

A NEW COMEDY
FROM THE GUYS THAT CREATED
SHAUN OF THE DEAD

THEY'RE BAD BOYS.
THEY'RE DIE HARDS.
THEY'RE LETHAL WEAPONS.

THEY ARE...

HOT FUZZ

SIMON PEGG NICK FROST

ROGUE PICTURES PRESENTS IN ASSOCIATION WITH STUDIOCANAL A WORKING TITLE PRODUCTION IN ASSOCIATION WITH BIG TALK PRODUCTIONS
"HOT FUZZ" SIMON PEGG NICK FROST JIM BROADBENT NINA GOLD WRITTEN BY DAVID ARNOLD DIRECTED BY ANNIE HARDINE
CHRIS DICKENS MARKUS ROWLAND PRODUCED BY JESS HALL PRODUCED BY RONALDO VASCONCELLOS PRODUCED BY NATASCHA WHARTON
WRITTEN BY EDGAR WRIGHT & SIMON PEGG PRODUCED BY NIRA PARK TIM BEVAN ERIC FELLNER DIRECTED BY EDGAR WRIGHT



www.jointhefuzz.com

A NEW COMEDY
FROM THE GUYS THAT CREATED
SHAUN OF THE DEAD





**YOU CAN'T FIND
BUGS WITH SUPER
SIMPLE TECHNIQUES**

imgflip.com

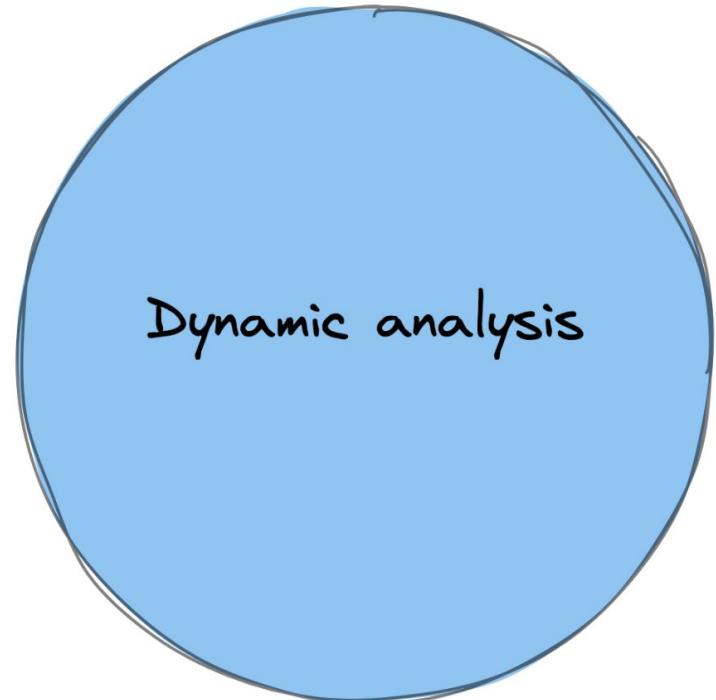
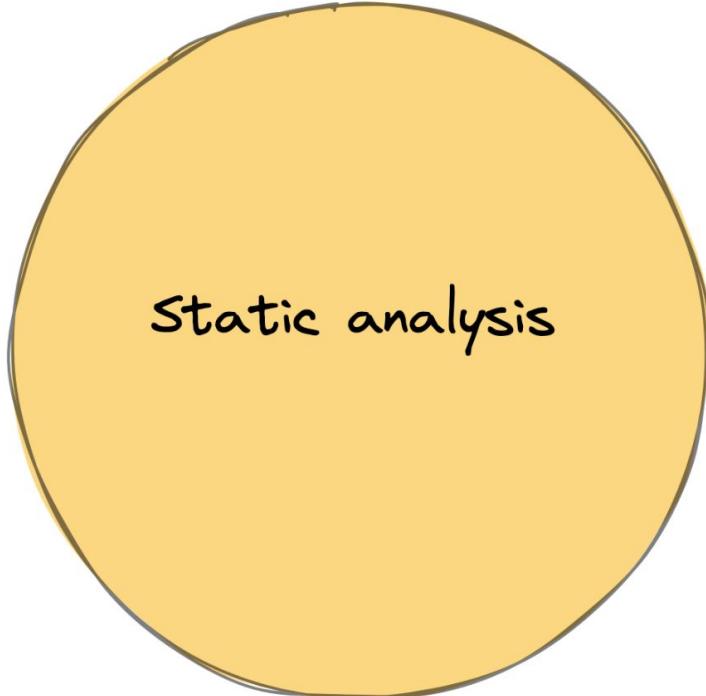


**FUZZER GO
BRRRRRRRRRR**

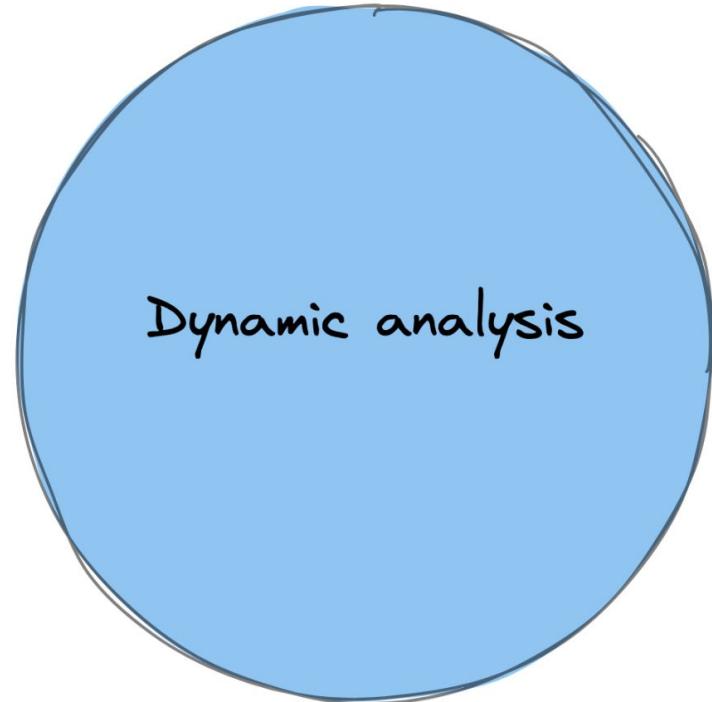
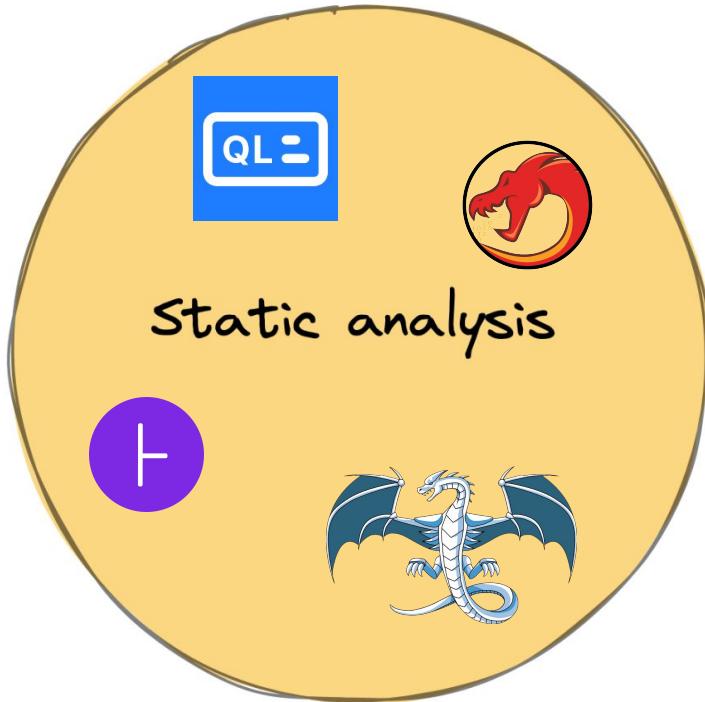
Outline

1. What is fuzzing?
2. Shades of fuzzers
 - o Black, grey, white
3. Fuzzing research state-of-the-art
4. Future directions

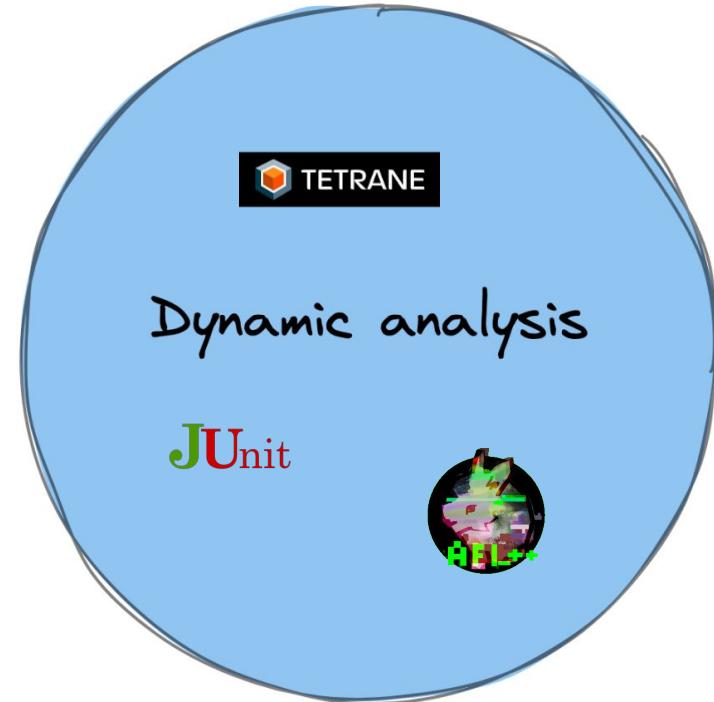
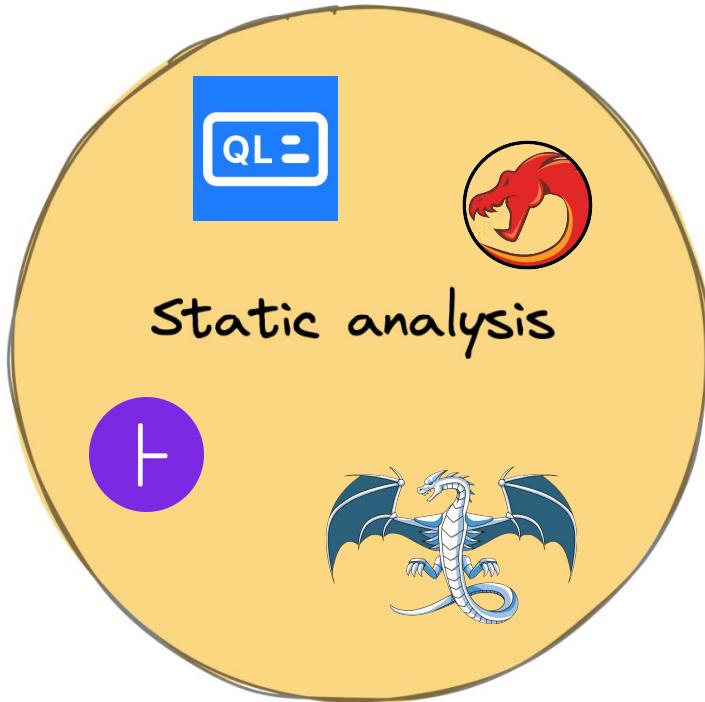
What is fuzzing?



What is fuzzing?



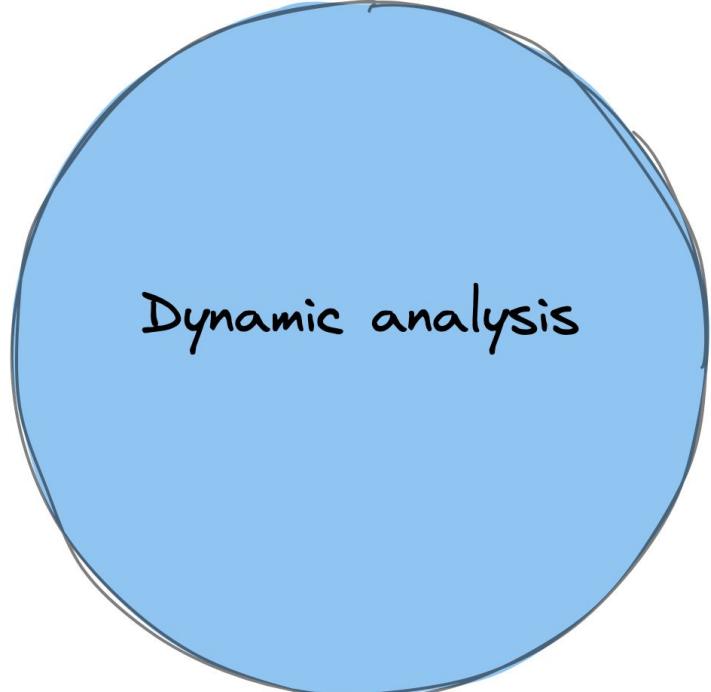
What is fuzzing?



What is fuzzing?

Pros

- No false positives
- Produces PoC
- Scalable



Dynamic analysis

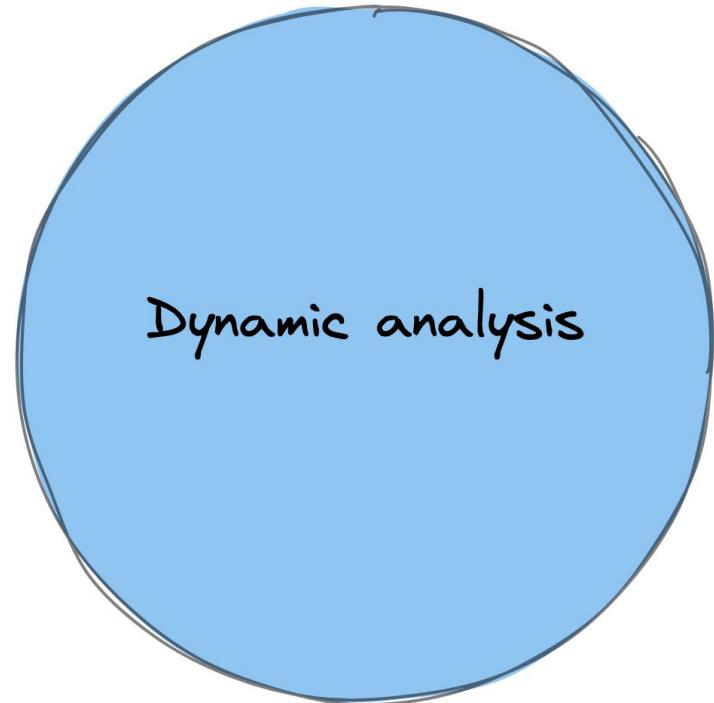
What is fuzzing?

Pros

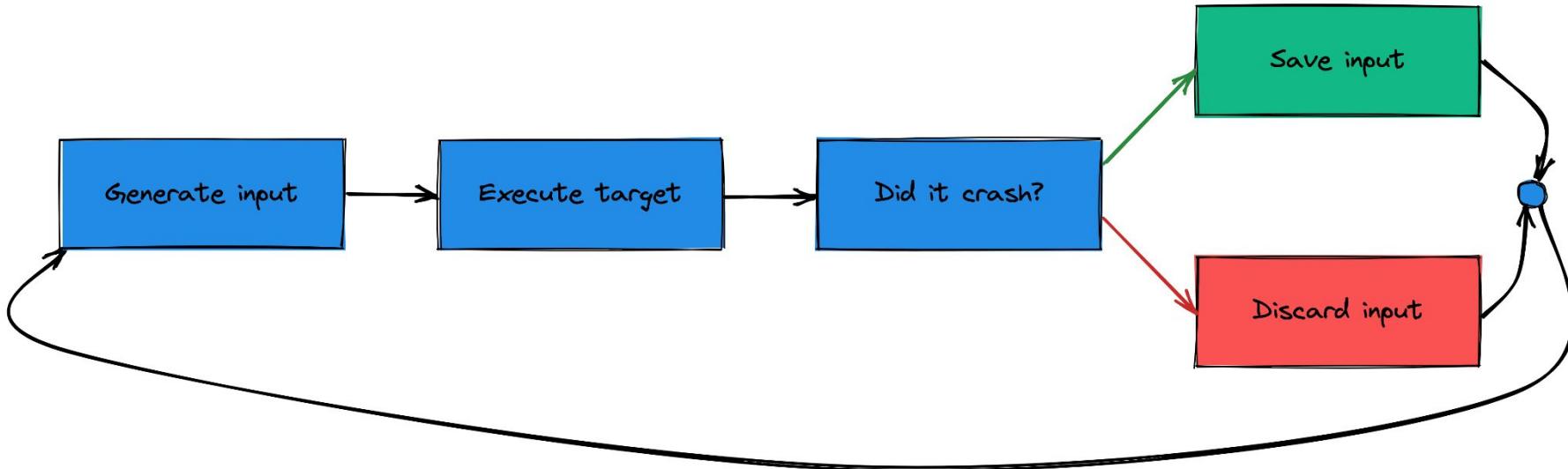
- No false positives
- Produces PoC
- Scalable

Cons

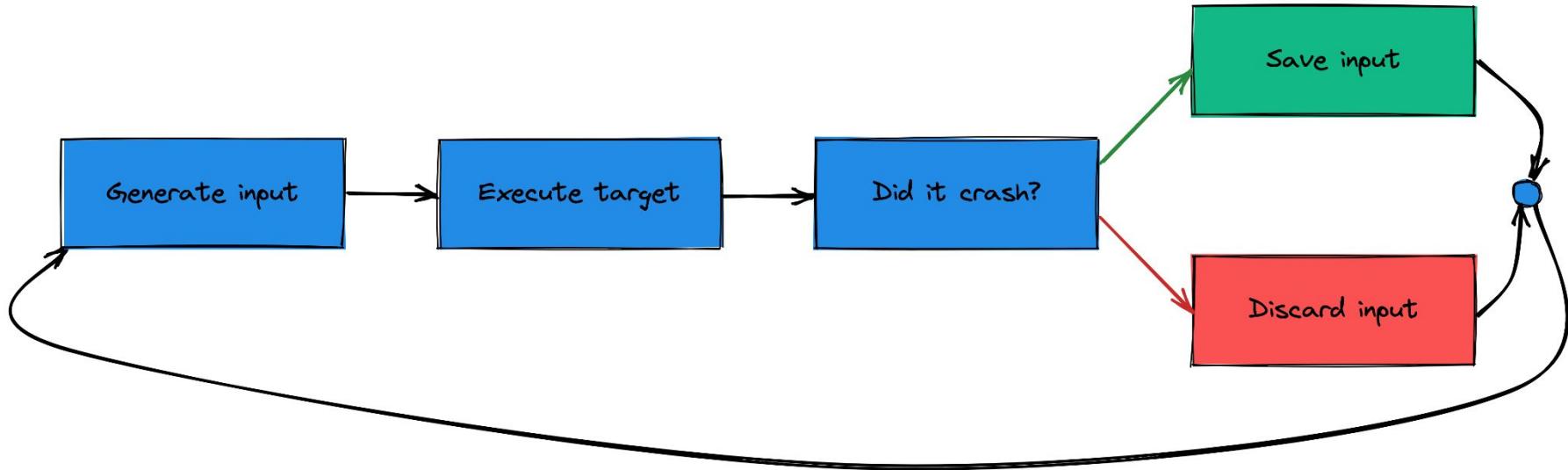
- Incomplete
- Requires buildable target
- Scalability



