselessly stuck revisiting that same program point in equivalent states, as illustrated in one of the contrived examples presented later in this section, but also as suggested by our experiments in Section 4.

The RWset algorithm is sound – relative to the base system – with respect to the effects that we observe in the program (in our particular implementation with respect to the basic branch coverage achieved in the program), as the RWset analysis only discards execution paths that are proven to generate the same effects as some previously explored path (e.g., that are proven to cover the very same basic blocks). The soundness guarantee is relative to the base system, because the RWset technique only discards redundant paths; if, for example, the base symbolic execution tool misses a non-redundant path due to imprecision in its analysis, then the RWset version of the same system will miss that path too.

In order to determine when we can stop executing a path, we apply the simple observation that deterministic code applied to the same input in the same internal state must compute the same result. For simplicity, assume for now that the *state* of a program is just the current set of path constraints (we discuss details concerning program states in the next section). If a path arrives at a program point in the same state as a previous instance, the system generates a test case, and then halts execution. We call such an event a *cache hit.* We generate a test case on a cache hit so that the prefix of the path (which so far has been unique) will be tested.

The attentive reader will note that, as discussed so far, such a cache hit will actually be fairly rare — the only reason a different path would execute is because of a branch, which would add the corresponding branch condition on one path and its negation on the other (e.g., $x = 1234$ and $x \neq 1234$), preventing most subsequent cache hits. We greatly increase the applicability of the basic idea by exploiting the following refinement: a value not subsequently read by

	Constraint Cache No refinement	Live vars	Refined cache
`1: x = read_sym_input();`	$\{x = *\}$	$\{x\}$	$\{x = *\}$
`2: if(x == 1234)`	$\{x = *\}$	$\{\}$	$\{\}$
`3:   printf("foo");`	$\{x = 1234\}$	$\{\}$	$\{\}$
`4: else printf("bar");`	$\{x \neq 1234\}$	$\{\}$	$\{\}$
`5: ...`	$\{x = 1234\}, \{x \neq 1234\}$	$\{\}$	$\{\}$ *HIT!*

Fig. 2: Constraint cache example using code from Figure 1 both with and without refinement. The constraint cache is used to truncate paths that reach a program point with the same constraints as some previous path. "Live vars" denotes the set of all variables read by code after a program point given a set of path constraints. Refinement considers two constraint sets equal if all constraints (transitively) involving live variables are equal.

the program can be dropped from the state, as it cannot affect any subsequent computation.

As the program executes, a *constraint cache* records all the states with which each program point was reached. When we get a cache hit (i.e., a path reaches a program point with the same constraint set as some previous path) we stop executing the path. We illustrate how RWset analysis works on the simple code example in Figure 1, both without and with the refinement discussed above. As shown in Figure 2, the initially empty constraint cache gets populated by the two code paths as follows. At line 1, EXE checks if the path constraint $\{x = *\}$ is already in the cache. Since this is not the case, the constraint set is added to this program point and execution continues. When line 2 is reached, EXE forks execution, adding the constraint $x = 1234$ on the first path, and $x \neq 1234$ on the second. Subsequently, the current constraint set for each path is added to the constraint cache: $\{x = 1234\}$ at line 3, and $\{x \neq 1234\}$ at line 4. Finally, when both paths reach line 5, they add their current constraint sets to the constraint cache.

Note that when the second path reaches line 5 with constraint set $\{x \neq 1234\}$, there is no cache hit, since the only constraint set already in the cache at this point is $\{x = 1234\}$. However, if we assume that the code from line 5 onward does not read `x` again (`x` is a *dead variable*), we can drop all the constraints involving `x`, thus changing the picture dramatically.

More precisely, before adding the constraint set to the cache, we intersect it with the current set of *live variables* (details on how liveness is computed in our frameworks are described in § 3.2). Since `x` is not read after line 2 it is not in the set of live variables, and at lines 2, 3, 4 and 5 we add the empty set to the constraint cache. In particular, when path 1 reaches line 5, it adds the empty set to the cache. Then, when path 2 reaches line 5 too, its constraint set is also the empty set, and thus the system gets a cache hit at this point, and stops executing path 2. As discussed earlier, when pruning path 2, EXE generates a test case to cover path 2's unique prefix – if we did not generate a test case, we would not have a test case that exercises the `printf` call at line 4. Note that pruning a path can save significant work since the number of paths the pruned path would otherwise spawn can increase exponentially with the number of symbolic branches hit by the path's suffix.

