

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 & WORKING TITLE PRODUCTIONS IN ASSOCIATION WITH BIG TALK PRODUCTIONS
"HOT FUZZ" SIMON PEGG NICK FROST JIM GIRDLEBERT JESSIE NINA GOLD AND DAVID ARNOLD MUSIC BY ANNE HARRIDGE
WRITTEN BY CHRIS DICKENS PRODUCED BY MARCUS ROWLAND PRODUCED BY JESS HALL EDITOR RONALDO VASCONCELLOS PRODUCTION DESIGNER TASHA WHARTON
DIRECTOR OF PHOTOGRAPHY EDGAR WRIGHT EXECUTIVE PRODUCERS NIRA PARK TIM BEVAN ERIC FELLNER PRODUCED BY EDGAR WRIGHT



www.jointhefuzz.com

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

POLICE

HOT FUZZ

SIMON

ROBIE PICTURES PRESENTS IN ASSOCIATION WITH STUDIOCANAL
HOT FUZZ SIMON PEGG NICK FROST JIM BRADBURY
WRITTEN BY CHRIS DICKENS DIRECTED BY MARCUS ROVLAND PRODUCED BY JESS HALE
CASTING BY EDGAR WRIGHT & SIMON PEGG COSTUME DESIGNER NIRA PARK, TIM

R
RESTRICTED
Under 17 requires accompaniment
of an adult guardian
Some material may be offensive
to parents and young persons

www.jointhefuzz.com



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

imgflip.com

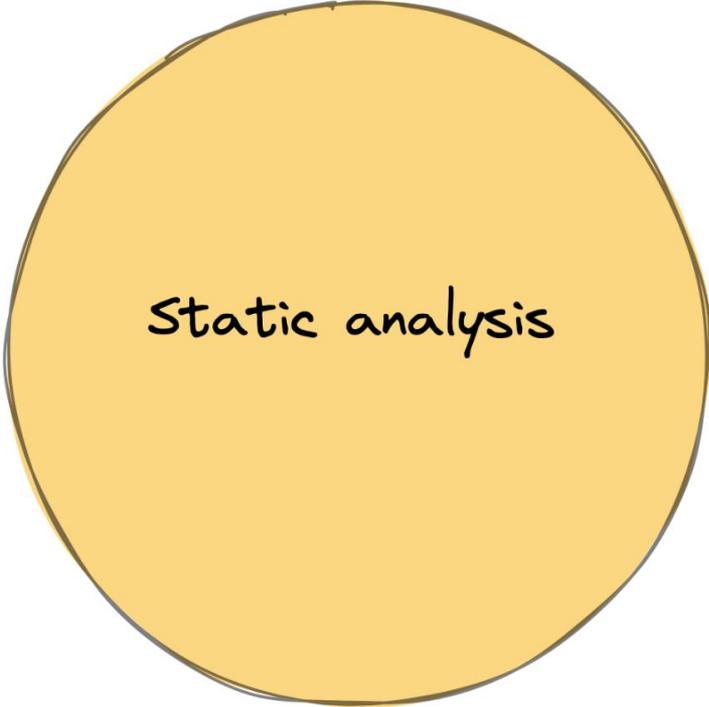


**FUZZER GO
BRRRRRRRRRR**

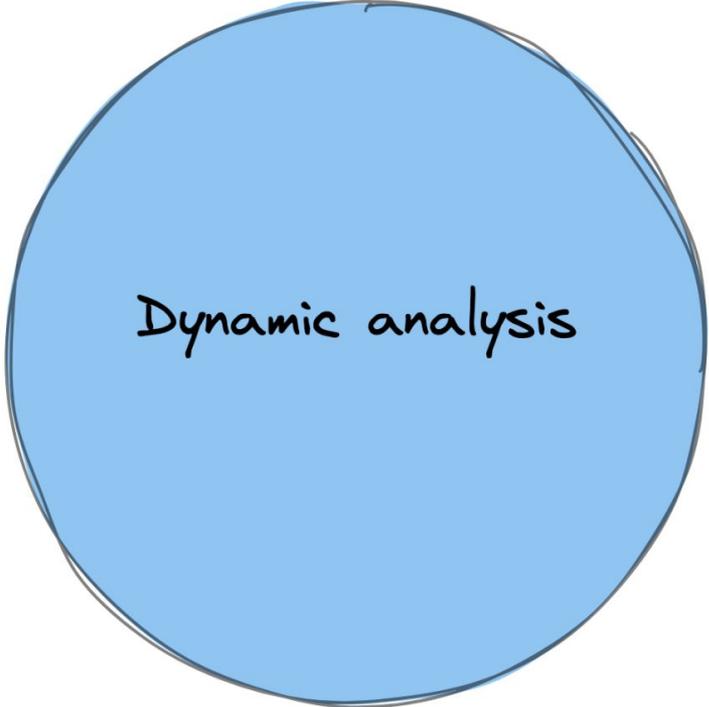
Outline

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

What is fuzzing?

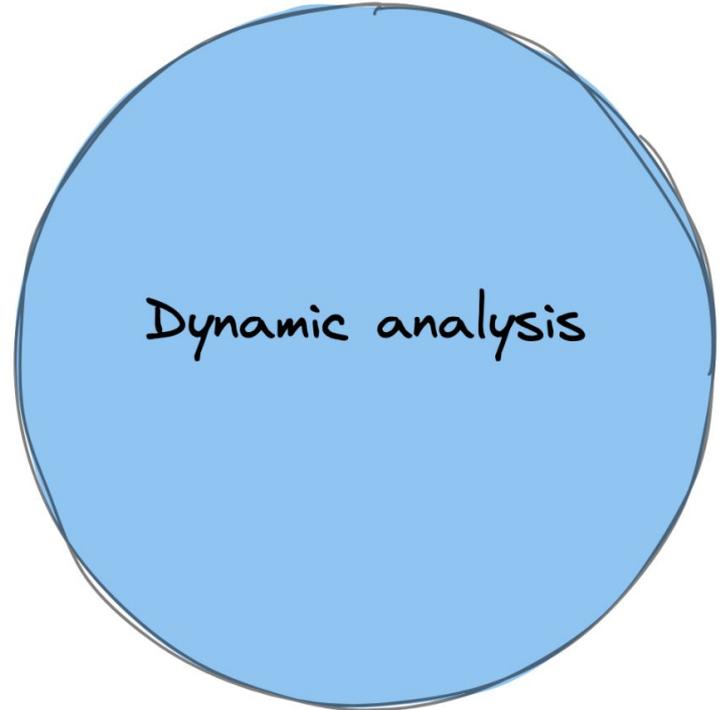
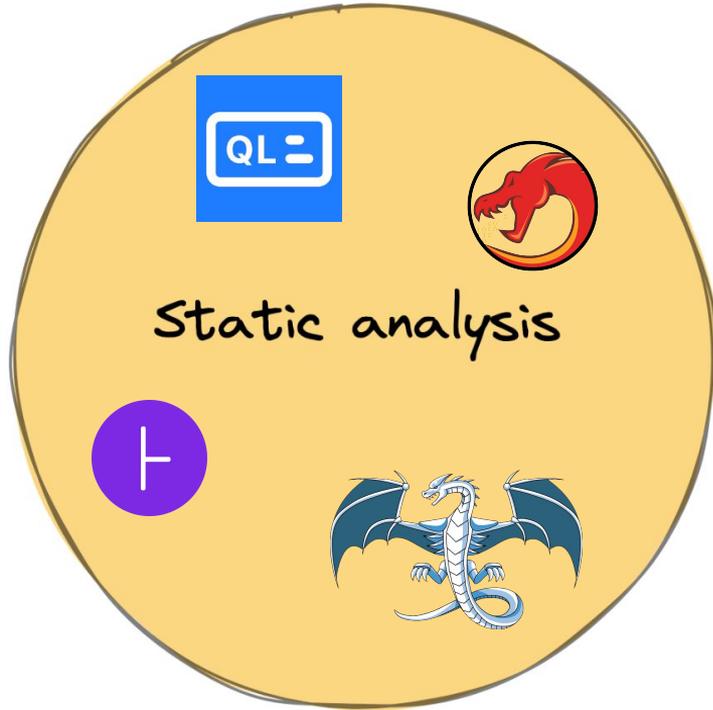