Our first fuzzer

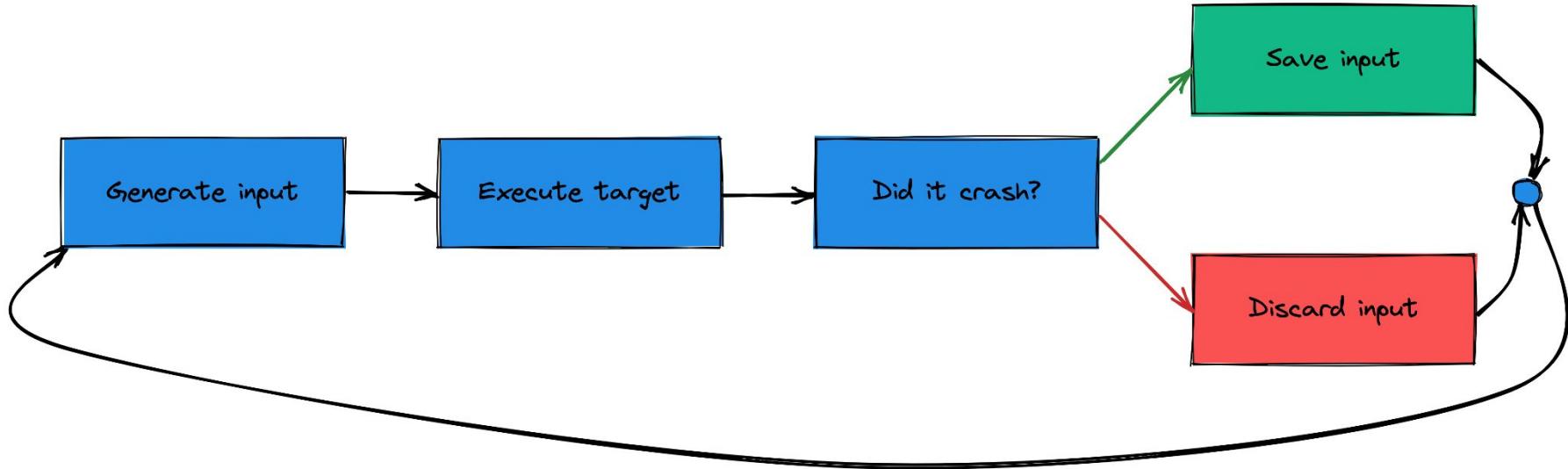


Our first fuzzer



How is this different to dynamic testing?

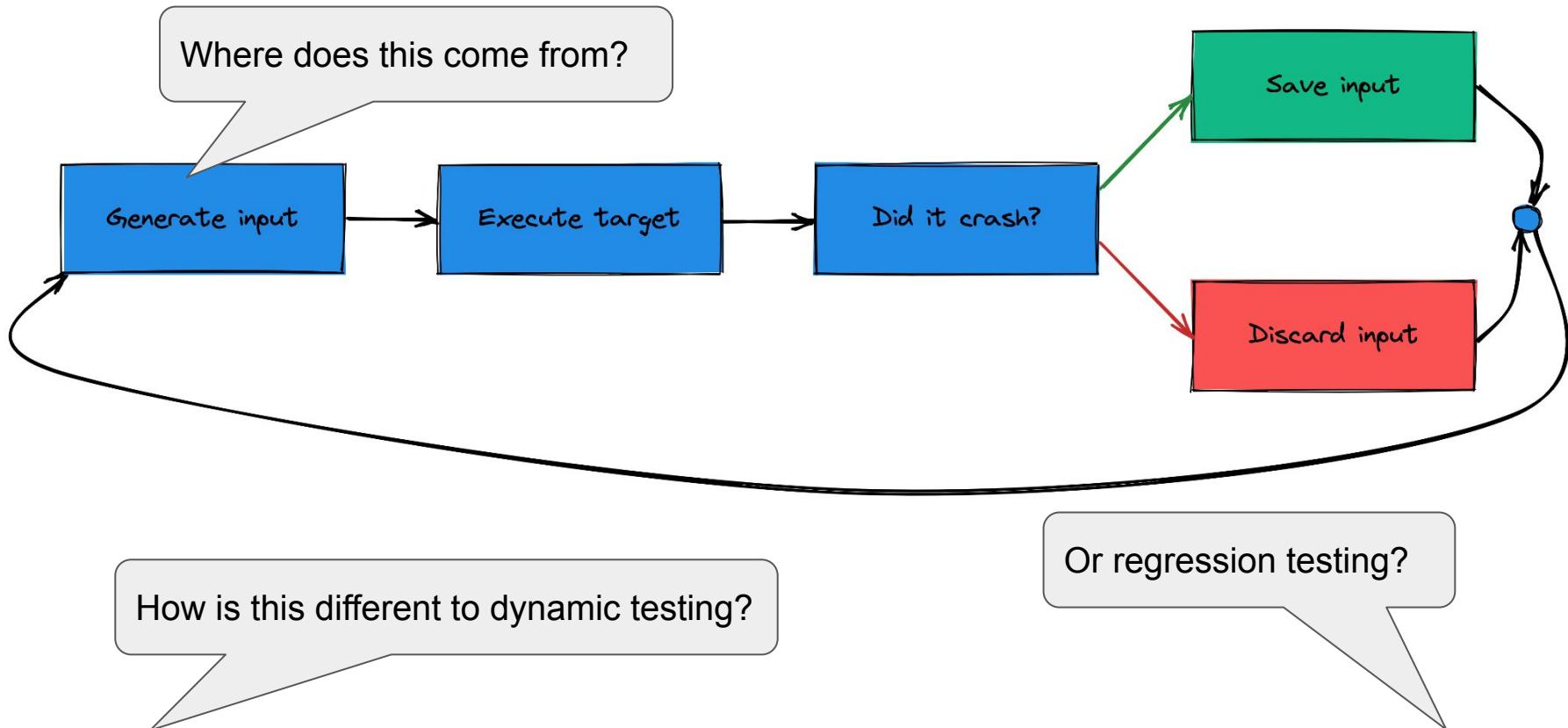
Our first fuzzer



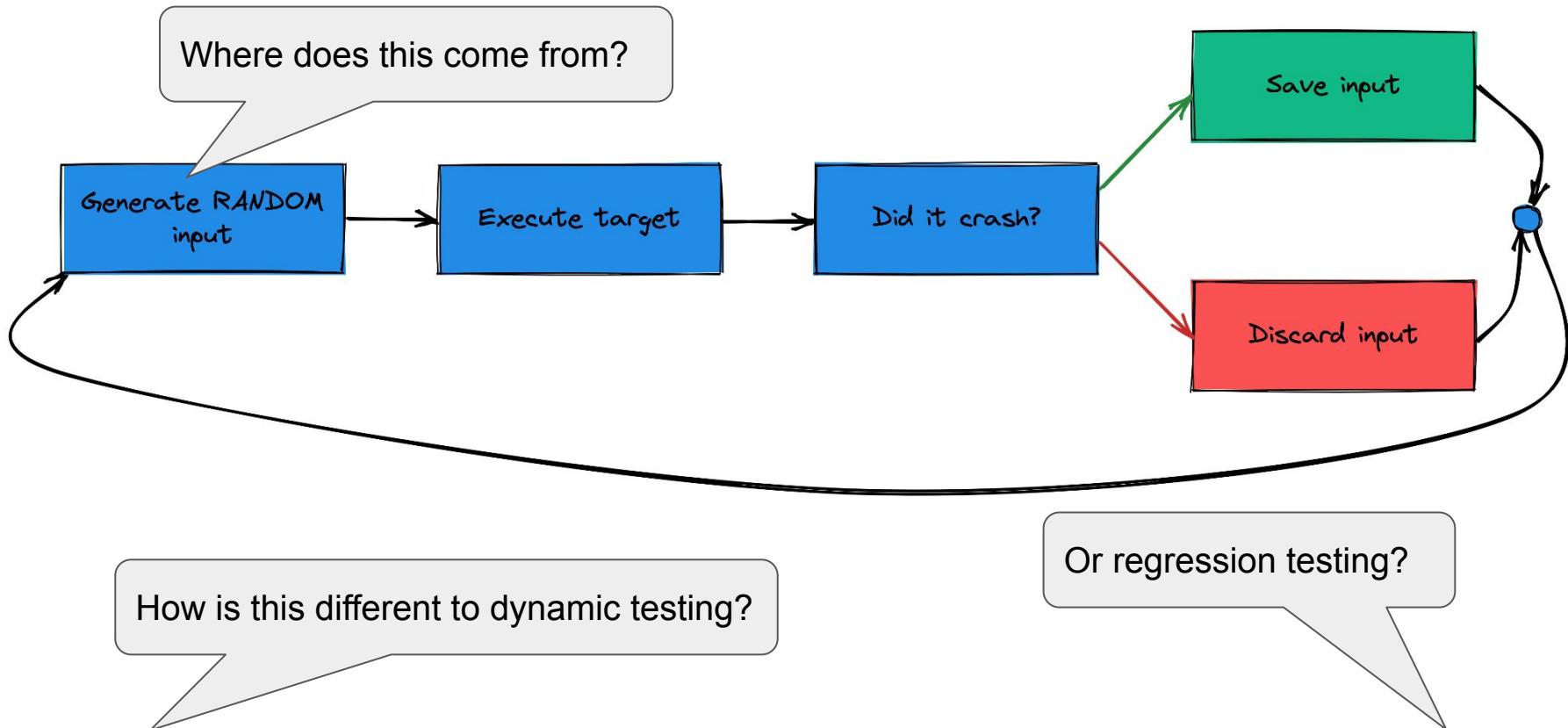
How is this different to dynamic testing?

Or regression testing?

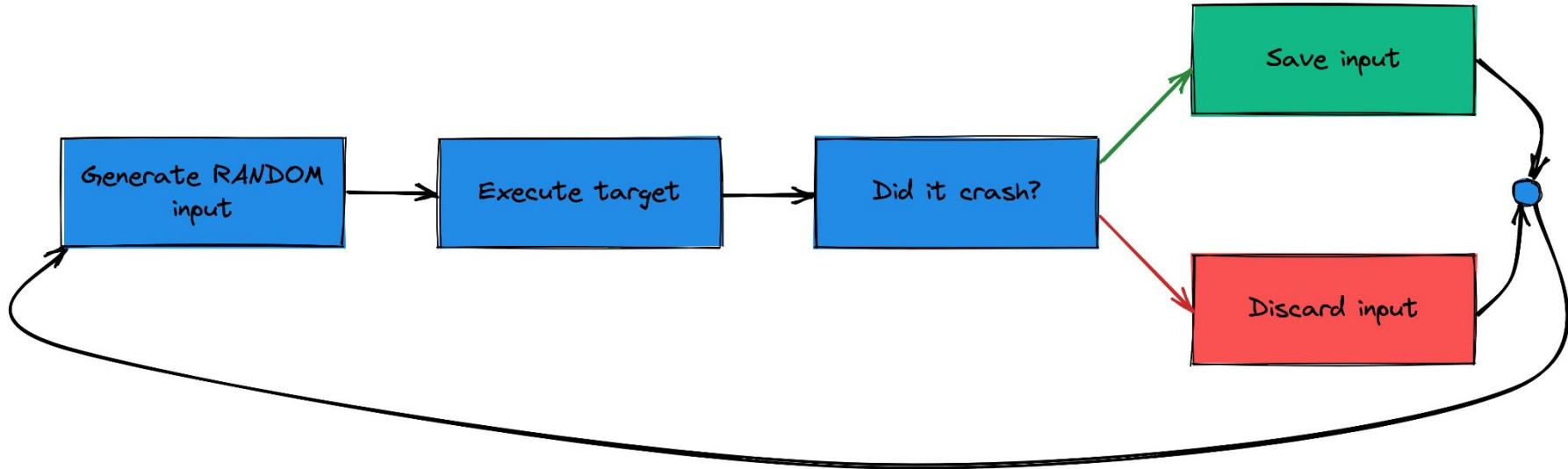
Our first fuzzer



Our first fuzzer



Our first fuzzer



A classic **generational blackbox** fuzzer

“An Empirical Study of the Reliability of Unix Utilities”

- Class project in 1988
“Advanced Operating Systems” course @ University Wisconsin
- Later published in 1990

When we use basic operating system facilities, such as the kernel and major utility programs, we expect them to have some degree of reliability. These programs are used frequently and this fact has reached the point where it is easy to apply (or commonly applied) to make a statement about

the Internet worm (the “gett footer” bug) [2,3]. We have found additional bugs that might indicate security holes. There are some of these bugs that are caused by input that might be carelessly typed—some strange and unexpected errors can be detected by this method of testing. Fourth, we sometimes inadvertently feed programs noisy input (e.g., trying to

the correctness of a program, we should probably use some form of formal verification. While the technology for program verification is still in its infancy, it is clear that there was still a need for some form of more complete testing. On a dark night, the author of this paper was at his computer, and his co-authors were logged on to his workstation via a dial-up line from home, and they ran the command “ls”. The phone line had some frequent spurious characters on the line.

The author had to race to see if he could type over the spurious characters before the noise scrambled the command. This line noise was a bit of a surprise, and the author was surprised that these spurious characters were causing programs to crash. These programs included a significant number of the standard operating system utilities. It is reasonable to expect that most utilities should not crash (“c”) due to receiving unusual input, they might exit with minimal error messages, but they should not crash. This kind of experience let us to believe that there might be serious bugs lurking in the systems that we regularly used.

The goal of our study was a systematic test of the utility versions running on various versions of the Unix operating system. The project proceeded in four steps: (1) programs were constructed to generate random characters, and to help identify the source of the noise; (2) these programs were used to test a large number of utilities on a common input stream to see if they crashed; (3) the strings (or types of strings) that crash these programs were identified; and (4) the causes of the

program crashes were identified and the common mistakes that cause them crash were categorized. As a result of testing almost seven different utility programs on seven versions of Unix™, we were able to crash more than 24% of the programs. Some of the excluded versions of Unix that under-

scored this study were some of our random testing and more traditional industrial software testing. While our testing strategy sounds like a random walk, the discovery of fatal program bugs is impressive. If we consider a program to be a state transition graph, then our testing strategy can be thought of as a random walk through the state space, testing for undefined states. Similar techniques have been used in areas such as network protocols and CPU cache management. When a module or network protocol, a module can be inserted in the data stream. This notion of correctness in our study.

A program is detected as faulty if it fails to produce the expected output. Our goal is to completely replace, or modify, existing test procedures. This type of study is important for several reasons: First, it contributes to the development of more sophisticated test cases against which researchers can evaluate more sophisticated testing and verification strategies. Second, the bugs that we found was caused by the same programming practice that we identified one of the security holes

Barton P. Miller, Lars Frederiksen and Bryan So

Study of the Reliability of Unix Utilities



An Empirical

program crashes were identified and the common mistakes that cause them crash were categorized. As a result of testing almost seven different utility programs on seven versions of Unix™, we were able to crash more than 24% of the programs. Some of the excluded versions of Unix that under-

scored this study were some of our random testing and more traditional industrial software testing. While our testing strategy sounds like a random walk, the discovery of fatal program bugs is impressive. If we consider a program to be a state transition graph, then our testing strategy can be thought of as a random walk through the state space, testing for undefined states. Similar techniques have been used in areas such as network protocols and CPU cache management. When a module or network protocol, a module can be inserted in the data stream. This notion of correctness in our study.

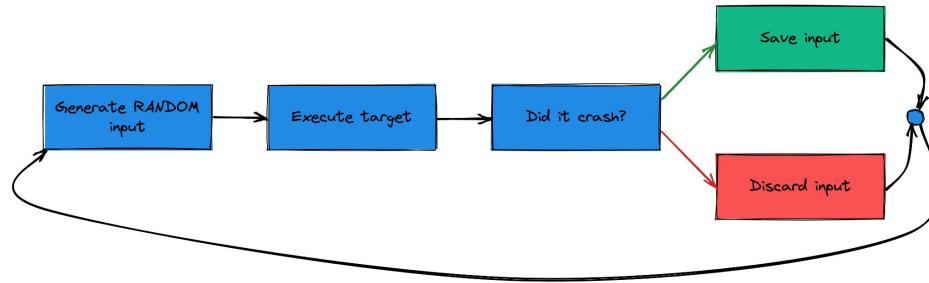
A program is detected as faulty if it fails to produce the expected output. Our goal is to completely replace, or modify, existing test procedures. This type of study is important for several reasons: First, it contributes to the development of more sophisticated test cases against which researchers can evaluate more sophisticated testing and verification strategies. Second, the bugs that we found was caused by the same programming practice that we identified one of the security holes

UNIX is a trademark of AT&T Bell Laboratories

Blackbox fuzzing

Pros

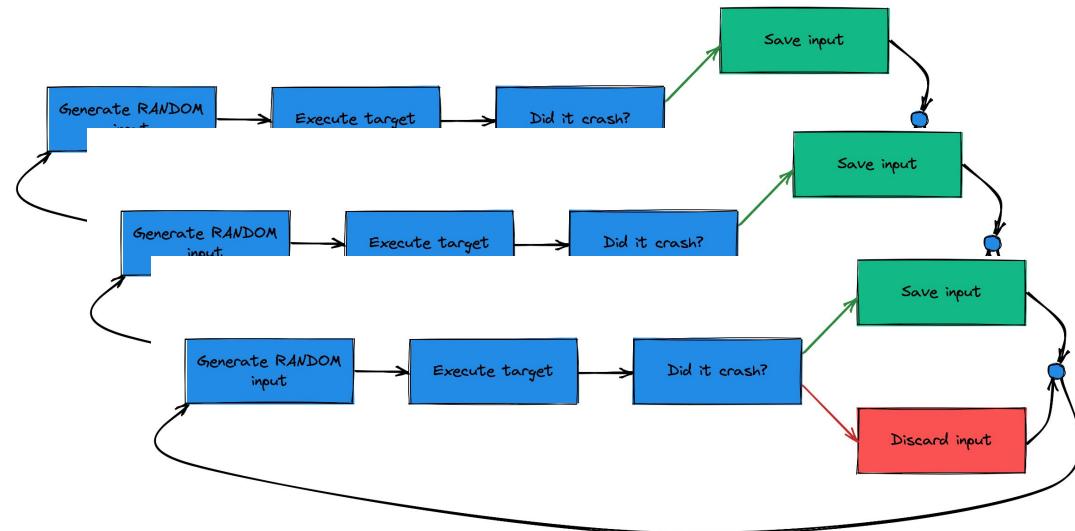
- Simple
- Fast
- Embarrassingly parallel



Blackbox fuzzing

Pros

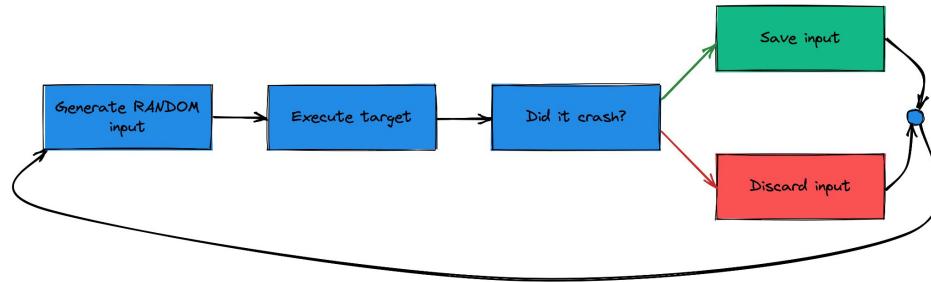
- Simple
- Fast
- Embarrassingly parallel



Blackbox fuzzing

Cons

- Generate mostly rubbish
- No notion of “progress”
- Only detect SIGSEGV

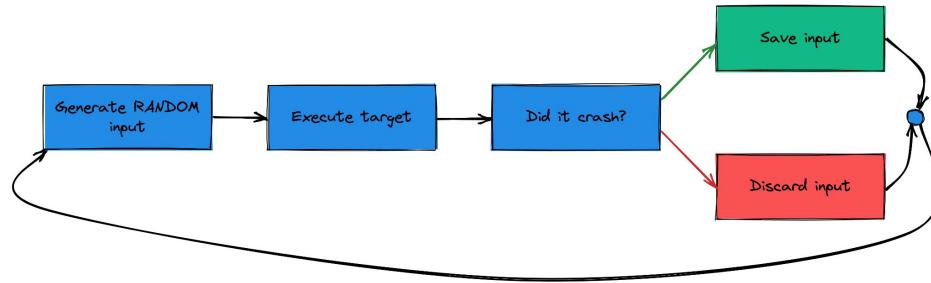


Can we do better?