As positive as it is to truncate paths spawned at if-statements, it is even better to truncate loops. As an example, consider a common style of event processing loop that shows up frequently in device drivers and networking applications where the code spins in an infinite loop, reading data and then processing it:

```
while(1) {
   x = read_data();  // x is symbolic.
   process(x);
}
```

Here, a naive constraint-based execution system will never terminate, since it will keep reading new symbolic data and generating new paths. A widely-used hack for handling such loops is to traverse them a fixed number of times. Unfortunately, such blind truncation can easily miss interesting paths. In contrast, as long as the loop has a finite number of states (or more precisely, as long as it is observed in a finite number of ways), RWset analysis will automatically determine when the loop reaches a fixed point and terminate it afterwards. Note that while the code above is contrived, the problem is very real: handling such loops in device drivers and networking applications was a primary motivation to build RWset analysis in EXE.

As an even more common pattern, consider the case of a loop that uses a symbolic variable as a loop bound, as in the following code where we assume the constraint that $n < 10$:

```
...
for(i = 0; i < n; i++)
  foo();
...no reads of i, n...
```

When running this loop, EXE will spawn ten new executions, one for every feasible loop exit, each with different constraints on n (that is, $NOT(0 < n)$, $NOT(1 < n)$, etc). If there are no subsequent reads of `i` or `n`, then RWset analysis will prune all but one of these ten new executions, thus saving an enormous amount of subsequent work.

3 Key Implementation Details

This section discusses some key implementation details that are critical in making the approach scale to real applications. To make the exposition clearer, Table 1 groups the terms used by this section for ease of reference.

3.1 Program states

This section discusses state representation issues.

Term	Definition
Program point (§ 3.1)	A context-sensitive MD4 hash of the program counter and callstack.
Path constraints (§ 2.1)	All constraints accumulated on a given path thus far.
Writeset (§ 3.1)	The set of concrete values written to concrete memory locations by a given path thus far.
Readset (§ 3.2)	All locations read after a program point given a program state.
Program state (§ 3.1)	A program point plus its writeset and path constraints Two program paths with identical program states must produce identical subsequent effects.

Table 1: Terminology

Handling mixed symbolic and concrete execution. If execution happened entirely symbolically, path constraints would provide a complete description of the current program state and would be the only thing stored in the constraint cache. However, for efficiency, we want to do as many things concretely as possible. While conceptually concrete values can be viewed as equality constraints (e.g., if variable `x` has the value `4`, this could be represented by the constraint $x = 4$), it is more efficient to separate the symbolic and concrete cases. Thus, a program state includes both the current path constraints (the symbolic state) and the values of all concrete memory locations (the concrete state).

Because the concrete state can be huge, we do not record it directly but instead only track the set of values written along the path — i.e., the path's difference from the initial concrete state all paths begin in. We call this set the *writeset*. When a concrete value x is assigned to a memory location v, we add the pair (v, x) to the writeset. We reduce spurious differences between writesets by removing a memory location from a writeset in two cases. First, when it is deallocated (by function call return or explicit heap deallocation) since we know these elements cannot be read later (a separate component of EXE catches use-after-free errors). Note that we only remove these values from the writeset, not from the path constraints, since deallocating a variable should have no impact on previously formulated constraints. To make this point clearer, assume we add the constraint that $x < y$ and then deallocate `y`; the constraint $x < y$ should definitely remain. The second, implementation-specific removal happens whenever an operation makes a formerly concrete memory location symbolic: it this case we remove the concrete location from the writeset, since it will now appear in the path constraints. The simplest example of such an operation is assigning a symbolic value to a concrete location.

Callsite-aware caching. The state must also include some context-sensitive notion of the current program point. Otherwise, the constraint cache entries for a function generated from other callsites can cause us to falsely think we can prune execution. Truncating path exploration when we get a cache hit is only sound if the entire path suffix after the current program point is identical to some previously explored one. This is only guaranteed when the current call will return to the same callsites that generated the cache entry. For example, consider a program that has calls to both of the following functions:

```
a() {          b() {
  c();           c();
}              }
```

Assume the tool first follows a path that calls `a`, which will then call `c`, populating `c`'s constraint cache. Subsequently, it follows a path that calls `b` and, hence, `c`. If we ignore context, we may (incorrectly) get a cache hit in `c` on an entry added by `a`, and stop execution, despite the fact that returning to the call in `b` can produce a very different result with the current constraints than returning to the call in `a`. Our implementation handles this problem by associating a secure MD4 hash of the current callstack with each constraint cache entry. Other approaches are also possible.

