Realistic Bug Synthesis for Testing Bug-Finding Tools

Anonymous Author(s)

ABSTRACT

In spite of decades of research in bug detection tools, there is a surprising dearth of ground-truth corpora that can be used to evaluate the efficacy of such tools. Recently, systems such as LAVA and EvilCoder have been proposed to automatically inject bugs into software to quickly generate large bug corpora, but the bugs created so far differ from naturally occurring bugs in a number of ways. In this work, we propose a new automated bug injection system, Apocalypse, that uses formal techniques—symbolic execution, constraint-based program synthesis and model counting—to automatically inject *realistic* (uses the program's control and data-flow), *deep* (requires a long sequence of dependencies to be satisfied to fire), *uncorrelated* (each bug behaves independent of others), *reproducible* (comes with a trigger input) and *rare* (fires on a very few program inputs) bugs in large software code bases. In our evaluation, we inject bugs into thirty Coreutils programs as well as the TCAS test suite. We find that Apocalypse's bugs are highly realistic under a variety of metrics, that they do not favor a particular bug-finding strategy (unlike bugs produced by LAVA), and that they are more difficult to find than hand-injected bugs, requiring up around 240× more tests to discover with a state-of-the-art symbolic execution tool.

ACM Reference Format:

Anonymous Author(s). 2018. Realistic Bug Synthesis for Testing Bug-Finding Tools. In *Proceedings of The 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Bugs in software are widespread, and decades of research have gone into eliminating bugs through automated bug-finding tools such as static analyzers, runtime sanitizers, symbolic execution tools, and fuzzers. Despite this ongoing effort, there is a surprising dearth of ground-truth corpora with which we can evaluate the *efficacy* of such tools: most existing corpora are small, do not come with triggering inputs, or feature bugs that are unrealistic. Moreover, the value of any individual dataset drops over time as tools adapt to it. As a result, in many cases we are forced to judge bug-finding tools on how many previously unknown bugs they find—which leaves us in the dark about how many they missed, and will vary depending on the underlying defect rate of the software being analyzed.

The lack of ground-truth datasets also means that it is very difficult to perform large-scale studies of bug discovery. For example, we cannot run bug-finding tools on corpora of millions of bugs and then attempt to draw conclusions about their relative strengths and weaknesses, or statistically correlate features of bugs and programs with their difficulty of discovery.

ESEC/FSE 2018, 4–9 November, 2018, Lake Buena Vista, Florida, United States

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

In recent work, two new systems, LAVA [12] and EvilCoder [24], have sought to address the need for ground-truth corpora through *automated vulnerability addition*. Briefly, these techniques take existing programs and seed them with vulnerabilities, either by adding new, vulnerable code (in the case of LAVA) or by identifying and removing safety checks to make existing code vulnerable. These systems are an important step forward, but fall short in a number of ways.

We consider that injected bugs should have the following properties:

- **Realistic** An injected bug is realistic when it is possible to unearth it by practical bug detection techniques. For example, a bug guarded by a famous mathematical theorem (say, Fermat's Last Threorem) is not realistic as it requires the proof of a difficult mathematical theorem to detect the bug; common bugs in programs does not resemble such a case.
- **Deep** An injected bug must require a sequence of data and control flow conditions to be met for it to trigger. A bug guarded by a single branch condition is, for the same reason, not a good bug.
- **Uncorrelated** Multiple injected bugs must be uncorrelated; that is, finding one of the bugs by a tool should not increase (or decrease) the chances of catching the other injected bugs.
- **Reproducible** An injected bug must come with a triggering input that proves the existance of the bug.
- **Rare** The bug should be triggered on a very small fraction of all possible program inputs.

Considering these properties, we find that existing bug injection techniques can be improved in several ways. EvilCoder, for example, cannot currently produce provide triggering inputs, and hence fails to be *reproducible* (related technqiues, such as mutation testing [18], also fail to satisfy this requirement). And we find that LAVA's bugs, although rare, uncorrelated, and reproducible, fail to be sufficiently realistic and deep: the triggers used (a comparision against a 32-bit "magic" constant) are unusually difficult for techniques such as random testing to find, and although the bugs manifest deep within programs, the injected guard is a single branch that can be systematically targeted [29].

In this work, we introduce a new technique for bug injection, based on symbolic execution, program synthesis and uniform sampling. We build our ideas into a tool, Apocalypse, and use it to introduce bugs in thirty coreutils programs. Apocalypse uses constraint based program synthesis to embed a transition system, what we refer to as the *Error Transition System (ETS)*, on a judiciously chosen program path. When the program is executed, the ETS is advanced at certain locations along this execution path, leading to a crash if the final state is reached. The state transitions on the ETS are guarded by carefully synthesized predicates that ensure that a few executions can successfully reach the final state, and, therefore, trigger the bug. We do so by enabling the synthesis engine to perform a multi-variate hill climbing on the space of predicates at the transition locations—searching for predicates that prevent most executions from reaching the bug location. We estimate the set of inputs that a predicate "blocks" from reaching the error location by model counting (approximated by uniform sampling).

We use Apocalypse to inject multiple bugs in thirty Coreutils programs, and then attempt to detect these bugs using state-of-the-art symbolic execution (KLEE [7]) and greybox fuzzing (AFL [1]) tools. The bugs demonstrated discoverability, demonstrating that the system does not inject unrealistic bugs: KLEE and AFL were able to discover 31% and 38% of the bugs; at the same time, many bugs were elusive, showing that these bugs can act as subjects for further research: 47% of the bugs could not be discovered by either of the tools. Similar to real bugs, different bugs showed affinity to different tools: out of the 53% bugs discovered, 15% of the bugs could be discovered only by KLEE while 22% of the bugs could be found only by AFL. The Apocalypse injected bugs needed about 240× more tests to be discovered than the manually seeded bugs (on our benchmarks). We also compared our tool with LAVA and found that the bugs injected by LAVA tended to be biased to one of the tools: about 80% of the bugs injected by LAVA were discovered by KLEE while only about 41% of these bugs were discovered by AFL; the bugs injected by Apocalypse responded almost uniformly on both KLEE and AFL, showing the bug corpora produced by Apocalypse do not favor a particular bug-finding strategy.

The contributions of this paper are as follows:

- We propose a symbolic execution-based strategy to automatically inject realistic, deep, uncorrelated, reproducible and rare bugs in programs.
- We propose a model counting-based strategy to reduce the number of bug inducing inputs to make the injected bugs difficult to find.
- We build our ideas into a tool, Apocalypse, and use it to inject bugs into Coreutils programs.
- We attempt to discover the bugs injected using Apocalypse using a symbolic execution engine (KLEE) and a greybox fuzzer (AFL). The experiments demonstrate that our injected bugs indeed show properties close to real bugs.