Blackbox fuzzing

Cons

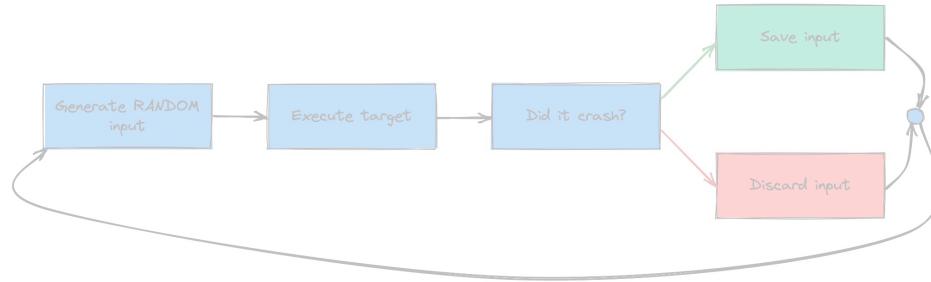
- Generate mostly rubbish
- No notion of “progress”
- Only detect SIGSEGV



Blackbox fuzzing

Cons

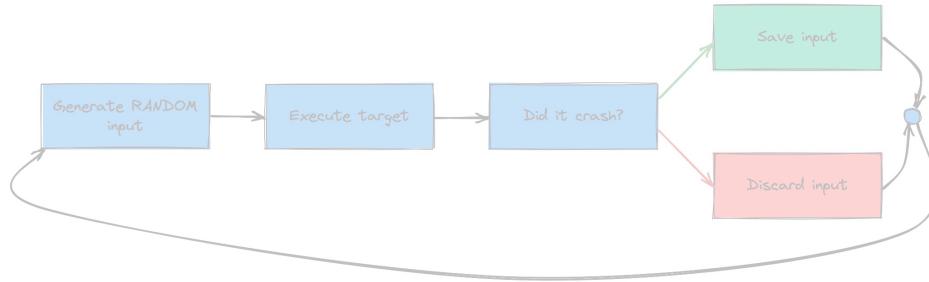
- Generate mostly rubbish
 - **Generate** **mutate**
- No notion of “progress”
- Only detect SIGSEGV



Blackbox fuzzing

Cons

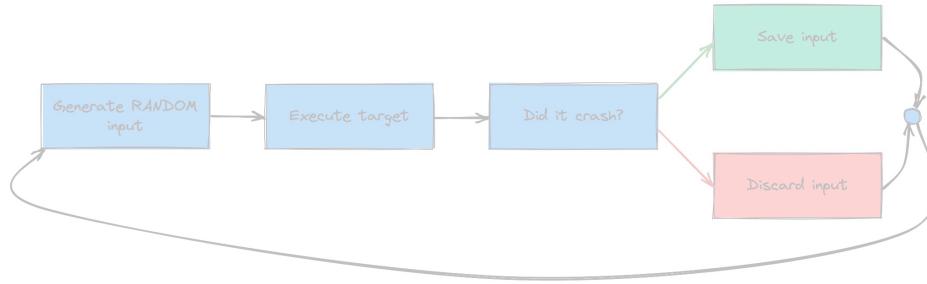
- Generate mostly rubbish
 - **Generate** **mutate**
- No notion of “progress”
 - Add a **feedback loop**
- Only detect **SIGSEGV**



Blackbox fuzzing

Cons

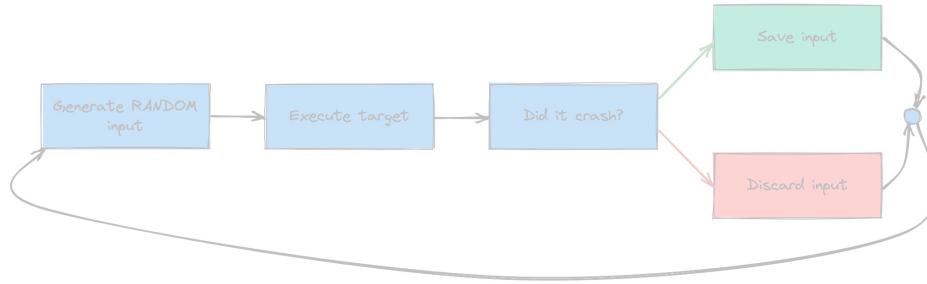
- Generate mostly rubbish
 - **Generate** **mutate**
- No notion of “progress”
 - Add a **feedback loop**
- Only detect **SIGSEGV**
 - Add a **sanitizer**



Blackbox fuzzing

Cons

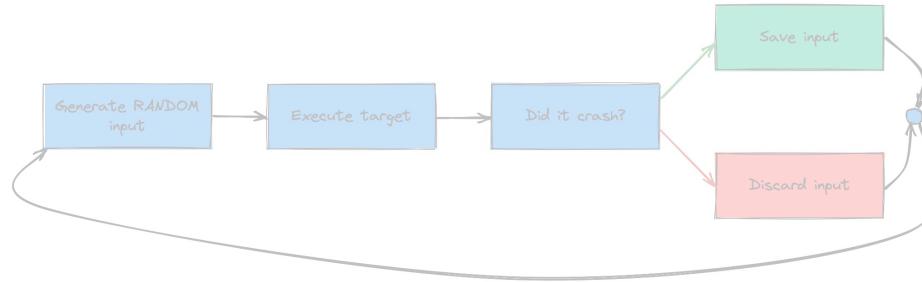
- Generate mostly rubbish
 - **Generate** **mutate**
- No notion of “progress”
 - Add a **feedback loop**
- Only detect SIGSEGV
 - Add a **sanitizer**



Blackbox fuzzing

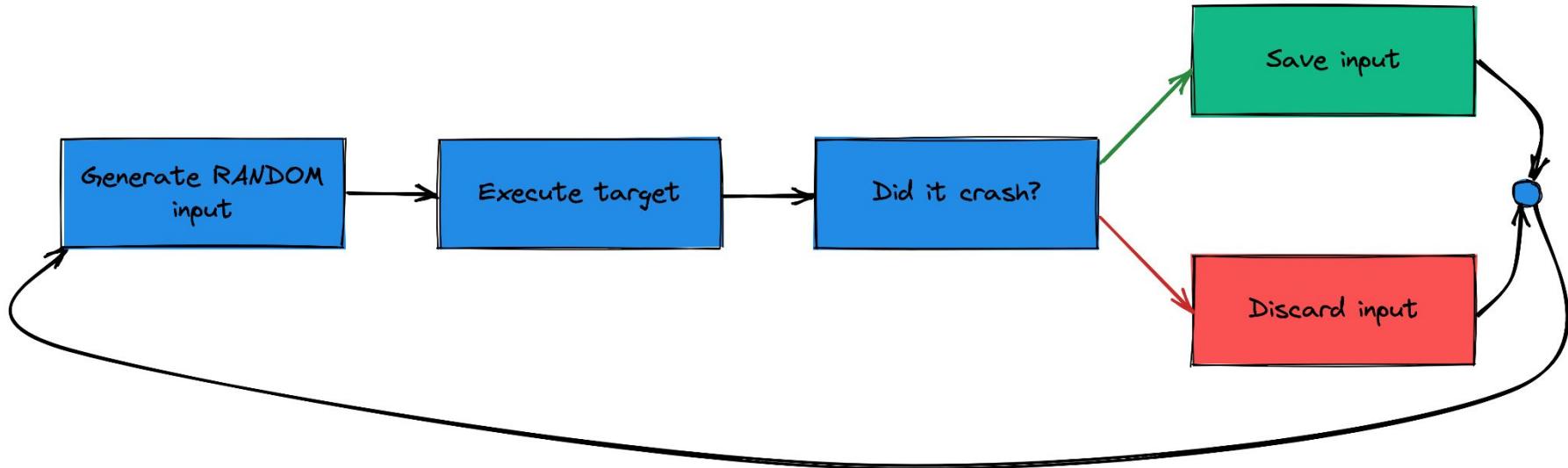
Cons

- Generate mostly rubbish
 - **Generate** **mutate**
- No notion of “progress”
 - Add a **feedback loop**
- Only detect `SIGSEGV`
 - Add a **sanitizer**

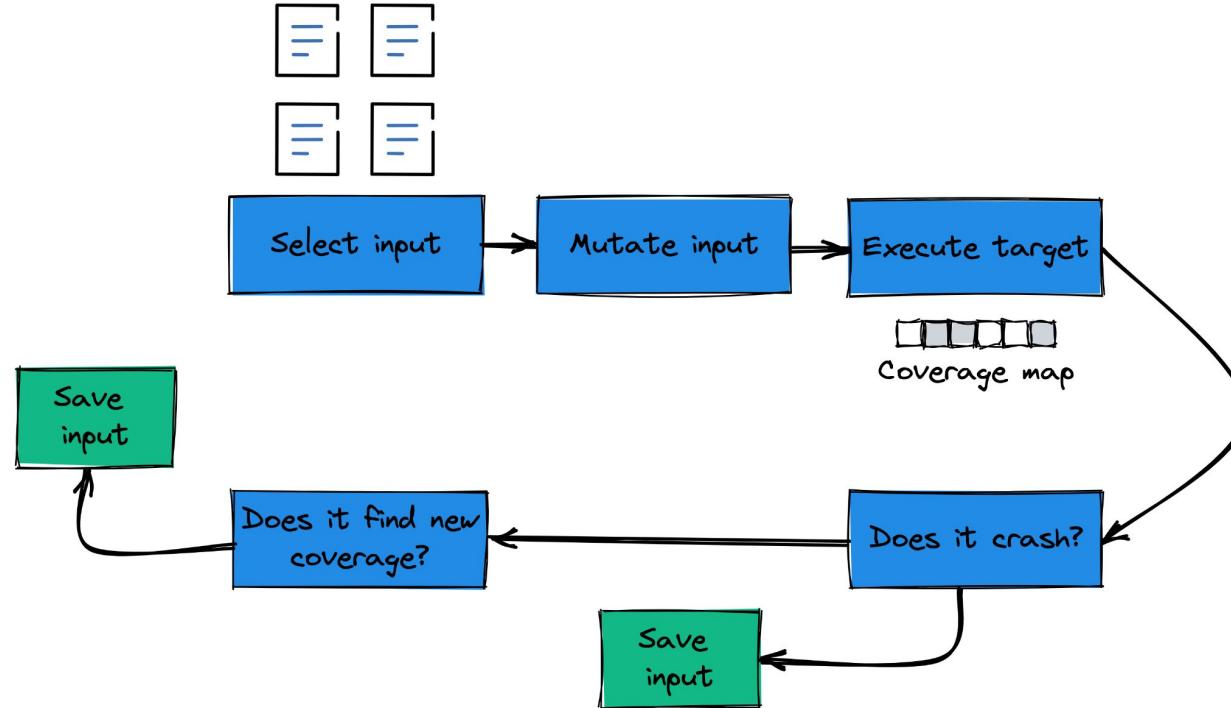


Mutational coverage-guided fuzzer
aka
greybox fuzzer

Blackbox fuzzing



Greybox fuzzing



Greybox fuzzing

```
american fuzzy lop 0.47b (readpng)

process timing
  run time : 0 days, 0 hrs, 4 min, 43 sec
  last new path : 0 days, 0 hrs, 0 min, 26 sec
  last uniq crash : none seen yet
  last uniq hang : 0 days, 0 hrs, 1 min, 51 sec

cycle progress
  now processing : 38 (19.49%)
  paths timed out : 0 (0.00%)

stage progress
  now trying : interest 32/8
  stage execs : 0/9990 (0.00%)
  total execs : 654k
  exec speed : 2306/sec

fuzzing strategy yields
  bit flips : 88/14.4k, 6/14.4k, 6/14.4k
  byte flips : 0/1804, 0/1786, 1/1750
  arithmetics : 31/126k, 3/45.6k, 1/17.8k
  known ints : 1/15.8k, 4/65.8k, 6/78.2k
  havoc : 34/254k, 0/0
  trim : 2876 B/931 (61.45% gain)

overall results
  cycles done : 0
  total paths : 195
  uniq crashes : 0
  uniq hangs : 1

map coverage
  map density : 1217 (7.43%)
  count coverage : 2.55 bits/tuple

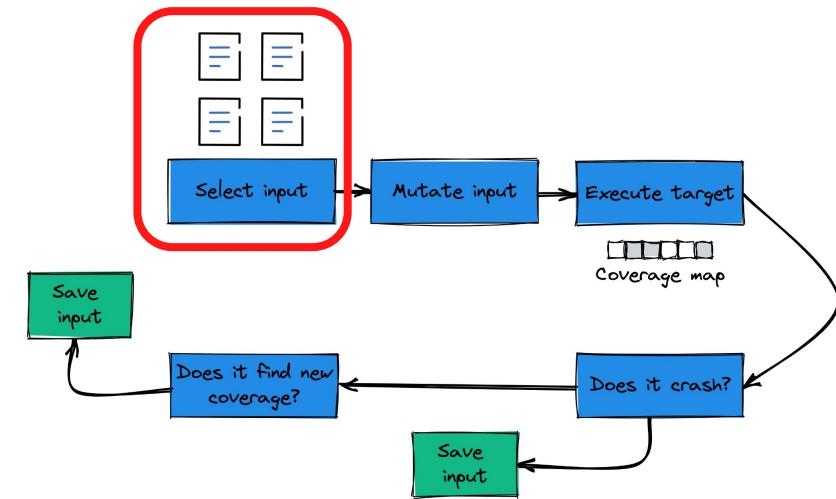
findings in depth
  favored paths : 128 (65.64%)
  new edges on : 85 (43.59%)
  total crashes : 0 (0 unique)
  total hangs : 1 (1 unique)

path geometry
  levels : 3
  pending : 178
  pend fav : 114
  imported : 0
  variable : 0
  latent : 0
```

Greybox fuzzing

Select input

- Rather than generating random data, mutate existing data

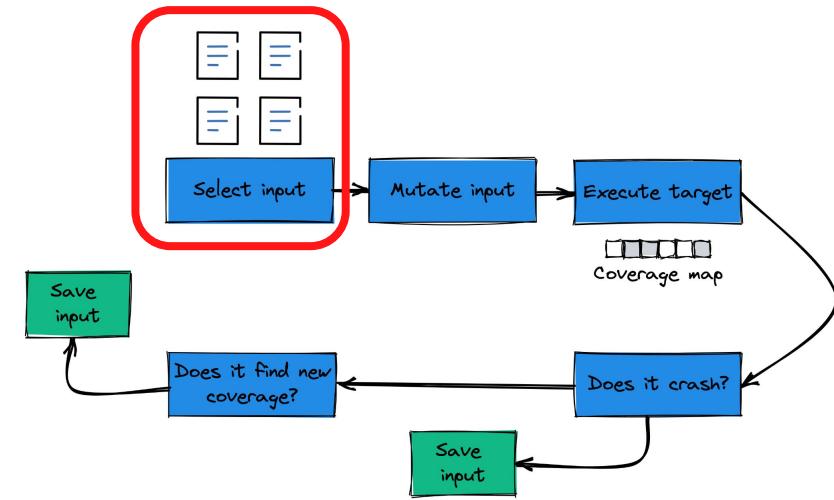


Greybox fuzzing

Select input

- Rather than generating random data, mutate existing data

Where do these initial inputs come from?



Seed selection

- In academic evaluations: “empty seed” common
- In practice: large corpora

Seed selection

- In academic evaluations: “empty seed” common
- In practice: large corpora

Which is better?

Seed selection

Optimizing Seed Selection for Fuzzing

Alexandre Rebert^{‡,§}
alex@forallsecure.com

Sang Kil Choi
sangkilc@cr

David Warren[†]
dwarren@cert.org

G
gg@c

Adrian Herrera
ANU & DST
Australia

C

Hendra Gunadi
ANU
Australia

Australia

Mathias Payer
EPFL
Switzerland

Australia

Shane Magrath
DST
Australia

Australia

Antony L. Hosking
ANU & CSIRO's Data61
Australia

Australia

[‡] Carnegie Mellon University
[†] Software I

Abstract

Randomly mutating well-formed program inputs, *fuzzing*, is a highly effective and widely used technique for finding bugs in software. Other than showing fuzz bugs, there has been little systematic effort in understanding the science of how to fuzz properly. In this work we focus on how to mathematically formulate and analyze critical aspect in fuzzing: how best to pick files to maximize the total number of bugs found in a fuzz campaign. We design and evaluate six different algorithms using over 650 CPU days on Amazon's Cloud (EC2) to provide ground truth. Overall, we find 240 bugs in 8 applications and show the choice of algorithm can greatly increase the # of bugs found. We also show that current seed selection strategies in Peach may fare no better than seeds at random. We make our data set available.

