EXERCISE 29: SOLUTION

SYMBOLIC EXECUTION REVIEW

How many states are in the symbolic execution tree for the following program?

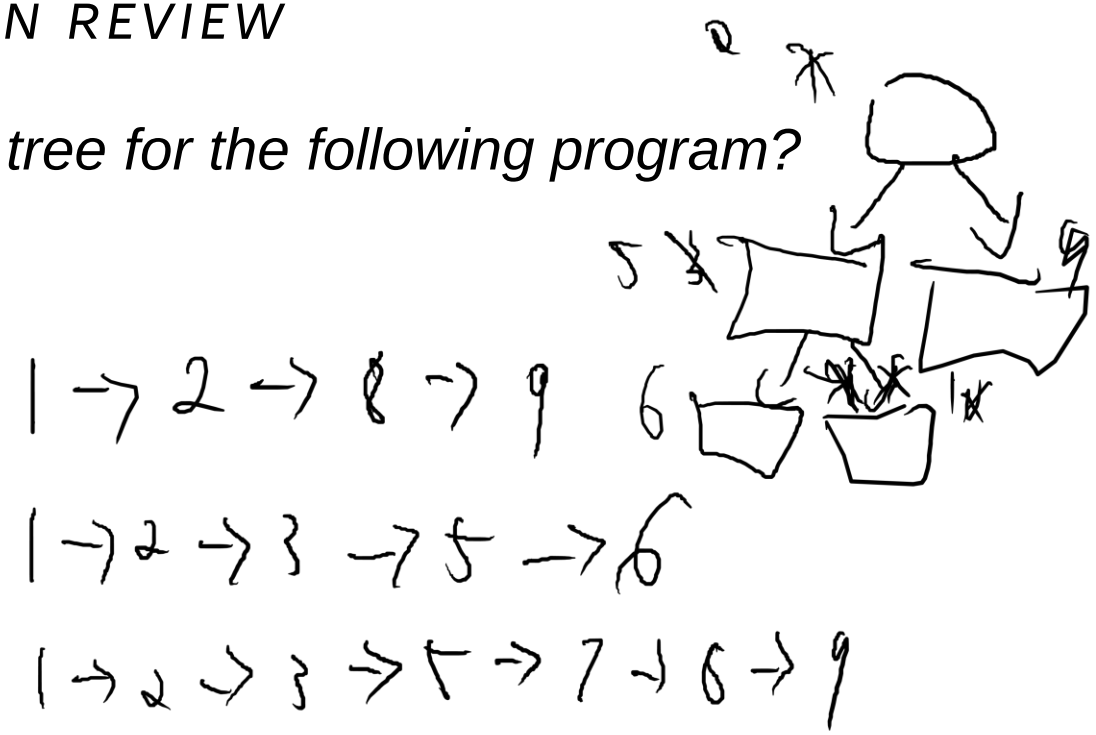
```
 1: int main(int argc){
 2:     if (argc > 2){
 3:         if (argc < 1){
 4:             return 1;
 5:         } else if (argc > 3){
 6:             return 2;
 7:         }
 8:     }
 9:     return argc;
10: }
11:
```

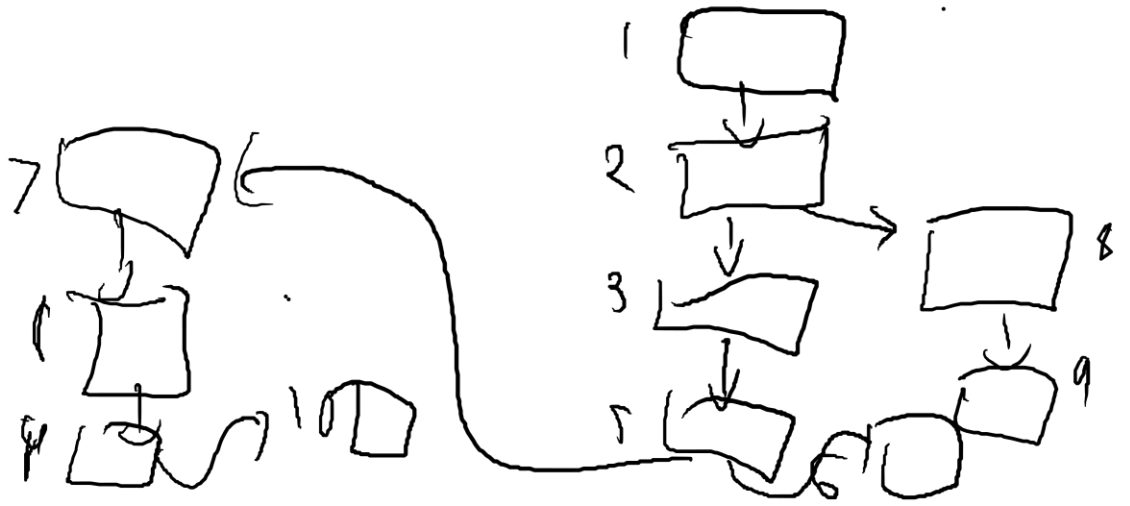
EXERCISE 29: SOLUTION

SYMBOLIC EXECUTION REVIEW

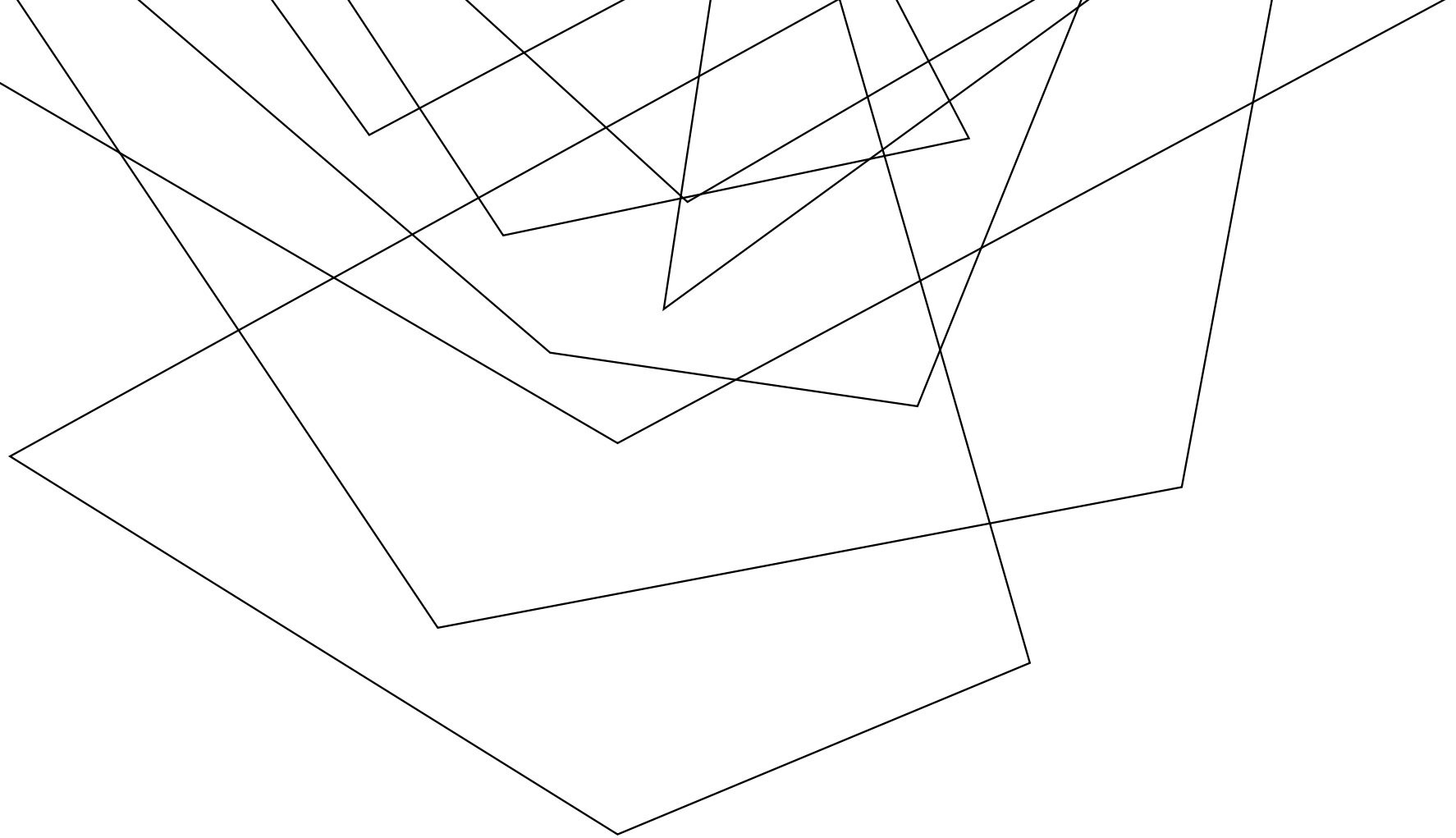
How many states are in the symbolic execution tree for the following program?

```
 1: int main(int argc){
 2:     if (argc > 2){
 3:         if (argc < 1){
 4:             return 1;
 5:         } else if (argc > 3){
 6:             return 2;
 7:         }
 8:     }
 9:     return argc;
10: }
11:
```