2 OVERVIEW

Our bug injection system, Apocalypse, begins with a concrete input and a program trace induced by that input. This input, which can be taken from the program's test suite or (as in our current implementation) discovered through symbolic execution, serves as a path along which we will add one or more bugs to the program. In the context of bug injection as a game between the injector (who wishes to add hard-to-find bugs to the program) and the bug-finder (who would like to find the bugs added by the injector), the concrete input serves as a source of *asymmetric advantage* in favor of the injector: armed with a concrete input, the injector has knowledge of an entire program path and all dynamic values along that path, whereas the bug-finder must search for the same program path among the space of all program paths.

Into the subset of the program described by the trace, Apocalypse embeds an *error transition system* (ETS) that incrementally advances a state machine towards an error state (i.e., the program point where the injected bug will manifest). Each transition in the state machine is triggered whenever pre-existing program variables meet certain conditions. To create a bug satisfying the requirements described in Section 1, these conditions must be *simple* (to match the complexity of real-world program conditionals), *non-trivial* (i.e., not always true or false), and *useful*, meaning that they are satisfied by a relatively small number of inputs.

Apocalypse achieves these goals by using *program synthesis* to create candidate conditions from variables that are in scope at different points (*transition points*) along the trace. Not all points in the trace are equally promising (in terms of the number of variables available) as transition points, and so we first scan the trace looking for program points that are deep in the call graph, guarded by many predicates, and have many variables in scope. To ensure that the synthesized constraints that trigger each transition meet our requirements, we use *model counting* to estimate the number of solutions to the conjunction of ETS constraints so far, and iteratively improve the constraint set by reducing the number of possible solutions.

The state machine itself is tracked using global program state. In order to match the surrounding program, the state should be stored in an appropriate representation. For example, if the program primarily manipulates integer values, we can use integer variables to track the state (and this case is what our current prototype supports). But we could also track state using string matching on some string embedded in the program, or the position of some node in an aggregate data structure such as a list or a tree, depending on what data structures and operations are already in the program. We give some concrete examples of state machine encodings in Section 3.7.

Finally, we create a buggy version of the program by adding, at each transition point, a snippet of code that checks one of our ETS conditions and then advances the state machine. When the state machine reaches its accepting state, we trigger buggy behavior in the program. In our current implementation, we simply add an *assert(false)*, but we could also add out of bounds memory accesses, integer errors, etc. depending on the type of bug detector under test.

Listing 1 shows a program with an injected bug; the statements synthesized by Apocalypse are shown commented in *green*; the program is instrumented with the ETS in Figure 1.

3 ALGORITHM

We define a *program trace* (or simply a *trace* Δ) as a sequence of dynamic instructions. We assume each trace Δ to have a triggering input `ip`, that causes the program to execute the given trace, and a symbolic path constraint, `sym_pc`, that encodes the conditions on the inputs that would follow the given trace. Given a map $\Gamma : V \mapsto \mathcal{E}$ from program variables $v_i \in V$ to symbolic expressions $s \in \mathcal{E}$, we use the notation $\Gamma[\phi(v_1, \ldots, v_n)]$ to denote the symbolic constraint formed by replacing each $v_i \in V$ by the respective symbolic expression from the map Γ.

3.1 Error Transition System

Apocalypse injects bugs in programs by interweaving an *Error Transition System* along a path in the program. The ETS is a tuple $(L, \mathcal{P}, \delta, l_0, l_{bug})$ where:

```
void ALIM ()
{
 othCap = climb + altVal;
 L6: /* if (ownRate < otherAlt && state == 6)
          state = 19; */
}

int InhibitBiasedClimb ()
{
 int up,down;
 up = upSep + 100 +altVal;
 down = upSep + OtherTrackedAlt;
 L16: /* if (othCap < climb && state == 16)
          state = 6;*/
 return (climb ? up : down);
}

void main()
{
 input(curSep, ownAlt, ownRate, otherAlt,
       altVal, upSep, downSep, othCap, climb);
L0:
 /* state = 16; */
 upPref = InhibitBiasedClimb() + downSep;
 upCross = ownAlt + otherAlt;
 ownRate = ownRate + curSep;
 ALIM();
 if (uppref > 5500){
   result = altVal;
 L19: /* if (climb == result && state == 19)
          state = 21;*/
  }
 L21: /* if (othAlt < upCros && state == 21)
          state = 30;*/
 upCross = ownAlt - otherAlt;
 L30: /*if (state == 30) assert(0);*/
}
```

Listing 1: Program with injected bug: the statements commented in *green* are statements inserted by Apocalypse (as per the ETS in Figure 1)

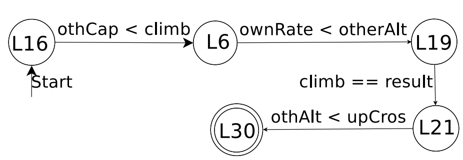

Figure 1: ETS for the program in Listing 1

Algorithm 1 Apocalypse

procedure Main($\mathcal{P}$)
  $\Delta \leftarrow$ IdentifyTrace($\mathcal{P}$)
  $L \leftarrow$ IdentifyTransitionPoints(Δ)
  $sym\_pc, \Lambda \leftarrow$ SymbolicExec(Δ, L)
  $\Omega \leftarrow$ SynthesizeETS(sym_pc, Λ)
  $\mathcal{P}' \leftarrow$ InstrumentETS($\mathcal{P}$, Ω)
return $\mathcal{P}'$
end procedure

- L, the set of states, corresponds to program locations that drive a transition on the ETS;
- $\mathcal{P}$ is a set of all predicates that can be constructed using the program variables;
- $\delta : L \times \mathcal{P} \to L$ is the transition function that dictates the transition on the ETS, given a predicates $p \in \mathcal{P}$ at a location $l_i \in L$;
- $l_0 \in L$ is the initial state of the ETS; the ETS is set to this state at the entry point of the program;
- $l_{bug} \in L$ is the program location that is instrumented by the buggy action (say simulating a program crash).

Figure 1 shows the ETS for the program in Listing 1: the labels `L6`, `L16` etc. (marked in *red*) show the transition locations where the ETS makes its moves; the program entry point (`L0`) sets the ETS to the initial state (`L16`). The transition on location `L16` is guarded by the predicate ($othCap < climb$); if this predicate holds, the ETS transitions to state `L6`. Finally, if the execution can drive the ETS to the final state (`L30`), an error is raised (say by simulating a crash or violating an assertion).

3.2 Identify a program trace

The algorithm driving Apocalypse is shown in Alg 1. Given a program $\mathcal{P}$, the algorithm starts off by using IdentifyTrace($\mathcal{P}$) to identifying a *trace* Δ on which an ETS will be embedded. IdentifyTrace($\mathcal{P}$) uses symbolic exploration to collect multiple possible intraprocedural paths in the program and selects a path based on the following parameters:

- **Complexity of the path**: We prefer program paths that contain a large number of dynamic instructions, pass through a large number of procedures and hit a large number of branching instructions. As this path represents the *secret* information that the adversary (bug detection tool) will need to discover, a complex path makes the injected bugs more elusive.
- **Number of *useful* variables**: This refers to the quality and quantity of the variables that are used by the participating instructions along this path. The quality of a variable is dictated by distance of the instruction that defines the variable in the *program dependence graph* from the input statements. In essence, it captures the "complexity" of constructing a required value into this variable from the program inputs. We select paths with abundant good-quality predicates, as these variables will eventually be used by the *ETS synthesizer* to construct transition predicates.