1 Introduction

Software bugs are expensive. A single software bug is enough to take down space crafts [2], make 1 centigears spin out of control [17], or recall 100,000 faulty cars resulting in billions of dollars in damage. In 2012, the software security market was estimated \$19.2 billion [1], and it is projected to double in size by 2016 due to a sequencing error [1]. The need for finding and fixing bugs in software they are exploited by attackers has led to the development of sophisticated automatic software testing tools.

Fuzzing is a popular and effective choice for bugs in applications. For example, fuzzing is a part of the overall quality checking process employed by Adobe [28], Microsoft [14], and Google [27], as

ABSTRACT

Mutation-based greybox fuzzing—unquestionably the most widely used fuzzing technique—uses a set of non-crashing seed inputs to find bugs in software. Other than showing fuzz bugs, there has been little systematic effort in understanding the science of how to fuzz properly. In this work we focus on how to mathematically formulate and analyze critical aspect in fuzzing: how best to pick files to maximize the total number of bugs found in a fuzz campaign. Little thought is given to *how* this seed choice affects the fuzzing process, and there is no consensus on which approach is best (or even a best approach exists).

We also show that current seed selection strategies in Peach may fare no better than seeds at random. We make our data set available.

Our results demonstrate that fuzzing outcomes vary significantly depending on the initial seeds used to bootstrap the fuzzer, with minimized corpora outperforming singleton, empty, and large (in the order of thousands of files) seed sets. Consequently, we encourage seed selection to be foremost in mind when evaluating/deploying fuzzers, and recommend that (a) seed choice be carefully considered and explicitly documented, and (b) never to evaluate fuzzers with only a single seed.

CCS CONCEPTS

• Software and its engineering → Software testing and debugging • Security and privacy → Software and application security
fuzzing, corpus minimization, software testing

Permission to make digital or hard copies of all or part of this work for personal use is granted free of fee by the copyright holders for those not for profit or commercial advantage. To copy otherwise, or to republish a page or pages, requires specific permission and/or a fee. Request permissions from permissions@acm.org.

ISBN 978-1-4503-4529-2/21/07

© 2021 Association for Computing Machinery

https://doi.org/10.1145/3460319.3464795

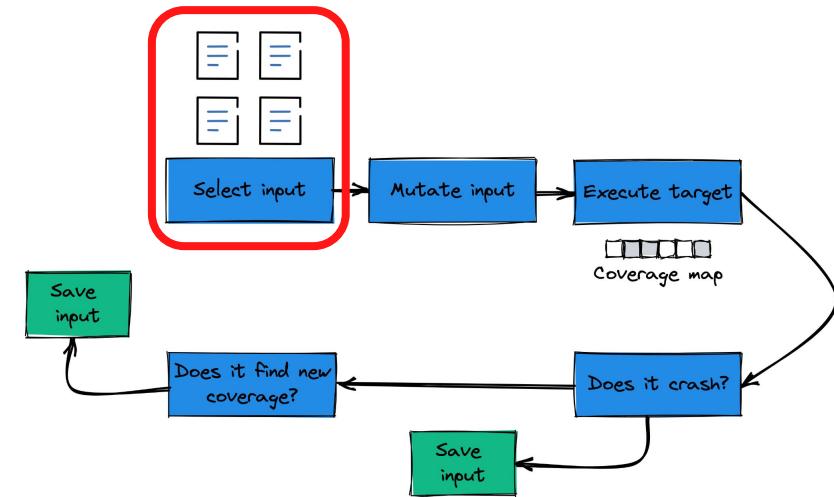
- Empty = easy to compare fuzzers
 - Only good for finding shallow bugs

- Too large corpus = slow fuzzer
- Sweet spot: Use a corpus minimizer
 - Doesn't matter which one

Greybox fuzzing

Select input

- Rather than generating random data, mutate existing data



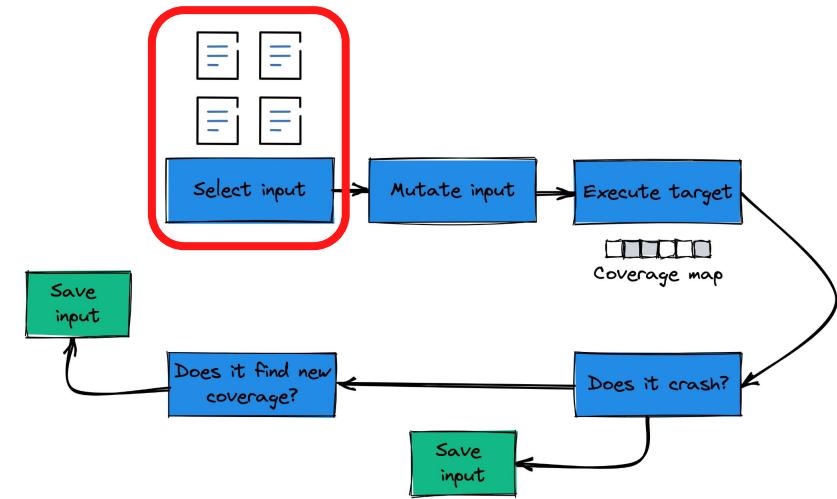
Greybox fuzzing

Select input

- Rather than generating random data, mutate existing data

How long do we focus on a seed?

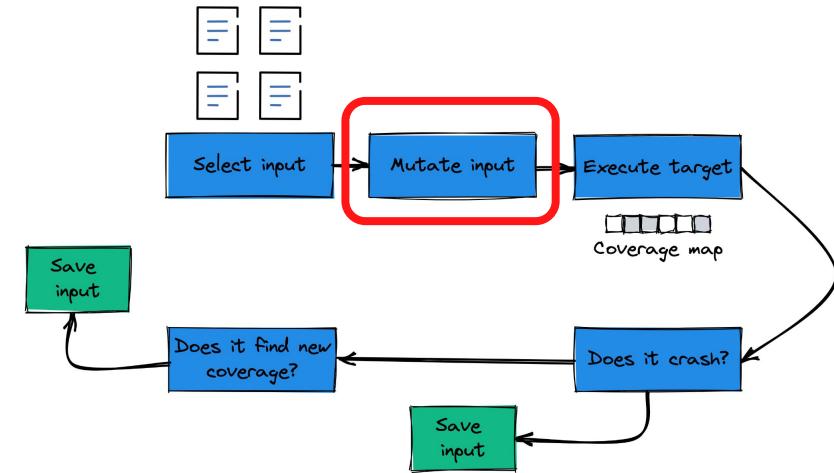
How do we select this seed?



Greybox fuzzing

Mutate input

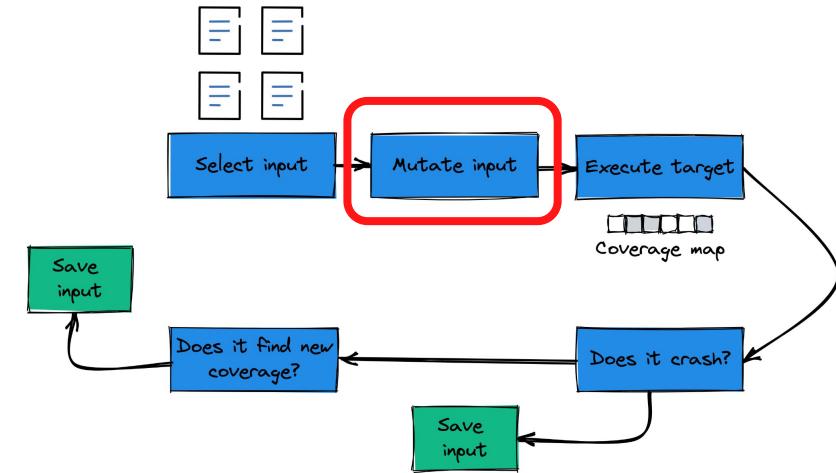
- Mutate enough to explore “interesting” states
- Don’t mutate too much, or we’ll just error out



Greybox fuzzing

Mutate input

- Mutate enough to explore “interesting” states
- Don’t mutate too much, or we’ll just error out



Where and how do we mutate?

Mutations

Structure agnostic

- Bit flip, byte/word/... substitution, repetition, splice

Structure aware

- Keyword substitution, grammar-based

Mutations

Structure agnostic

- Bit flip, byte/word/... substitution, repetition, splice
- Fast
- Simple to implement
- Destroys structure

Structure aware

- Keyword substitution, grammar-based
- Explore “deeper” code
- Require *a priori* knowledge

Mutations

Structure agnostic

- Bit flip, byte/word/... substitution, repetition, splice
- Fast
- Simple to implement
- Destroys structure

Structure aware

- Keyword substitution, **grammar-based**
- Explore “deeper” code
- Require *a priori* knowledge

Grammar-based fuzzing

- Many targets (e.g., JavaScript interpreter) accept input described by a **context-free grammar (CFG)**
 - Highly structured
 - Blind mutation will destroy structure

- Leverage CFG in mutation
 - “Lift” input to parse tree
 - Mutate parse tree(s)
 - Lower parse tree back to file

NAUTILUS: Fishing for Deep Bugs with Grammars



Gramatron: Effective Grammar-Aware Fuzzing

Co
Rut
come

Technis
patrick.jat

Abstract—For identifying bugs that are hard to fuzz, many fuzzers often and then semantic are passed, the fuzzers from execution of different many mutations. In contrast, all execution of error fuzzers are able to handle bugs. Gramatron, a fuzzing engine, mutation-based grammar fuzzer do not know what

In this paper, we propose the use of TDD-like mutation to increase the syntactic and of-concept coverage. Compared to the targets: Seven and one bug in the 2016 USD安6 combining configuration significantly outperforming two of them.

CCS CONCEPTS
• Fuzzing and its engineering → Sof
bugging • Security and privacy → Sof
security.

KEYWORDS

Fuzzing, grammar-aware, dynamic software

ACM Reference Format:

Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi,

Sergej Schumilo, Simon Wörner and Thorsten Holz

Network and Distril

24–27 February 2011

ISBN 978-1-4503-551-

https://doi.org/10.1145/1998793.1998814

© 2011 Association for Computing

Technology and Mechanics (ACM)

http://dx.doi.org/10.1145/1998793.1998814

3460/199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

199814

Grammar-based fuzzing

Pros

- Reach “deeper” code
- Can be used without coverage

Cons

- Require a priori knowledge of input format

Grammar-based fuzzing

Pros

- Reach “deeper” code
- Can be used without coverage

Cons

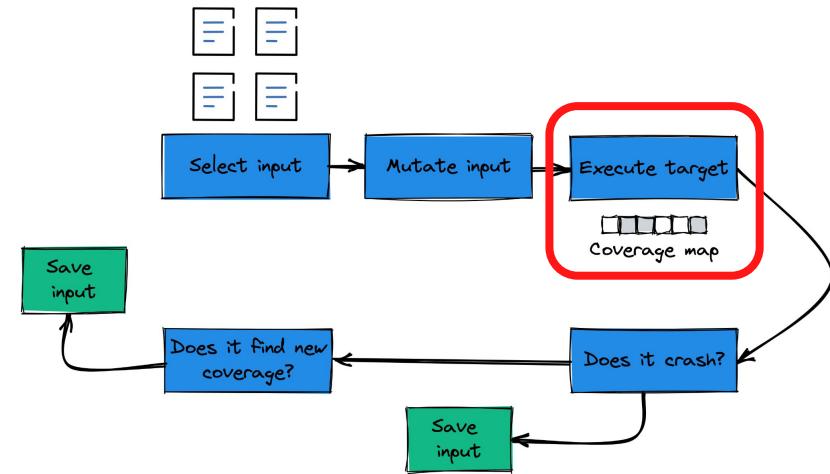
- Require a priori knowledge of input format

Some fuzzers try to “learn” this input format

Greybox fuzzing

Execute target

- Measure fuzzer “progress”
- Progress = code coverage



Coverage map

- Edge coverage is standard
- What if `# edges > sizeof(cov_map)`?
 - Must approximate
 - AFL uses a (lossy) hash function
- What if source is not available?
 - Use binary instrumentation (e.g., Intel PIN, DynamoRIO)

Coverage map

Edge coverage is a (relatively) poor approximation of a program's state space

Alternatives:

- Context-sensitive edge
- Path
- Data flow

Fuzzing with Data Dependency Information

Alessandro Mantovani
EURECOM
mantovan@eurecom.fr

Be Sensitive and Collaborative: Analyzing Impact of Coverage Metrics in Greybox Fuzzing

Jinghan Wang[†], Yue Duan[‡], Wei Song[†], Heng Yin[†], and Chengyu Song[†]

[†]UC Riverside [‡]Cornell University

[†]{jwang131,wsong008}@ucr.edu, {heng,csong}@cs.ucr.edu [‡]yd375@cornell.edu

Abstract—Recent advances in fuzz testing several forms of feedback mechanisms, fact that for a large range of programs at coverage alone is insufficient to reveal bugs by this line of research, we examined representations looking for a match between of the structure and adaptability to the testing. In particular, we believe that data and DDGs represent a good candidate for this information embedded by this data structure useful to find vulnerable constructs by inspection of def-use pairs that would be difficult for a trigger. Since some portions of the graph overlap with the control flow it is possible to reduce the additional instructions to only “interesting” data-flow dependencies the fuzzer to visit the code in a distinct standard methodology.

To test these observations, in this paper, a new approach that rewards fuzzer with code coverage information, but also in the data dependency graph, but also that the adoption of data dependency is coverage-guided fuzzing is a promising solution to discover bugs that would otherwise remain standard coverage approaches. This is due to 72 different vulnerabilities that our data-flow approach can identify when executed on three different datasets.

1. Introduction

In a society that makes software application core of many everyday activities is such software as secure as possible before to the public. This has led to a large amount focused on the development of increasing techniques to discover vulnerabilities, software testing [36], [60], [77], symbolic [62], [71] and dynamic analysis [73].

In the context of dynamic analysis, propagation of coverage is used to measure certain input produces in the software. One of the possible metrics is *path coverage*: all independent paths present in a program. In software testing, the community has coverage for test generation [64], [71] of automatically producing inputs that code locations. The main limitation of

Coverage-guided is most common technique, which decide-essential parameter of results. While there are many different ways to affect the fuzzing it is unclear whether it is superior to all the first systematical age metrics in fuzzing discuss the concept of metrically compare different coverage met-

study on these metrics: LAVA-M dataset, and of 221 binaries). We have limited resources metric has its unique of branches (thus vulnerability analysis incur a high run-time penalty, impeding fuzzer throughput. Lightweight data-flow alternatives to control-flow remain unexplored.

We present DATAFLOW, a greybox fuzzer driven by lightweight data-flow propagation. Whereas control-flow edges represent the order of operations in a program, a data-flow edges capture the dependencies between operations that produce data values and the operations that consume them; instead, they may produce (consumes) multiple values. Thus, data-flow edges capture behaviors not visible as control flow and intuitively discovers more or different bugs. Moreover, we establish a framework for reasoning about data-flow coverage, which computational cost of exploration to be balanced with precision.

We perform a preliminary evaluation of DATAFLOW, comparing fuzzers driven by control flow, taint analysis (both approximate and exact), and data flow. Our initial results suggest that, so far, pure coverage remains the best coverage metric for finding bugs in benchmarks. However, data-flow coverage does show promise in targets where control flow is decoupled from semantics (e.g., parsers). Further evaluation and analysis on a wider range of targets is required.