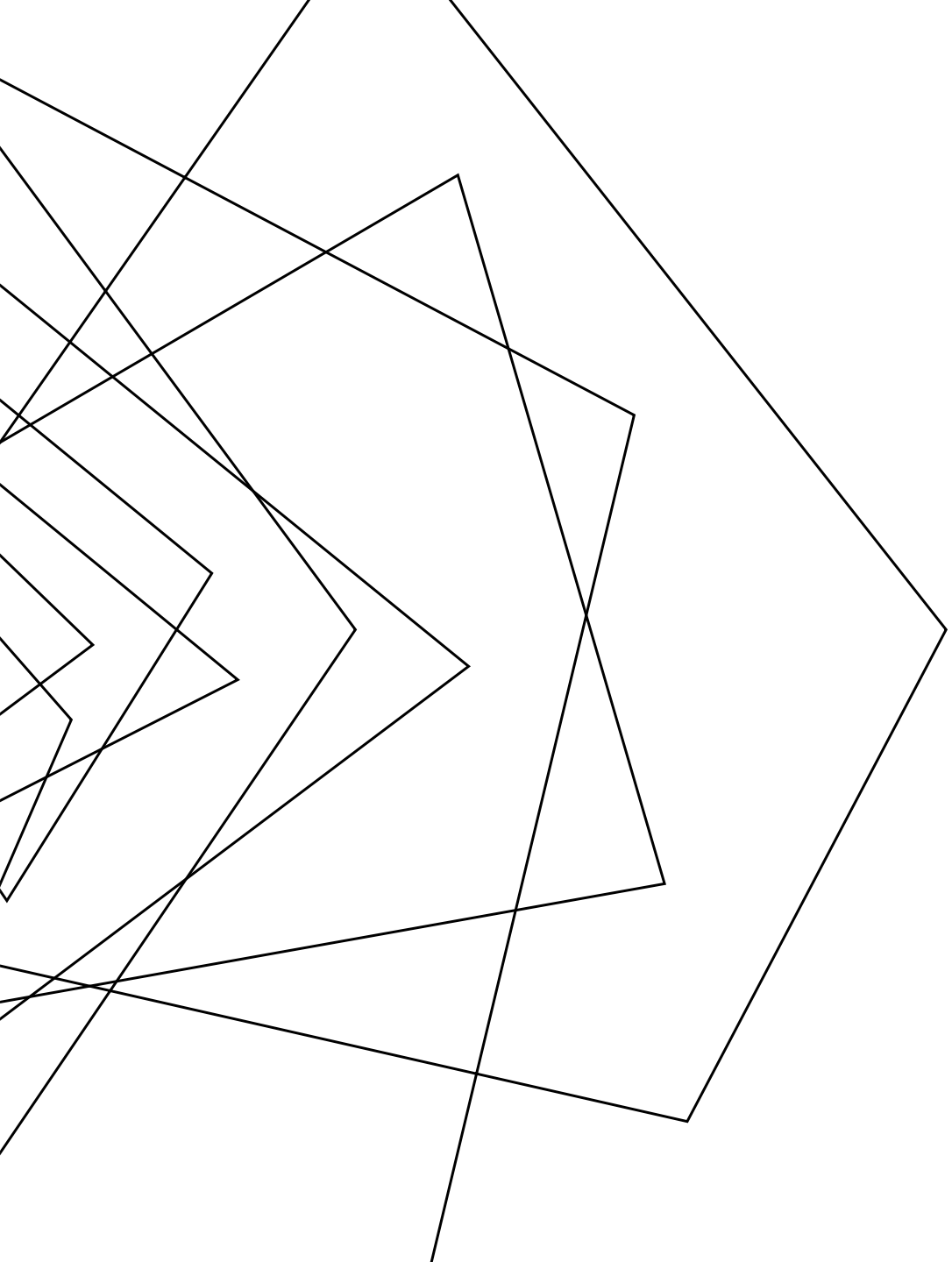
ADMINISTRIVIA AND ANNOUNCEMENTS



CONCOLIC EXECUTION

EECS 677: Software Security Evaluation

Drew Davidson



WHERE WE'RE AT

(BEYOND?) DYNAMIC ANALYSIS

- generating test cases

PREVIOUSLY: SYMBOLIC EXECUTION

OUTLINE / OVERVIEW

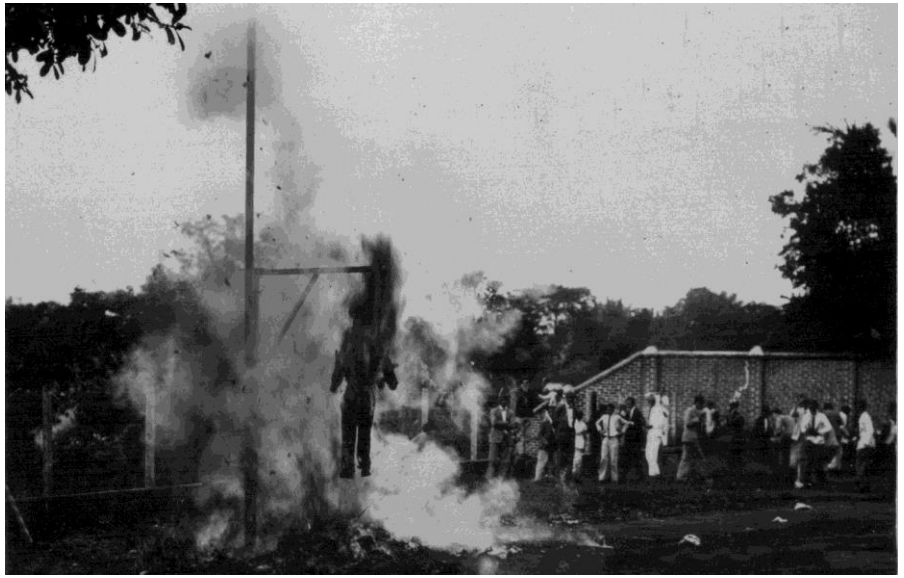
Advance Abstract States across the program

Split abstract states according to predicates to enhance coverage

Use an SMT Solver to determine if the path constraint is feasible

Sound and Complete modulo termination

Stealth limitation of testing as dynamic analysis as well

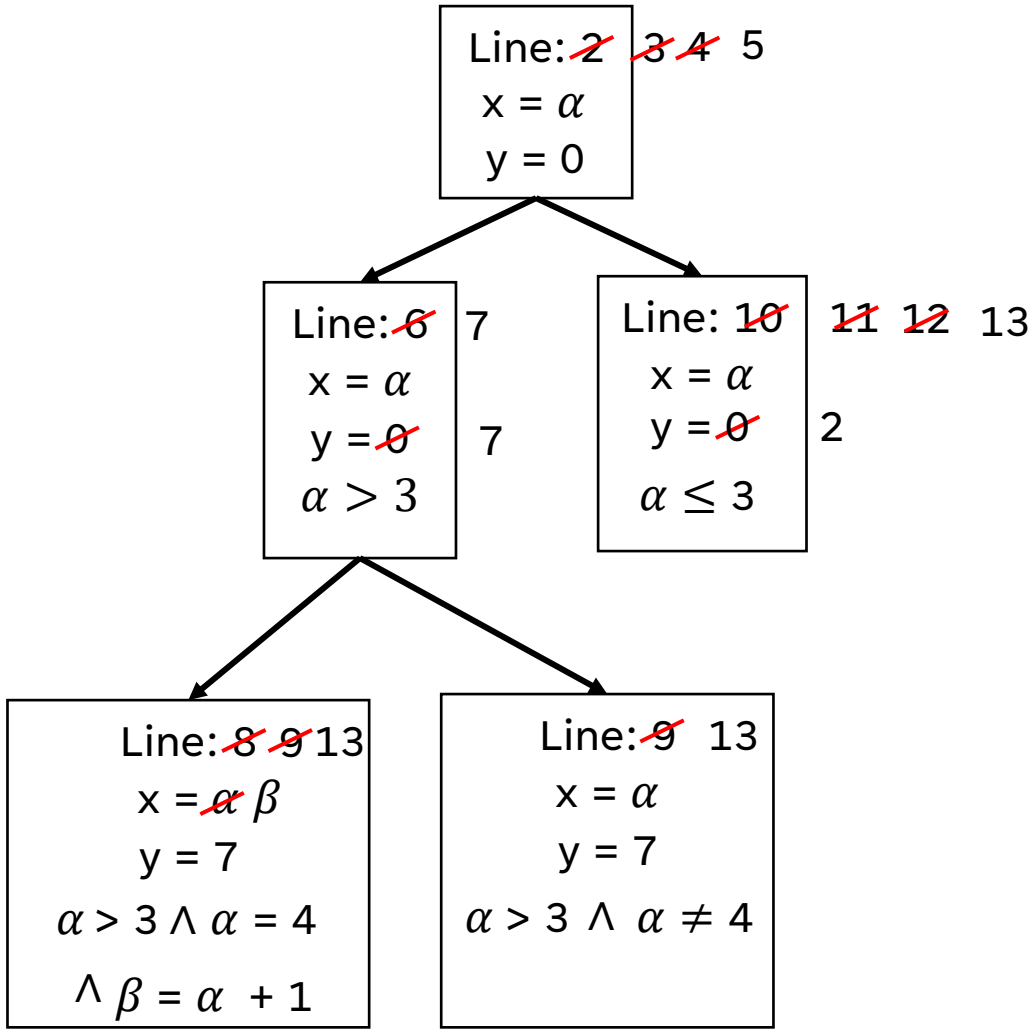


Symbolic Execution =/= Burning in Effigy

THE SYMBOLIC STATE TREE

OUTLINE / OVERVIEW

```
 1: #include "stdio.h"
 2: int main(){
 3:     int x = getchar();
 4:     int y = 0;
 5:     if (x > 3){
 6:         y = 7;
 7:         if (x == 4){
 8:           x++;
 9:         }
10:     } else {
11:         y = 2;
12:     }
13:     return y;
14: }
```