Granularity. Our cache tracks values at the byte level. One extreme would be to track the values associated with each bit. While this adds more precision, it was not clear the increase in bookkeeping was worth it. We could also choose a more coarse-grained approach, such as tracking constraints at the variable or memory object level, which would decrease the amount of bookkeeping, but unfortunately would miss many redundant paths, since often only some parts of a memory object are dead, but not the entire object. We picked byte-level granularity because it seems to be the right trade-off between memory consumption and precision, and because it's a straightforward match of C's view of memory.

3.2 Live variables

We call the set of locations read after a program point the *readset* at that program point; any value in the program state not in this set can be discarded. Thus, the more precise (smaller) we can make the readset, the more irrelevant parts of the current state we can discard and the more likely we are to get cache hits and prune an exponential number of redundant paths.

One approach to computing the readset would be to use a static live variable analysis. Unfortunately, doing so would be incredibly imprecise — for example, often the heap contains most of the program state, which such an analysis typically gives up on. Instead, we compute the locations dynamically, which turns out to be both cheap and very accurate. The basic algorithm is as follows. At a given program point, we do a complete depth-first (DFS) traversal of all paths after that point. The union of all values read by these paths is the readset for that program point, and any part of the current state not observed by this readset can be discarded. As a simple but effective optimization, as we propagate the readset backwards up each path, we remove from it all locations that are deallocated or overwritten. For example, a read of `z` will be removed from the readset if we hit an assignment to `z`.

The reader may be concerned about whether this algorithm is sound when the DFS traversal does not achieve full branch coverage, such as when some path constraints make some branches infeasible. For example, assume we traverse the following code with the constraint that $x \neq 12$:

```
    ...
    // after DFS from this point, the readset will be {x}
1:  if(x == 12)
2:    if(y == 34)    // constraint x!=12 makes this branch unreachable
3:      printf("hello\n");
   ... no further reads of x or y ...
```

In this case, we will never execute the branch at line 2, so y will not be in the readset, and will be discarded from the current program state. Will this cause us to discard states that could reach line 2? The answer is no: since x is in the readset, the only states that will be discarded at line 2 are those that have an equivalent set of constraints on x, i.e, those for which $x \neq 12$. But these states don't satisfy the branch at line 1 and so will not execute the branch at line 2 either. Recursively applying this argument can easily be used to formally prove that the dynamic algorithm is sound even when it does not explore all branches.

3.3 Symbolic memory references

Symbolic memory references complicate both readset and writeset construction. Assume we have an array reference a[i] where i is symbolic. If the only constraint on i is that it is in-bounds, a[i] can refer to any location in a. Even if the constraints on i limit it to a narrow range, the cost of precisely determining this range (via expensive constraint solver interactions) often means that we must accept imprecision. For reads, this imprecision inflates the readset in two ways. First, we must conservatively assume a[i] can reference any location in a unless we can prove otherwise. Thus, a single reference can pull all of a into the readset. Second, when propagating the readset back up a path, when we hit an assignment to a[i] we cannot remove any element from the readset unless we can prove a[i] overwrites it. As a result, in our implementation, assignments to arrays at symbolic offsets (indices) do not filter the readset at all.

Similarly, such assignments identically prevent removing elements from the writeset. Recall that assigning a symbolic value to x causes x to be removed from the writeset and added as a path constraint instead. However, when we assign to a[i] we can only remove an element from the writeset if we can prove that a[i] must overwrite it.

3.4 State refinement

Given a program state and a readset, we remove irrelevant parts of the program state as follows:

1 Concrete state: keep the locations in the intersection of the readset and writeset.

2 Symbolic state: keep the transitive closure of all constraints that overlap with the readset. For example, if the readset is $\{x\}$ and the current path constraint is: $\{x < y, y < 10, z < w\}$, our transitive closure would be $\{x < y, y < 10\}$. Note that taking the intersection instead of the transitive closure, would produce constraint sets that allow solutions illegal in the original path constraint.