3.3 Identify transition points in the program

Next, IdentifyTransitionPoints(Δ) attempts to find good program locations on the error trace to embed ETS transitions. A location is selected if it meets the following criteria:

- Abundant "useful" variables are available at that program location;
- The program location is deep in the call graph, making it hard for bug detection tools to reach this location;
- The program location appears deep in the control dependence graph; a location deep in the control dependence graph is guarded by multiple predicates, making reachability challenging for bug-detection tools.

The above metrics on identifying a trace and transition locations can be tuned to inject bugs of varying degrees of hardness, thereby allowing one to gauge the effectiveness of different bug detection techniques. In this project, we have attempted to inject bugs that are harder to find; we plan to investigate on the above questions in future work.

3.4 Collect Symbolic Constraints

In the next phase, we run a symbolic execution engine on trace Δ to collect the following:

- **Symbolic Path Condition (*sym_pc*)**: The path condition (*sym_pc*) for the trace (Δ) contains a symbolic summarization of all possible input values that would drive a program execution along Δ.
- **Symbolic Expression Dictionary (Λ)**: This dictionary $\Lambda : L \mapsto (V \mapsto \mathcal{E})$ maps each identified transition location $l_i \in L$ in the program to a dictionary of symbolic expressions $\mathcal{E}$ for each program variable $v \in V$.
- **Concrete Value Dictionary (C)**: This dictionary $C : L \mapsto (V \mapsto \nu)$ maps each identified transition location $l_i \in L$ in the program to a dictionary of concrete values ν observed for each program variable $v \in V$ along the execution trace.

3.5 Synthesize the Error Transition System (ETS)

In this phase, we use constraint solving to synthesize an Error Transition System (ETS) that can be embedded in the program. The synthesis algorithm is shown in Algorithm 2.

Synthesis of the ETS essentially involves identification of the transition predicates that guard the automata transitions. The identified predicates should satisfy the following properties:

- **Simple**: The predicate should be simple to compute so as to not change the dataflow behaviour of the existing program by much.
- **Non-Trivial**: The predicates should be non-trivial; for example, $(x \geq x)$, $(x + 42 > x)$ etc. should not be produced.
- **Useful**: The predicate should effectively reduce the number of inputs that could trigger the bug.

For all the transition locations $l_i \in L$, let $pred : L \to \texttt{Predicate}$ denote a dictionary of the predicates synthesized, such that $pred[i]$ is the predicate at the i^{th} location (l_i). This map is initialized to $pred : L \to \texttt{true}$.

We synthesize the predicates for the different locations in a round-robin manner; each predicate, $pred[i]$, is synthesized subject to the current values of all other predicates. The predicate for the k^{th} transition location (denoted as $(v_1\ op\ v_2)^k$) is synthesized using the following **synthesis condition**:

[Equation **Synth**]

$$(v_1\ op\ v_2)^k \equiv \exists_{\alpha_1,\ldots,\alpha_n}\ \ (sym\_pc \wedge \prod_{l_i \in L,\, i \neq k} pred[i] \\ \wedge [\![\ C_k(v_1)\ op\ C_k(v_2)\ ]\!]) \\ \wedge \neg [\![\ \Lambda_k(v_1)\ op\ \Lambda_k(v_2)\ ]\!]$$

where

$$[\![\ \mathcal{V}_1\ op\ \mathcal{V}_2\ ]\!] = \begin{cases} [\![\mathcal{V}_1]\!]\ <\ [\![\mathcal{V}_2]\!] & \text{for } op =' <' \\ [\![\mathcal{V}_1]\!]\ \leq\ [\![\mathcal{V}_2]\!] & \text{for } op =' \leq' \\ [\![\mathcal{V}_1]\!]\ =\ [\![\mathcal{V}_2]\!] & \text{for } op =' =' \end{cases}$$

The above constraint synthesizes a guard $(v_1\ op\ v_2)$ for the k^{th} transition location if there exists a feasible execution (i.e. feasible values of the input symbolic variables $\alpha_1, \ldots, \alpha_n$) that meet the following conditions:

- The first two terms ensure that this execution takes the same execution path as the triggering input (so as to satisfy *sym_pc*) and also satisfies the guard conditions synthesized for all other transition locations ($\wedge \prod_{l_i \in L,\, i \neq k} pred[i]$);
- The next term ensures that the concrete values (corresponding to the seed input) of the variables v_1 and v_2 satisfy the synthesized guard condition; this term ensures that the injected bug would be triggered by this seed input;
- The final term ensures that there exists a feasible execution along the `false` path of the synthesized guard; the term is designed to ensure that the symbolic values of the variables v_1 and v_2 are capable of generating an execution along the `false` branch of the guard condition; this prevents generation of trivial predicates that are always `true`.

Now, we lay out the complete synthesis algorithm in Algorithm 2. Lines 3-4 initialize the dictionary *pred*; it also initializes the dictionary *sols* that maintains the set of solutions that we have already seen earlier. Then, the procedure enters into an iterative refinement loop to inductively search for *good* guards for transition predicates: for each transition location $l_k \in L$, Apocalypse tries to find a feasible guard $(v_1\ op\ v_2)$ (as per Eqn Synth). To ensure monotonicity, it searches for a solution (line 10) while ensuring that any new solution does not include a solution that we have seen earlier (cached in sol_k).

If Ψ is satisfiable, the predicate is extracted (from the model associated with Ψ); else we move to the next location. The *pred* and *sols* dictionaries are finally updated at lines 19-20 as per the new solution found.

Figure 2 provides a simplified view on the operation of our synthesizer: assuming `Trigger` as the seed input, the synthesis constraint attempts to seach for a point `P1` and predicate (denoted by the line) that divides the input space into two partitions: *white*-region that would induce the bug, and *blue*-region that would not. The existence of the point `P1` is important to prevent generation of trivial predicates that do not divide the input space (say as lines that are tangents to the input space). Futher, in the next iteration, futher *shrink* the bug-inducing region by searching for another point `P2`—from the *bug-inducing region*—such that the new predicate separates `Trigger` from `P2`.