THIS TIME: ENHANCING SYMBOLIC EXECUTION

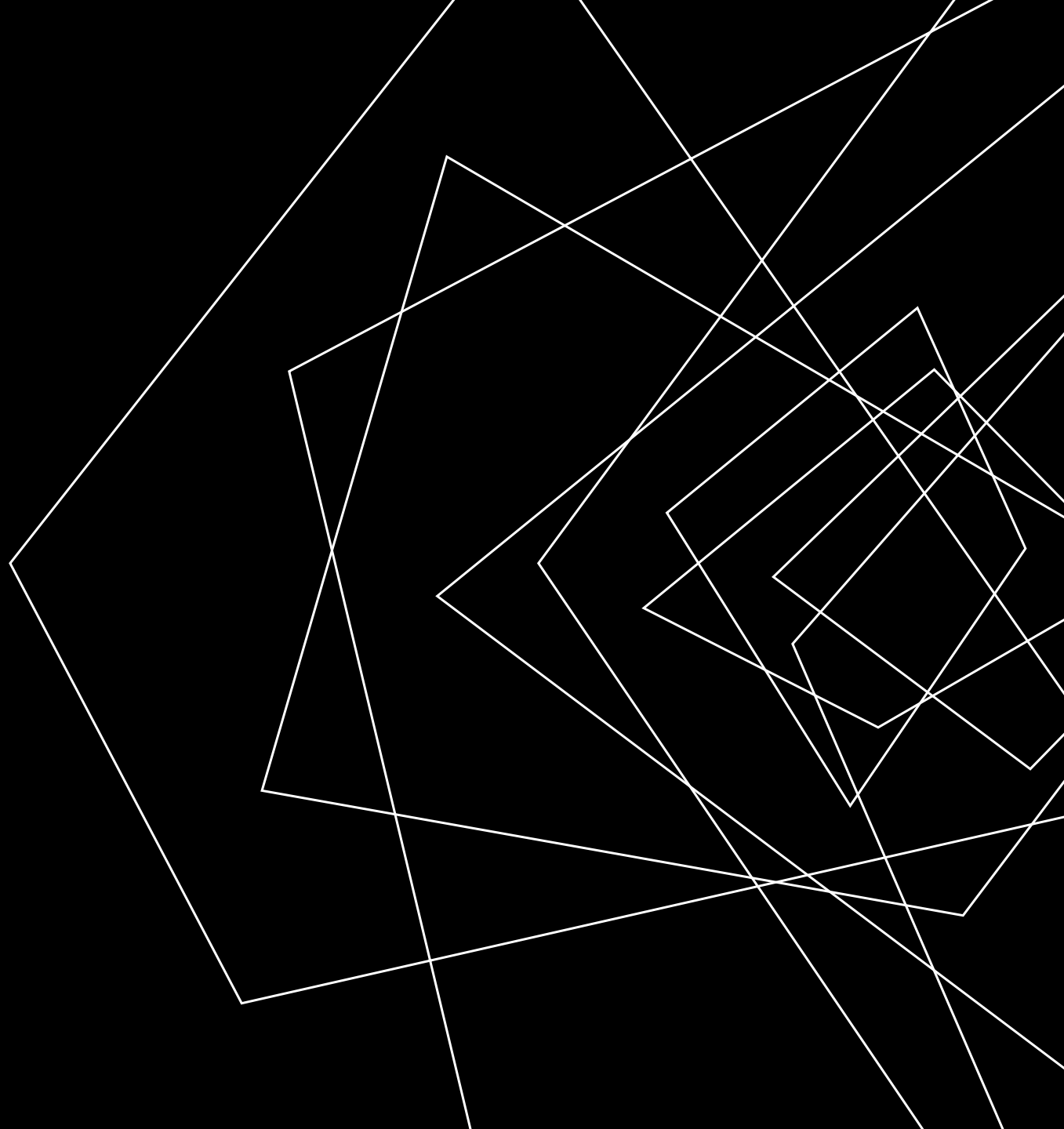
OUTLINE / OVERVIEW

TURNING THE SYMBOLIC EXECUTION CONCEPT INTO A USABLE TOOL

LIMITATIONS / IMPROVEMENTS OF THE ANALYSIS TECHNIQUE

LECTURE OUTLINE

- Generating test cases
- Analysis Termination
- Concolic Execution



GENERATING TEST CASES

CONCOLIC EXECUTION

WAIT A MINUTE... WE'RE SUPPOSED TO BE BUILDING A TEST SUITE!


```
 1: #include "stdio.h"
 2: int main(){
`3:     int x = getchar();
 4:     int y = 0;
 5:     if (x > 3){
 6:         y = 7;
 7:         if (x == 4){
 8:           x++;
 9:         }
10:     } else {
11:         y = 2;
12:     }
13:     return y;
14: }
```

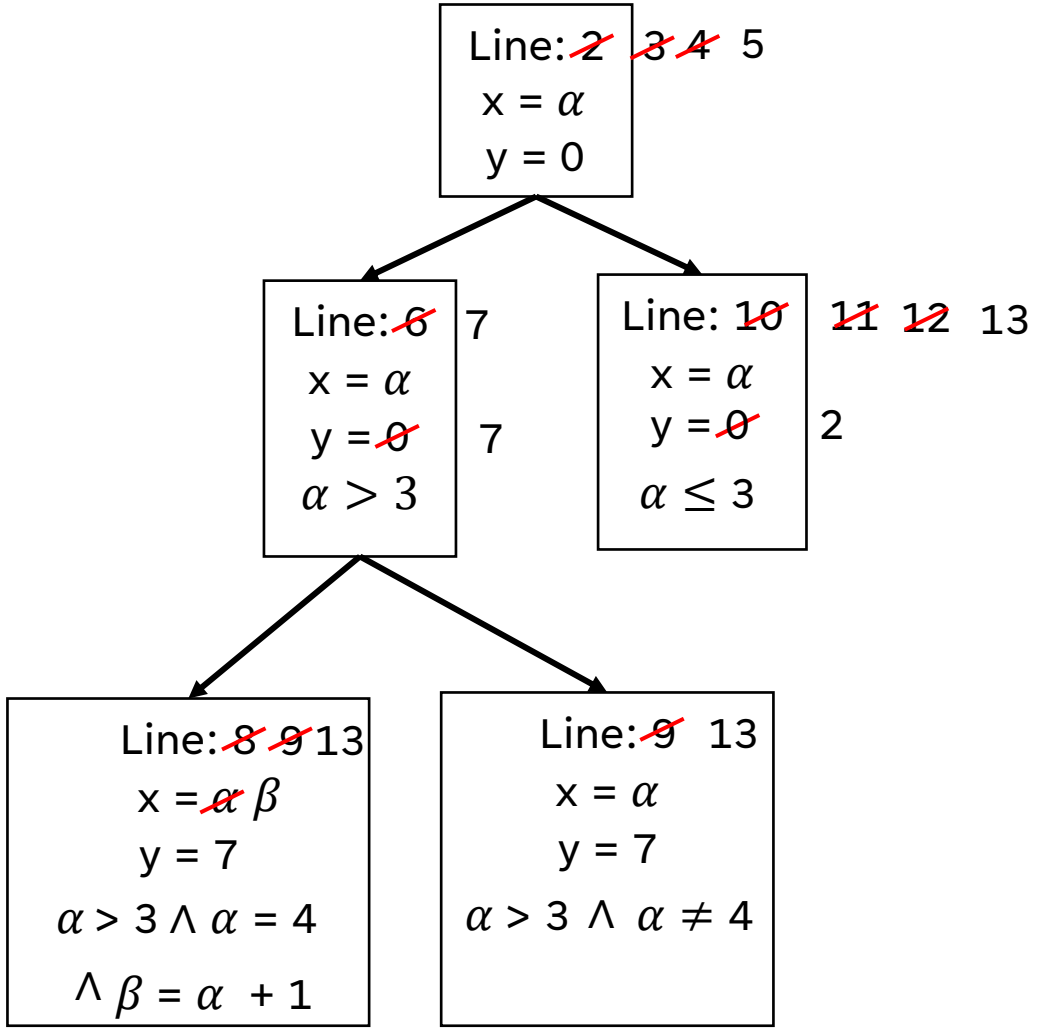
GENERATING TEST CASES

CONCOLIC EXECUTION

WAIT A MINUTE... WE'RE SUPPOSED TO BE BUILDING A TEST SUITE!