3.5 Abstraction issues

For space reasons, we have currently taken a very literal view of what the program state is, what reads and writes are, what is considered a cache hit, and what the effects of a path are. One can, of course, profitably vary all of these, depending on the requirements of the testing process. We consider two first-order decisions.

First, what effects of computation are we interested in? The literal view is everything. We can also consider things more abstractly. For example, one may consider only the effects that affect branch coverage, or those that expose bugs in the program. Deciding what effects to focus on determines what reads (or writes) we must consider: if a read cannot affect the given metric, then the read can be ignored. For example, if we are aiming for branch coverage, we can ignore all reads not involved in an eventual control dependency.

Second, what is a cache hit? Thus far we have assumed two states are equal if they match exactly. We can however, improve on this definition. One sound improvement is to notice that two sets of constraints are equal if they would cause the same effect. For example, if one path has $x < 10$ and another $x < 20$ and the only subsequent use of `x` is a branch comparison $x < 30$, then we could consider these two constraints to be equal since they both satisfy the condition.

3.6 Summary

We now summarize the basic RWset implementation in EXE. We represent the symbolic state by the current path constraint, the concrete state by the writeset, and the program point by the callstack and program counter. Each context-sensitive program point records all previous states it has been visited in, and associates with each of these entries the complete set of reads (observations) done by all subsequent paths when reached in this state (the readset). We determine if we have already seen a state by comparing it against each of these entries after first intersecting it with the entry's associated readset. If we get a hit, we generate a test case (to exercise the path up to this point) and terminate further exploration of this path. Otherwise we continue.

4 Evaluation

This section evaluates the effectiveness of RWset analysis on real code, using a mix of server and library code and operating system device drivers. The results show that the technique gives an order of magnitude reduction in the number of tests needed to reach the same number of branches, and often achieves branch coverage out-of-reach for the base version of EXE. All experiments were performed on a dual-core 3.2 GHz Intel Pentium D machine with 2 GB of RAM, and 2048 KB of cache.

4.1 Server and library code

Our first experiments measure the improvement given by RWset analysis on five medium-sized open-source benchmarks previously used to evaluate the base version of EXE [4]: `bpf`, the Berkeley Packet Filter; `udhcpd`, a DHCPD server; `expat`, an XML parser library; `tcpdump`, a tool for printing out headers of network packets matching a boolean expression; and `pcre`, the Perl Compatible Regular Expression library.

We ran each of these benchmarks for roughly 30 minutes each with the base version of EXE, and recorded: (1) the (maximum) branch coverage achieved, and (2) how many test cases were necessary to achieve this coverage (note that sometimes we generate more than this number of test cases in 30 minutes, but the extra tests don't hit any new branches). The one exception was PCRE, which we ran longer until it generated 30,000 test cases in order to have a meaningful comparison between the base and RWset versions. The second column of Table 2 gives the number of branches hit by these runs and the third column gives the number of test cases.

We then reran each benchmark using the RWset version of EXE and recorded the number of test cases necessary to achieve the same coverage as the base version did in half an hour. The last column of Table 2 gives the percentage of test cases needed for the RWset version to match the coverage from the base run. As the table shows, the improvement can be substantial: `tcpdump` only needs 11.4% the number of test cases to get equivalent coverage (249 vs 2175 tests) and `bpf` needs 16%. In fact, with the exception of `pcre`, all benchmarks need less than half the number of test cases with the RWset version.

We also measured the number of distinct states visited by the RWset version relative to the base. The graphs, shown in Figure 3, indicate that without RWset analysis the system wastes enormous resources constantly revisiting redundant states, thus generating many irrelevant test cases.

	Base		RWset
	`Branches`	`Tests`	`% tests needed`
tcpdump	123	2175	11.4%
bpf	171	6333	16.2%
expat	472	677	31.1%
udhcpd	166	225	49.7%
pcre	1268	26,596	72.2%

Table 2: Number of tests in RWset mode necessary to achieve the same coverage as in the base system.

Finally, we measured the runtime overhead of our RWset implementation by running an EXE version that performs all computations RWset requires (constructing readsets and writesets, checking for cache hits), but without pruning any paths. Thus, this version generates exactly the same tests as the base version of EXE, while paying the full cost of RWset analysis. Our measurements show that for all benchmarks the average overhead is at most 4.38%.