To estimate the usefulness of the predicate, we perform a *hill-climing search* over the multi-variate predicate space corresponding to each location (lines 16-19). The hill-climing search uses a model-counter to estimate the number of feasible inputs corresponding to the newly synthesized predicate and the older predicate (cached in $pred[k]$); we always choose a predicate that *shrinks* the space of bug inducing inputs. The search is designed similar to a Gibbs sampler [cite] for multi-variate problems, wherein we make the

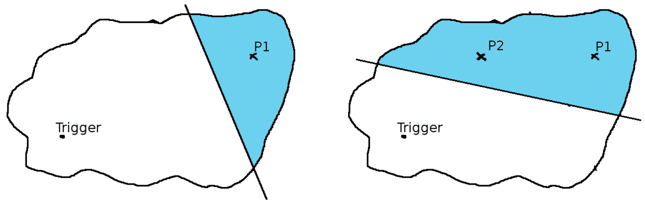

Figure 2: Input space pruning

decision about one variable conditioned on the current values of every other variable.

As calling a model-counter twice in each loop iteration is quite expensive, in our implementation, we approximate the relative usefulness of the predicates by performing uniform sampling on *sym_pc* to create a sampled space of inputs that follow the same path as the seed input. Then, we generate and execute a program to count the number of inputs that are satisfied by the competing guard predicates; we show a sketch of our generated programs in Listing 2.

Note that the algorithm generates the guard predicates in a manner such that the seed input ends up as the trigger for the bug. This was important as the symbolic execution engines often concretize parts of the execution (for instance, at external library calls, floating-point operations, system calls etc.). This causes the map Λ to not have a complete symbolic model, thereby leading to path divergence [15].

Let us explain the problem: in Listing 3, assuming *sqrt* is an external function, its output will concretized. Hence, the path condition (of an execition where both branches evaluate to `true`) would be an *incomplete* $(a > 42)$ (rather than $(a > 42) \wedge (c > 10)$). If we desire to get inputs that would trigger the bug by hitting locations `L0`, `L6`, and `L10`, we would need to solve the respective path condition $(a > 42)$ for which a constraint solver can return a solution `a = 95, z = 0` (unaware of the branch constraint $(c > 10)$). This input fails to trigger the bug as the program *diverges* to a different path at the branch "`if (c > 10)`". We handle this problem by transforming the seed input (which is consistent with all concretizations) into the trigger for the bug.

For simplicity, the above algorithm assumes that each program location is hit at most once. In our actual implementation, if a program location is hit multiple times (say in a loop or a procedure), we use only the first few (bounded) instances when the location is reached as possible transition points.

```
int approxModelCount(oldPred,newPred){
 /* "sample" is a set of inputs constructed by
  * uniform sampling on the path condition */
 while(inp = sample.next()){
  if(oldPred[inp]) countOld++;
  if(newPred[inp]) countNew++;
 }
 return (countNew < countOld);
}
```

Listing 2: Approximate model counting

Algorithm 2 Synthesis Algorithm

procedure SynthesizeETS(sym_pc, Λ, C, V)
  for all $l_i \in L$ **do**
    $pred[i] \leftarrow$ true
    $sols[i] \leftarrow \{\}$
  end for
  $tries \leftarrow 0$
  while $tries < MAX\_TRIES$ **do**
    $tries \leftarrow tries + 1$
    for all $l_k \in L$ **do**
      $\Psi \leftarrow (v_1\ op\ v_2)^k \wedge \prod_{\vec{\beta} \in sols[k]} \neg\vec{\beta}[(v_1\ op\ v_2)]$
      if IsSAT Ψ **then**
        $p \leftarrow \Psi[(v_1\ op\ v_2)]$
      else
        continue
      end if
      $\Phi \leftarrow sym\_pc \wedge \prod_{l_i \in L, i \neq k} pred[i]$
      $count_{old} \leftarrow ModelCount(\Phi \wedge pred[k])$
      $count_{new} \leftarrow ModelCount(\Phi \wedge p)$
      if $count_{new} < count_{old}$ **then** $pred[k] \leftarrow p$
      $sols[k] \leftarrow sols[k] \cup \Psi[\alpha_1, \ldots, \alpha_n]$
    end for
  end while
  return $pred$
end procedure

```
L0:
/*state = 6*/
make_symbolic(a, z);
if(a > 42) c = sqrt(z);
if(c >10){
L6:
/*if(c < a && state == 6) state = 9*/
}
L10:
/*if( state == 9) assert(0)*/
```

Listing 3: Problem of path divergence

3.6 Embed the synthesized ETS in the program

In the final phase, Apocalypse embeds the synthesized ETS in the program by instrumenting the transition locations with guarded state transitions as per the synthesized ETS. Listings 4 and 5 show two possible instrumentation schemes: Listing 5 is a better scheme as it avoids creating path explosion, and hence, creates buggy programs that are closer to the input program in terms of the total number of paths.

Different instrumentation scheme can be adopted to camouflage the ETS transitions: for example, Listing 6 for string programs and Listing 7 for bit manipulating programs. As our current prototype was meant to study the properties of our injected bugs for automated bug detection systems (and not human subjects), all our experiments were conducted on the instrumentation scheme shown in Listing 5.

3.7 Running Example

To begin with, Apocalypse needs to be provided with a *seed input* that drives the program through an path on which we are interested in inducing an error condition; a *good* path for error-injection can be discovered by symbolic execution (see §3.2). Let us assume that in this case we select the seed inputs as (`curSep=1258, ownAlt=897,`

```c
int state;

void buggy(){
if (p1 && state == 0)
 state = 5;
if (p2 && state == 5)
 state = 9;
if (p3 && state == 9)
 state = -1;
if (p4 && state == -1)
 crash();
}
```

Listing 4: ETS encoding for integer programs

```c
int state;

void buggy(){
 state += 3*(p1 * !(state - 0));
 state += 5*(p2 * !(state - 3));
 state += 3*(p3 * !(state - 8));
 state += -11*(p4 * !(state - 11))

 if (state == 0)
  crash();

}
```

Listing 5: ETS smart encoding for integer programs

```c
char str[100] = "hello\0world\0for\0bug"
void buggyFunction(){
 if(p1 && (cmp(str+loc,"hello") == 0))
  loc += (strlen(str)+1);
 if(p2 && (cmp(str+loc,"world") ==0))
  loc += strlen(str);
 if(p3 && (cmp(str+loc,"for") ==0))
  loc += strlen(str);
 if(p4 && (cmp(str+loc,"bug") ==0))
  crash();
}
```

Listing 6: ETS encoding for string based programs

```c
int state = 0xffff0011;
void buggy(){
 if(p1 && (state
       & 0x0000ffff == 3))
  state |= 0xff000022;
 if(p2 && (state
       & 0x0000ffff == 0))
  state |= 0xff00ff00;
 if(p3 && (state
       & 0x0000ffff == 3))
  crash();
}
```

Listing 7: ETS encoding for bit manipulating programs

`ownRate=174, otherAlt=7253, altVal=1, upSep=629, downSep=5000, otherRAC=0, climb=1`).