... instead, we generated a symbolic execution tree

```
 1: #include "stdio.h"
 2: int main(){
`3:     int x = getchar();
 4:     int y = 0;
 5:     if (x > 3){
 6:         y = 7;
 7:         if (x == 4){
 8:           x++;
 9:         }
10:     } else {
11:         y = 2;
12:     }
13:     return y;
14: }
```



FROM TREES TO TESTS

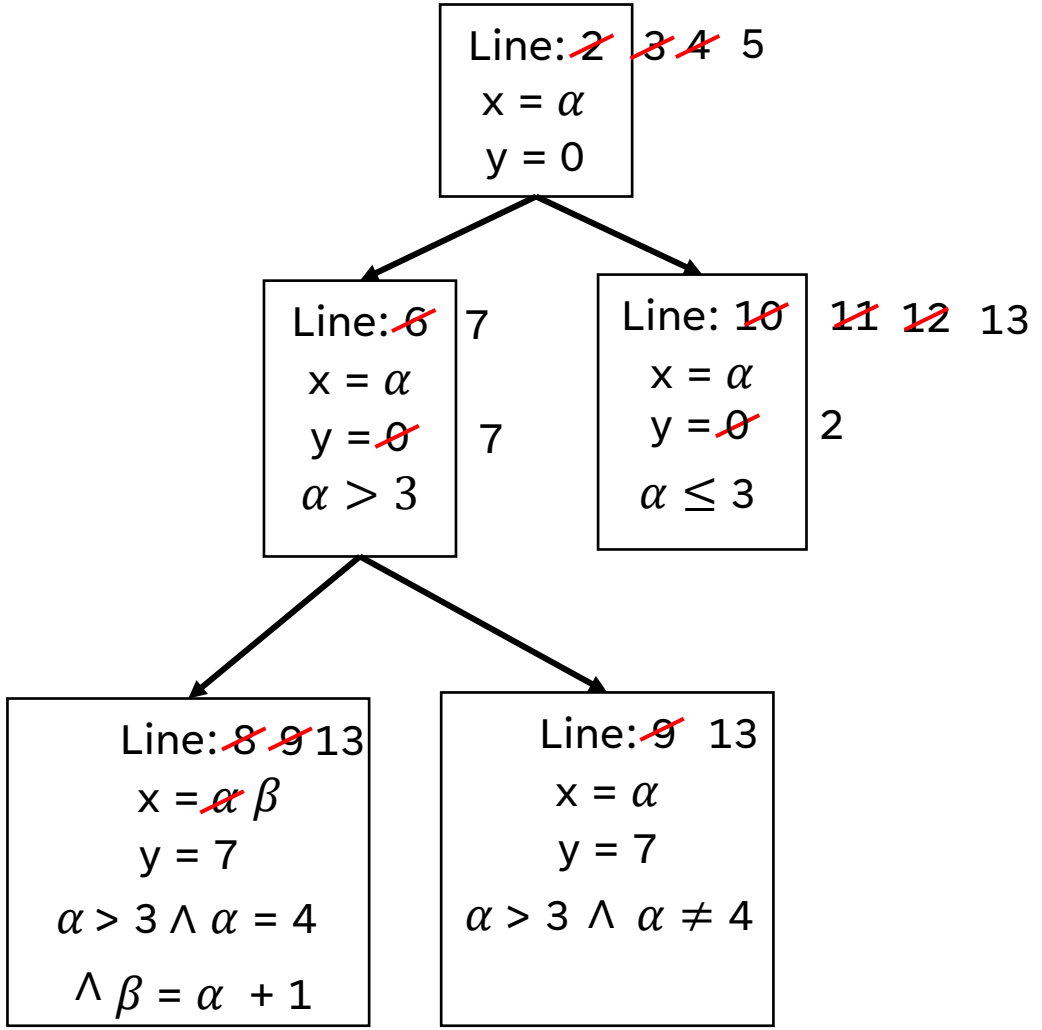
CONCOLIC EXECUTION

WAIT A MINUTE... WE'RE SUPPOSED TO BE BUILDING A TEST SUITE!

... instead, we generated a symbolic execution tree

consider a program trace

```
int X = getchar();
return 1/X;
```



FROM TREES TO TESTS

CONCOLIC EXECUTION

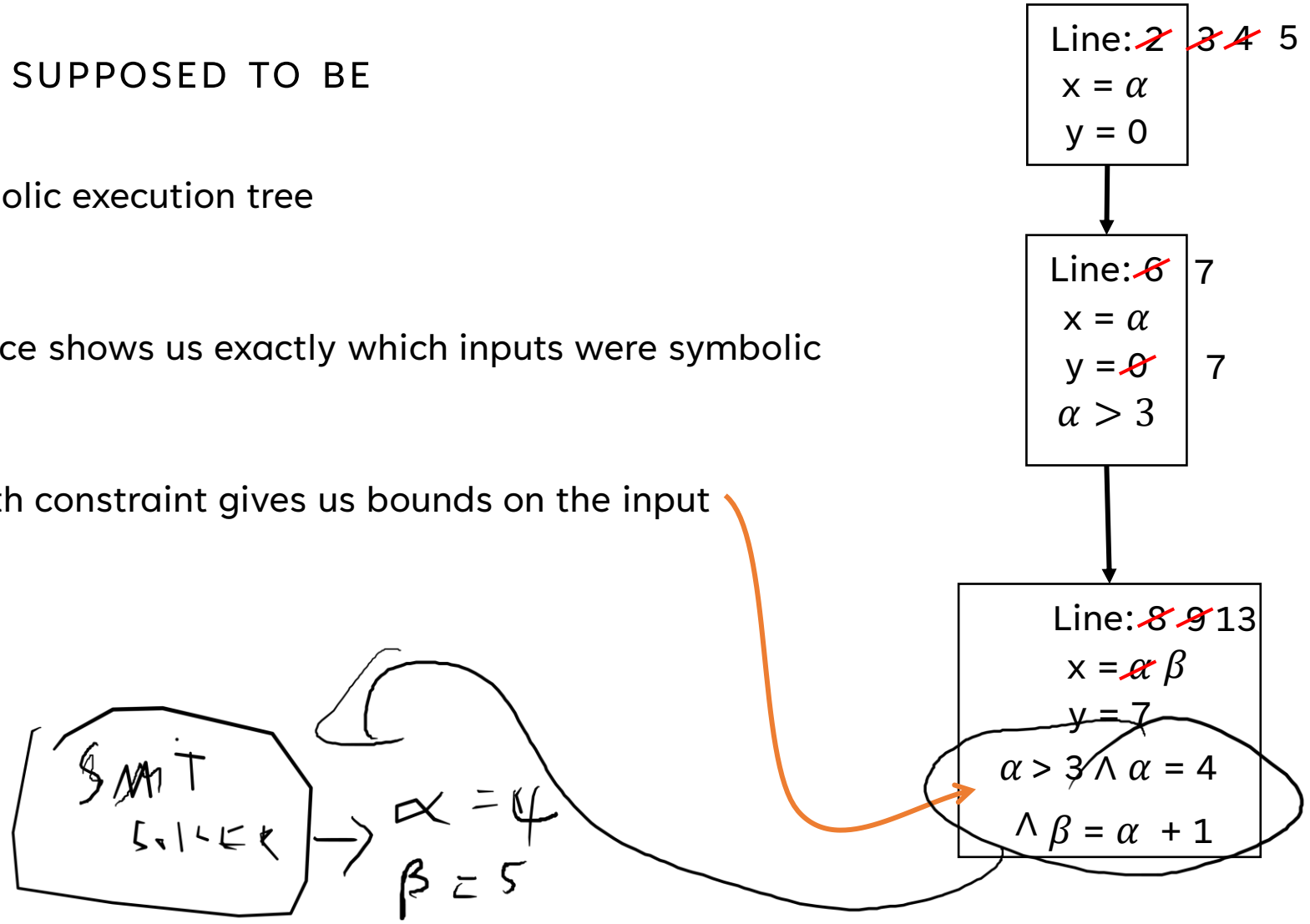
WAIT A MINUTE... WE'RE SUPPOSED TO BE BUILDING A TEST SUITE!

... instead, we generated a symbolic execution tree

consider a program trace

The trace shows us exactly which inputs were symbolic

The path constraint gives us bounds on the input

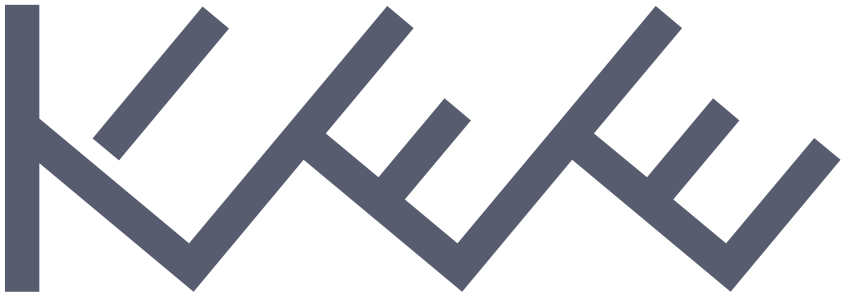


REPRESENTATIVE TOOL: KLEE

CONCOLIC EXECUTION

A STATE-OF-THE-ART SYMBOLIC EXECUTION ENGINE

http://klee.doc.ic.ac.uk/



KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs

Cristian Cadar, Daniel Dunbar, Dawson Engler *
Stanford University

Abstract

We present a new symbolic execution tool, KLEE, capable of automatically generating tests that achieve high coverage on a diverse set of complex and environmentally-intensive programs. We used KLEE to thoroughly check all 89 stand-alone programs in the GNU COREUTILS utility suite, which form the core user-level environment installed on millions of Unix systems, and arguably are the single most heavily tested set of open-source programs in existence. KLEE-generated tests achieve high line coverage — on average over 90% per tool (median: over 94%) — and significantly beat the coverage of the developers' own hand-written test suite. When we did the same for 75 equivalent tools in the BUSYBOX embedded system suite, results were even better, including 100% coverage on 31 of them.