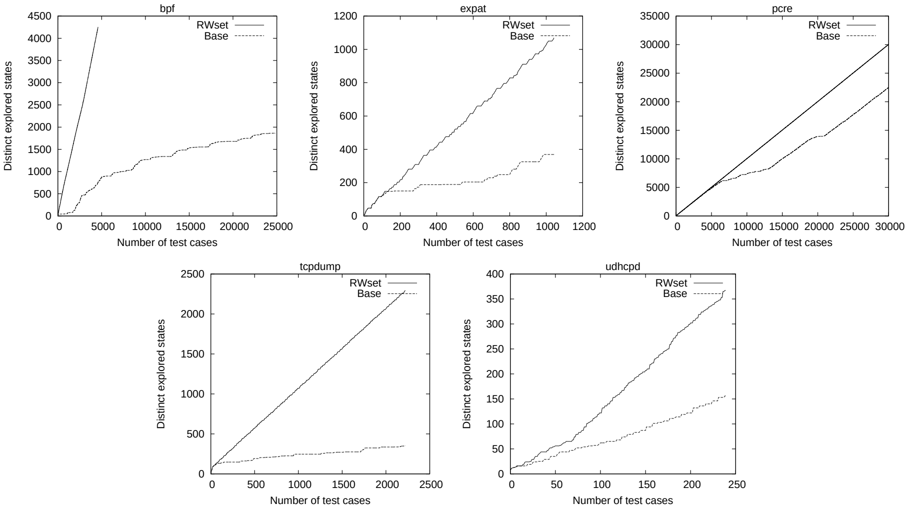

Fig. 3: Distinct explored states over number of test cases for the base system versus the RWset version for the server and library code benchmarks. With the exception of PCRE, the base system without RWset wastes much of it time exploring provably redundant states.

4.2 Device drivers

We also checked OS-level code by applying EXE to three Minix device drivers. Minix 3 [16, 10], is an open source, Unix-like, microkernel-based operating system, with a kernel of under 4000 lines of code, and almost all functionality – including device drives – running in user space. [1]

Drivers make up the bulk of modern operating systems and are notoriously buggy [1, 15]. Drivers are an interesting case for systems such as EXE because, while drivers ostensibly require a physical version of the device they are intended to drive, they only interact with the device through memory-mapped I/O, which mechanically looks like a memory array with special read and write semantics. Thus, we can effectively test a driver by marking this array as symbolic and running the driver inside our symbolic environment.

The Minix driver interface makes this approach easy to apply. Minix drivers are built as standalone processes that use a small message-passing interface to communicate with the rest of the system. Their organization mirrors that of many network servers: a main dispatch loop that waits for incoming messages from other processes, the kernel or the hardware (the kernel translates hardware interrupts into messages as well) and processes the incoming requests. Thus, applying EXE to these drivers was relatively easy: for each read the driver does, we just return a symbolic message.

[1] We have a lot of experience checking Linux drivers but switched to Minix because of the first author's affiliation with the Minix group. We do not expect our results to change for Linux.

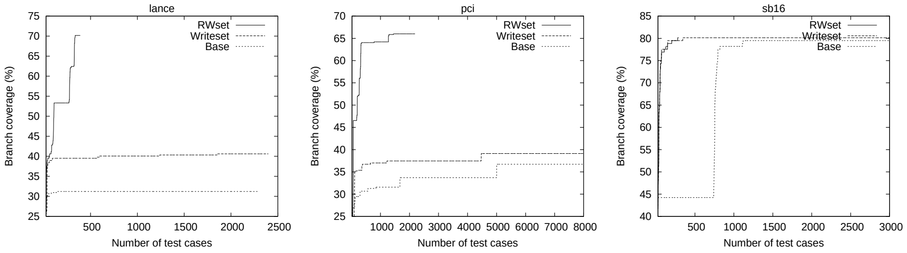

Fig. 4: Branch coverage (in percentage) over the number of test cases generated for the device drivers, comparing the base version of EXE and the RWset version with and without the use of readsets. In the first two cases, the full RWset system quickly gets branch coverage dramatically beyond that of the base system or writeset alone.