Static analysis

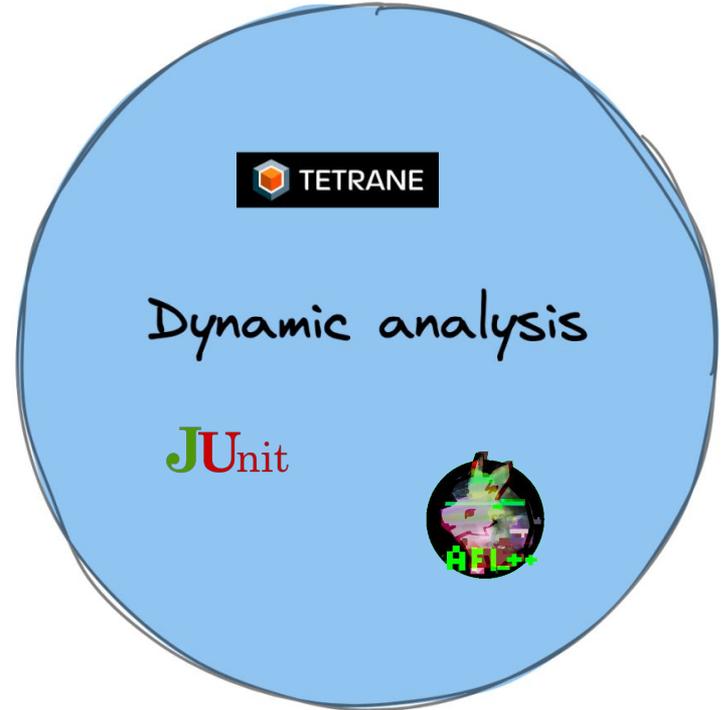
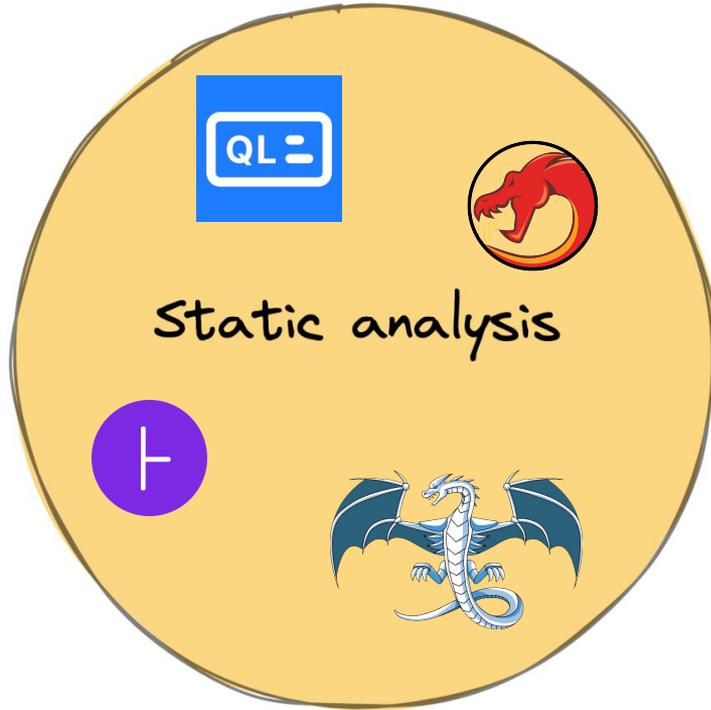


Dynamic analysis

What is fuzzing?



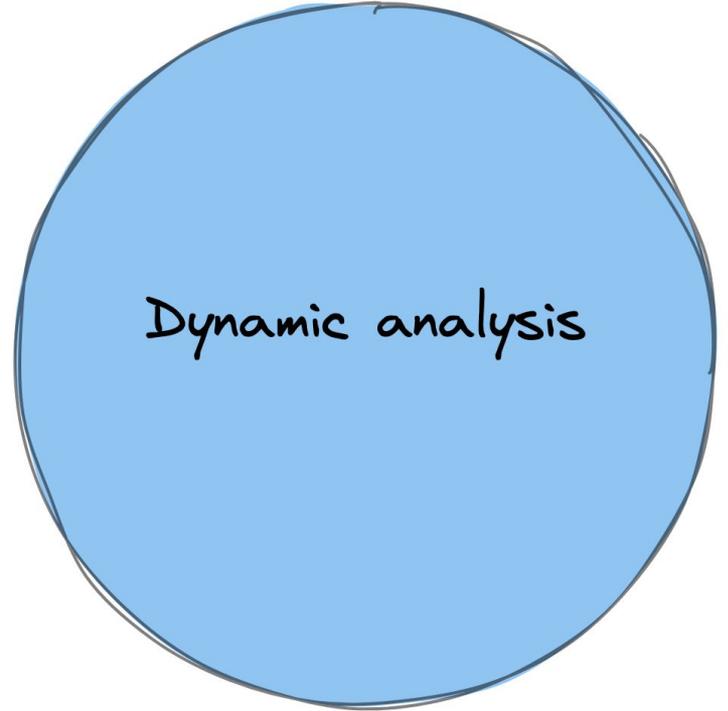
What is fuzzing?



What is fuzzing?

Pros

- No false positives
- Produces PoC
- Scalable



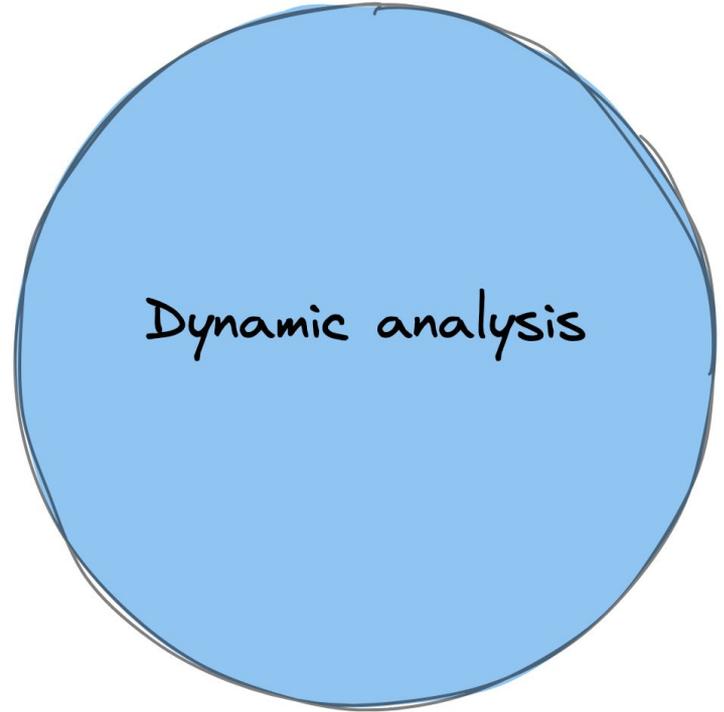
What is fuzzing?

Pros

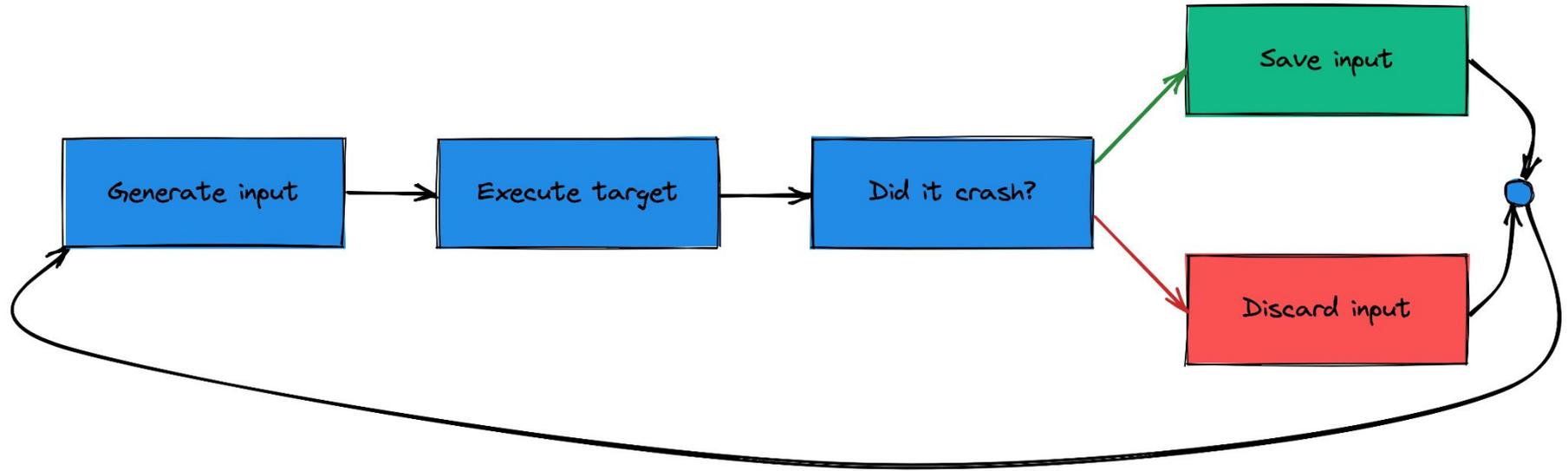
- No false positives
- Produces PoC
- Scalable

Cons

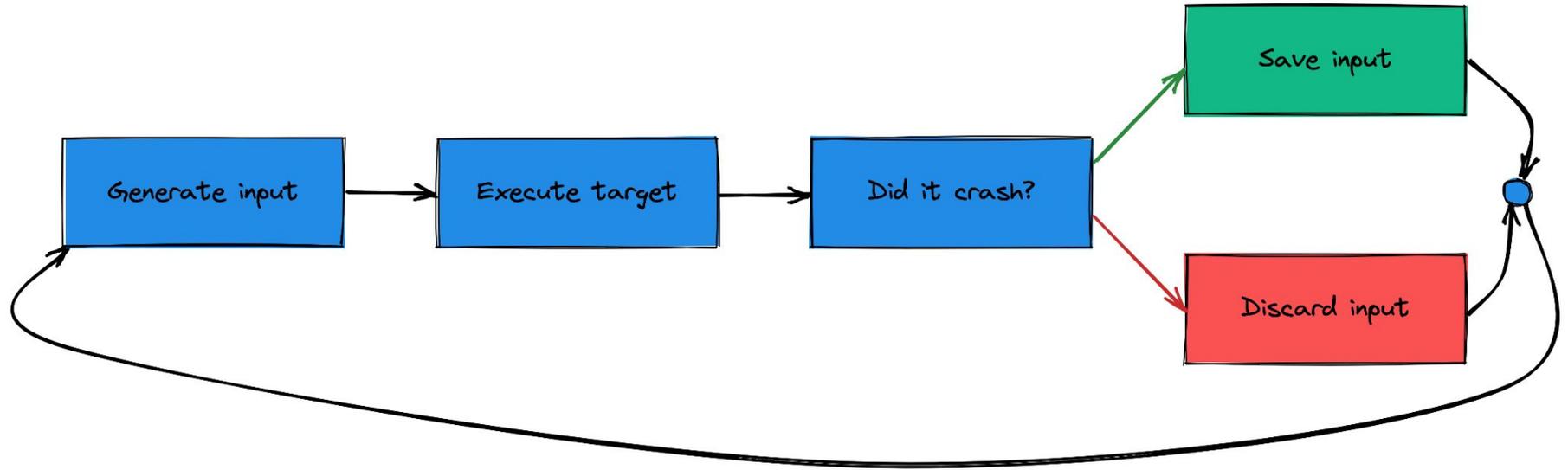
- Incomplete
- Requires buildable target
- Scalability