We also used KLEE as a bug finding tool, applying it to 452 applications (over 430K total lines of code), where it found 56 serious bugs, including three in COREUTILS that had been missed for over 15 years. Finally, we used KLEE to crosscheck purportedly identical BUSYBOX and COREUTILS utilities, finding functional correctness errors and a myriad of inconsistencies.

1 Introduction

Many classes of errors, such as functional correctness bugs, are difficult to find without executing a piece of code. The importance of such testing — combined with the difficulty and poor performance of random and manual approaches — has led to much recent work in using *symbolic execution* to automatically generate test inputs [11, 14–16, 20–22, 24, 26, 27, 36]. 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 initially allowed to be "anything." They substitute program inputs with symbolic values and replace corresponding concrete program operations with ones that manipulate symbolic values. When program execution branches based on a symbolic value, the system (conceptually) follows both branches, on each path maintaining a set of constraints called the *path condition* 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 condition for concrete values. Assuming deterministic code, feeding this concrete input to a raw, unmodified version of the checked code will make it follow the same path and hit the same bug.

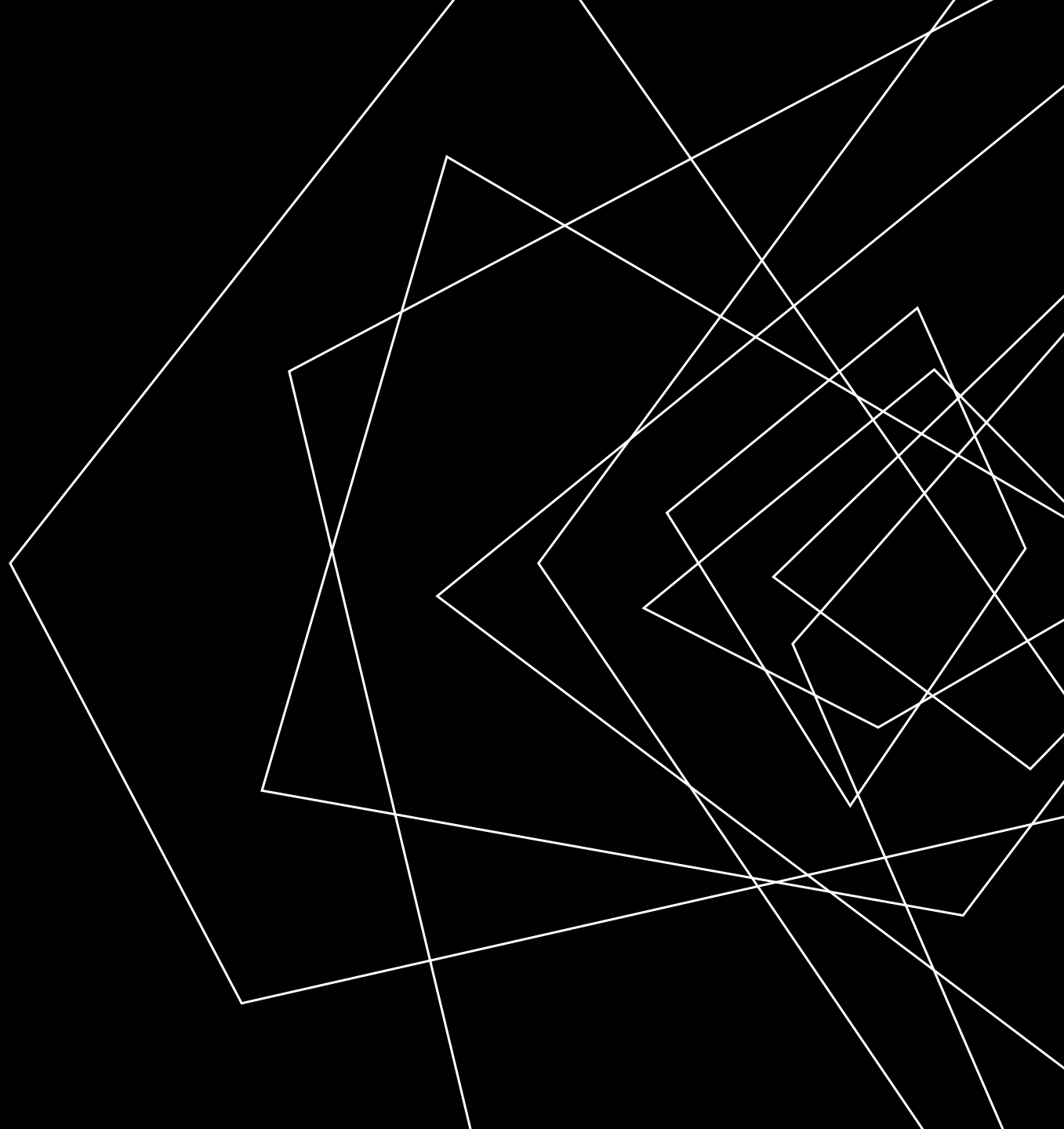
Results are promising. However, while researchers have shown such tools can sometimes get good coverage and find bugs on a small number of programs, it has been an open question whether the approach has any hope of consistently achieving high coverage on real applications. Two common concerns are (1) the exponential number of paths through code and (2) the challenges in handling code that interacts with its surrounding environment, such as the operating system, the network, or the user (colloquially: "the environment problem"). Neither concern has been much helped by the fact that most past work, including ours, has usually reported results on a limited set of hand-picked benchmarks and typically has not included any coverage numbers.

This paper makes two contributions. First, we present a new symbolic execution tool, KLEE, which we designed for robust, deep checking of a broad range of applications, leveraging several years of lessons from our previous tool, EXE [16]. KLEE employs a variety of constraint solving optimizations, represents program states compactly, and uses search heuristics to get high code coverage. Additionally, it uses a simple and straightforward approach to dealing with the external environment. These features improve KLEE's performance by over an order of magnitude and let it check a broad range of system-intensive programs "out of the box."

*Author names are in alphabetical order. Daniel Dunbar is the main author of the KLEE system.

LECTURE OUTLINE

- Generating test cases
- Analysis Termination
- Concolic Execution



TERMINATION

OUTLINE / OVERVIEW

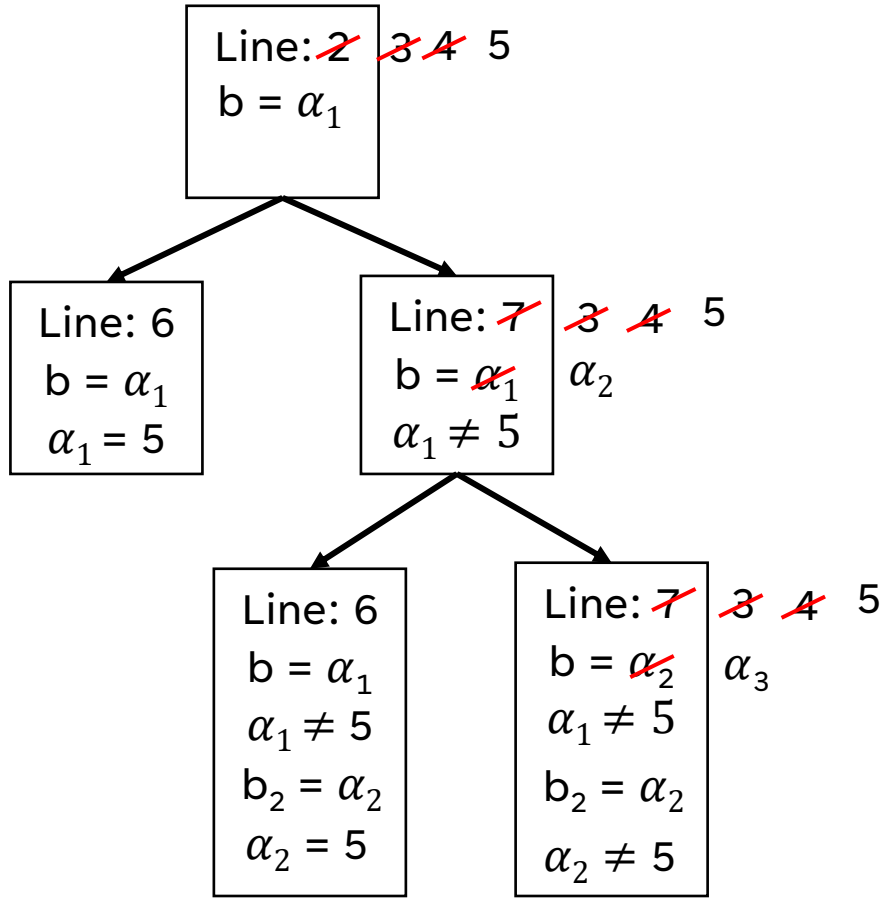
ONE ADVANTAGE OF SYMBOLIC EXECUTION:

Partial credit

WE CAN GUARANTEE TERMINATION AT THE EXPENSE OF COMPLETENESS

Quit after a threshold is met (Tree size? Clock time?)

```
1: #include "stdio.h"
2: int main(){
3:     while(true){
4:         int b = getchar();
5:         if (b == 5){
6:             return 5;
7:         }
8:     }
9: }
```



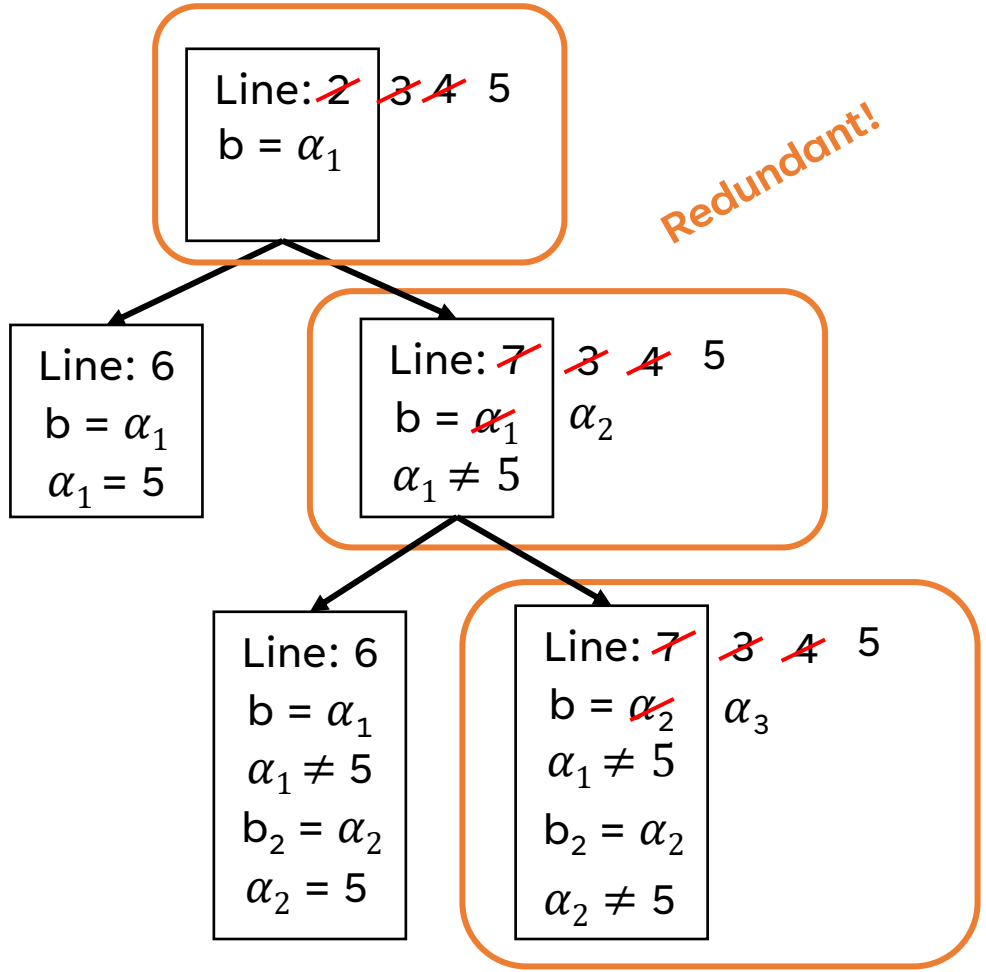
STATE PRUNING

SYMBOLIC EXECUTION

TERMINATION INSIGHT: A REDUNDANT STATE HAS REDUNDANT SUCCESSORS

* With proper environmental handling

```
1: #include "stdio.h"
2: int main(){
3:     while(true){
4:       int b = getchar();
5:       if (b == 5){
6:         return 5;
7:       }
8:     }
9: }
```



RESEARCH DIRECTION: “FIE ON FIRMWARE”

FUZZING

SYMBOLIC EXECUTION FOR “EXOTIC” ENVIRONMENTS

```
int a = *(0x400080)

int b = *(0x400080)
```

STATE PRUNING: LIMITATION

OUTLINE / OVERVIEW

SERIOUS PROGRAMS LIKELY HAVE STATE SPACE EXPLOSION

States are too complicated to prune.

```
1: #include "stdio.h"
2: int main(){
3:     int i = 0;
4:     while(int b = getchar()){
5:       i++;
6:       if (b == 5){
7:         return i;
8:       }
9:     }
10: }
```

STATE PRIORITIZATION

OUTLINE / OVERVIEW

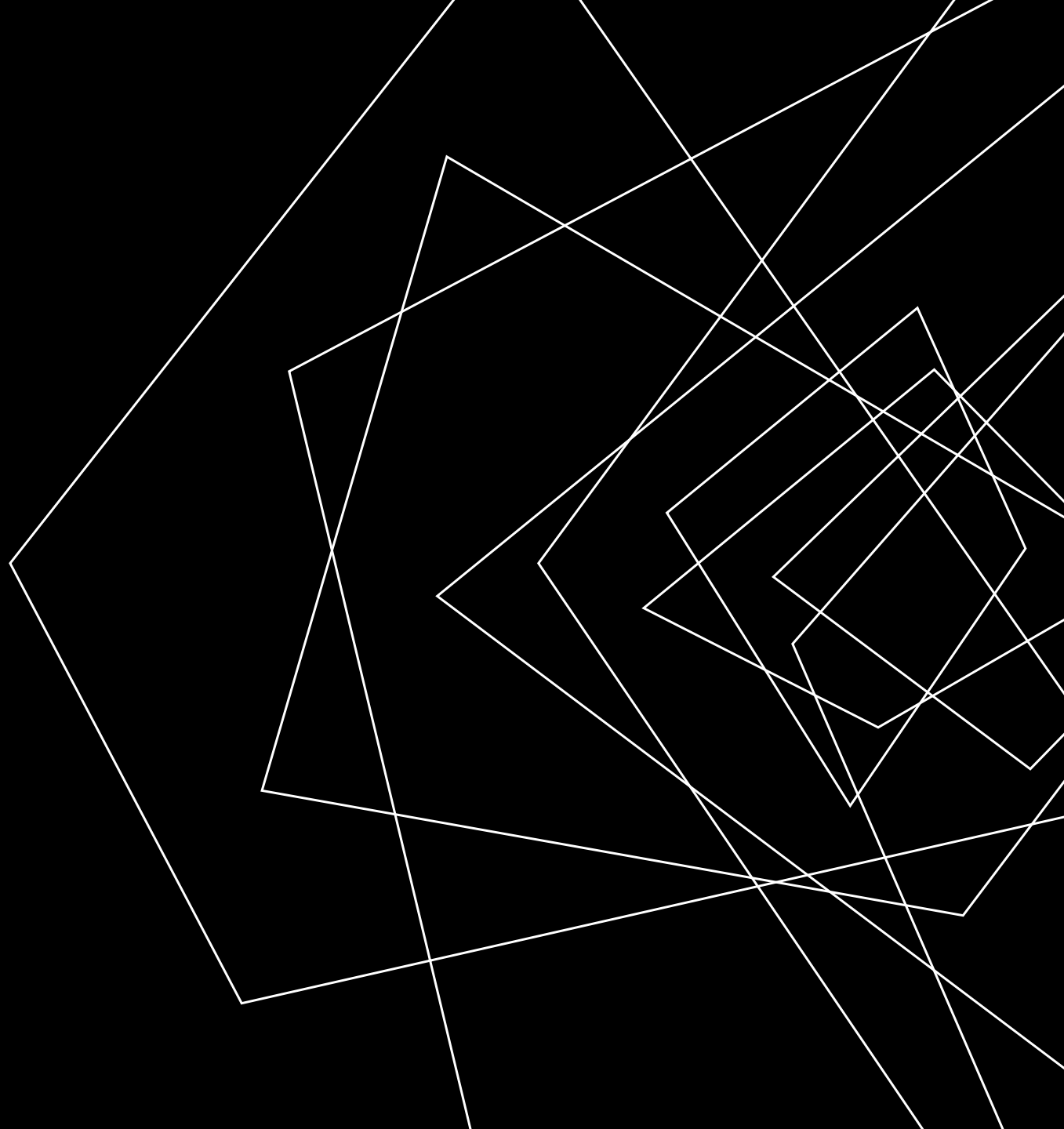
WHICH IS THE "BEST" STATE TO ADVANCE?