1 Introduction

Greybox fuzzing is a technique that has been implemented in various tools such as Google’s *PathFinder* [1], *Trail of Bits* [2], *Challenge* [3], *Grind* [4], *Grind* [5], *Grind* [6], *Grind* [7], *Grind* [8], *Grind* [9], *Grind* [10], *Grind* [11], *Grind* [12], *Grind* [13], *Grind* [14], *Grind* [15], *Grind* [16], *Grind* [17], *Grind* [18], *Grind* [19], *Grind* [20], *Grind* [21], *Grind* [22], *Grind* [23], *Grind* [24], *Grind* [25], *Grind* [26], *Grind* [27], *Grind* [28], *Grind* [29], *Grind* [30], *Grind* [31], *Grind* [32], *Grind* [33], *Grind* [34], *Grind* [35], *Grind* [36], *Grind* [37], *Grind* [38], *Grind* [39], *Grind* [40], *Grind* [41], *Grind* [42], *Grind* [43], *Grind* [44], *Grind* [45], *Grind* [46], *Grind* [47], *Grind* [48], *Grind* [49], *Grind* [50], *Grind* [51], *Grind* [52], *Grind* [53], *Grind* [54], *Grind* [55], *Grind* [56], *Grind* [57], *Grind* [58], *Grind* [59], *Grind* [60], *Grind* [61], *Grind* [62], *Grind* [63], *Grind* [64], *Grind* [65], *Grind* [66], *Grind* [67], *Grind* [68], *Grind* [69], *Grind* [70], *Grind* [71], *Grind* [72], *Grind* [73], *Grind* [74], *Grind* [75], *Grind* [76], *Grind* [77], *Grind* [78], *Grind* [79], *Grind* [80], *Grind* [81], *Grind* [82], *Grind* [83], *Grind* [84], *Grind* [85], *Grind* [86], *Grind* [87], *Grind* [88], *Grind* [89], *Grind* [90], *Grind* [91], *Grind* [92], *Grind* [93], *Grind* [94], *Grind* [95], *Grind* [96], *Grind* [97], *Grind* [98], *Grind* [99], *Grind* [100], *Grind* [101], *Grind* [102], *Grind* [103], *Grind* [104], *Grind* [105], *Grind* [106], *Grind* [107], *Grind* [108], *Grind* [109], *Grind* [110], *Grind* [111], *Grind* [112], *Grind* [113], *Grind* [114], *Grind* [115], *Grind* [116], *Grind* [117], *Grind* [118], *Grind* [119], *Grind* [120], *Grind* [121], *Grind* [122], *Grind* [123], *Grind* [124], *Grind* [125], *Grind* [126], *Grind* [127], *Grind* [128], *Grind* [129], *Grind* [130], *Grind* [131], *Grind* [132], *Grind* [133], *Grind* [134], *Grind* [135], *Grind* [136], *Grind* [137], *Grind* [138], *Grind* [139], *Grind* [140], *Grind* [141], *Grind* [142], *Grind* [143], *Grind* [144], *Grind* [145], *Grind* [146], *Grind* [147], *Grind* [148], *Grind* [149], *Grind* [150], *Grind* [151], *Grind* [152], *Grind* [153], *Grind* [154], *Grind* [155], *Grind* [156], *Grind* [157], *Grind* [158], *Grind* [159], *Grind* [160], *Grind* [161], *Grind* [162], *Grind* [163], *Grind* [164], *Grind* [165], *Grind* [166], *Grind* [167], *Grind* [168], *Grind* [169], *Grind* [170], *Grind* [171], *Grind* [172], *Grind* [173], *Grind* [174], *Grind* [175], *Grind* [176], *Grind* [177], *Grind* [178], *Grind* [179], *Grind* [180], *Grind* [181], *Grind* [182], *Grind* [183], *Grind* [184], *Grind* [185], *Grind* [186], *Grind* [187], *Grind* [188], *Grind* [189], *Grind* [190], *Grind* [191], *Grind* [192], *Grind* [193], *Grind* [194], *Grind* [195], *Grind* [196], *Grind* [197], *Grind* [198], *Grind* [199], *Grind* [200], *Grind* [201], *Grind* [202], *Grind* [203], *Grind* [204], *Grind* [205], *Grind* [206], *Grind* [207], *Grind* [208], *Grind* [209], *Grind* [210], *Grind* [211], *Grind* [212], *Grind* [213], *Grind* [214], *Grind* [215], *Grind* [216], *Grind* [217], *Grind* [218], *Grind* [219], *Grind* [220], *Grind* [221], *Grind* [222], *Grind* [223], *Grind* [224], *Grind* [225], *Grind* [226], *Grind* [227], *Grind* [228], *Grind* [229], *Grind* [230], *Grind* [231], *Grind* [232], *Grind* [233], *Grind* [234], *Grind* [235], *Grind* [236], *Grind* [237], *Grind* [238], *Grind* [239], *Grind* [240], *Grind* [241], *Grind* [242], *Grind* [243], *Grind* [244], *Grind* [245], *Grind* [246], *Grind* [247], *Grind* [248], *Grind* [249], *Grind* [250], *Grind* [251], *Grind* [252], *Grind* [253], *Grind* [254], *Grind* [255], *Grind* [256], *Grind* [257], *Grind* [258], *Grind* [259], *Grind* [260], *Grind* [261], *Grind* [262], *Grind* [263], *Grind* [264], *Grind* [265], *Grind* [266], *Grind* [267], *Grind* [268], *Grind* [269], *Grind* [270], *Grind* [271], *Grind* [272], *Grind* [273], *Grind* [274], *Grind* [275], *Grind* [276], *Grind* [277], *Grind* [278], *Grind* [279], *Grind* [280], *Grind* [281], *Grind* [282], *Grind* [283], *Grind* [284], *Grind* [285], *Grind* [286], *Grind* [287], *Grind* [288], *Grind* [289], *Grind* [290], *Grind* [291], *Grind* [292], *Grind* [293], *Grind* [294], *Grind* [295], *Grind* [296], *Grind* [297], *Grind* [298], *Grind* [299], *Grind* [300], *Grind* [301], *Grind* [302], *Grind* [303], *Grind* [304], *Grind* [305], *Grind* [306], *Grind* [307], *Grind* [308], *Grind* [309], *Grind* [310], *Grind* [311], *Grind* [312], *Grind* [313], *Grind* [314], *Grind* [315], *Grind* [316], *Grind* [317], *Grind* [318], *Grind* [319], *Grind* [320], *Grind* [321], *Grind* [322], *Grind* [323], *Grind* [324], *Grind* [325], *Grind* [326], *Grind* [327], *Grind* [328], *Grind* [329], *Grind* [330], *Grind* [331], *Grind* [332], *Grind* [333], *Grind* [334], *Grind* [335], *Grind* [336], *Grind* [337], *Grind* [338], *Grind* [339], *Grind* [340], *Grind* [341], *Grind* [342], *Grind* [343], *Grind* [344], *Grind* [345], *Grind* [346], *Grind* [347], *Grind* [348], *Grind* [349], *Grind* [350], *Grind* [351], *Grind* [352], *Grind* [353], *Grind* [354], *Grind* [355], *Grind* [356], *Grind* [357], *Grind* [358], *Grind* [359], *Grind* [360], *Grind* [361], *Grind* [362], *Grind* [363], *Grind* [364], *Grind* [365], *Grind* [366], *Grind* [367], *Grind* [368], *Grind* [369], *Grind* [370], *Grind* [371], *Grind* [372], *Grind* [373], *Grind* [374], *Grind* [375], *Grind* [376], *Grind* [377], *Grind* [378], *Grind* [379], *Grind* [380], *Grind* [381], *Grind* [382], *Grind* [383], *Grind* [384], *Grind* [385], *Grind* [386], *Grind* [387], *Grind* [388], *Grind* [389], *Grind* [390], *Grind* [391], *Grind* [392], *Grind* [393], *Grind* [394], *Grind* [395], *Grind* [396], *Grind* [397], *Grind* [398], *Grind* [399], *Grind* [400], *Grind* [401], *Grind* [402], *Grind* [403], *Grind* [404], *Grind* [405], *Grind* [406], *Grind* [407], *Grind* [408], *Grind* [409], *Grind* [410], *Grind* [411], *Grind* [412], *Grind* [413], *Grind* [414], *Grind* [415], *Grind* [416], *Grind* [417], *Grind* [418], *Grind* [419], *Grind* [420], *Grind* [421], *Grind* [422], *Grind* [423], *Grind* [424], *Grind* [425], *Grind* [426], *Grind* [427], *Grind* [428], *Grind* [429], *Grind* [430], *Grind* [431], *Grind* [432], *Grind* [433], *Grind* [434], *Grind* [435], *Grind* [436], *Grind* [437], *Grind* [438], *Grind* [439], *Grind* [440], *Grind* [441], *Grind* [442], *Grind* [443], *Grind* [444], *Grind* [445], *Grind* [446], *Grind* [447], *Grind* [448], *Grind* [449], *Grind* [450], *Grind* [451], *Grind* [452], *Grind* [453], *Grind* [454], *Grind* [455], *Grind* [456], *Grind* [457], *Grind* [458], *Grind* [459], *Grind* [460], *Grind* [461], *Grind* [462], *Grind* [463], *Grind* [464], *Grind* [465], *Grind* [466], *Grind* [467], *Grind* [468], *Grind* [469], *Grind* [470], *Grind* [471], *Grind* [472], *Grind* [473], *Grind* [474], *Grind* [475], *Grind* [476], *Grind* [477], *Grind* [478], *Grind* [479], *Grind* [480], *Grind* [481], *Grind* [482], *Grind* [483], *Grind* [484], *Grind* [485], *Grind* [486], *Grind* [487], *Grind* [488], *Grind* [489], *Grind* [490], *Grind* [491], *Grind* [492], *Grind* [493], *Grind* [494], *Grind* [495], *Grind* [496], *Grind* [497], *Grind* [498], *Grind* [499], *Grind* [500], *Grind* [501], *Grind* [502], *Grind* [503], *Grind* [504], *Grind* [505], *Grind* [506], *Grind* [507], *Grind* [508], *Grind* [509], *Grind* [510], *Grind* [511], *Grind* [512], *Grind* [513], *Grind* [514], *Grind* [515], *Grind* [516], *Grind* [517], *Grind* [518], *Grind* [519], *Grind* [520], *Grind* [521], *Grind* [522], *Grind* [523], *Grind* [524], *Grind* [525], *Grind* [526], *Grind* [527], *Grind* [528], *Grind* [529], *Grind* [530], *Grind* [531], *Grind* [532], *Grind* [533], *Grind* [534], *Grind* [535], *Grind* [536], *Grind* [537], *Grind* [538], *Grind* [539], *Grind* [540], *Grind* [541], *Grind* [542], *Grind* [543], *Grind* [544], *Grind* [545], *Grind* [546], *Grind* [547], *Grind* [548], *Grind* [549], *Grind* [550], *Grind* [551], *Grind* [552], *Grind* [553], *Grind* [554], *Grind* [555], *Grind* [556], *Grind* [557], *Grind* [558], *Grind* [559], *Grind* [560], *Grind* [561], *Grind* [562], *Grind* [563], *Grind* [564], *Grind* [565], *Grind* [566], *Grind* [567], *Grind* [568], *Grind* [569], *Grind* [570], *Grind* [571], *Grind* [572], *Grind* [573], *Grind* [574], *Grind* [575], *Grind* [576], *Grind* [577], *Grind* [578], *Grind* [579], *Grind* [580], *Grind* [581], *Grind* [582], *Grind* [583], *Grind* [584], *Grind* [585], *Grind* [586], *Grind* [587], *Grind* [588], *Grind* [589], *Grind* [590], *Grind* [591], *Grind* [592], *Grind* [593], *Grind* [594], *Grind* [595], *Grind* [596], *Grind* [597], *Grind* [598], *Grind* [599], *Grind* [600], *Grind* [601], *Grind* [602], *Grind* [603], *Grind* [604], *Grind* [605], *Grind* [606], *Grind* [607], *Grind* [608], *Grind* [609], *Grind* [610], *Grind* [611], *Grind* [612], *Grind* [613], *Grind* [614], *Grind* [615], *Grind* [616], *Grind* [617], *Grind* [618], *Grind* [619], *Grind* [620], *Grind* [621], *Grind* [622], *Grind* [623], *Grind* [624], *Grind* [625], *Grind* [626], *Grind* [627], *Grind* [628], *Grind* [629], *Grind* [630], *Grind* [631], *Grind* [632], *Grind* [633], *Grind* [634], *Grind* [635], *Grind* [636], *Grind* [637], *Grind* [638], *Grind* [639], *Grind* [640], *Grind* [641], *Grind* [642], *Grind* [643], *Grind* [644], *Grind* [645], *Grind* [646], *Grind* [647], *Grind* [648], *Grind* [649], *Grind* [650], *Grind* [651], *Grind* [652], *Grind* [653], *Grind* [654], *Grind* [655], *Grind* [656], *Grind* [657], *Grind* [658], *Grind* [659], *Grind* [660], *Grind* [661], *Grind* [662], *Grind* [663], *Grind* [664], *Grind* [665], *Grind* [666], *Grind* [667], *Grind* [668], *Grind* [669], *Grind* [670], *Grind* [671], *Grind* [672], *Grind* [673], *Grind* [674], *Grind* [675], *Grind* [676], *Grind* [677], *Grind* [678], *Grind* [679], *Grind* [680], *Grind* [681], *Grind* [682], *Grind* [683], *Grind* [684], *Grind* [685], *Grind* [686], *Grind* [687], *Grind* [688], *Grind* [689], *Grind* [690], *Grind* [691], *Grind* [692], *Grind* [693], *Grind* [694], *Grind* [695], *Grind* [696], *Grind* [697], *Grind* [698], *Grind* [699], *Grind* [700], *Grind* [701], *Grind* [702], *Grind* [703], *Grind* [704], *Grind* [705], *Grind* [706], *Grind* [707], *Grind* [708], *Grind* [709], *Grind* [710], *Grind* [711], *Grind* [712], *Grind* [713], *Grind* [714], *Grind* [715], *Grind* [716], *Grind* [717], *Grind* [718], *Grind* [719], *Grind* [720], *Grind* [721], *Grind* [722], *Grind* [723], *Grind* [724], *Grind* [725], *Grind* [726], *Grind* [727], *Grind* [728], *Grind* [729], *Grind* [730], *Grind* [731], *Grind* [732], *Grind* [733], *Grind* [734], *Grind* [735], *Grind* [736], *Grind* [737], *Grind* [738], *Grind* [739], *Grind* [740], *Grind* [741], *Grind* [742], *Grind* [743], *Grind* [744], *Grind* [745], *Grind* [746], *Grind* [747], *Grind* [748], *Grind* [749], *Grind* [750], *Grind* [751], *Grind* [752], *Grind* [753], *Grind* [754], *Grind* [755], *Grind* [756], *Grind* [757], *Grind* [758], *Grind* [759], *Grind* [760], *Grind* [761], *Grind* [762], *Grind* [763], *Grind* [764], *Grind* [765], *Grind* [766], *Grind* [767], *Grind* [768], *Grind* [769], *Grind* [770], *Grind* [771], *Grind* [772], *Grind* [773], *Grind* [774], *Grind* [775], *Grind* [776], *Grind* [777], *Grind* [778], *Grind* [779], *Grind* [780], *Grind* [781], *Grind* [782], *Grind* [783], *Grind* [784], *Grind* [785], *Grind* [786], *Grind* [787], *Grind* [788], *Grind* [789], *Grind* [790], *Grind* [791], *Grind* [792], *Grind* [793], *Grind* [794], *Grind* [795], *Grind* [796], *Grind* [797], *Grind* [798], *Grind* [799], *Grind* [800], *Grind* [801], *Grind* [802], *Grind* [803], *Grind* [804], *Grind* [805], *Grind* [806], *Grind* [807], *Grind* [808], *Grind* [809], *Grind* [810], *Grind* [811], *Grind* [812], *Grind* [813], *Grind* [814], *Grind* [815], *Grind* [816], *Grind* [817], *Grind* [818], *Grind* [819], *Grind* [820], *Grind* [821], *Grind* [822], *Grind* [823], *Grind* [824], *Grind* [825], *Grind* [826], *Grind* [827], *Grind* [828], *Grind* [829], *Grind* [830], *Grind* [831], *Grind* [832], *Grind* [833], *Grind* [834], *Grind* [835], *Grind* [836], *Grind* [837], *Grind*

Coverage map

Edge coverage is a (relatively) poor approximation of a program's state space

Alternatives:

- Context-sensitive edge
- Path
- Data flow

Accuracy vs performance

Fuzzing with Data Dependency Information

Alessandro Mantovani
EURECOM
mantovan@eurecom.fr

Be Sensitive and Collaborative:
Analyzing Impact of Coverage Metrics in Greybox Fuzzing

Jinghan Wang[†], Yue Duan[‡], Wei Song[†], Heng Yin[†], and Chengyu Song[†]

[†]UC Riverside [‡]Cornell University

[†]{jwang131,wsong008}@ucr.edu, {heng,csong}@cs.ucr.edu [‡]yd375@cornell.edu

Registered Report: DATAFLOW
Towards a Data-Flow-Guided Fuzzer

Adrian Herrera
ANU & DST
adrian.herrera@anu.edu.au

Mathias Payer
EPFL
mathias.payer@nebelwelt.net

Antony L. Hosking
ANU
antony.hosking@anu.edu.au

Abstract—Recent advances in fuzz testing several forms of feedback mechanisms, fact that for a large range of programs at coverage alone is insufficient to reveal bugs by this line of research, we examined representations looking for a match between of the structure and adaptability to the testing. In particular, we believe that data and (DDGs) represent a good candidate for this information embedded by this data structure useful to find vulnerable constructs by sections of def-use pairs that would be difficult fuzzer to trigger. Since some portions of graph overlap with the control flow of t possible to reduce the additional instructions to only “interesting” data-flow dependencies the fuzzer to visit the code in a distinct standard methodology.

To test these observations, in this paper, a new approach that rewards fuzzer with code coverage information, but also in the data dependency graph, but also that the adoption of data dependency is coverage-guided fuzzing is a promising solution to discover bugs that would otherwise remain standard coverage approaches. This is due to 72 different vulnerabilities that our data-flow approach can identify when executed on three different datasets.

1. Introduction

In a society that makes software application core of many every-day activities is such software as secure as possible before to the public. This has led to a large amount focused on the development of increasing techniques to discover vulnerabilities, software testing [36], [60], [77], symbolic [62], [71] and dynamic analysis [73].

In the context of dynamic analysis, propagation of coverage is used to measure certain input produces in the software. One of the possible metrics is *path coverage*: all independent paths present in a program. In software testing, the community has coverage for test generation [64], [71] of automatically producing inputs that code locations. The main limitation of

Coverage-guided g most common technique metric, which decide essential parameter of results. While there are many different coverage metrics known about how they affect the fuzzing it is unclear whether it is superior to all the first systematical age metrics in fuzzing discuss the concept of metrically compare different coverage met

study on these metrics: LAVA-M dataset, and of 221 binaries). We have limited resources metric has its unique of branches (thus full grand slam coverage also explore combin

cross-leading, and the fuzzing-based approach of binaries in the CGC that combines fuzzing time, our approach is

We perform a preliminary evaluation of DATAFLOW, comparing fuzzers driven by control flow, taint analysis (both approximate and exact), and data flow. Our initial results suggest that, so far, pure coverage remains the best coverage metric for finding bugs in targets with complex control flow. However, data-flow coverage does show promise in targets where control flow is decoupled from semantics (e.g., parsers). Further evaluation and analysis on a wider range of targets is required.

1 Introduction

Greybox fuzzing is a technique that has been implemented in various tools such as Google's *Mutation* [1], *Trail of Bits* [2], *Challenge* (CGC), *gr* [3], *gr* [4] to be more effective [5] and symbolic execution [6].

USENIX Association

Abstract—Coverage-guided greybox fuzzers rely on feedback derived from control-flow graphs to target specific bugs. These fuzzer are capable of providing feedback offering only a coarse approximation of program behavior. Data-flow intuitively more accurately characterizes program behavior. Despite this advantage, fuzzers driven by data-flow coverage have received comparatively little attention, appearing mainly in academic papers. This is despite the fact that data-flow execution incur a high run-time penalty, impeding fuzzer throughput. Lightweight data-flow alternatives to control-flow fuzzing remain unexplored.

We present DATAFLOW, a greybox fuzzer driven by lightweight data-flow propagation. Whereas control-flow edges represent the order of operations in a program, data-flow edges capture the dependencies between operations that produce data values and the operations that consume them; instead, there may be multiple data-flow edges between two operations. Thus, data-flow coverage captures behaviors not visible as control flow and intuitively discovers more or different bugs. Moreover, we establish a framework for reasoning about data-flow coverage, which includes computational cost of exploration to be balanced with precision.

We perform a preliminary evaluation of DATAFLOW, comparing fuzzers driven by control flow, taint analysis (both approximate and exact), and data flow. Our initial results suggest that, so far, pure coverage remains the best coverage metric for finding bugs in targets with complex control flow. However, data-flow coverage does show promise in targets where control flow is decoupled from semantics (e.g., parsers). Further evaluation and analysis on a wider range of targets is required.

1. Introduction

Fuzzers are an indispensable tool in the software-testing toolbox. The idea of fuzzing—to test a target program by subjecting it to a large number of randomly-generated inputs—is not new. It has been used to detect bugs back to an assignment in a graduate Advanced Operating Systems class [1]. These fuzzers were relatively simple (consisting of a loop that generates random inputs and sends them to the target). They did not model program or input structure, and could only observe the input/output behavior of the target. In contrast, modern fuzzers use sophisticated

International Fuzzing Workshop (FUZZING2022)
24 April 2022, San Diego, CA, USA
ISBN 1-891562-77-4
<https://doi.org/10.14722/fuzzing2022.23001>
www.ndss-symposium.org

program analyses to model program and input structure, and continuously gather dynamic information about the target.