Our first fuzzer

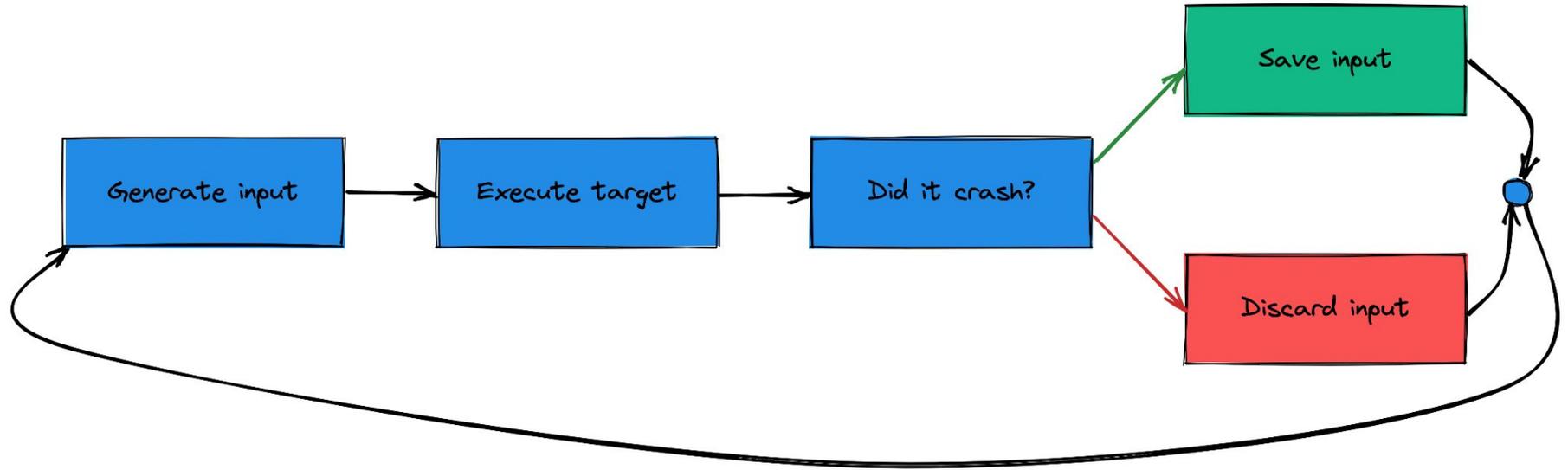


Our first fuzzer



How is this different to dynamic testing?

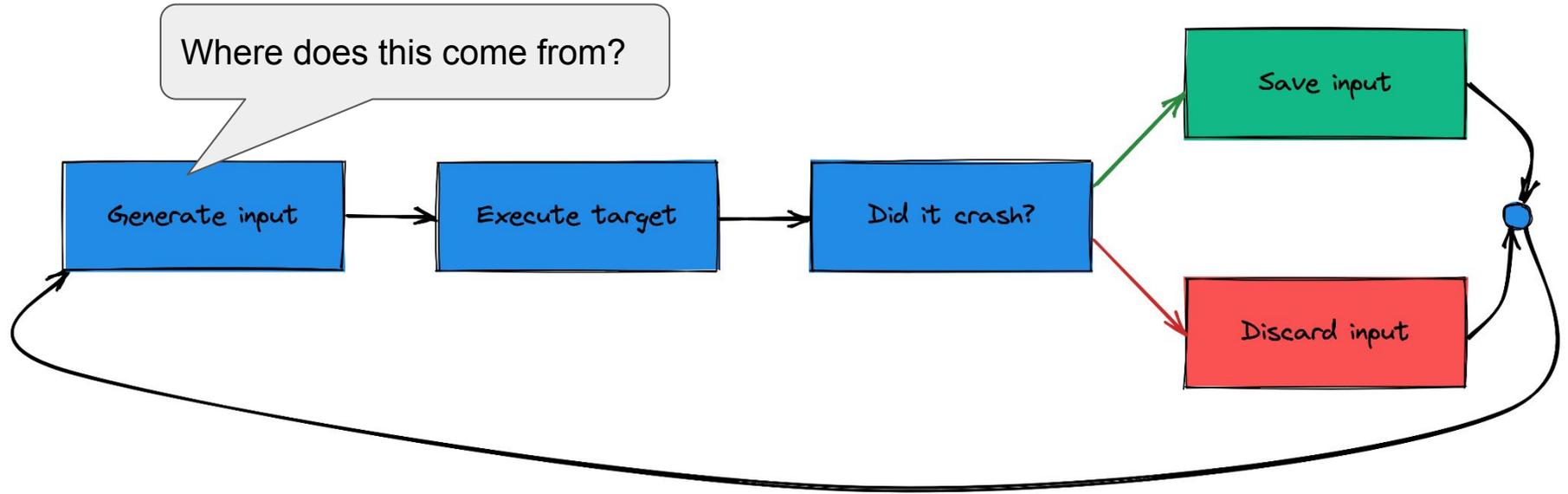
Our first fuzzer



How is this different to dynamic testing?

Or regression testing?

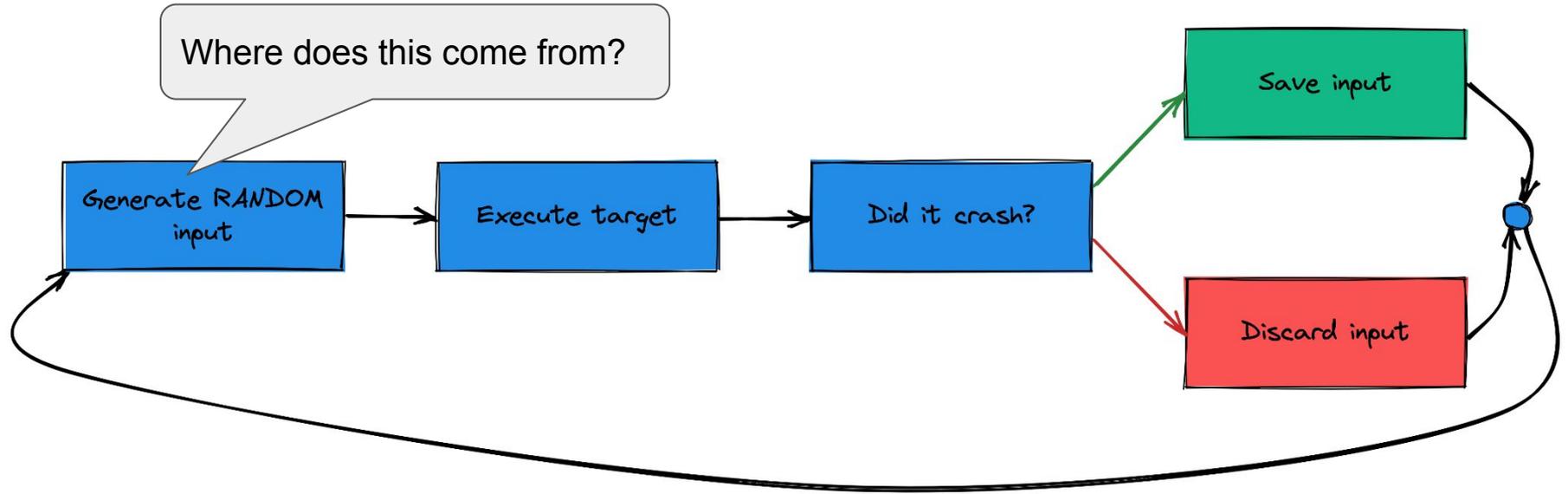
Our first fuzzer



How is this different to dynamic testing?

Or regression testing?

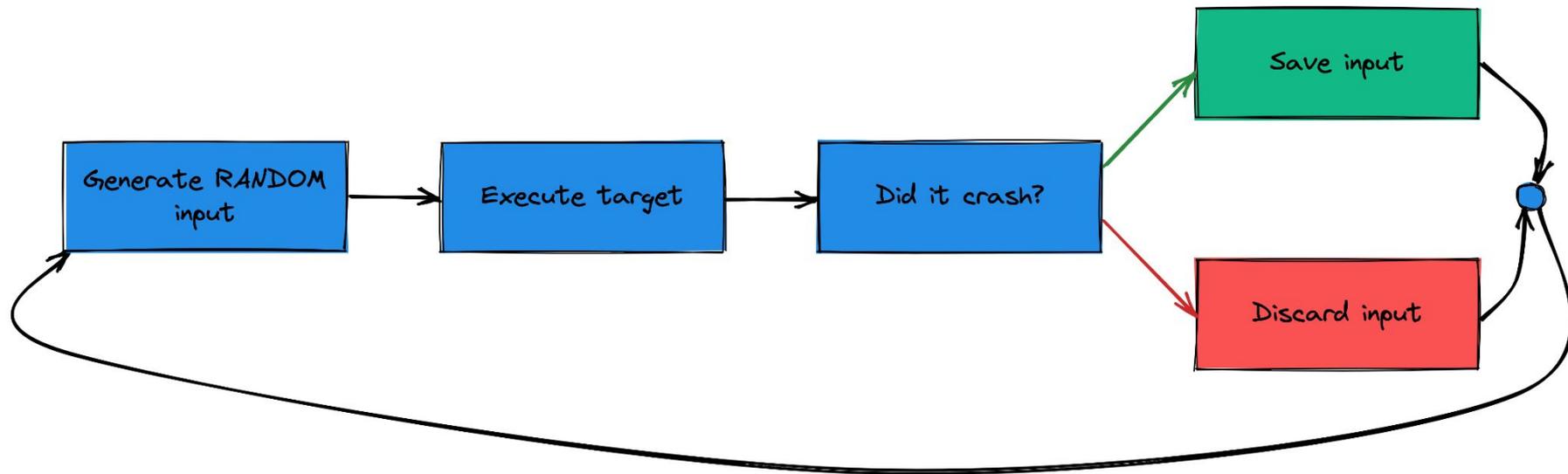
Our first fuzzer



How is this different to dynamic testing?