Akin to the fuzzing heuristics

```
 1: #include "stdio.h"
 2: int main(){
 3:     int i = 0;
 4:     while(int b = getchar()){
 5:       i++;
 6:       if (b == 5){
 7:         return i;
 8:       }
 9:     }
10:     foo();
11: }
```

LECTURE OUTLINE

- Generating test cases
- Analysis Termination
- Concolic Execution



ANALYSIS SCOPE / LIMITS

OUTLINE / OVERVIEW

OVERBURDENING THE SOLVER

At some point, the path constraint becomes unwieldly

```
1: #include "stdio.h"
2: int main(int argv, const char * argv){
3:     int i = 0;
4:     if(sha1sum(argv[1]) == 0xf572d396fae9206628714fb2ce00f72e94f2258f){
5:         return i / 0;
6:     }
7: }
```

CONCOLIC EXECUTION

OUTLINE / OVERVIEW

Concrete + symbolic

Big Idea

Replace a symbolic value with a representative concrete value

Benefits

Increased coverage (at the cost of completeness)

Can still pair with termination thresholds

```
1: #include "stdio.h"
2: int main(int argc, const char * argv){
3:     int i = 0;
4:     if(sha1sum(argv[1]) == 0xf572d396fae9206628714fb2ce00f72e94f2258f){
5:         return i / 0;
6:     }
7: }
```

CONCOLIC EXECUTION BEYOND THE SOLVER

OUTLINE / OVERVIEW

AUTOMATICALLY DETERMINE CRASH VALUES

EXE: Automatically Generating Inputs of Death

Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, Dawson R. Engler
Computer Systems Laboratory
Stanford University
Stanford, CA 94305, U.S.A
{cristic, vganesh, piotrek, dill, engler} @cs.stanford.edu

ABSTRACT

This paper presents EXE, an effective bug-finding tool that automatically generates inputs that crash real code. Instead of running code on manually or randomly constructed input, EXE runs it on symbolic input initially allowed to be "anything." As checked code runs, EXE tracks the constraints on each symbolic (i.e., input-derived) memory location. If a statement uses a symbolic value, EXE does not run it, but instead adds it as an input-constraint; all other statements run as usual. If code conditionally checks a symbolic expression, EXE forks execution, constraining the expression to be true on the true branch and false on the other. Because EXE reasons about all possible values on a path, it has much more power than a traditional runtime tool: (1) it can force execution down any feasible program path and (2) at dangerous operations (e.g., a pointer dereference), it detects if the current path constraints allow *any* value that causes a bug. When a path terminates or hits a bug, EXE automatically generates a test case by solving the current path constraints to find concrete values using its own co-designed constraint solver, STP. Because EXE's constraints have no approximations, 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 (assuming deterministic code).

EXE works well on real code, finding bugs along with inputs that trigger them in: the BSD and Linux packet filter implementations, the `udhcpd` DHCP server, the `pcre` regular expression library, and three Linux file systems.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools, Symbolic execution*

General Terms

Reliability, Languages

Keywords

Bug finding, test case generation, constraint solving, symbolic execution, dynamic analysis, attack generation.

CCS'06, October 30–November 3, 2006, Alexandria, Virginia, USA.

1. INTRODUCTION

Attacker-exposed code is often a tangled mess of deeply-nested conditionals, labyrinthine call chains, huge amounts of code, and frequent, abusive use of casting and pointer operations. For safety, this code must exhaustively vet input received directly from potential attackers (such as system call parameters, network packets, even data from USB sticks). However, attempting to guard against all possible attacks adds significant code complexity and requires awareness of subtle issues such as arithmetic and buffer overflow conditions, which the historical record unequivocally shows programmers reason about poorly.

Currently, programmers check for such errors using a combination of code review, manual and random testing, dynamic tools, and static analysis. While helpful, these techniques have significant weaknesses. The code features described above make manual inspection even more challenging than usual. The number of possibilities makes manual testing far from exhaustive, and even less so when compounded by programmer's limited ability to reason about all these possibilities. While random "fuzz" testing [35] often finds interesting corner case errors, 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. Dynamic tools require test cases to drive them, and thus have the same coverage problems as both random and manual testing. Finally, while static analysis benefits from full path coverage, the fact that it inspects rather than executes code means that it reasons poorly about bugs that depend on accurate value information (the exact value of an index or size of an object), pointers, and heap layout, among many others.

This paper describes EXE ("EXecution generated Executions"), an unusual but effective bug-finding tool built to deeply check real code. The main insight behind EXE is that code can *automatically* generate its own (potentially highly complex) test cases. Instead of running code on manually or randomly constructed input, EXE runs it on *symbolic* input that is initially allowed to be "anything." As checked code runs, if it tries to operate on symbolic (i.e., input-derived) expressions, EXE replaces the operation with its corresponding input-constraint; it runs all other operations as usual. When code conditionally checks a symbolic expression, EXE forks execution, constraining the expression to be true on the true branch and false on the other. When a path terminates or hits a bug, EXE automatically generates a test case that will run this path by solving the path's con-

ANALYSIS SCOPE / LIMITS

OUTLINE / OVERVIEW

OVERBURDENING THE SOLVER

At some point, the path constraint becomes unwieldly

THERE MAY BE CODE OUTSIDE THE ANALYSIS ENGINE

What happens on a network call?

```
1: #include "stdio.h"
2: int main(int argc, const char * argv){
3:     int i = networkRead(argv[1]);
4:     if(i == 7){
5:         return i / 0;
6:     }
7: }
```

CONCOLIC EXECUTION: BEYOND THE SOLVER

OUTLINE / OVERVIEW

AT SOME POINT, MAYBE WE JUST GUESS

Pick a random value

```
1: #include "stdio.h"
2: int main(int argc, const char * argv){
3:     int i = networkRead(argv[1]);
4:     if(i = 7){
5:         return i / 0;
6:     }
7: }
```

S2E: WHOLE-SYSTEM SYMBOLIC EXECUTION!

OUTLINE / OVERVIEW

BIG IDEA: USE THE REAL SYSTEM

Back out assumptions

S2E: A Platform for In-Vivo Multi-Path Analysis of Software Systems

Vitaly Chipounov, Volodymyr Kuznetsov, George Candea
School of Computer and Communication Sciences
École Polytechnique Fédérale de Lausanne (EPFL), Switzerland
{vitaly.chipounov,vova.kuznetsov,george.candea}@epfl.ch

Abstract

This paper presents S^{2}E, a platform for analyzing the properties and behavior of software systems. We demonstrate S^{2}E's use in developing practical tools for comprehensive performance profiling, reverse engineering of proprietary software, and bug finding for both kernel-mode and user-mode binaries. Building these tools on top of S^{2}E took less than 770 LOC and 40 person-hours each.

S^{2}E's novelty consists of its ability to scale to large real systems, such as a full Windows stack. S^{2}E is based on two new ideas: *selective symbolic execution*, a way to automatically minimize the amount of code that has to be executed symbolically given a target analysis, and relaxed *execution consistency models*, a way to make principled performance/accuracy trade-offs in complex analyses. These techniques give S^{2}E three key abilities: to simultaneously analyze entire families of execution paths, instead of just one execution at a time; to perform the analyses in-vivo within a real software stack—user programs, libraries, kernel, drivers, etc.—instead of using abstract models of these layers; and to operate directly on binaries, thus being able to analyze even proprietary software.

Conceptually, S^{2}E is an automated path explorer with modular path analyzers: the explorer drives the target system down all execution paths of interest, while analyzers check properties of each such path (e.g., to look for bugs) or simply collect information (e.g., count page faults). Desired paths can be specified in multiple ways, and S^{2}E users can either combine existing analyzers to build a custom analysis tool, or write new analyzers using the S^{2}E API.

Categories and Subject Descriptors D.2.4 [*Software/Program Verification*]

General Terms Reliability, Verification, Performance, Security

1. Introduction

System developers routinely need to analyze the behavior of what they build. One basic analysis is to *understand observed behavior*, such as why a given web server is slow on a SPECweb benchmark. More sophisticated analyses aim to *characterize future behavior* in previously unseen circumstances, such as what will a web server's maximum latency and minimum throughput be, once deployed at a customer site. Ideally, system designers would also like to be able to do quick *what-if analyses*, such as determining whether aligning a certain data structure on a page boundary will avoid all cache misses and thus increase performance. For small programs, experienced developers can often reason through some of these questions based on code alone. The goal of our work is to make it feasible to answer such questions for large, complex, real systems.