To embed an *Error Transition System* (ETS) along this path, Apocalypse also needs a set of *good* program locations to drive the ETS transitions (see §3.3). Our system identifies the lines marked as `L16`, `L6`, `L19` and `L21` as the *transition* locations.

Armed with the seed inputs and the set of transition locations, Apocalypse runs symbolic execution along the seed path to collect the symbolic path condition (*sym_pc*), and the symbolic and concrete expression maps (Λ and C) (see Table 3).

Apocalypse, now, synthesizes an ETS as follows: for the location L16, it finds a predicate `othCap < climb` to move the transition system by a step. It does so by building a **synthesis constraint** that ensures that the predicate is simple, non-trivial (disallowing predicates like (`upPref` $\geq$ `downSep`) that are invariants) and useful (discussed next). Similarly, it synthesizes predicates (`ownRate < otherAlt`), (`upPref < upCros`) and (`othCap < climb`) for locations `L6`, `L19` and `L21`.

Next, Apocalypse makes more passes over these locations in a search for better predicates. Attempting another synthesis cycle over L19 (and disallowing the previous solution), it synthesizes a new predicate (`climb == result`). Now, it checks the model count for:

$sym\_pc : (\alpha_{10} \neq -1) \land (\alpha_{11} \neq 0) \land (\alpha_6 + \alpha_8 > 5400) \land (\alpha_5 - \alpha_3 > 0) \land (\alpha_8 \neq 0) \land (\alpha_6 \neq 0)$

$\Psi_1 : sym\_pc \land (\alpha_{11} == \alpha_6) \land (\alpha_{10} < \alpha_{11}) \land (\alpha_0 + \alpha_4 < \alpha_5) \land (\alpha_5 < \alpha_3 + \alpha_5)$, and,

$\Psi_2 : sym\_pc \land (\alpha_6 + \alpha_7 + \alpha_8 + 100 < \alpha_3 + \alpha_5) \land (\alpha_{10} < \alpha_{11}) \land (\alpha_0 + \alpha_4 < \alpha_5) \land (\alpha_5 < \alpha_3 + \alpha_5)$.

In this case, it finds that the model count of Ψ_2 is smaller than that of Ψ_1, and hence it goes about replacing the older (`upPref < upCros`) by the newer (`climb == result`) predicate. On the other hand, if the count of Ψ_1 was smaller, it would have rejected it and persisted with the older predicate. This hill climbing over the multivariate space of predicates at the different locations allows us to "shrink" the space of inputs that would trigger the bug. Table 1 shows the set of all predicates produced by Apocalypse, with the one finally selected marked in *blue*. In our experiments, this procedure increased the bug detection time of the injected bugs by about 390× (on AFL). The synthesized ETS is shown in Figure 1.

Finally, the generated ETS is inserted into the existing code. We show the statements injected by Apocalypse as comments (in *green*) in Listing 1; these statements drive the program to a crash at `L30`.

Table 1: Synthesized predicates at ETS locations

Loc	Predicates	Loc	Predicates
L16	othCap < climb	L6	ownRate < otherAlt
L19	climb == result upPref < upCros	L21	othAlt < upCros climb < othCap

Table 2: Symbolic and Concrete inputs for the trace

Variable	Value	Variable	Value
curSep	$(\alpha_0, 1258)$	ownAlt	$(\alpha_3, 897)$
ownrate	$(\alpha_4, 174)$	otherAlt	$(\alpha_5, 7253)$
altVal	$(\alpha_6, 1)$	upSep	$(\alpha_7, 629)$
downSep	$(\alpha_8, 5000)$	othCap	$(\alpha_{10}, 0)$
climb	$(\alpha_1 1, 1)$		

Table 3: Symbolic and concrete maps

Variable	Loc16		Loc6		Loc19		Loc21	
	Sym	Conc	Sym	Conc	Sym	Conc	Sym	Conc
othCap	α_{10}	0	$\alpha_6 + \alpha_{11}$	2	$\alpha_6 + \alpha_{11}$	2	$\alpha_6 + \alpha_{11}$	2
ownRate	α_4	174	$\alpha_0 + \alpha_4$	1432	$\alpha_0 + \alpha_4$	1432	$\alpha_0 + \alpha_4$	1432
climb	α_{11}	1	α_{11}	1	α_{11}	1	α_{11}	1
othAlt	α_5	7253	α_5	7253	α_5	7253	α_5	7253
up	$\alpha_6 + \alpha_7 + 100$	730						
down	$\alpha_5 + \alpha_7$	7882						
uppref					$\alpha_6 + \alpha_7 + \alpha_8 + 100$	5730	$\alpha_6 + \alpha_7 + \alpha_8 + 100$	5730
upCros					$\alpha_3 + \alpha_5$	8150	$\alpha_3 + \alpha_5$	8150
result					α_6	1	α_6	1

4 EXPERIMENTS

Apocalypse is based on multiple tools: it uses Clang [2] for instrumentation (for dynamic analysis for selecting good transition locations as well as for embedding the ETS in the program). We modified Crest [6] for running symbolic execution to collect the symbolic path conditions and the expression maps. The ETS synthesizer uses Z3 [9] for constraint solving. We use a modifed version of Boolector [23] to create SAT encodings of SMT constraints, and use QuickSampler [13] for uniform sampling on the boolean path conditions.

For the purpose of our experiments, we insert *assert(false)* statements at our bug injection points. Our experiments were conducted on a Intel Xeon(R) which has 2GHz clock frequency machine with 12 cores and 32GB main memory. To understand the quality of the bugs injected by Apocalypse, we attempt to uncover the injected bugs using two popular bug finding techniques:

- **Symbolic Execution**: We use the state-of-the-art symbolic execution engine KLEE [7] to unearth the bugs. KLEE is run with the default search strategy within a timeout of 1 hr.
- **Greybox Fuzzing**: Coverage-guided fuzzing tools perform executions on randomly mutated inputs, guided by coverage

metrics. We use the popular fuzzer AFL for our experiments, running it with default settings and a timeout of 1 hr.

Our experiments attempt to answer the following research questions:

RQ1 Are our automatically injected bugs realistic?
RQ2 Is there any correlation between multiple injected bugs?
RQ3 Are the bugs injected by Apocalypse reproducible?
RQ4 Are our bugs deeper and rarer than manually seeded bugs?
RQ5 What is the effect of sampling on the difficulty of an injected bug?
RQ6 How does Apocalypse compare with state-of-the-art bug injection tools?

4.1 RQ1: Realism of our injected bug

We demonstrate that our bugs are realistic by using two state-of-the-art bug detection tools, KLEE (based on symbolic execution) and AFL (employing greybox fuzzing) on discovering the bugs injected by Apocalypse. We injected bugs in the GNU Coreutils programs [4]. We use Apocalypse to inject four bugs in each program; we then used KLEE [7] and AFL [1] for one hour each to discover the bugs. The results are shown in Figure 3.