Or regression testing?

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 a high degree of reliability. These parts of the system are used frequently and this frequent use implies that the programs are well tested and working correctly. To make a systematic statement about

Unix operating system. The project proceeded in four steps: (1) programs were constructed to generate random characters, and to help test interactive utilities; (2) these programs were used to test a large number of utilities on random input strings to see if they crashed; (3) the strings (or types of strings) that crash these programs were identified; and (4) the causes of the

to the Internet worm the “grog fingers” bug [25]. We have found additional bugs that might indicate future security holes. Third, some of the crashes were caused by input that might be carelessly typed—some strange and unexpected errors were uncovered by this method of testing. Fourth, we sometimes inadvertently feed programs noisy input (e.g., trying to

An Empirical

the correctness of a program, we should probably use some form of formal verification. While the technology for program verification is advancing, it has not yet reached the point where it is easy to apply (or commonly applied) to large systems.

A recent experience led us to believe that, while formal verification of a complete set of operating system utilities was too onerous a task, there was still a need for some form of more complete testing. On a dark and stormy night one of the authors was logged on to his workstation on a dial-up line from home and the rain had affected the phone lines, there were frequent spurious characters on the line. The author had to race to see if he could type a sensible sequence of characters before the noise scrambled the command. This line noise was not surprising, but we were surprised that these spurious characters were causing programs to crash. These programs included a significant number of basic operating system utilities. It is reasonable to expect that basic utilities should not crash (“core dump”), on receiving unusual input, they might exit with minimal error messages, but they should not crash. This experience led us to believe that there might be serious bugs lurking in the systems that we regularly used.

This scenario motivated a systematic test of the utility programs running on various versions of the

program crashes were identified and the common mistakes that cause these crashes were categorized. As a result of testing almost 50 different utility programs on seven versions of UnixSM, we were able to crash more than 24% of these programs. Our testing included versions of Unix that underwent commercial product testing. A hypothesis of this project is a list of bug reports (and fixes) for the crashed programs and a set of tools available to the systems community.

There is a rich body of research on program testing and verification. Our approach is not a substitute for a formal verification or testing procedures, but rather an inexpensive mechanism to identify bugs and increase overall system reliability. We are using a coarse notion of correctness in our study. A program is detected as faulty only if it crashes or hangs (loops indefinitely). Our goal is to complement, not replace, existing test procedures.

This type of study is important for several reasons. First, it contributes to the testing community a large list of real bugs. These bugs can provide test cases against which researchers can evaluate more sophisticated testing and verification strategies. Second, one of the bugs that we found was caused by the same programming practice that provided one of the security holes (as is a weakness of C.I.T. Bell Laboratories).

edit or view an object module). In these cases, we would like some meaningful and predictable response. Fifth, noisy phone lines are a reality, and major utilities like shells and editors should not crash because of them. Last, we were interested in the interactions between our random testing and more traditional industrial software testing.

While our testing strategy sounds somewhat naive, its ability to discover fatal program bugs is impressive. If we consider a program to be a complex finite state machine, then our testing strategy can be thought of as a random walk through the state space, searching for undefined states. Similar techniques have been used in areas such as network protocols and CPU cache testing. When testing network protocols, a module can be inserted in the data stream. This module randomly perturbs the packets (either destroying them or modifying them) to test the protocol's error detection and recovery features. Random testing has been used in evaluating complex hardware, such as multiprocessor cache coherence protocols [4]. The state space of the device, when combined with the memory architecture, is large enough that it is difficult to generate systematic tests. In the multiprocessor example, random generation of test cases helped cover a large part of the state space and, simplify the generation of test cases.



Barton P. Miller, Lars Fredriksen and Bryan So

Study of the Reliability of

UNIX Utilities

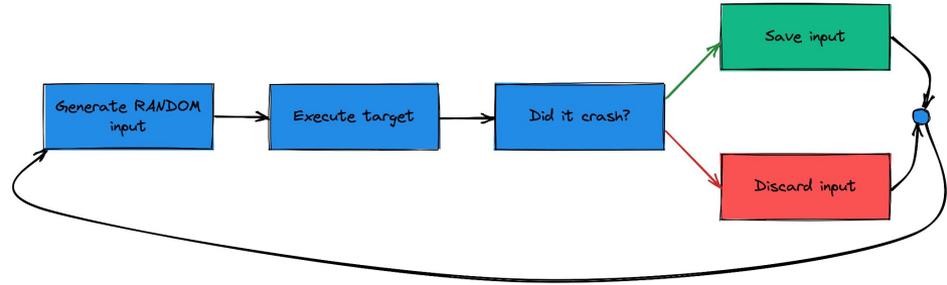
COMMUNICATIONS OF THE ACM, December 1990, Vol. 33, No. 12

33

Blackbox fuzzing

Pros

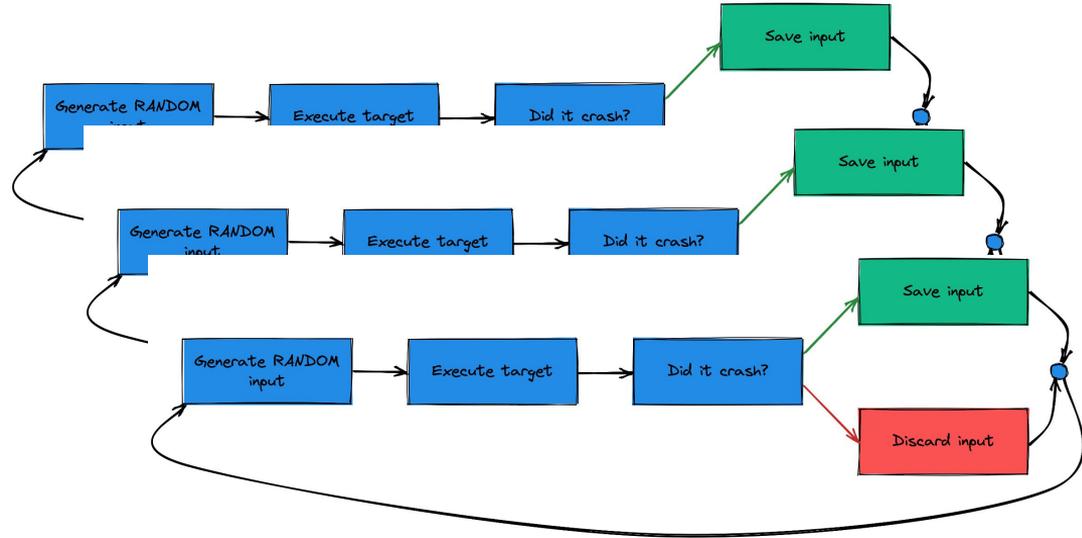
- Simple
- Fast
- Embarrassingly parallel



Blackbox fuzzing

Pros

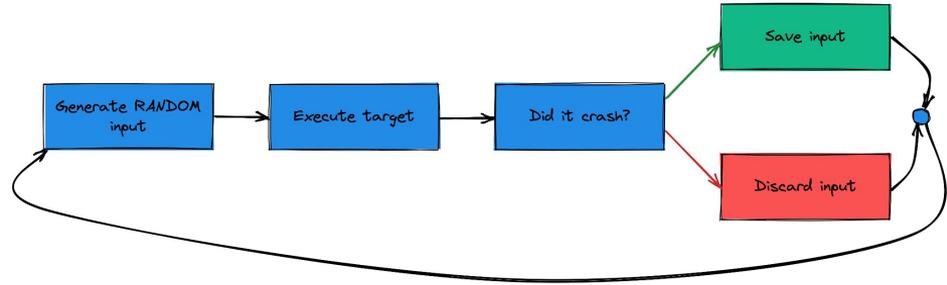
- Simple
- Fast
- Embarrassingly parallel



Blackbox fuzzing

Cons

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

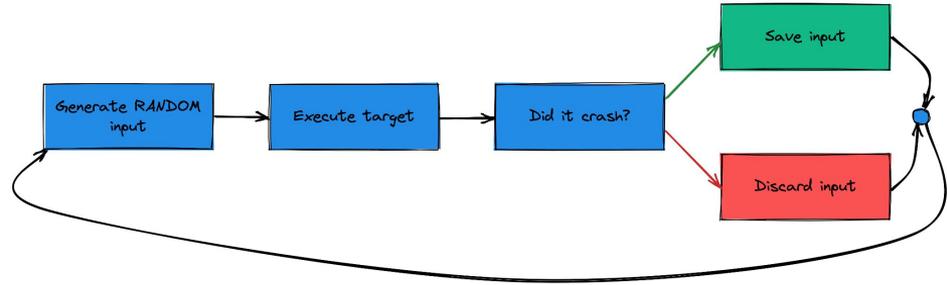


Can we do better?