Leveraging dynamic information drives fuzzer efficiency. For example, *coverage-guided greybox fuzzers*—perhaps the most widely-used class of fuzzer—track code paths executed by the target. This allows the fuzzer to focus its mutations on interesting parts of the program. As a result, fewer bugs are found in code never executed, so maximizing the amount of code executed should maximize the number of bugs found. Code coverage serves as an approximation of program behavior, and expanding code coverage implies exploring program behaviors.

Coverage-guided greybox fuzzers are now pervasive. Their success [2] can be attributed to one fuzzer in particular: American Fuzzy Lop (AFL) [3]. AFL is a greybox fuzzer that uses lightweight instrumentation to track edges covered in the target. It uses a copy-on-write (COW) mechanism to keep data-flow coverage captures behaviors not visible as control flow and intuitively discovers more or different bugs. Moreover, we established a framework for reasoning about data-flow coverage, which includes computational cost of exploration to be balanced with precision.

We present DATAFLOW, a greybox fuzzer driven by lightweight data-flow propagation. Whereas control-flow edges represent the order of operations in a program, data-flow edges capture the dependencies between operations that produce data values and the operations that consume them; instead, there may be multiple data-flow edges between two operations. Thus, data-flow coverage captures behaviors not visible as control flow and intuitively discovers more or different bugs. Moreover, we established a framework for reasoning about data-flow coverage, which includes computational cost of exploration to be balanced with precision.

In fuzzing, data flow typically takes the form of dynamic taint analysis (DTA). Here, the target's input data is tainted at its definition site and tracked as it is accessed and used in memory. Unfortunately, accurate DTA is difficult to achieve and expensive to compute. For example, accurate fault-based DTA is expensive [18, 19] and its accuracy highly variable across implementations [18, 20]. Moreover, several real-world programs fail to complete under DTA, increasing deployability concerns. Thus, most widely-deployed greybox fuzzers (e.g., AFL [3], libFuzzer [21], and hongfuzz [22]) eschew DTA in favor of higher fuzzing throughput.

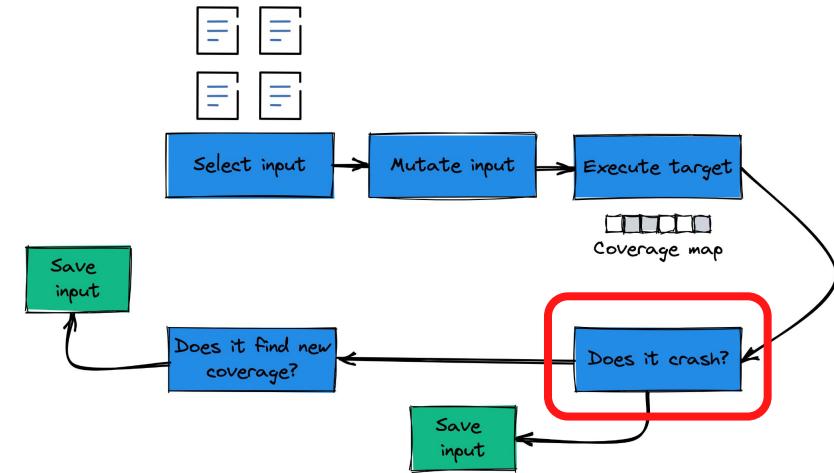
While lightweight alternatives to DTA exist (e.g., Reproducer [23], OpenOoze [19]), the full potential of control- vs. data-flow based fuzzer coverage metrics have not yet been thoroughly explored. To support this exploration, we

¹Miller et al.'s original fuzzer [1] is now known as a *blocker* fuzzer, because it has no knowledge of the target's internals.

Greybox fuzzing

Does it crash?

- Classic memory-safety violation
 - SIGSEGV

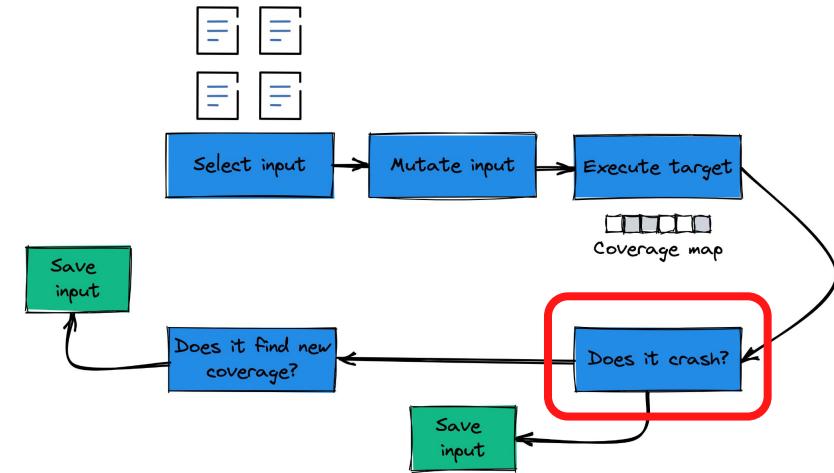


Greybox fuzzing

Does it crash?

- Classic memory-safety violation
 - SIGSEGV

What about other bug types?



Sanitization

- Allow for additional security policies to be defined and checked at runtime
 - Typically compiler-based (e.g., LLVM)
 - But don't have to be

SoK: Sanitizing for Security

Dokyung Song, Julian Lettner, Prabhu Rajasekaran,
Yeoul Na, Stijn Volckaert, Per Larsen, Michael Fraaije
University of California, Irvine

{dokyungs, jlettner, rajasekp, yeouln, stijnv, perl, franz}@uci.edu

RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization

and bugs that elude detection by the actual executive. We observe incorrect padding of a vast number of sanitizers and refined by practitioners with sanitizers of many issues. Specifically, security vulnerabilities of compatibility properties.

I. IN
C and C++ remain the most popular software such as, compilers, and browsers. A significant portion of the world's software is written in C and C++. On the flip side, memory corruption is a common error in almost every aspect. In the case of meeting these requirements, the code vulnerable at the same time, need sophisticated [1]-[4] such as Address Space Layout Randomization (ASLR), Pointer Protection Program (PPP) as functions pointers, and *if*-*else* of the program.

critical role in finding well-understood, while such minimization is often executed such as AFL [6] leverage coverage tools such as Address Sanitizer (ASan) accesses for possible violations. T₄: Type confusion; Bad casting; Type safety; Typecasting; Static_cast; Dynamic_cast; Reinterpret_cast.

1. INTRODUCTION

many software testing criteria: (i) test resulting in shallow coverage close (ii) rely on dynamic binary translation the binary at prohibitively high runt for AFL fuzzing in QEMU mode use `uncond static` requiring broad C++ is well suited for large software projects as it combines high level modularity and abstraction with low level memory access and permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full
reference on the first page of the copy.

use unsound static rewriting based
on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or
republish, to post on servers or to redistribute to lists, requires prior specific permission
and/or a fee. Request permission from [permissions@sagepub.com](http://permissions.drm.sagepub.com).
CCS12, Oct. 30–Nov. 3, 2012, Dallas, TX, USA
© 2012 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ISBN 978-1-4503-1496-8/12/10...\$15.00
[DOI !\[\]\(5ed4e929f29e555498ea58f609acf062_img.jpg\) 10.1145/2388914.2388920](http://doi.acm.org/10.1145/2388914.2388920)

performance. Common examples of C++ software include Google Chrome, MySQL, the Oracle Java Virtual Machine, and Firefox, all of which form the basis of daily computing uses for end-users.

The runtime performance efficiency and backwards compatibility to C come at the price of safety: enforcing memory and type safety is left to the programmer. This lack of safety leads to type confusion vulnerabilities that can be abused to attack programs, allowing the attacker to gain full privileges of these programs. Type confusion vulnerabilities are a challenging mixture between lack of type and memory safety.

Generally, type confusion vulnerabilities are, as the name implies, vulnerabilities that occur when one data type is mistaken for another due to unsafe typecasting, leading to a reinterpretation of the underlying type representation in semantically mismatching contexts.

For instance, a program may cast an instance of a parent class to a descendant class, even though this is neither safe nor allowed at the programming language level if the parent class lacks some specific methods. Such type confusion vulnerabilities are not only wide-spread, as Google Chrome (CVE-2017-2053), Adobe Flash (CVE-2017-2095), Webkit (CVE-2017-2415), Microsoft Internet Explorer (CVE-2015-6184) and PHP (CVE-2016-3185), but also severely critical (e.g., many are demonstrated to be easily exploitable due to erroneous runtime behaviors).

Previous research efforts tried to address to problems through various mechanisms. Existing solutions can be categorized into two types (1) mechanisms that identify objects through existing fields embedded in the objects (such as visible pointers) [6, 29, 38] and (2) mechanisms that leverage disjoint metadata [15, 21] or polymorphic objects [1]. First, solutions that rely on the existing object format have the advantage of avoiding expensive runtime object tracking to maintain disjoint metadata. Unfortunately, these solutions only support polymorphic objects which have a specific form at runtime that allows object identification through their visible pointer. As most software mixes both polymorphic and non-polymorphic objects, these solutions are limited in practice – either developers must manually blacklist unsupported classes or programs end up having unexpected crashes at runtime. Therefore, recent state-of-the-art detectors leverage disjoint metadata for type information. Upon object allocation, the runtime system records the true type of the object in a disjoint metadata table. This approach indeed does not

Sanitization

- Allow for additional security policies to be defined and checked at runtime
- Typically compiler-based (e.g., LLVM)
 - But don't have to be

What can we check for?

<p>SoK: Sanitizing for Security</p> <p>Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, Michael Franz University of California, Irvine {dokyungs,jlettner,rajasekp,yeouln,stijnv,perl,franz}@uci.edu</p> <p>RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization</p> <p>Sushant Dinesh Purdue University Nathan Burow Purdue University Dongyan Xu Purdue University Mathias Payer EPFL</p> <p>HexType: Efficient Detection of Type Confusion Errors for C++</p> <p>Yuseok Jeon Purdue University jeon41@purdue.edu Priyam Biswas Purdue University biswas12@purdue.edu Byoungyoung Lee Purdue University byoungyoung@purdue.edu Mathias Payer Purdue University mathias.payer@nebeldwelt.net Scott Carr Purdue University carr27@purdue.edu</p> <p>ABSTRACT Type confusion, often combined with use-after-free, is the main attack vector to compromise modern C++ software like browsers or virtual machines. Typecasting is a core principle that enables modularity in C++. For performance, most typecasts are only checked statically, i.e., the check only tests if a cast is allowed for the given type hierarchy, ignoring the actual running time of the object. Using an object of an incompatible base type instead of a derived type results in type confusion. Attackers abuse such type confusion issues to attack popular products including Adobe Flash, PHP, Google Chrome, or Firefox.</p> <p>We propose to make all type checks explicit, replacing static checks with full runtime type checks. To minimize the performance impact of our mechanism HexType, we develop both low-overhead data structures and compiler optimizations. To maximize detection coverage, we handle specific object allocation patterns, e.g., placement new or reinterpret_cast which are not handled by other mechanisms.</p> <p>Our prototype results show that, compared to prior work, HexType has at least 1.1 - 6.1 times higher coverage on Firefox benchmarks. For SPEC CPU2006 benchmarks with overhead, we show a 2 - 33x times reduction in overhead. In addition, HexType discovered 4 new type confusion bugs in Qt and Apache Xerces-C++.</p> <p>CCS CONCEPTS • Security and privacy → Systems security; Software and application security;</p> <p>KEYWORDS Type confusion; Bad casting; Type safety; Typecasting; Static_cast; Dynamic_cast; Reinterpret_cast</p> <p>1 INTRODUCTION C++ is well suited for large software projects as it combines high level modularity and abstraction with low memory access and permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyright © 2017, Association for Computing Machinery or its affiliates. Abstracting from the code or otherwise copying or republishing it without permission is expressly prohibited. Abusing rights with or without permission. To do otherwise or republish, to post on servers or to redistribute to lists, requires prior specific permission or reprint permission. Requests to reprint should be addressed to permissions@acm.org. CCS'17 Oct 30-Nov 3, 2017, Dallas, TX, USA © 2017 Copyright held by the owner/authors. Publication rights licensed to ACM. ISBN 978-1-4503-4946-8/17/10. \$15.00 DOI http://dx.doi.org/10.1145/3133963.3134062</p>	<p>Abstract—The C and C++ languages are notoriously insecure yet remain resort to a variety of practices in adversarial contexts. These include <i>i</i> analysis, Dynamic bug fuzzer can find bugs that elude <i>i</i> observe the actual program directly or through incorrect <i>p</i>. A vast number of sanitizers and their variants are proposed by prior overview of sanitizers with security issues. Specifically, the security vulnerabilities, the security vulnerability and compatibility property.</p> <p>I. INTRODUCTION C and C++ remain as systems software such as libraries and browsers. A and leave the programme hardware. On the flip side, every security researcher has been faced with undefined behavior, etc. In short of meeting these tests, make the code vulnerable.</p> <p>At the same time, more sophisticated [1]–[4] solutions such as Address Sp and Data Execution Prevent as Return-Oriented Program such as function pointers control-flow of the program. Data-Oriented Programming can be invoked on legal c program by corrupting on legal c program.</p> <p>As a result, static analysis tools to identify what is deployed in production program analysis, dynamic Static tools analyze the results that are conservative of the code [5]–[9]. In contrast, called “sanitizers”—and output a precise analysis.</p> <p>Sanitizers are now in many vulnerability disclosure and critical role in finding not well-understood, while mitigations Stack Canaries [3], or CFI [4], [5] they cannot pinpoint underlying, combine a feedback-guided fuzzer query information about the executable such as AFL [6] leverage coverage tools such as Address Sanitizer (accesses to possible violations. It as compiler-passes to instrument resulting in low runtime overhead, binary software testing either: (i) resulting in shallow coverage close (ii) rely on dynamic binary translation the binary at prohibitively high runt for AFL fuzzing in QEMU mode use unsound static rewriting based</p> <p>to determine disjoint metadata. Unfortunately, these solutions only support polymorphic objects which have a specific form at runtime that allows object identification through their viable pointer. As most software mixes both polymorphic and non-polymorphic objects, these solutions are limited in practice as either developers must manually block specific classes or programs and hope for unexpected crashes at runtime. Therefore, recent state-of-the-art detectors leverage disjoint metadata for type information. Upon object allocation, the runtime system records the true type of the object in a disjoint metadata table. This approach indeed does not</p>
--	--

Sanitization

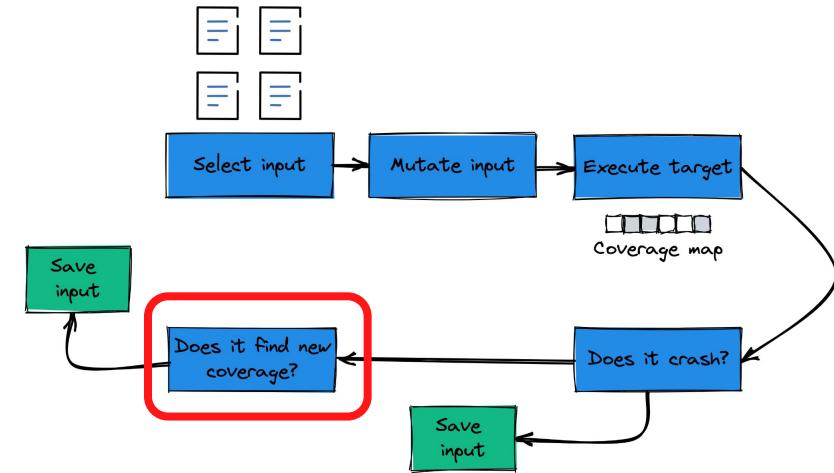
Anything we can encode as an **invariant**

- Address Sanitizer (ASan)
- Undefined behavior Sanitizer (UBSan)
- Memory Sanitizer (MSan)
- LeakSanitizer (LSan)
- ThreadSanitizer (TSan)

Greybox fuzzing

Does it find new coverage?

- Save input
- Return to start



What about...

- Non-file, non-*nix fuzzing
 - E.g., network services, OS kernel, IoT, ...
- Overcoming “roadblocks”
 - E.g., complex conditionals

*nix file fuzzing

- Primary focus of academic research
- Assumes an “obvious” entry point
 - AFL-style fuzzing: main + fread
 - libFuzzer: dedicated LLVMFuzzerTestOneInput
- Commonly assumes source code

*nix file fuzzing

- Primary focus of academic research
- Assumes an “obvious” entry point
 - AFL-style fuzzing: main + fread
 - libFuzzer: dedicated LLVMFuzzerTestOneInput
- Commonly assumes source code

What is the entry point for a network service / OS kernel / IoT device? 🤔

Network apps

Challenges

- State
- Setup/teardown connection cost
- What is “coverage”?

Solutions

- Snapshots
 - No need to start from scratch each time
- Annotate/infer states

FIRM-AFL: High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation

Yaowen Zheng^{1,2,3*}, Ali Davanloo¹, Beijing Kinsins¹

³ School of Cyber
fzhangyaowen,zhuhongshu@

Abstract

Cyber attacks against IoT devices are often carried out by exploiting software vulnerabilities. Fuzzing is an effective software testing method for vulnerability discovery. In this work, we propose FIRM-AFL, a high-throughput greybox fuzzer. It addresses two fundamental problems. First, it addresses compatibility issues between OSes and emulators. Second, it addresses the performance bottleneck caused by system-mode emulation called augmented process emulation. mode emulation and user-mode emulation provide system-mode emulation and high throughput respectively. Our evaluations show that FIRM-AFL can significantly improve the exploitability of vulnerabilities in IoT programs: (1) it achieves up to 8 times higher than system fuzzing; and (2) FIRM-AFL is able to fuzz much faster than system-mode emulation and is able to find 0-day vulnerabilities.

1 Introduction

The security impact of IoT devices on society has been increasing rapidly. By 2020, the number of connected IoT devices will reach 20 billion [10]. This creates a surface leaving almost everybody at risk. Hackers leverage the lack of security to create large botnets (e.g., Mirai, VPNF). Malware attacks exploit the vulnerabilities to penetrate into the IoT devices. As defenders to discover vulnerabilities them before attackers.

*The work was done while visiting University of Washington.
Corresponding author

1 Introduction

Security vulnerabilities often arise inside operating systems. A vulnerability can completely compromise a popular technique fixing such critical bugs. Fuzzers focus primarily on the OS kernel and user-space.

Incremental Snapshot Systems (EuroSys'20)

New York, NY, USA, 18–22 July 2022, Virtual, South Korea

© 2022 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503

https://doi.org/10.1145/3533767.3534376

1 INTRODUCTION

Fuzzing is an effective technique for testing software systems, with popular fuzzers such as AFL and LibFuzzer having found thousands of bugs in both open-source and commercial software. For instance,

INSTA '22, July 18–22, 2022, Virtual, South Korea

For the purpose of open access, the author has applied a Creative Commons Attribution (CC BY) license to any Author Accepted Manuscript version arising.