The first bar for each benchmark shows the time spent by AFL to hit each of the bugs, normalized to the time taken to reach the last bug (or timeout when no bug is found). The second bar for each benchmark shows the number of test cases generated by KLEE till we find the *first* test that reveals a bug, normalized to the number of tests required to reach the last bug.

For example, for the experiments on KLEE (second bars), in the program `cat`, KLEE is able to find only one bug (so the bar for the first bug reaches all the way to one). In cases where all bugs are found, for example `df`, the first bug is found after 15% of the total time, the second bug is found at 32%, the third is found at 64% of total testcases till we found last bug. Cases where we are are unable to find any bug, like `test`, are shown as timeout.

For the experiments involving AFL (first bars): Because AFL generates a test only for a failing execution, we show the amount of time spent to reach a bug (instead of the number of tests). The first bar of each cluster shows how much relative time AFL has invested to find the each bug compared to the last bug. If AFL is not able to find any bug, the whole bar is set to “timeout”. For example, in the case of `cat`, the first bug is found at 2%, the second bug is found at 7%, and the third is found at 46% of the time at which we found the final bug.

Overall, KLEE could find 31% of the bugs while AFL found 38.33% of the bugs; 36% of the bugs was found only by one of the two tools while 47% of the bugs could not be found by either. This illustrates common traits exhibited by real bugs:

- **Discoverability:** State-of-the-art bug detection tools have been successful in dicovering many bugs in large programs. Even for our injected bugs, all in all, 53% of the bugs are discovered by at least one of the tools.
- **Elusiveness:** Certain bugs are still elusive, showing that these injected bugs (resembling real bugs) can now be employed to stress tools for new bug detection techniques; about 47% of the bugs could not be discovered by either of the tools.
- **Affinity to tools:** Certain bugs are more likely to be found by one type of technique than by another; 22% of the bugs were only discovered by AFL while 15% of the bugs were only discovered by KLEE.
- **Variance in tool effort:** Some bugs require more effort to be discovered than others; on the discovered bugs, AFL shows a standard deviation of 583 seconds (on a total running time of 1 hour for each program).

4.2 RQ2: Correlation of bugs injected by Apocalypse

Figure 3 shows that the number of tests (using KLEE) and the time taken (by AFL) to discover the different bug is almost uniformly distributed; also, in many cases, even after discovering a few bugs, the tools fail to unearth the rest of the bugs. This shows that there exists almost no correlation among the different bugs injected by Apocalypse. Together, KLEE and AFL are able to catch 53% of all bugs; there are 7 programs (out of 30) where none of the tools is able to catch any bug.

4.3 RQ3: Reproducibility

Because Apocalypse generates triggering inputs for each bug it creates, reproducibility is satisfied by construction. Nevertheless, we checked that the generated inputs really did trigger each bug, and found that we could reproduce all the injected bugs.

4.4 RQ4: Comparison with manually seeded bug

To compare with manually seeded bugs, we used the TCAS [11] benchmark. TCAS contains 41 buggy versions, each buggy version containing exactly one manually seeded bug. As the seeded bugs produce an incorrect output (but not a crash or assertion failure), we use KLEE to generate a set of tests; any test that that produces an incorrect output or reaches our injected bug location is designated as a failing test.

We use Apocalypse to inject two additional bugs into each TCAS version. KLEE was able to discover all the injected bugs as well as the manually seeded bugs in all versions except versions 33 and 38. Figure 5 shows the number of test cases KLEE had to generate before hitting the test case that triggers the bug. This experiment shows the elusiveness of our bugs with respect to the manually seeded ones: on an average, the bugs injected by Apocalypse require 240× more tests than the manually seeded bugs.

Table 4 shows the rarity of our bugs: this figure shows the number of generated test cases on which a bug induced a failure. On an average, the bugs injected by Apocalypse induce failures on 30× fewer tests over the manually seeded bugs.

4.5 RQ5: Searching for stronger transition predicates

Figure 4 shows how our hill climbing search for guard conditions improves the rarity of the bugs on the different versions of the TCAS program. We conducted the experiment by comparing the bugs generated when we always picked the first predicate found (red line) versus when the searcher is switched on (blue line). The

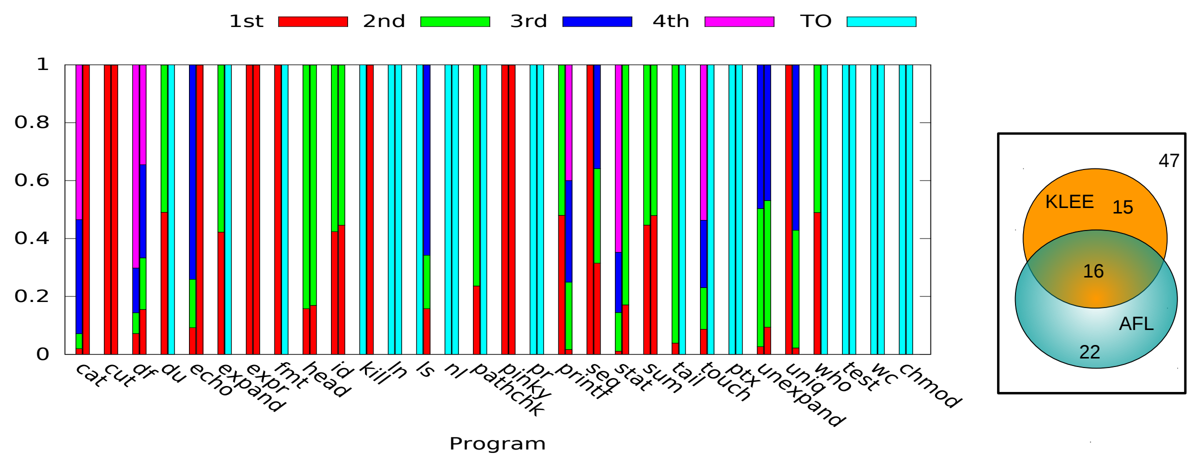

Figure 3: Normalised time or testcase to find each bug. The first bar is for AFL and the second for KLEE; the Venn Diagram shows the distribution of the bugs found by KLEE and AFL.

search for good predicates is effective as it increases the time needed to find the triggering inputs for the bugs by about 390× on average.

4.6 RQ6: Comparison with LAVA

We compare Apocalypse with LAVA on nine Coreutils programs. We discuss the results on the two tools separately:

- **KLEE**: KLEE uses constraint solving to discover newer paths. In LAVA, as the bug location is essentially guarded by a magic number, it is quite easy for KLEE to use the constraint solver to "guess" this magic number. Hence, on most of the benchmarks, KLEE is able to discover almost all bugs. On an average, KLEE discovers 80% of the bugs.
- **AFL**: AFL uses random mutations to discover test cases (guided by coverage information). Hence, AFL finds it hard to guess the magic numbers by random mutations, thereby finding many fewer bugs. On an average, AFL discovers 41% of the bugs.