Blackbox fuzzing

Cons

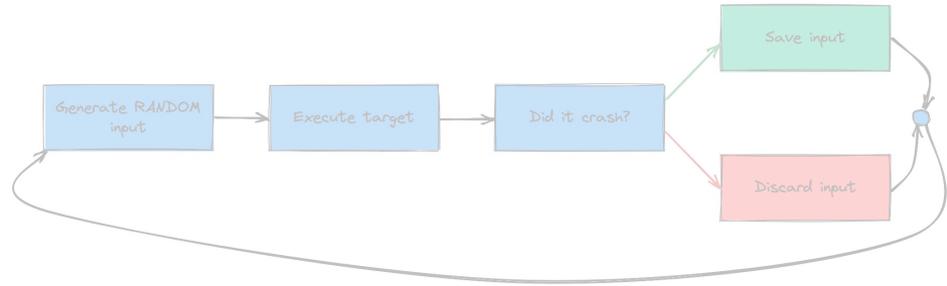
- Generate mostly rubbish
- No notion of “progress”
- Only detect `SIGSEGV`



Blackbox fuzzing

Cons

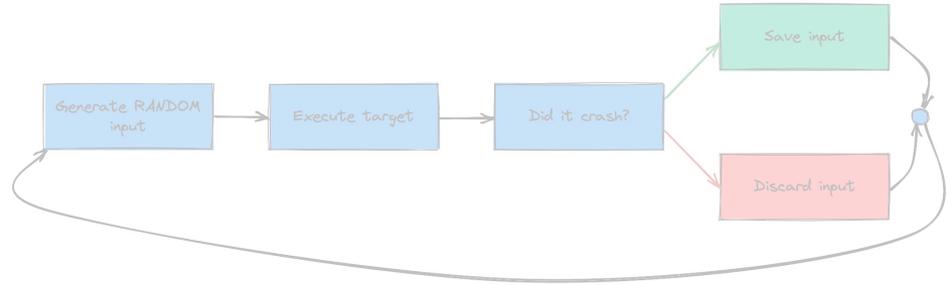
- Generate mostly rubbish
 - ~~Generate~~ mutate
- No notion of “progress”
- Only detect SIGSEGV



Blackbox fuzzing

Cons

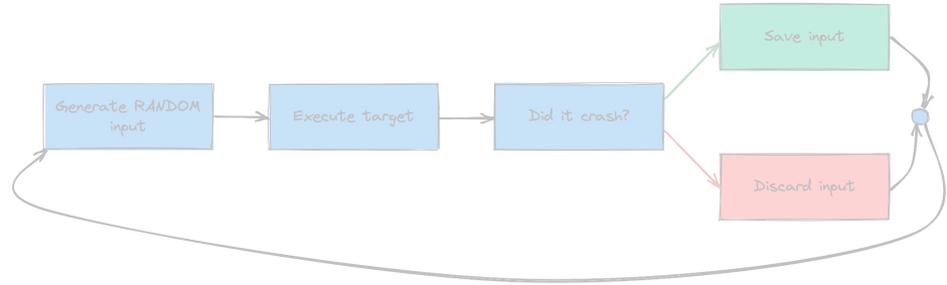
- Generate mostly rubbish
 - ~~Generate~~ mutate
- No notion of “progress”
 - Add a **feedback loop**
- Only detect SIGSEGV



Blackbox fuzzing

Cons

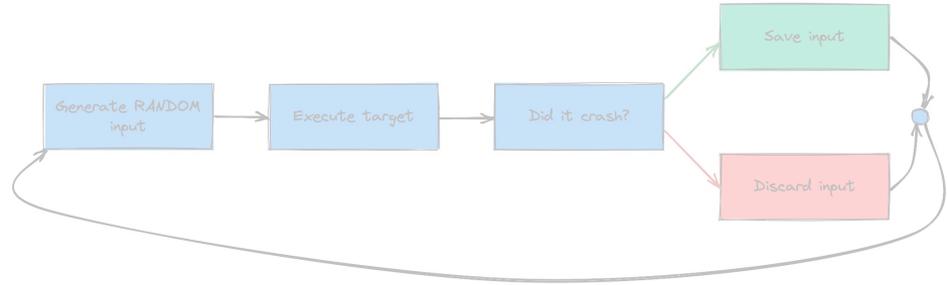
- Generate mostly rubbish
 - ~~Generate~~ **mutate**
- No notion of “progress”
 - Add a **feedback loop**
- Only detect SIGSEGV
 - Add a **sanitizer**



Blackbox fuzzing

Cons

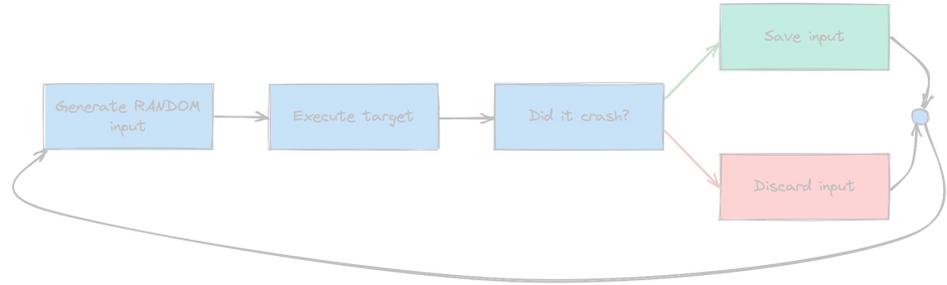
- Generate mostly rubbish
 - **Generate mutate**
- No notion of “progress”
 - Add a **feedback loop**
- Only detect SIGSEGV
 - Add a **sanitizer**