The definitive version of this paper will be published in the Proceedings of the 1st ACM SIGSOF

International Symposium on Software Testing and Analysis (INSTA '22), July 18–22, 2022, Virtual, South Korea.

https://doi.org/10.1145/3533767.3534376

2 FROM AFL TO AFLNET TO SNAPFUZZ

In this section, we first discuss how AFL and AFLNet work, focusing

on their internal architecture and performance implications, and then provide an overview of SnapFuzz's architecture and main contributions.

MoonShine: Optimizing OS Fuzzer Seed Selection with Trace Distillation

Shankara Pailoor, Andrew Aday, and Suman Jana
Columbia University

Nyx-Net: Network Fuzzing with Incremental Snapshots

Sergej Schumilo¹, Cornelius Aschermann¹, Andrea Jemmett², Ali Abbasi¹, and Thorsten Holz³
¹Ruhr-Universität Bochum, ²Vrije Universiteit Amsterdam
³CISPA Helmholtz Center for Information Security

SnapFuzz: High-Throughput Fuzzing of Network Applications

Anastasios Andronidis
Imperial College London
London, United Kingdom
a.andronidis@imperial.ac.uk

Cristian Cadar
Imperial College London
London, United Kingdom
c.cadar@imperial.ac.uk

ABSTRACT

In recent years, fuzz testing has benefited from increased computational power and important algorithmic advances, leading to systems that have discovered many critical bugs and vulnerabilities in production software. Despite these successes, not all applications can be fuzzed efficiently. In particular, stateful applications such as network protocol implementations are constrained by a low fuzzing throughput and the need to develop complex fuzzing harnesses that involve custom time delays and clean-up scripts.

In this paper, we propose *SnapFuzz*, a novel framework for fuzzing application-layer protocols. SnapFuzz relies on a robust infrastructure that transforms slow asynchronous network communication into fast synchronous communication, snapshots the target at the latest point at which it is safe to do so, speeds up file operations by redirecting them to a custom in-memory filesystem, and removes the need for many fragile modifications, such as configuring time delays or writing clean-up scripts.

Using *SnapFuzz*, we fuzz five popular networking applications: Lighttpd, Tomcat, Nginx, LibreSSL, and Dnsperf. We report impressive performance speedups of 62.8x, 41.2x, 36.6x, 24.6x, and 8.4x, respectively, with significantly simpler fuzzing harnesses in all cases. Due to its advantages, *SnapFuzz* has also found 12 extra crashes compared to AFLNet in these applications.

CCS Concepts: • Testing • Software and its engineering → Software testing and debugging • Security and privacy → Systems security

KEYWORDS

Fuzzing, network protocol implementations, stateful applications, ACM Reference Format

Anastasios Andronidis and Cristian Cadar. 2022. SnapFuzz: High-Throughput Fuzzing of Network Applications. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISTAT '22)*, July 18–22, 2022, Virtual, New York, NY, USA, 12 pages.

https://doi.org/10.1145/3533767.3534376

1 INTRODUCTION

Fuzzing is an effective technique for testing software systems, with popular fuzzers such as AFL and LibFuzzer having found thousands of bugs in both open-source and commercial software. For instance,

INSTA '22, July 18–22, 2022, Virtual, South Korea

For the purpose of open access, the author has applied a Creative Commons Attribution (CC BY) license to any Author Accepted Manuscript version arising.

The definitive version of this paper will be published in the Proceedings of the 1st ACM SIGSOF

International Symposium on Software Testing and Analysis (INSTA '22), July 18–22, 2022, Virtual, South Korea.

https://doi.org/10.1145/3533767.3534376

2 FROM AFL TO AFLNET TO SNAPFUZZ

In this section, we first discuss how AFL and AFLNet work, focusing

on their internal architecture and performance implications, and then provide an overview of SnapFuzz's architecture and main contributions.

OS kernel

Challenges

- Measuring coverage
- Performance
- Seeds?

Solutions

- kCOV + kASan
- Hypervisor + PMU
- Seeds = syscall traces

FIRM-AFL: High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation

Yaowen Zheng^{1,2,3*}, Ali Davari¹, Beijing Kinsins

³ School of Cyber
{zhengyaowen,zhuhong}so

Abstract

Cyber attacks against IoT devices are often carried out by exploiting software vulnerabilities. Fuzzing is an effective software testing methodology for vulnerability discovery. In this work, we first propose a high-throughput greybox fuzzer (FIRM-AFL) to address two fundamental problems. First, it addresses compatibility issues between Fuzzing and Process Emulation. Second, it addresses the performance bottleneck caused by system-mode emulation called augmented process emulation, mode emulation and user-mode emulation. Our evaluations show that FIRM-AFL significantly improves the exploitability of vulnerabilities in IoT programs: (1) it achieves average 8.2 times higher than system fuzzing; and (2) FIRM-AFL is able to fuzz much faster than system-mode emulation and is able to find 0-day vulnerabilities.

1 Introduction

The security impact of IoT devices on society has been increasing rapidly. By 2020, the number of connected IoT devices will reach 20 billion [10]. This creates a surface leaving almost everybody at risk. Hackers leverage the lack of security in IoT devices to create large botnets (e.g., Mirai, VPNf). Malware attacks exploit the vulnerabilities to penetrate into the IoT devices. As defenders to discover vulnerabilities them before attackers.

*The work was done while visiting University of Washington.
Corresponding author

USENIX Association

ACM Reference Format:
Sergej Schumilo, Cobus Bassi, and Thorsten Holz. 2022. MoonShine: Optimizing OS Fuzzer Seed Selection with Trace Distillation. In Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISTFA '22), July 18–22, 2022, Virtual, South Korea. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3533767.3534376>

CCS Concepts: • Test and validation

Keywords: Testing

Security vulnerability after-fuzz inside operating system is a major concern. It is especially dangerous if it can completely compromise the system. A popular technique to fix such critical vulnerabilities is to use fuzzer focus primarily on the OS kernel and use

MoonShine: Optimizing OS Fuzzer Seed Selection with Trace Distillation

Shankara Pailoor, Andrew Aday, and Suman Jana
Columbia University

Abstract

OS fuzzers primarily target the OS kernel for vulnerability discovery. In this work, we propose a novel fuzzer, MoonShine, which addresses two fundamental issues. First, it addresses compatibility issues between Fuzzing and Process Emulation. Second, it addresses the performance bottleneck caused by system-mode emulation called augmented process emulation, mode emulation and user-mode emulation. Our evaluations show that MoonShine significantly improves the exploitability of vulnerabilities in IoT programs: (1) it achieves average 8.2 times higher than system fuzzing; and (2) MoonShine is able to fuzz much faster than system-mode emulation and is able to find 0-day vulnerabilities.

Abstract

Coverage-guided fuzzer and we have made significant progress recently. However, network service providers often run on hand-coded sequences of system calls. Unfortunately, the diversity of the sequences limits the effectiveness of the fuzzer.

In this paper, we propose a new approach for distilling seed traces of real-world programs across different platforms. We implemented our approach in Nx-Net and found that Nx-Net is able to targets that no other fuzzer can. Nx-Net is able to span multiple servers and protocols simultaneously, such as Lighttpd, Firefox's IPC mechanism, and the network protocol implementations are constrained by a low fuzzing throughput and the need to develop complex fuzzing harnesses that involve custom time delays and clean-up scripts.

In this paper, we present SnapFuzz, a novel framework for application-specific network fuzzing. SnapFuzz is built on top of a robust infrastructure that transforms slow asynchronous network communication into fast synchronous communication, snapshots the target at the latest point at which it is safe to do so, speeds up file operations by redirecting them to a custom in-memory filesystem, and removes the need for many fragile modifications, such as configuring time delays or writing clean-up scripts.

Using SnapFuzz, we fuzzed five popular networking applications: Lighttpd, Tomcat, MySQL, PostgreSQL, and Elasticsearch. We report impressive performance speedups of 62.8x, 41.2x, 36.6x, 24.6x, and 8.4x, respectively, with significantly simpler fuzzing harnesses in all cases. Due to its advantages, SnapFuzz has also found 12 extra crashes compared to AFLNet in these applications.

CCS CONCEPTS

• Software and its engineering → Software testing and debugging; Security and privacy → Systems security.

KEYWORDS

Fuzzing, network protocol implementations, stateful applications

ACM Reference Format

Anastasios Andronidis and Cristian Cadar. 2022. SnapFuzz: High-Throughput Fuzzing of Network Applications. In Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISTFA '22), July 18–22, 2022, Virtual, South Korea. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3533767.3534376>

CCS CONCEPTS

• Software and its engineering → Software testing and debugging; Security and privacy → Systems security.

KEYWORDS

Fuzzing, network protocol implementations, stateful applications

ACM Reference Format

Anastasios Andronidis and Cristian Cadar. 2022. SnapFuzz: High-Throughput Fuzzing of Network Applications. In Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISTFA '22), July 18–22, 2022, Virtual, South Korea. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3533767.3534376>

INSTA '22, July 18–22, 2022, Virtual, South Korea

© 2022 Copyright held by the author(s). Publication rights licensed to ACM.

For the purpose of open access, the author has applied a Creative Commons Attribution (CC BY) license to any Author Accepted Manuscript version arising.

The definitive version of this article is available at <https://doi.org/10.1145/3533767.3534376>.

INSTA '22, July 18–22, 2022, Virtual, South Korea

© 2022 Copyright held by the author(s). Publication rights licensed to ACM.

For the purpose of open access, the author has applied a Creative Commons Attribution (CC BY) license to any Author Accepted Manuscript version arising.

The definitive version of this article is available at <https://doi.org/10.1145/3533767.3534376>.

INSTA '22, July 18–22, 2022, Virtual, South Korea

© 2022 Copyright held by the author(s). Publication rights licensed to ACM.

For the purpose of open access, the author has applied a Creative Commons Attribution (CC BY) license to any Author Accepted Manuscript version arising.

The definitive version of this article is available at <https://doi.org/10.1145/3533767.3534376>.

INSTA '22, July 18–22, 2022, Virtual, South Korea

© 2022 Copyright held by the author(s). Publication rights licensed to ACM.

For the purpose of open access, the author has applied a Creative Commons Attribution (CC BY) license to any Author Accepted Manuscript version arising.

The definitive version of this article is available at <https://doi.org/10.1145/3533767.3534376>.

INSTA '22, July 18–22, 2022, Virtual, South Korea

© 2022 Copyright held by the author(s). Publication rights licensed to ACM.

For the purpose of open access, the author has applied a Creative Commons Attribution (CC BY) license to any Author Accepted Manuscript version arising.

The definitive version of this article is available at <https://doi.org/10.1145/3533767.3534376>.

INSTA '22, July 18–22, 2022, Virtual, South Korea

© 2022 Copyright held by the author(s). Publication rights licensed to ACM.

For the purpose of open access, the author has applied a Creative Commons Attribution (CC BY) license to any Author Accepted Manuscript version arising.

The definitive version of this article is available at <https://doi.org/10.1145/3533767.3534376>.

INSTA '22, July 18–22, 2022, Virtual, South Korea

© 2022 Copyright held by the author(s). Publication rights licensed to ACM.

For the purpose of open access, the author has applied a Creative Commons Attribution (CC BY) license to any Author Accepted Manuscript version arising.

The definitive version of this article is available at <https://doi.org/10.1145/3533767.3534376>.

INSTA '22, July 18–22, 2022, Virtual, South Korea

© 2022 Copyright held by the author(s). Publication rights licensed to ACM.

For the purpose of open access, the author has applied a Creative Commons Attribution (CC BY) license to any Author Accepted Manuscript version arising.

The definitive version of this article is available at <https://doi.org/10.1145/3533767.3534376>.

INSTA '22, July 18–22, 2022, Virtual, South Korea

© 2022 Copyright held by the author(s). Publication rights licensed to ACM.

For the purpose of open access, the author has applied a Creative Commons Attribution (CC BY) license to any Author Accepted Manuscript version arising.

The definitive version of this article is available at <https://doi.org/10.1145/3533767.3534376>.

INSTA '22, July 18–22, 2022, Virtual, South Korea

© 2022 Copyright held by the author(s). Publication rights licensed to ACM.

For the purpose of open access, the author has applied a Creative Commons Attribution (CC BY) license to any Author Accepted Manuscript version arising.

The definitive version of this article is available at <https://doi.org/10.1145/3533767.3534376>.

INSTA '22, July 18–22, 2022, Virtual, South Korea

© 2022 Copyright held by the author(s). Publication rights licensed to ACM.

For the purpose of open access, the author has applied a Creative Commons Attribution (CC BY) license to any Author Accepted Manuscript version arising.

The definitive version of this article is available at <https://doi.org/10.1145/3533767.3534376>.

INSTA '22, July 18–22, 2022, Virtual, South Korea

© 2022 Copyright held by the author(s). Publication rights licensed to ACM.

For the purpose of open access, the author has applied a Creative Commons Attribution (CC BY) license to any Author Accepted Manuscript version arising.

The definitive version of this article is available at <https://doi.org/10.1145/3533767.3534376>.

INSTA '22, July 18–22, 2022, Virtual, South Korea

© 2022 Copyright held by the author(s). Publication rights licensed to ACM.

For the purpose of open access, the author has applied a Creative Commons Attribution (CC BY) license to any Author Accepted Manuscript version arising.

The definitive version of this article is available at <https://doi.org/10.1145/3533767.3534376>.

INSTA '22, July 18–22, 2022, Virtual, South Korea

© 2022 Copyright held by the author(s). Publication rights licensed to ACM.

For the purpose of open access, the author has applied a Creative Commons Attribution (CC BY) license to any Author Accepted Manuscript version arising.

The definitive version of this article is available at <https://doi.org/10.1145/3533767.3534376>.

INSTA '22, July 18–22, 2022, Virtual, South Korea

© 2022 Copyright held by the author(s). Publication rights licensed to ACM.

For the purpose of open access, the author has applied a Creative Commons Attribution (CC BY) license to any Author Accepted Manuscript version arising.

The definitive version of this article is available at <https://doi.org/10.1145/3533767.3534376>.

INSTA '22, July 18–22, 2022, Virtual, South Korea

© 2022 Copyright held by the author(s). Publication rights licensed to ACM.

For the purpose of open access, the author has applied a Creative Commons Attribution (CC BY) license to any Author Accepted Manuscript version arising.

The definitive version of this article is available at <https://doi.org/10.1145/3533767.3534376>.

INSTA '22, July 18–22, 2022, Virtual, South Korea

© 2022 Copyright held by the author(s). Publication rights licensed to ACM.

For the purpose of open access, the author has applied a Creative Commons Attribution (CC BY) license to any Author Accepted Manuscript version arising.

The definitive version of this article is available at <https://doi.org/10.1145/3533767.3534376>.

INSTA '22, July 18–22, 2022, Virtual, South Korea

© 2022 Copyright held by the author(s). Publication rights licensed to ACM.

For the purpose of open access, the author has applied a Creative Commons Attribution (CC BY) license to any Author Accepted Manuscript version arising.

The definitive version of this article is available at <https://doi.org/10.1145/3533767.3534376>.

INSTA '22, July 18–22, 2022, Virtual, South Korea

© 2022 Copyright held by the author(s). Publication rights licensed to ACM.

For the purpose of open access, the author has applied a Creative Commons Attribution (CC BY) license to any Author Accepted Manuscript version arising.

The definitive version of this article is available at <https://doi.org/10.1145/3533767.3534376>.

INSTA '22, July 18–22, 2022, Virtual, South Korea

© 2022 Copyright held by the author(s). Publication rights licensed to ACM.

For the purpose of open access, the author has applied a Creative Commons Attribution (CC BY) license to any Author Accepted Manuscript version arising.

The definitive version of this article is available at <https://doi.org/10.1145/3533767.3534376>.

INSTA '22, July 18–22, 2022, Virtual, South Korea

© 2022 Copyright held by the author(s). Publication rights licensed to ACM.

For the purpose of open access, the author has applied a Creative Commons Attribution (CC BY) license to any Author Accepted Manuscript version arising.

The definitive version of this article is available at <https://doi.org/10.1145/3533767.3534376>.

INSTA '22, July 18–22, 2022, Virtual, South Korea

© 2022 Copyright held by the author(s). Publication rights licensed to ACM.

For the purpose of open access, the author has applied a Creative Commons Attribution (CC BY) license to any Author Accepted Manuscript version arising.

The definitive version of this article is available at <https://doi.org/10.1145/3533767.3534376>.

INSTA '22, July 18–22, 2022, Virtual, South Korea

© 2022 Copyright held by the author(s). Publication rights licensed to ACM.

For the purpose of open access, the author has applied a Creative Commons Attribution (CC BY) license to any Author Accepted Manuscript version arising.

The definitive version of this article is available at <https://doi.org/10.1145/3533767.3534376>.

INSTA '22, July 18–22, 2022, Virtual, South Korea

© 2022 Copyright held by the author(s). Publication rights licensed to ACM.

For the purpose of open access, the author has applied a Creative Commons Attribution (CC BY) license to any Author Accepted Manuscript version arising.

The definitive version of this article is available at <https://doi.org/10.1145/3533767.3534376>.

INSTA '22, July 18–22, 2022, Virtual, South Korea

© 2022 Copyright held by the author(s). Publication rights licensed to ACM.

For the purpose of open access, the author has applied a Creative Commons Attribution (CC BY) license to any Author Accepted Manuscript version arising.

The definitive version of this article is available at <https://doi.org/10.1145/3533767.3534376>.

INSTA '22, July 18–22, 2022, Virtual, South Korea

© 2022 Copyright held by the author(s). Publication rights licensed to ACM.

For the purpose of open access, the author has applied a Creative Commons Attribution (CC BY) license to any Author Accepted Manuscript version arising.

The definitive version of this article is available at <https://doi.org/10.1145/3533767.3534376>.

INSTA '22, July 18–22, 2022, Virtual, South Korea

© 2022 Copyright held by the author(s). Publication rights licensed to ACM.

For the purpose of open access, the author has applied a Creative Commons Attribution (CC BY) license to any Author Accepted Manuscript version arising.

The definitive version of this article is available at <https://doi.org/10.1145/3533767.3534376>.

INSTA '22, July 18–22, 2022, Virtual, South Korea

© 2022 Copyright held by the author(s). Publication rights licensed to ACM.

For the purpose of open access, the author has applied a Creative Commons Attribution (CC BY) license to any Author Accepted Manuscript version arising.

The definitive version of this article is available at <https://doi.org/10.1145/3533767.3534376>.

INSTA '22, July 18–22, 2022, Virtual, South Korea

© 2022 Copyright held by the author(s). Publication rights licensed to ACM.

For the purpose of open access, the author has applied a Creative Commons Attribution (CC BY) license to any Author Accepted Manuscript version arising.