As can be seen, the bugs injected by LAVA, in general, show affinity towards a certain tool (KLEE); this raises the question of realism. Over a set of injected bugs, each bug may show affinity towards a certain tool, but overall, all bugs injected by a tool should be unbiased. For the bugs injected by Apocalypse, though a certain bug may be discovered by a certain tool more easily than the other, overall both tools are almost equally effective (30% of bugs discovered by KLEE, 47% of bugs discovered by AFL on these nine programs) at discovering the bugs injected by our tool. This shows that Apocalypse injects bugs that are more "realistic" than those injected by LAVA.

5 RELATED WORK

The work most directly related to our current work is LAVA [12] and EvilCoder [24]. As we discuss the relationship of our work to these systems in detail in elsewhere in the paper, we omit a complete discussion here, noting only that while our system shares the goals of this prior work, we improve upon the state of the art

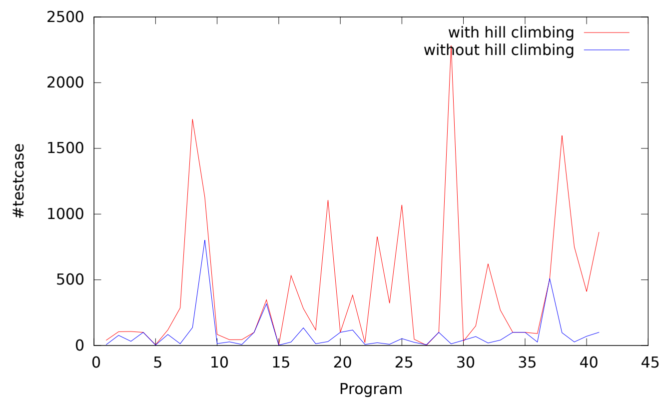

Figure 4: Effect of hill climbing approach on time to found bug using AFL in TCAS

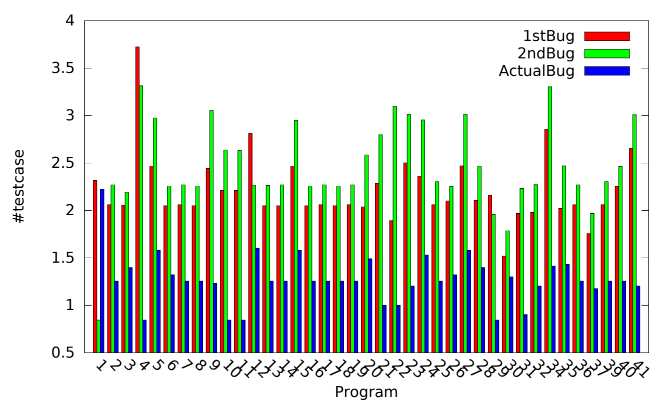

Figure 5: First testcase number on which bug is found

Table 4: Number of KLEE generated testcases that reveal our bugs (F, S) and manually seeded bug (AC). (F,S) and AC denote the number of failing testcases on our 1st (F) and 2nd (S) injected bug, and manually seeded bug.

V	(F,S)	#AC	V	(F,S)	#AC	V	(F,S)	#AC
1	(251, 1)	232	14	(1, 1)	39	27	(1, 19)	659
2	(1, 1)	160	15	(1, 19)	658	28	(2, 2)	302
3	(1, 1)	41	16	(1, 1)	43	29	(1, 1)	97
4	(4, 3)	374	17	(1, 1)	37	30	(1, 1)	56
5	(1, 19)	654	18	(1, 1)	33	31	(1, 1)	35
6	(1, 1)	30	19	(1, 1)	45	32	(1, 12)	81
7	(1, 1)	41	20	(1, 1)	124	34	(341, 2)	1805
8	(1, 1)	39	21	(9, 1)	162	35	(2, 2)	273
9	(117, 1)	883	22	(1, 1)	172	36	(1, 1)	429
10	(1, 1)	418	23	(1, 1)	424	37	(5, 5)	24
11	(1, 1)	1110	24	(156, 1)	937	39	(1, 1)	209
12	(1, 13)	1783	25	(1, 1)	213	40	(1, 1)	345
13	(1, 1)	310	26	(1, 1)	291	41	(1, 1)	577

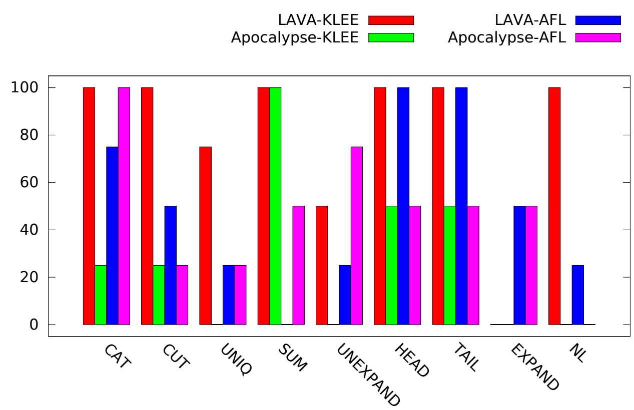

Figure 6: % of Apocalypse injected and LAVA bug found by KLEE and AFL

by generating deep, realistic bugs that do not favor a particular bug-finding approach.

Bug injection systems are intended to automate the creation of bug corpora; however, it should be noted that there are a number of existing public corpora of buggy programs as well, and studies have been performed to evaluate bug-finders using these systems. For example, Wilander and Kamkar [30, 31] performed a pair of studies using synthetic bugs that evaluated the effectiveness of static and dynamic bug-finding tools. NIST's SAMATE group hosts a collection of buggy software data sets known as the Software Assurance Reference Datasets (SARD) [5]. And in 2016, DARPA hosted an automated bug-finding Cyber Grand Challenge (CGC) [3]; this competition resulted in a collection of 247 programs with known vulnerabilities and triggering inputs, and has been used extensively since its creation for evaluating new bug-finding techniques [26, 28, 29]. The CGC corpus is very high quality, but it is expected that bug-finding software will eventually improve to be able to find all known bugs in the 247 programs. And all of these corpora suffer from one or more of the following issues: they contain few programs or each individual program is small, the bugs may be shallow or unrealistic, or the bugs may not come with triggering inputs.

The field of *mutation testing* [10, 17–19], in which random *mutation operators* are applied to a program. The resulting (presumably incorrect) program is then run against its test suite in an attempt to judge the robustness of the test suite. In some sense, bug injection is an extension of mutation testing, in that it automatically creates buggy versions of a program. However, the effects of the mutants created by mutation testing are difficult to predict, and do not come with triggering test cases—in other words, they fail to be reproducible (under the definition given in Section 1). And while mutation testing is good for evaluating the quality of a test suite, it is less clear how to apply it to the task of evaluating effectiveness of a bug-finding system such as KLEE [7].