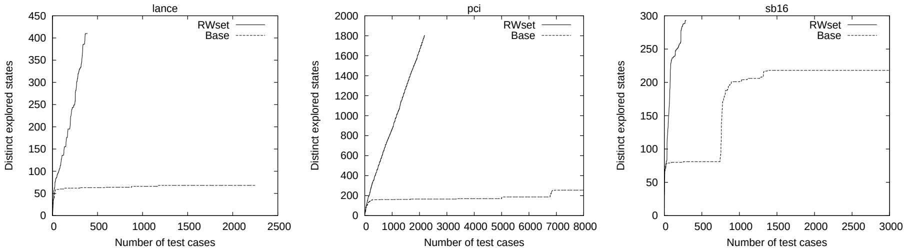

Fig. 5: Distinct explored states over number of test cases generated for the device drivers, comparing the base and the RWset versions of EXE. As with the server experiments, RWset typically both explores many more distinct states than the base system and does so very quickly.

We made two modifications to the driver code. First, we performed simple "downscaling," such as reducing data buffer sizes, which only required changing a few constant values in header files. Second, we fixed the number of iterations in the dispatch loop and limited the search depth (expressed in number of symbolic branches). These latter changes were needed in order to do a better comparison between the base and the RWset versions of EXE. Without them, the base version gets stuck in infinite cycles. As a result of these changes, our experiments *underestimate* the improvement given by the RWset technique.

We run each device driver for one hour in three modes. As before, we use the base and the RWset versions of EXE. In addition, we also measure the effectiveness of using the readset to filter irrelevant state details by disabling this step and only pruning paths when states are exactly equal (*writeset*).

We statically counted the total number of branches in each driver (using a compiler pass) and then, for the three runs, recorded the branches hit by each generated test case. Figure 4 uses this data to plot the cumulative percentage of code branches covered by the given number of test cases. As the figure shows: (1) the RWset version outperforms the base version and (2) a lot of this improvement comes from the readset filtering. For `lance` and `pci`, the base version keeps visiting the same branches, and cannot achieve more than 40% branch coverage in the first hour. In the case of sb16, the base version does not fare as poorly, although it still doesn't achieve as much coverage as the RWset version.

Figure 5 shows the number of distinct states visited by the base and the RWset versions of EXE. The most striking feature of the graph is how quickly the base version gets stuck, repeatedly visiting states that are provably the same and thus generating large numbers of redundant test cases.

While testing these device drivers, we also checked for bugs. Unsurprisingly, the bug count improves with the amount of branch coverage achieved. While we only looked for simple low-level errors (such as assert() failures, null pointer dereferences, buffer overflows and out-of-bounds errors) the RWset version still found thirteen unique, confirmed bugs. On the other hand, the base version of EXE missed all but four of these.

5 Related Work

The idea for RWset was inspired by two bug-finding systems the authors were involved with [9, 18]. The first system [9] statically found bugs by pushing user-written compiler extensions down all paths in a checked program. For scalability, it tracked the internal state of these checkers and stopped exploring paths when a basic block was revisited in the same state. We reused this caching idea to do path truncation dynamically. In retrospect, adding a variation of the readset calculation presented in this paper would have likely made a big improvement in this static system since it would allow it to discard many more paths in a simple way, and transparently scale up with the power of any underlying path-sensitivity added to the system. The second system [18] provided the idea of using read and write sets when computing state equivalence. It dynamically computed such information in terms of the disk blocks a file system repair program read and wrote as a way to determine when crashing and restarting such a program during repair would compute identical results to not crashing. The use of read and write sets let it save enormous amounts of work, leading us to try a similar approach for memory (rather than disk) in our more general checking context.

Recent work on constraint-based execution tools have approached the path explosion problem in a variety of ways. Two methods that use heuristics to guide path exploration are [4] (which attempts to explore paths that hit less-often executed statements) and [13] (which combines symbolic execution with random testing). We view these techniques as largely complementary to the RWset analysis: one can use RWset analysis to discard irrelevant paths and then use these techniques to prioritize the residue.

Another approach, which like RWset analysis uses a static analysis-inspired technique to attack path explosion, tests code compositionally by reusing function summaries [7]. Roughly speaking, it does a bottom-up analysis that records the result of analyzing a function at a given callsite and, if it encounters another call to the same function with the same inputs, reuses the result of this analysis. If we regard program suffixes as functions taking as arguments the current state of the program, then [7] becomes equivalent to our RWSet technique without the readset refinement. We believe the function summary approach could also use a variation of readsets to prune out irrelevant details, and thus both get more summary hits and remove unneeded path constraints.