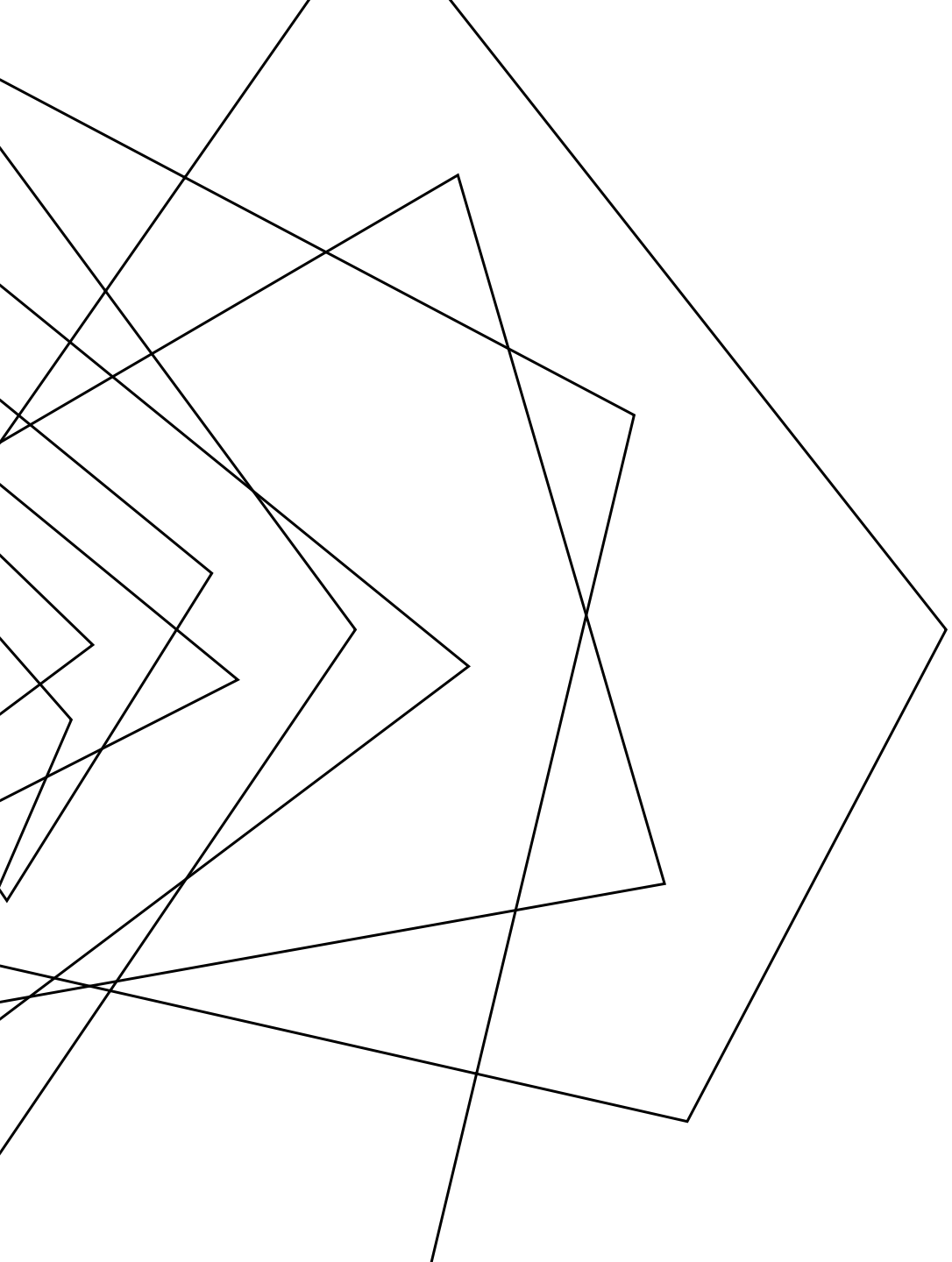
We introduce in this paper a platform that enables easy construction of analysis tools (such as oprofile, valgrind, bug finders, or reverse engineering tools) that simultaneously offer the following three properties: (1) they efficiently analyze entire families of execution paths; (2) they maximize realism by running the analyses within a real software stack; and (3) they are able to directly analyze binaries. We explain these properties below.

First, predictive analyses often must reason about entire *families of paths* through the target system, not just one path. For example, security analyses must check that there exist no corner cases that could violate a desired security policy; recent work has employed model checking [29] and symbolic execution [11] to find bugs in real systems—these are all multi-path analyses. One of our case studies demonstrates multi-path analysis of performance properties: instead of profiling solely one execution path, we derive performance envelopes that characterize the performance of entire families of paths. Such analyses can check real-time requirements (e.g., that an interrupt handler will never exceed a given bound on execution time), or can help with capacity planning (e.g., determine how many web servers to provision for a web farm). In the end, properties shown to hold for *all* paths constitute proofs, which are in essence the ultimate prediction of a system's behavior.

Second, an accurate estimate of program behavior often requires taking into account the *whole environment* surrounding the analyzed program: libraries, kernel, drivers, etc.—in other words, it requires in-vivo[1] analysis. Even small programs interact with their environment (e.g., to read/write files or send/receive network packets), so understanding program behavior requires understanding the nature of these interactions. Some tools execute the real environment, but allow calls from different execution paths to interfere inconsistently with each other [12, 18]. Most approaches abstract away the environment behind a model [2, 11], but writing abstract models is labor-intensive (taking in some cases multiple person-years [2]), models are rarely 100% accurate, and they tend to lose

ASPLOS'11, March 5–11, 2011, Newport Beach, California, USA.

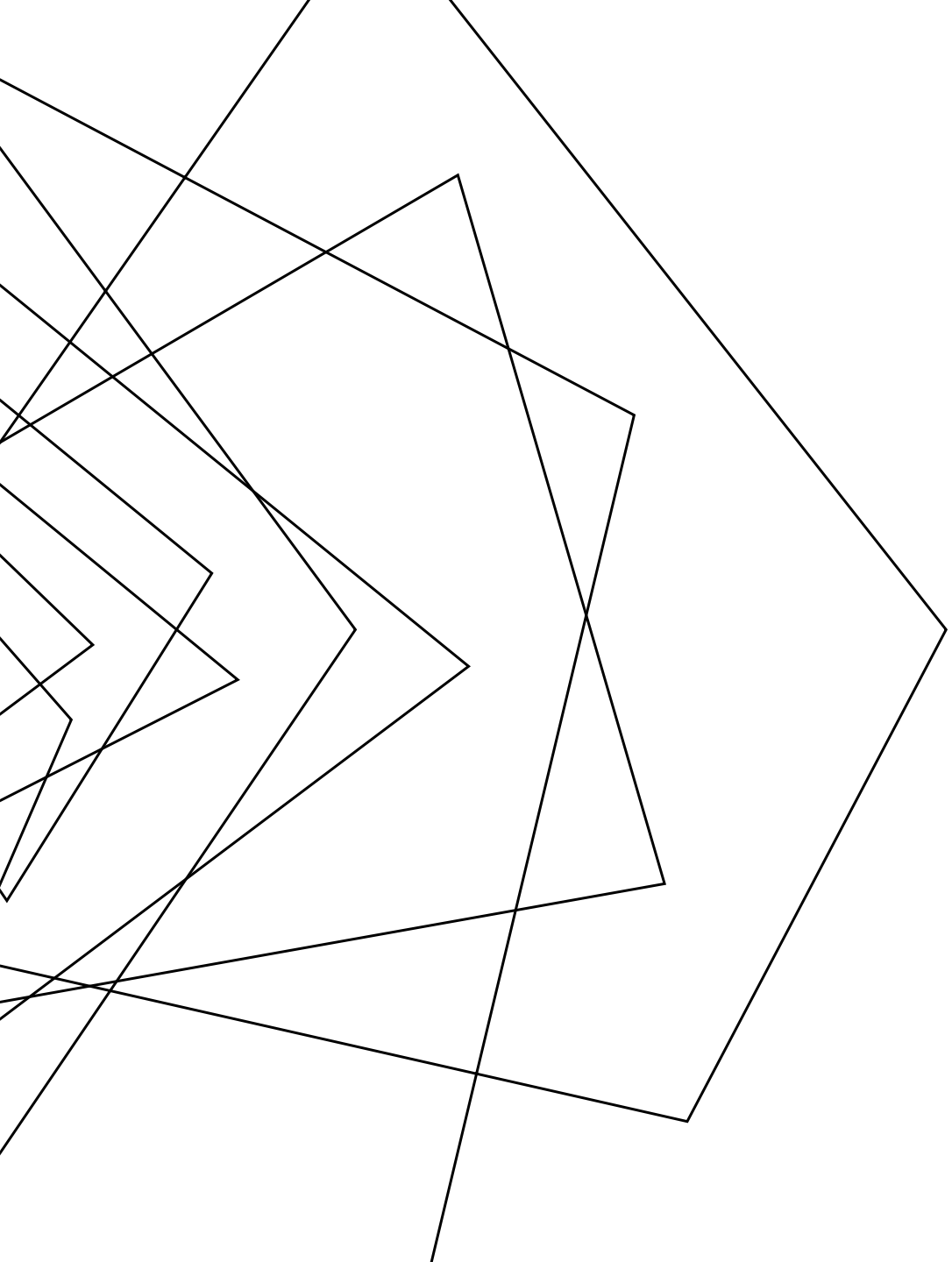
[1] *In vivo* is Latin for "within the living" and refers to experimenting using a whole live system; *in vitro* uses a synthetic or partial system. In life sciences, in vivo testing—animal testing or clinical trials—is often preferred, because, when organisms or tissues are disrupted (as in the case of in vitro settings), results can be substantially less representative. Analogously, in-vivo program analysis captures all interactions of the analyzed code with its surrounding system, not just with a simplified abstraction of that system.



WRAP-UP

SYMBOLIC EXECUTION

Exercise each feasible program path



NEXT TIME...

SAT SOLVERS

Peeking inside the magic box that determines if an equation is feasible