Blackbox fuzzing

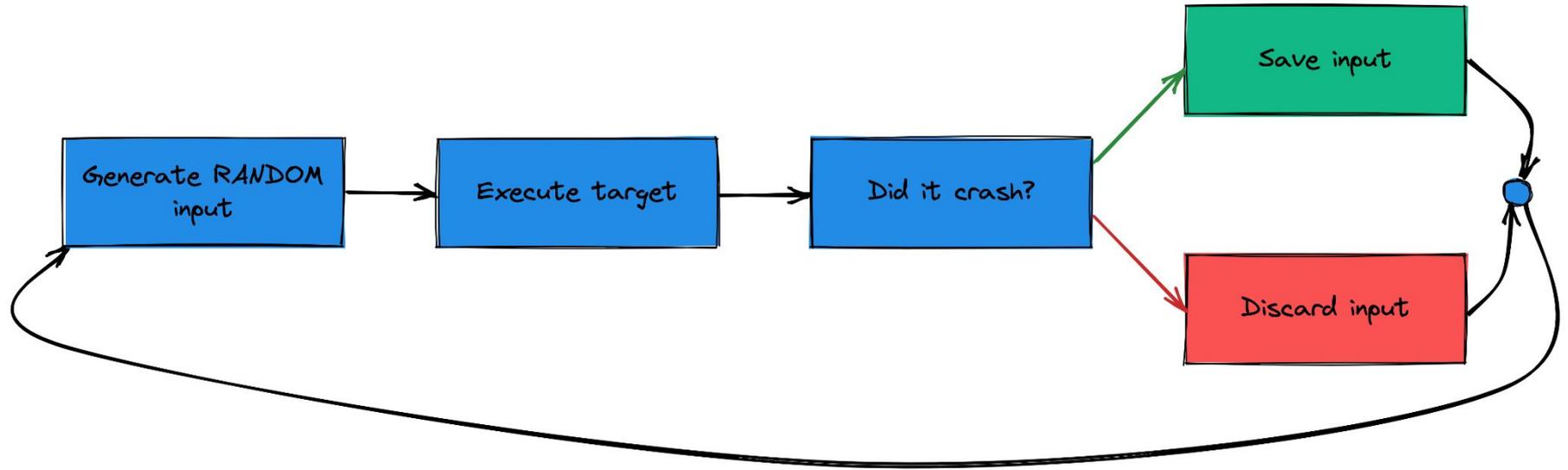
Cons

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

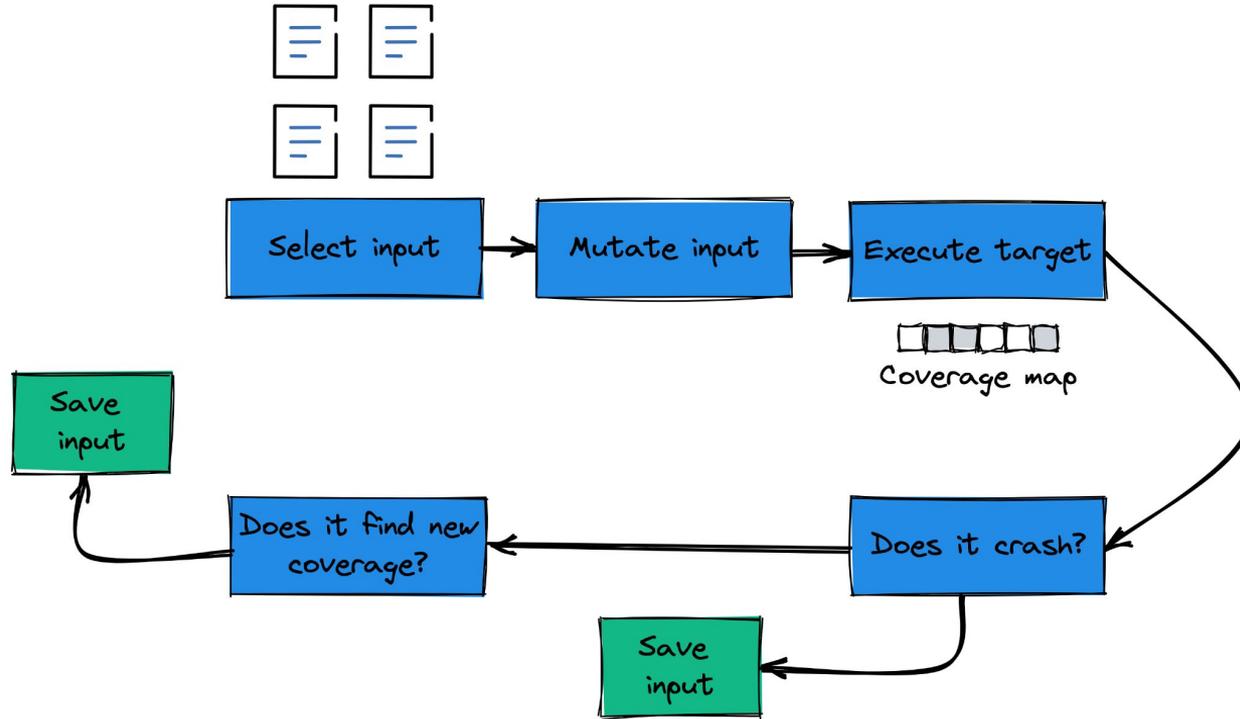


Mutational coverage-guided fuzzer
aka
greybox fuzzer

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	overall results cycles done : 0 total paths : 195 uniq crashes : 0 uniq hangs : 1
cycle progress now processing : 38 (19.49%) paths timed out : 0 (0.00%)	map coverage map density : 1217 (7.43%) count coverage : 2.55 bits/tuple
stage progress now trying : interest 32/8 stage execs : 0/9990 (0.00%) total execs : 654k exec speed : 2306/sec	findings in depth favored paths : 128 (65.64%) new edges on : 85 (43.59%) total crashes : 0 (0 unique) total hangs : 1 (1 unique)
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)	path geometry levels : 3 pending : 178 pend fav : 114 imported : 0 variable : 0 latent : 0

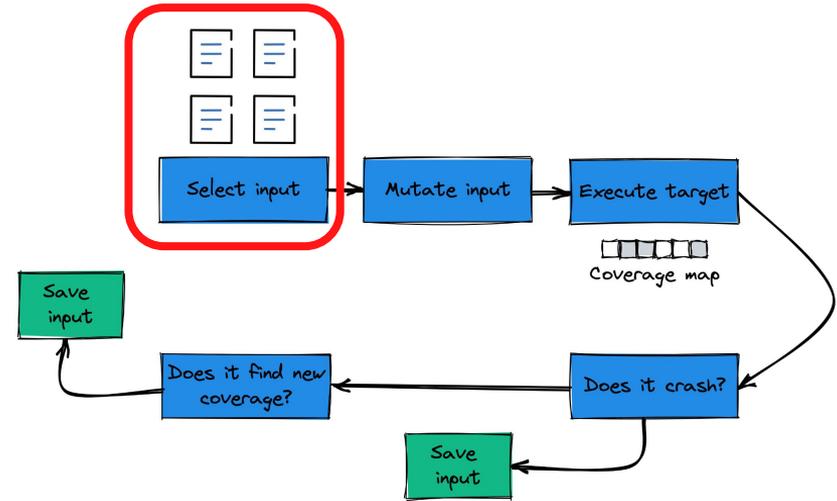
Save input

Save input

Greybox fuzzing

Select input

- Rather than generating random data, mutate existing data

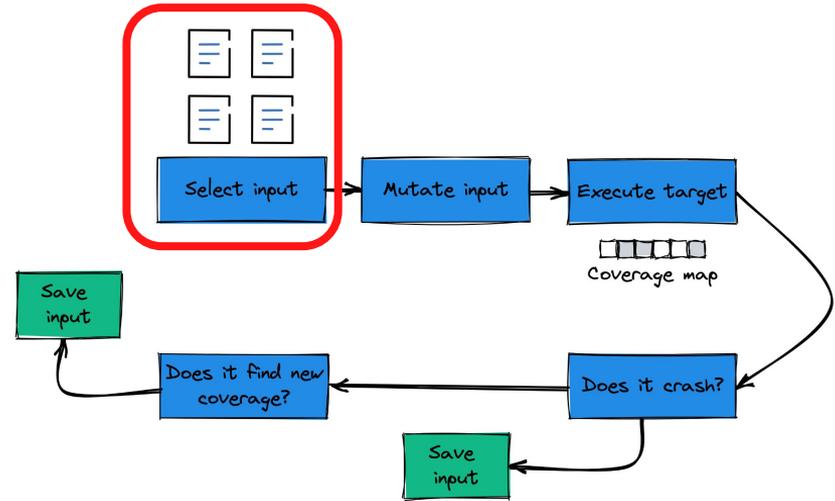


Greybox fuzzing

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^{‡5} Sang Kil [†]
alex@forallsecure.com sangkilic@cr

David Warren[†] G
dwarren@cert.org gg@c

[‡] Carnegie Mellon University
[†] Software 1

Seed Selection for Successful Fuzzing

Adrian Herrera
ANU & DST
Australia

Michael Norrish
CSIRO's Data61 & ANU
Australia

Hendra Gunadi
ANU
Australia

Mathias Payer
EPFL
Switzerland

Shane Magrath
DST
Australia

Antony L. Hosking
ANU & CSIRO's Data61
Australia

ACM Reference Format:

Adrian Herrera, Hendra Gunadi, Shane Magrath, Michael Norrish, Mathias Payer, and Antony L. Hosking. 2021. Seed Selection for Successful Fuzzing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '21)*, July 11–17, 2021, Virtual, Denmark. ACM, New York, NY, USA, 14 pages. <https://doi.org/10.1145/3460319.3464785>

1 INTRODUCTION

Fuzzing is a dynamic analysis technique for finding bugs and vulnerabilities in software, triggering crashes in a target program by subjecting it to a large number of (possibly malformed) inputs. *Mutation-based* fuzzing typically uses an initial set of valid seed inputs from which to generate new seeds by random mutation. Due to their simplicity and ease-of-use, mutation-based greybox fuzzers such as AFL [74], honggfuzz [64], and libFuzzer [61] are widely deployed, and have been highly successful in uncovering thousands of bugs across a large number of popular programs [6, 16]. This success has prompted much research into improving various aspects of the fuzzing process, including mutation strategies [9, 42], energy assignment policies [15, 25], and path exploration algorithms [14, 73]. However, while researchers often note the importance of high-quality input seeds and their impact on fuzzer performance [37, 56, 58, 67], few studies address the problem of *optimal design and construction of corpora* for mutation-based fuzzers [56, 58], and none assess the precise impact of these corpora in coverage-guided mutation-based greybox fuzzing.

Intuitively, the collection of seeds that form the initial corpus should generate a broad range of observable behaviors in the target. Similarly, candidate seeds that are behaviorally similar to one another should be represented in the corpus by a single seed. Finally, both the total size of the corpus and the size of individual seeds should be minimized. This is because previous work has demonstrated the impact that file system contention has on industrial-scale fuzzing. In particular, Xu et al. [71] showed that the overhead from opening/closing test-cases and synchronization between workers each introduced a 2x overhead. Time spent opening/closing test-cases and synchronization is time diverted from mutating inputs and expanding code coverage. Minimizing the total corpus size and the size of individual test-cases reduces this wastage and enables time to be (better) spent on finding bugs.

Under these assumptions, simply gathering as many input files as possible is not a reasonable approach for constructing a fuzzing corpus. Conversely, these assumptions also suggest that beginning with the “empty corpus” (e.g., consisting of one zero-length file) may be less than ideal. And yet, as we survey here, the majority of published research uses either (a) the “singleton corpus” (e.g., a single seed representative of the target program’s input format),

- 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

Abstract

Randomly mutating well-formed program inputs plays a highly effective and widely used role to find bugs in software. Other than showing fuzzer 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 about one critical aspect in fuzzing: how best to pick files to maximize the total number of bugs found a fuzz campaign. We design and evaluate six different algorithms using over 650 CPU days on Amazon Compute Cloud (EC2) to provide ground truth. Overall, we find 240 bugs in 8 applications and show the choice of algorithm can greatly increase the number of bugs found. We also show that current seed strategies as found in Peach may fare no better than seeding at random. We make our data set an publicly available.

1 Introduction

Software bugs are expensive. A single software is enough to take down spacecrafts [2], make a centrifuges 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 at \$19.2 billion [13], and recent forecasts predict a increase in the future despite a sequestering economy. 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 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—relies on a set of non-crashing seed inputs (a corpus) to bootstrap the bug-finding process. When evaluating a fuzzer, common approaches for constructing this corpus include: (i) using an empty file; (ii) using a single seed representative of the target’s input format; or (iii) collecting a large number of seeds (e.g., by crawling the Internet). 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 if a best approach exists).

To address this gap in knowledge, we systematically investigate and evaluate how seed selection affects a fuzzer’s ability to find bugs in *real-world software*. This includes a systematic review of seed selection practices used in both evaluation and deployment contexts, and a large-scale empirical evaluation (over 33 CPU-years) of six seed selection approaches. These six seed selection approaches include three *corpus minimization* techniques (which select the smallest subset of seeds that trigger the same range of instrumentation data points as a full corpus).

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 information protection → Software and application security.