The definitive version of this article is available at <https://doi.org/10.1145/3533767.3534376>.

INSTA '22, July 18–22, 2022, Virtual, South Korea

© 2022 Copyright held by the author(s). Publication rights licensed to ACM.

For the purpose of open access, the author has applied a Creative Commons Attribution (CC BY) license to any Author Accepted Manuscript version arising.

The definitive version of this article is available at <https://doi.org/10.1145/3533767.3534376>.

INSTA '22, July 18–22, 2022, Virtual, South Korea

© 2022 Copyright held by the author(s). Publication rights licensed to ACM.

For the purpose of open access, the author has applied a Creative Commons Attribution (CC BY) license to any Author Accepted Manuscript version arising.

The definitive version of this article is available at <https://doi.org/10.1145/3533767.3534376>.

INSTA '22, July 18–22, 2022, Virtual, South Korea

© 2022 Copyright held by the author(s). Publication rights licensed to ACM.

For the purpose of open access, the author has applied a Creative Commons Attribution (CC BY) license to any Author Accepted Manuscript version arising.

The definitive version of this article is available at <https://doi.org/10.1145/3533767.3534376>.

INSTA '22, July 18–22, 2022, Virtual, South Korea

© 2022 Copyright held by the author(s). Publication rights licensed to ACM.

For the purpose of open access, the author has applied a Creative Commons Attribution (CC BY) license to any Author Accepted Manuscript version arising.

The definitive version of this article is available at <https://doi.org/10.1145/3533767.3534376>.

INSTA '22, July 18–22, 2022, Virtual, South Korea

© 2022 Copyright held by the author(s). Publication rights licensed to ACM.

For the purpose of open access, the author has applied a Creative Commons Attribution (CC BY) license to any Author Accepted Manuscript version arising.

The definitive version of this article is available at [https://doi.org/10.1145/3533767.](https://doi.org/10.1145/3533767.3534376)

IoT

Challenges

- Measuring coverage
- Performance
- Seeds?

Solutions

- QEMU (slow / incomplete)
- Avatar² orchestration

FIRM-AFL: High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation

Yaowen Zheng^{1,2,3*}, Ali Davar¹,
Beijing Key
Ins

³ School of Cyber
fzhangyaowen,zhuhongso

MoonShine: Optimizing OS Fuzzer Seed Selection with Trace Distillation

Shankara Pailoor, Andrew Aday, and Suman Jana
Columbia University

Abstract

Cyber attacks against IoT devices are often carried out by exploiting software vulnerabilities. Fuzzing is an effective software testing methodology for discovering such vulnerabilities. In this work, we first propose a high-throughput greybox fuzzer (FIRM-AFL) to address two fundamental problems. First, it addresses compatibility issues between the OS kernel and user-mode emulators. Second, it addresses the performance bottleneck caused by system-mode emulation called augmented process emulation, mode emulation and user-mode emulation. FIRM-AFL provides high throughput and low latency. Our evaluations show that FIRM-AFL can significantly improve the exploitability of vulnerabilities in IoT programs: (1) it achieves up to 8.2 times higher than system fuzzing; and (3) FIRM-AFL is able to fuzz much faster than system-mode emulators and is able to find 0-day vulnerabilities.

1 Introduction

The security impact of IoT devices on society has been increasing rapidly. By 2020, the number of connected IoT devices will reach 20 billion [10]. This creates a surface leaving almost everybody at risk. Hackers leverage the lack of security in IoT devices to create large botnets (e.g., Mirai, VPNf). Malware attacks exploit the vulnerability of IoT devices to penetrate into the IoT devices. As defenders to discover vulnerabilities them before attackers.

*The work was done while visiting University of Washington.
Corresponding author

USENIX Association

ACM Reference Format:
Sergej Schumilo, Co
basi, and Thorsten Holz
Incremental Snapsh
puter Systems (Euro
New York, NY, USA,

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

CCS Concepts: • Testin

• Software and its engineering • Software testing and de

bugging • Security and privacy • Systems, comput

Keywords: Testing

Security vulnerability detection, fuzzer, incremental snapshotting, network communication, system-level testing, software testing and analysis.

ACM Reference Format:

Anastasios Andronidis and Cristian Cadar. 2022. SnapFuzz: High-Throughput Fuzzing of Network Applications. In Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISTFA '22), July 18–22, 2022, Virtual, South Korea. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3533767.3534376

1 INTRODUCTION

Fuzzing is an effective technique for testing software systems, with

popular fuzzers such as AFL and LibFuzzer having found thousands

of bugs in both open-source and commercial software. For instance,

INSTA '22, July 18–22, 2022, Virtual, South Korea

© 2022 Copyright held

ACM ISBN 978-1-4503

https://doi.org/10.1145/3533767.3534376

2 FROM AFL TO AFLNET TO SNAPFUZZ

In this section, we first discuss how AFL and AFLNet work, focusing

on their internal architecture and performance implications, and then provide an overview of SnapFuzz's architecture and main contributions.

Abstract

OS fuzzers primarily target the OS kernel rather than user-space. To address this limitation, we propose MoonShine, a novel OS fuzzer that generates good seeds by learning from the behavior of existing fuzzers. MoonShine uses a coverage-guided approach to select seeds that cover more code than previous fuzzers. We evaluate MoonShine across various benchmarks and show that it finds more bugs than existing fuzzers.

In this paper, we propose MoonShine, a novel OS fuzzer that generates good seeds by learning from the behavior of existing fuzzers. MoonShine uses a coverage-guided approach to select seeds that cover more code than previous fuzzers. We evaluate MoonShine across various benchmarks and show that it finds more bugs than existing fuzzers.

Abstract

Coverage-guided fuzzer and we have recently. However, test network service methods. In this implementation of MoonShine, we propose a novel approach that can span multiple servers, and the Process Communal-of-the-art method up to 300x and co Nx-Net is able to targets that no other fuzzer can. Nx-Net to play a significant role in the development of complex fuzzing harnesses that involve custom time delays and clean-up scripts.

In this paper, we propose MoonShine, a novel OS fuzzer that generates good seeds by learning from the behavior of existing fuzzers. MoonShine uses a coverage-guided approach to select seeds that cover more code than previous fuzzers. We evaluate MoonShine across various benchmarks and show that it finds more bugs than existing fuzzers.

ABSTRACT

In recent years, fuzz testing has benefited from increased computational power and important algorithmic advances, leading to systems that have discovered many critical bugs and vulnerabilities in production software. Despite these successes, not all applications can be fuzzed efficiently. In particular, stateful applications such as network protocol implementations are constrained by a low fuzzing throughput and the need to develop complex fuzzing harnesses that involve custom time delays and clean-up scripts.

In this paper, we present SnapFuzz, a novel fuzzer framework for testing application-layer protocols. SnapFuzz relies on a robust infrastructure that transforms slow asynchronous network communication into fast synchronous communication, snapshots the target at the latest point at which it is safe to do so, speeds up file operations by redirecting them to a custom in-memory filesystem, and removes the need for many fragile modifications, such as configuring time delays or writing clean-up scripts.

Using SnapFuzz, we fuzz five popular networking applications: Lighttpd, Tomcat, Apache, MySQL, and PostgreSQL. We report impressive performance speedups of 62.8x, 41.2x, 36.6x, 24.6x, and 8.4x, respectively, with significantly smaller fuzzing harnesses in all cases. Due to its advantages, SnapFuzz has also found 12 extra crashes compared to AFLNet in these applications.

CCS CONCEPTS

• Software and its engineering • Software testing and de

bugging • Security and privacy • Systems, comput

Keywords: Testing

Network protocol implementations, stateful applications

ACM Reference Format:

Anastasios Andronidis and Cristian Cadar. 2022. SnapFuzz: High-Throughput Fuzzing of Network Applications. In Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISTFA '22), July 18–22, 2022, Virtual, South Korea. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3533767.3534376

1 INTRODUCTION

Fuzzing is an effective technique for testing software systems, with

popular fuzzers such as AFL and LibFuzzer having found thousands

of bugs in both open-source and commercial software. For instance,

INSTA '22, July 18–22, 2022, Virtual, South Korea

© 2022 Copyright held

ACM ISBN 978-1-4503

https://doi.org/10.1145/3533767.3534376

2 FROM AFL TO AFLNET TO SNAPFUZZ

In this section, we first discuss how AFL and AFLNet work, focusing

on their internal architecture and performance implications, and then provide an overview of SnapFuzz's architecture and main contributions.

Google has discovered over 25,000 bugs in their products and over 22,000 bugs in open-source code using greybox fuzzing [18]. Unfortunately, not all software can benefit from such fuzzing campaigns. One important class of software, network protocol implementations, is difficult to fuzz. There are two main difficulties: the fact that in-depth testing of such applications needs to be aware of the network protocols they implement (e.g., FTP, DICOM, SIP), and the fact that they have side effects, such as writing data to the file system, when performing fuzzing tests [18].

There are two main approaches for testing such software in a meaningful way. One approach, adopted by Google's OSS-Fuzz, is to write unit-level test drivers that interact with the software via its API [21]. While such an approach can be effective, it requires significant manual effort, and does not perform system-level testing where actual server instances interact with actual clients.

A second approach, used by AFLNet [30], performs system-level testing by injecting random message exchanges between them, which nevertheless follow the underlying network protocol. Furthermore, it does so without needing a specification of the protocol, but rather by using a corpus of real message exchanges between server and clients. AFLNet's approach has significant advantages, requiring less manual effort and performing end-to-end testing at the protocol level.

While AFLNet makes important advances in terms of fuzzing

network protocols, it has two main limitations. First, it requires users to specify the network protocols they want to fuzz, to ensure the protocol is followed, and to write clean-up scripts to reset the state across fuzzing iterations. Second, it has poor fuzzing performance, caused by asynchronous network communication, various time delays, and expensive file system operations, among others.

SnapFuzz addresses both of these challenges through a robust

architecture that transforms slow asynchronous network communi

cation into fast synchronous communication, speeds up file oper

ations, and interacts with the network for clearing up file system

filesystems and improving other metrics such as delaying and au

tomating the forkserver placement, correctly handling signal pro

tection and eliminating developer-added delays.

These improvements significantly simplify the construction of

fuzzing harnesses for network applications and dramatically im

prove fuzzing throughput in the range of 8.4x to 62.8x (mean:

30.6x) for a set of five popular server benchmarks.

INSTA '22, July 18–22, 2022, Virtual, South Korea

© 2022 Copyright held

ACM ISBN 978-1-4503

https://doi.org/10.1145/3533767.3534376

2 FROM AFL TO AFLNET TO SNAPFUZZ

In this section, we first discuss how AFL and AFLNet work, focusing

on their internal architecture and performance implications, and then provide an overview of SnapFuzz's architecture and main contributions.

INSTA '22, July 18–22, 2022, Virtual, South Korea

© 2022 Copyright held

ACM ISBN 978-1-4503

https://doi.org/10.1145/3533767.3534376

Overcoming “roadblocks”

Program constraints that are hard to meet

Solutions

- Whitebox fuzzing
 - Concolic execution
 - Rewrite the target 😊

Driller: Augmenting Fuzzing Through Selective Symbolic Execution

Nick Stephens,
Jacana Books

{stephe}

Abstract—Memory corruption vulnerability presents risk in software, which attaches unauthorized access to confidential data with access to sensitive data are becoming number of potentially exploitable sites resulting in a greater need for automation. DARPA recently funded a competition in prize money, to further research vulnerability finding and patching, research in this area. Current techniques include static, dynamic, and hybrid, each having their own advantages. A common limitation of systems design trigger vulnerabilities is that they or struggle to exercise deeper paths in the code.

We present Driller, a hybrid solution which leverages fuzzing and symbol resolution in a complementary manner, to find bugs used to exercise *compartments*, concilic execution is used to generate complex checks separating the compartments of two techniques, avoiding the path explosion inherent in instrumenting every function. We explore to only one of the paths deemed interesting by the project, to generate inputs for conditions that the evaluate Driller on 126 applications event of the DARPA Cyber Grand Challenge, efficacy by identifying the same number of bugs as the top-scoring team at the same time, as the top-scoring team.

J. J. JONES

1 Introduction

Despite efforts to increase the against security flaws, vulnerability commonplace. In fact, in recent security vulnerabilities has increased. Furthermore, despite the introductory and execution redirection mitigation flaws account for over a third of all in the last year [14].

Permit is granted to freely reproduce all or part of purposes is granted provided that copies bear on the first page. Reproduction for commercial, without the prior written consent of the In author (for reproduction of an entire paper or if the paper was not published with the In permission of the 24th February 2016, San Diego, Copyright 2016 Internet Society, ISBN 1-891 http://ds3.dsi.org/pn/10.47272/2016.23368 (I.E.

Symbolic execution with SYMCC: Don't interpret, compile!



Sebastian Poeplau Aurélien Francillon

Aurélien Francillo

T-Fuzz: fuzzing by program transformation

Hui Li
Purdue U
peng124@p

Yan Shoshitaishvili
Arizona State University
yans@asu.edu

Mathias Payer
Purdue University
mathias.payer@nebelwelt.net

fuzzing has become a standard in software development to improve reliability and security [3], [4].

Fuzzers can be roughly divided into two categories based on how inputs are produced: generational fuzzers and mutational fuzzers.

Abstract—Fuzzing is a simple yet effective approach to discover software bugs utilizing randomly generated inputs. However, it is limited by coverage and cannot find bugs hidden in deep execution paths of the program because the randomly generated inputs fail complex *sanity checks*, e.g., checks on magic values, checksums, or hashes.

To improve coverage, existing approaches rely on imprecise heuristics or complex initial mutation strategies (e.g., symbolic execution, search analysis) to identify many mutants. In contrast, PEACH takes a different angle by removing sanity checks in the target program. T-Fuzz leverages a coverage guided fuzzer to generate inputs. Whenever the fuzzer can no longer trigger new code paths, a light-weight, dynamic tracing based technique detects the input checks that the fuzzer generated inputs hit. These checks are then removed from the transformed program. This allows the fuzzer to explore the transformed program, allowing the code protected by the removed checks to be triggered and potential bugs discovered.

Fuzzing transformed programs to find bugs poses two challenges: (1) removal of checks leads to over-approximation and false positives, and (2) even for true bugs, the crashing input on the transformed program may not trigger the bug in the original

program. As an auxiliary post-processing step, T-Fuzz leverages a symbolic execution-based approach to filter out false positives and report only real bugs.

By transforming the program as well as *mutating the input*, T-Fuzz covers more code and finds more true bugs than any previous technique. We have evaluated T-Fuzz on the DARPA Cyber Grand Challenge dataset, LAVA-M dataset and 4 real-world programs (pngfix, tiff2png, magick and pdffontinfo). For the CGC challenge, T-Fuzz finds bugs in 166 binaries, Driller in 121, and AFL in 105. In addition, found 3 new bugs in previously-fuzzed programs and libraries.

J. INTRODUCTION

Fuzzing is an automated software testing technique that discovers faults by providing randomly-generated inputs to a program. It has been proven to be simple, yet effective [1], [2]. With the reduction of computational costs, fuzzing has become increasingly useful for both hackers and software vendors, who use it to discover new bugs/vulnerabilities in software. As such

Permission to freely reproduce all or part of this paper for noncommercial purposes is granted provided that copies bear this notice and the full citation on the first page. Reproduction for commercial purposes is strictly prohibited without the prior written consent of the Internet Society, the first-named author (for reproduction of an entire paper only), and the author's employer if the

Whitebox fuzzing

Symbolic execution

- Translate expressions into **symbolic formulae**
- Program paths accumulate formulae into **constraints**
- Constraints are solved (via a **SAT / SMT solver**)

Challenges

- Expensive / slow
- Modeling “external environment”

Concolic fuzzing

Concolic = **concrete + symbolic**

- Symbolic values augmented with concrete values
- Can always fall back to concrete values

Solutions

- Angora: Treat solver as optimization problem
- SymCC: Compiles concolic executor into the binary
- JIGSAW: JIT compile constraints 

What about...

- Directed fuzzers?
- Determining when we've "fuzzed enough"?
- Benchmarking fuzzers?

What about...

- Directed fuzzers?
- Determining when
- Benchmarking fu

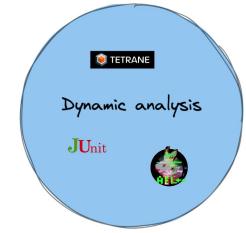
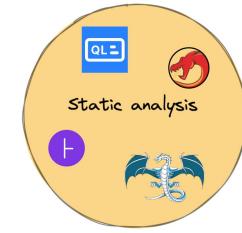


Conclusions

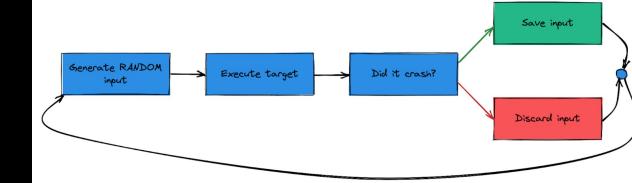
- Fuzzing research has progressed in leaps and bounds
 - No longer just “file-based + *nix-based”
- Still many open problems
- Balance between **performance** and **accuracy**



What is fuzzing?



Our first fuzzer

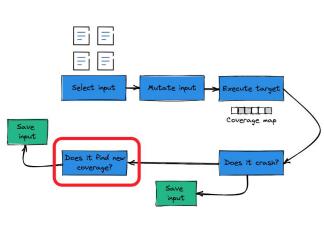


This is a classic **generational blackbox** fuzzer

Greybox fuzzing

Does it find new coverage?

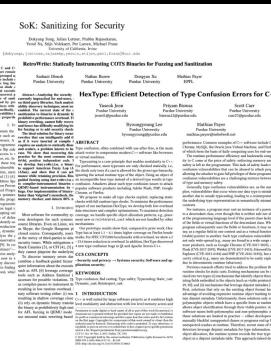
- Save input
 - Return to start



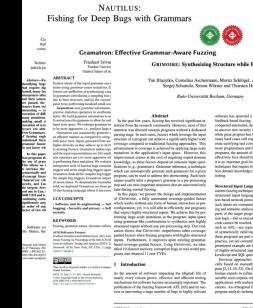
Sanitization

- Allow for additional security policies to be defined and checked at runtime
 - Typically compiler-based (e.g., LLVM) but don't have to be

What can we check for?



Grammar-based fuzzing



- Many targets (e.g., JavaScript interpreter) accept input described by a **context-free grammar (CFG)**
 - Highly structured
 - Blind mutation will destroy structure
 - Leverage CFG in mutation
 - “Lift” inputs to parse tree
 - Mutate parse tree(s)
 - Lower parse tree back to file

Conclusions

- Fuzzing research has progressed in leaps and bounds
 - No longer just “file-based + *nix-based”
 - Still many open questions
 - Balance between **performance** and **accuracy**