Finally, our bug injection is based on the core techniques of *program synthesis* and *model counting*. Techniques for automatically generating programs have a long history (dating back perhaps as early as 1957, if one includes Church's discussion of the problem of circuit synthesis [8]), but have recently seen a flurry of activity due to the emergence of fast SAT and SMT solvers combined with the work of Solar-Lezama [27], which showed that program synthesis could be cast as a in terms of satisfiability. Since then, program synthesis has been applied to a wide variety of problems, including automating string processing in spreadsheets [16], heap-manipulations [25] and automated program repair [20, 22]. Model counting [13] and uniform sampling [21] have elicited a huge interest due to their applications in bayesian inference and probabilistic programming. Model counting has been successfully employed for probabilistic symbolic execution [14] that assigns probabilities to program paths to aid understanding. We use model counting in a similar context to synthesize a low probability path for the buggy executions.

6 DISCUSSION

We show some of the bugs Apocalypse inserted in a few of the coreutils programs. As can be seen, the predicates are quite non-trivial, spanning array accesses, pointer deferences and access to fields of aggregate structures. We believe that a large corpus of injected bugs will help us better understand the pecularities and relative merits of different bug-detection tools. There exist threats to validity to our experimental results, in particular from the choice of the benchmark programs and the seed inputs. We were careful to select a large number of program and inject multiple bugs in them; nevertheless, more extensive experiments can be performed.

REFERENCES

[1] American Fuzzy Lop (AFL) Fuzzer. `http://lcamtuf.coredump.cx/afl/technical_details.txt`. ; accessed 21-Jan-2018.
[2] clang: a C language family frontend for LLVM. http://clang.llvm.org/ ; accessed 21-Sept-2017.
[3] DARPA. Cyber Grand Challenge. http://archive.darpa.mil/cybergrandchallenge/.
[4] GNU Coreutils Program. `http://sir.unl.edu/portal/index.php` ; accesed 15-Sep-2018.
[5] NIST Software Assurance Reference Dataset Project. `https://samate.nist.gov/SRD/` ;.
[6] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 443–446, Washington, DC, USA, 2008. IEEE Computer Society.
[7] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and*

```
num_input_files = argc − optind;
...
case 'r': input_reference = 1;
...
state12 += (num_input_files > input_reference )
                 * (449- 614) * ¬(state12 -614)
```

(a) ptx

```
case 'a': need_deadprocs = 1;
...
trimmed_name = extract_trimmed_name(utemp_buffer);
...
state12 += (trimmed_name[2] == need_deadprocs)
                 * ( 431 - 428 ) * ( ! ( state12 - 428 ));
```

(b) who

```
case 'd':
    output_unique = 1
...
thisfield = find_field(thisline);
...
state12 += (thisfield[1] == output_unique) *
                ( 458 - 457 ) * ( ! ( state12 - 457 ));
```

(c) uniq

```
if (*optarg == '+') from_start = 1;
...
if (!(xstrtod (optarg, &s, c_strtod) && 0 <= s))
    sleep_interval = s;
...
state12 += (*sleep_interval == from_start ) *
                ( 621 - 668 ) * ( ! ( state12 - 668 ));
```

(d) tail

```
case f:  interactive = 1;
...
ok = ((symbolic_link ? symlink (source, dest)
             : link (source, dest)))
...
state12 += ( remove_existing_files < ok ) *
                ( 136 - 112 ) * ( ! ( state12 - 112 ));
```

(e) nl

```
...
scan_arg(argv[1]);
...
scan_arg(argv[2]);
...
state12 += (argv[optind][1] == argv[++optind][2]) *
                (112 - 353)* !(state12 - 353)
```

(f) seq

Figure 7: Bugs injected by Apocalypse on some of the Coreutils programs

Implementation, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[8] Alonzo Church. Application of recursive arithmetic to the problem of circuit synthesis. Summaries of the Summer Institute of Symbolic Logic, 1957.

[9] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[10] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, April 1978.

[11] Hyunsook Do, Sebastian G. Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435, 2005.

[12] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan. Lava: Large-scale automated vulnerability addition. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 110–121, May 2016.

[13] Rafael Dutra, Kevin Laeufer, Jonathan Bachrach, and Koushik Sen. Efficient Sampling of SAT Solutions for Testing. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '18, 2018.

[14] Jaco Geldenhuys, Matthew B. Dwyer, and Willem Visser. Probabilistic symbolic execution. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 166–176, New York, NY, USA, 2012. ACM.

[15] Patrice Godefroid, Michael Y. Levin, and David Molnar. Automated whitebox fuzz testing, 01 2008.

[16] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 317–330, New York, NY, USA, 2011. ACM.

[17] R. G. Hamlet. Testing programs with the aid of a compiler. *IEEE Trans. Softw. Eng.*, 3(4):279–290, July 1977.

[18] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, Sept 2011.

[19] Richard Lipton. Fault diagnosis of computer programs. Student report, Carnegie Mellon Univ., 1971.

[20] Fan Long and Martin Rinard. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 166–178, New York, NY, USA, 2015. ACM.

[21] Kuldeep S Meel, Moshe Y Vardi, Supratik Chakraborty, Daniel J Fremont, Sanjit A Seshia, Dror Fried, Alexander Ivrii, and Sharad Malik. Constrained sampling and counting: Universal hashing meets sat solving. 2016.

[22] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: Program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 772–781, Piscataway, NJ, USA, 2013. IEEE Press.

[23] Aina Niemetz, Mathias Preiner, and Armin Biere. Boolector 2.0 system description. *Journal on Satisfiability, Boolean Modeling and Computation*, 9:53–58, 2014 (published 2015).

[24] Jannik Pewny and Thorsten Holz. Evilcoder: Automated bug insertion. In *Proceedings of the 32Nd Annual Conference on Computer Security Applications*, ACSAC '16, pages 214–225, New York, NY, USA, 2016. ACM.

[25] Subhajit Roy. From concrete examples to heap manipulating programs. In Francesco Logozzo and Manuel Fähndrich, editors, *Static Analysis*, pages 126–149, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[26] Yan Shoshitaishvili, Michael Weissbacher, Lukas Dresel, Christopher Salls, Ruoyu Wang, Christopher Kruegel, and Giovanni Vigna. Rise of the HaCRS: Augmenting autonomous cyber reasoning systems with human assistance. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 347–362, New York, NY, USA, 2017. ACM.

[27] Armando Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, University of California, Berkeley, 2008.

[28] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Network and Distributed Systems Symposium (NDSS)*, 2016.

[29] László Szekeres. *Memory corruption mitigation via hardening and testing*. PhD thesis, Stony Brook University, 2017.

[30] John Wilander and Mariam Kamkar. A comparison of publicly available tools for static intrusion prevention. In *Proceedings of the 7th Nordic Workshop on Secure IT Systems*, 2002.

[31] John Wilander and Mariam Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *Proceedings of the 10th Network and Distributed System Security Symposium (NDSS)*, 2003.