KEYWORDS

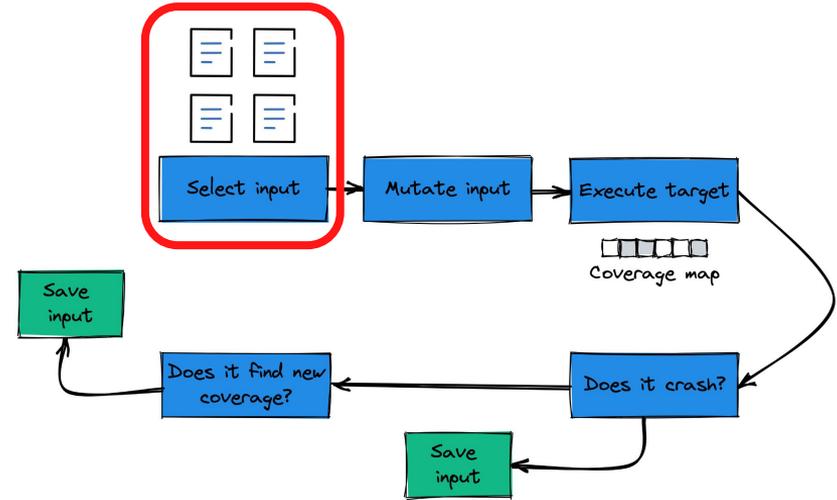
fuzzing, corpus minimization, software testing

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 for components of this work owned by others than ACM 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 permissions from permissions@acm.org.
ISSTA '21, July 11–17, 2021, Virtual, Denmark.
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8459-9/21/07.
<https://doi.org/10.1145/3460319.3464785>

Greybox fuzzing

Select input

- Rather than generating random data, mutate existing data



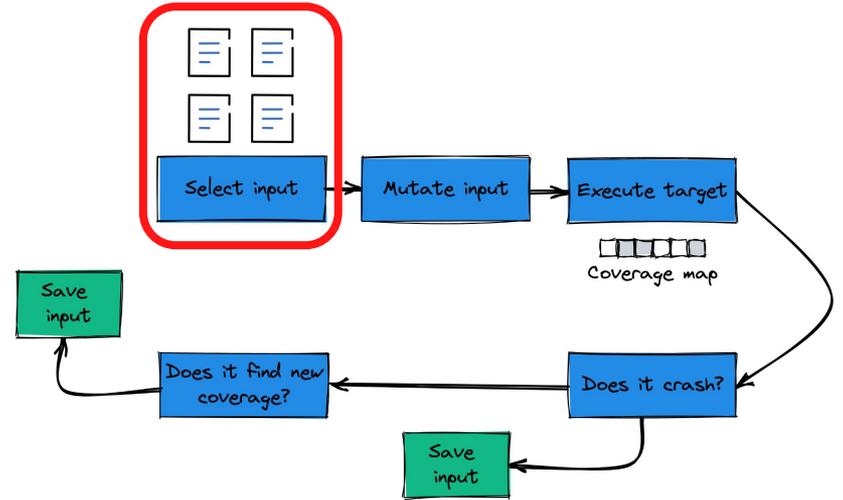
Greybox fuzzing

Select input

- Rather than generating random data, mutate existing data

How long do we focus on a seed?

How do we select this seed?



Power scheduling

- Power schedule = amount of energy assigned to an input
 - Decrease energy each execution
 - When energy = 0, change inputs

● Examples

- Markov chain
- Multi-arm bandit
- Machine learning
- Heuristics

Coverage-Based Greybox Fuzzing as Markov Chain

Marcel Böhme¹, Van-Thuan Pham², and Abhik Roychoudhury

Abstract—Coverage-based Greybox Fuzzing (CGF) generated by slightly mutating a seed input. If the seed is discarded, we observe that most tests exercise more paths with the same number of tests by gray CGF using a Markov chain model which specifies exercises path i . Each state (i.e., seed) has an energy that CGF is considerably more efficient if energy is monotonically every time that seed is chosen. Extending AFL to 24 hours, AFLFast exposes 39 unreported CVEs 7x faster than AFL, AFLFast prc AFLFast to the symbolic executor Klee. In terms of same subject programs that were discussed in the Klee while a combination of both tools achieves by

Index Terms—Vulnerability detection, fuzzing, pm

1 INTRODUCTION

RECENTLY, there has been a controversial efficiency of symbolic execution-based fuzzers versus more lightweight greybox fuzzer. Symbolic execution is a systematic effort to st behaviors and thus considerably more effective most vulnerabilities were exposed by partly fuzzers that do not leverage any program. It turns out that even the most effective test efficient than blackbox fuzzing if the time spent a test case takes too long [4]. Symbolic execution effective because each new test exercises a different program. However, this effectiveness comes at the cost of spending significant time doing program analysis solving. Blackbox fuzzing, on the other hand, does not require any program analysis and generates orders of magnitude more tests in the same time.

Coverage-based Greybox Fuzzing (CGF) is a more effective at path exploration by sacrificing time for program analysis. CGF uses (binary) instrumentation to determine a unique path for the path that is exercised by an input. New seeds by slightly mutating the provided seed also call the new tests as fuzz. If some fuzz exercises

• The authors are with the Department of Computer Science, National University of Singapore, Singapore. E-mail: marcel, thuan, abhik@comp.nus.edu.sg.
Manuscript received 11 Aug. 2017; revised 4 Dec. 2017; accepted Date of publication 20 Dec. 2017; date of current version 22 Feb. 2018.
Corresponding author: Marcel Böhme.
Recommended for acceptance by X. Zhang.
For information on obtaining reprints of this article, please visit www.ieee.org and reference the Digital Object Identifier (DOI) number 10.1109/SE.2017.82841.

EcoFuzz: Adaptive Energy-Saving Greybox Fuzzing as a Variant of the Adversarial Multi-Armed Bandit

Tai Yue, Pengfei Wang, Yong Tang*, Enze Wang, Bo Yu, Kai Lu, Xu Zhou

(yuetai17, p.

Abst

Fuzzing is one of the most effective security vulnerabilities. As a greybox fuzzer, AFL is a high technique. However, AFL all the number of test cases generate the high-frequency just the energy allocation, thus of energy. Moreover, the coverage-based greybox fuzzer This paper presents a variant of Bandit model for modeling A We first explain the challenge rithm by using the reward process for discovering a new path three states of the seeds set an scheduling algorithm as well strategy. These approaches an in an adaptive energy-saving g EcoFuzz is examined against 14 real-world subjects over 49 results. EcoFuzz could attain AFL with reducing 32% test c Besides, EcoFuzz identified 17 tils and other software. We a some IoT devices and found a component.

1 Introduction

Fuzzing is an automated software and effective for detector which was first devised by B. Since then, fuzzing has been d of the most effective technique Fuzzing (CGF) has attracted se

*Corresponding author

Yuekang Li
University of Science and Technology of China
Nanyang Technological University Singapore

Xiuheng Wu
Nanyang Technological University Singapore

Haijun Wang
Nanyang Technological University Singapore

Yinxing Xue¹
University of Science and Technology of China

Cen Zhang
Nanyang Technological University Singapore

Yang Liu
Nanyang Technological University Singapore
Zhejiang Sci-Tech University China

Hongxu Chen
Nanyang Technological University Singapore

Xiaofei Xie
Nanyang Technological University Singapore

ABSTRACT

Existing greybox fuzzers mainly utilize program coverage as the goal to guide the fuzzing process. To maximize their outputs, coverage-based greybox fuzzers need to evaluate the quality of seeds properly, which involves making two decisions: 1) which is the most promising seed to fuzz next (seed prioritization), and 2) how many efforts should be made to the current seed (power scheduling). In this paper, we present our fuzzer, CEREBRO, to address the above challenge. For the seed prioritization problem, we propose an online multi-objective based algorithm to balance various metrics such as code complexity, coverage, execution time, etc. To address the power scheduling problem, we introduced the concept of input potential to measure the complexity of uncovered code and propose a cost-effective algorithm to update it dynamically. Unlike previous approaches where the fuzzer evaluates an input solely based on the execution traces that it has covered, CEREBRO is able to foresee the benefits of fuzzing the input by adaptively evaluating its input potential. We perform a thorough evaluation for CEREBRO on 8 different real-world programs. The experiments show that CEREBRO can find more vulnerabilities and achieve better coverage than state-of-the-art fuzzers such as AFL and AFLFast.

CCS CONCEPTS

• Security and privacy — Vulnerability scanners.

*Corresponding Author

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. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions.acm.org.
ESEC/PSE '18, August 26–30, 2018, Tallinn, Estonia.
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5724-3/18/08...\$15.00
<https://doi.org/10.1145/3338066.3338975>



CEREBRO: Context-Aware Adaptive Fuzzing for Effective Vulnerability Detection

Yuekang Li
University of Science and Technology of China
Nanyang Technological University Singapore

Xiuheng Wu
Nanyang Technological University Singapore

Haijun Wang
Nanyang Technological University Singapore

Yinxing Xue¹
University of Science and Technology of China

Cen Zhang
Nanyang Technological University Singapore

Yang Liu
Nanyang Technological University Singapore
Zhejiang Sci-Tech University China

Hongxu Chen
Nanyang Technological University Singapore

Xiaofei Xie
Nanyang Technological University Singapore

KEYWORDS

Fuzz Testing, Software Vulnerability

ACM Reference Format:

Yuekang Li, Yinxing Xue, Hongxu Chen, Xiuheng Wu, Cen Zhang, Xiaofei Xie, Haijun Wang, and Yang Liu. 2018. CEREBRO: Context-Aware Adaptive Fuzzing for Effective Vulnerability Detection. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18)*, August 26–30, 2018, Tallinn, Estonia. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3338066.3338975>

1 INTRODUCTION

Fuzzing, or fuzz testing, is progressively gaining popularity in both industry and academia since proposed decades before [1]. Various fuzzing tools (fuzzers) have been springing up to fulfill different testing scenarios in recent years [2]. These fuzzers can be classified as blackbox, whitebox, and greybox based on the awareness of the structural information about the program under test (PUT). Blackbox fuzzers [3] have no knowledge about the internals of PUT. So they can scale up but may not be effective. On the contrary, whitebox fuzzers utilize heavy-weight program analysis techniques (e.g. symbolic execution tree [1]) to improve effectiveness at the cost of scalability. To have the best of both worlds, greybox fuzzers (GBFs), such as AFL [5], are advocated to achieve scalability yet effectiveness. Fig. 1 depicts the workflow of greybox fuzzing.