More generally, the idea of pruning equivalent states is an old one and has shown up in many different situations. A context closely related to ours is the use of state caching in explicit state model checking (e.g., [17]), which tracks the states generated by an abstract model of a system and does not explore the successors of an already-seen state. State caching is often improved through dead variable elimination. In most systems, this is accomplished by running a standard static live variable analysis before model checking begins, as in SPIN and Bebop [11, 2]. In [12], the system uses runtime information to eliminate infeasible paths at various points in the program in order to improve the results of the static live variable analysis. While such pruning helps, we expect the significant imprecision inherent to a static live variable forces this approach to miss many pruning opportunities. However, comparing the two techniques is hard as the benchmarks used in [12] seem to be on the order of a hundred lines of code or less, with at most three loops per program.

We note that while in hindsight it may appear clear that state caching is worth applying to constraint-based tools, the context seems different enough that, while all authors of such tools that we talked to complained about the path explosion problem, no one suggested using a state-caching approach.

A final model checking technique related to RWset analysis is *partial order reduction* [6], which skips redundant states by exploiting the fact that if two actions are independent then the order in which they occur does not matter. The two approaches should work well together: partial order reduction is a “horizontal” approach that eliminates path interleavings, while the RWset technique is a “vertical” one that truncates the remaining paths.

6 Conclusion

While constraint-based execution is a promising approach for automatically generating test cases to cover all program paths, it faces significant scalability challenges for checking large applications. This paper introduces RWset analysis, a technique for detecting and pruning large numbers of redundant paths. RWset analysis tracks all the reads and writes performed by the checked program and uses this information to truncate a path as soon as it determines that the path will execute equivalently to some previously explored one.

We measured the effectiveness of our RWset implementation by applying it to server, library, and device driver code. Our results show that RWset analysis can reduce the tests needed to reach a given number of branches by an order of magnitude, and often achieves branch coverage out-of-reach for the base version of the system, which easily gets stuck revisiting provably redundant states.

References

[1] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *EuroSys*, April 2006.

[2] T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN 2000 Workshop on Model Checking of Software (LNCS 1885, Springer)*, August/September 2000.

[3] C. Cadar and D. Engler. Execution generated test cases: How to make systems code crash itself. In *Proceedings of the 12th International SPIN Workshop on Model Checking of Software*, August 2005.

[4] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler. EXE: Automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, October-November 2006.

[5] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *Computer Aided Verification (CAV '07)*, Berlin, Germany, July 2007. Springer-Verlag.

[6] P. Godefroid. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*, volume 1032. Springer-Verlag Inc., New York, NY, USA, 1996.

[7] P. Godefroid. Compositional dynamic test generation. In *Proceedings of the 34th Symposium on Principles of Programming Languages (POPL'07)*, Jan. 2007.

[8] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, Chicago, IL USA, June 2005. ACM Press.

[9] 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*, 2002.

[10] J. N. Herder. Towards a true microkernel operating system. Master's thesis, Vrije Universiteit Amsterdam, 2005.

[11] G. J. Holzmann. The engineering of a model checker: the Gnu i-protocol case study revisited. 1999.

[12] M. Lewis and M. Jones. A dead variable analysis for explicit model checking. In *In Proceedings of the ACM SIGPLAN 2006 Workshop on Partial Evaluation and Program*, 2006.

[13] R. Majumdar and K. Sen. Hybrid concolic testing. In *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*, May 2007.

[14] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *In 5th joint meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'05)*, Sept. 2005.

[15] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering device drivers. In *OSDI*, pages 1–16, Dec. 2004.

[16] A. S. Tanenbaum and A. S. Woodhull. *Operating Systems Design and Implementation*. Prentice Hall, 3rd edition, 2006.

[17] U. Stern and D.L. Dill. Improved Probabilistic Verification by Hash Compaction. In *Correct Hardware Design and Verification Methods*, volume 987, pages 206–224. Springer-Verlag, 1995.

[18] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using model checking to find serious file system errors. In *Symposium on Operating Systems Design and Implementation*, December 2004.