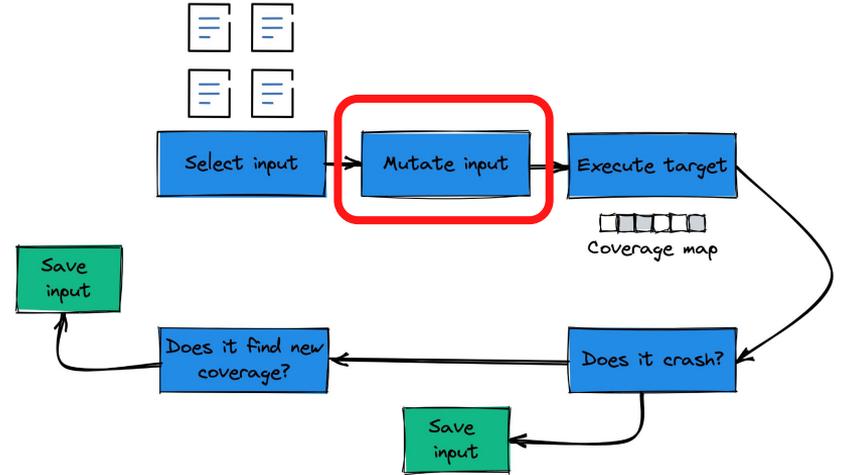
A recent trend in academia is to make greybox fuzzing whiter with various light-weight program analysis. For example, VUZZER [6], STRELLER [7], and ANGGRA [8] mainly help GBFs to penetrate path constraints via modifications on the seed mutator and feedback collector modules in Fig. 1. However, based on the nature that fuzzing's results are strong related with the seeds', the effects of all the works on these modules can be further maximized by enhancing the seeds'

¹In this paper, we denote all the files fed to the PUT by fuzzers as inputs, and only those inputs kept by fuzzers for subsequent mutations as seeds.

Greybox fuzzing

Mutate input

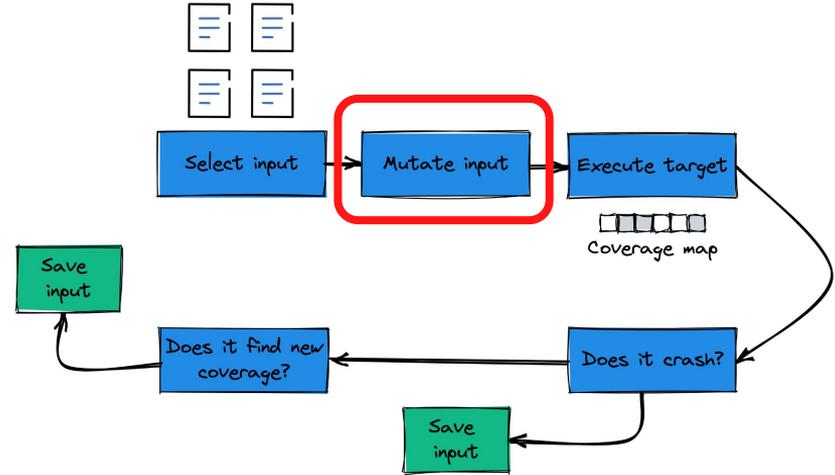
- Mutate enough to explore “interesting” states
- Don’t mutate too much, or we’ll just error out



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

Co
Rul
come

Technis
patrick.ja

Abstract—Fu
identifying bugs
that require big
fuzzed, many fu
interpreters ofte
and then seman
are passed, the
fuzzers from ext
interesting — c
execution of diff
many mutations
making small c
execution of erro
fuzzers are able
Free Grammars,
of fuzzing engi
mutational fuzz
grammar fuzzer
do not know wh

In this paper
fuzz programs th
the use of gram
This allows us t
to increase the
syntactically and
of-concept fuzze
ChakraCore (th
murlby, and Liu
the targets: Sive
and one in Lua, 1
2600 USD and 6
combining conte
significantly out
an order of ma
factor of two wh

Software co
life. Hence, the

Network and Distel
24-27 February 201
ISBN 1-891962-55-
https://doi.org/10
www.ndso-symposi

This work is licensed under a Creative Commons
Attribution License.
© 2021 Copyright held by the owner/authors.
ACM ISBN 978-1-60959-344-1
https://doi.org/10.1145/3460319.3464614

NAUTILUS: Fishing for Deep Bugs with Grammars

Gramatron: Effective Grammar-Aware Fuzzing

Prashast Srivas
Purdue Univers
United States of A

ABSTRACT
Fuzzers aware of the input grammar can e
states using grammar-aware mutations. E
fuzzers are ineffective at synthesizing conq
(i) grammars introducing a sampling bias (r
due to their structure, and (ii) the current
parse trees performing localized small-sca
Gramatron uses grammar automata
gressive mutation operators to synthesize
faster. We build grammar automata to ac
It restructures the grammar to allow for un
input state space. We redesign grammar-av
to be more aggressive, i.e., perform large-s
Gramatron can consistently generate e
an efficient manner as compared to using
with parse trees. Inputs generated from s
higher diversity as they achieve up to 24.2
to existing fuzzers. Gramatron makes imp
and the input representations are 24% small
mutations in the application’s input space.
However, this improvement comes at the cost of requiring expert domain
knowledge, as these fuzzers depend on structure input specifications (e.g., grammars). Grammar inference, a technique which can automatically generate such grammars for a given program, can be used to address this shortcoming. Such techniques usually infer a program’s grammar in a pre-processing step and can miss important structures that are uncovered only later during normal fuzzing.

CCS CONCEPTS
• Software and its engineering → Sof
ware; • Security and privacy → Sof
tware security.

KEYWORDS
Fuzzing, grammar-aware, dynamic software
analysis, Reference Format:
Prashast Srivastava and Mathias Payer. 2021. *Gram
Aware Fuzzing*. In *Proceedings of the 30th ACM/S
on Software Testing and Analysis (EASTA ’21)*.
Denmark, ACM, New York, NY, USA, 13 pages.
9800193.3464614



This work is licensed under a Creative Commons
Attribution License.
© 2021 Copyright held by the owner/authors.
ACM ISBN 978-1-60959-344-1
https://doi.org/10.1145/3460319.3464614

GRIMOIRE: Synthesizing Structure while Fuzzing

Tim Blazytko, Cornelius Aschermann, Moritz Thörling, Ali Abbasi,
Sergej Schumilo, Simon Wörner and Thorsten Holz

Ruhr-Universität Bochum, Germany

Abstract

In the past few years, fuzzing has received significant attention from the research community. However, most of this attention was directed towards programs without a dedicated parsing stage. In such cases, fuzzers which leverage the input structure of a program can achieve a significantly higher code coverage compared to traditional fuzzing approaches. This advancement in coverage is achieved by applying large-scale mutations in the application’s input space. However, this improvement comes at the cost of requiring expert domain knowledge, as these fuzzers depend on structure input specifications (e.g., grammars). Grammar inference, a technique which can automatically generate such grammars for a given program, can be used to address this shortcoming. Such techniques usually infer a program’s grammar in a pre-processing step and can miss important structures that are uncovered only later during normal fuzzing.

In this paper, we present the design and implementation of GRIMOIRE, a fully automated coverage-guided fuzzer which works without any form of human interaction or pre-configuration; yet, it is still able to efficiently test programs that expect highly structured inputs. We achieve this by performing large-scale mutations in the program input space using grammar-like combinations to synthesize new highly structured inputs without any pre-processing step. Our evaluation shows that GRIMOIRE outperforms other coverage-guided fuzzers when fuzzing programs with highly structured inputs. Furthermore, it improves upon existing grammar-based coverage-guided fuzzers. Using GRIMOIRE, we identified 19 distinct memory corruption bugs in real-world programs and obtained 11 new CVEs.

1 Introduction

As the amount of software impacting the digital life of nearly every citizen grows, effective and efficient testing mechanisms for software become increasingly important. The publication of the fuzzing framework AFL [65] and its success at uncovering a huge number of bugs in highly relevant



software has spawned a large body of research on effective feedback-based fuzzing. AFL and its derivatives have largely conquered automated, dynamic software testing and are used to uncover new security issues and bugs every day. However, while great progress has been achieved in the field of fuzzing, many hard cases still require manual user interaction to generate satisfying test coverage. To make fuzzing available to more programmers and thus scale it to more and more target programs, the amount of expert knowledge that is required to effectively fuzz should be reduced to a minimum. Therefore, it is an important goal for fuzzing research to develop fuzzing techniques that require less user interaction and, in particular, less domain knowledge to enable more automated software testing.

Structured Input Languages. One common challenge for current fuzzing techniques are programs which process highly structured input languages such as interpreters, compilers, text-based network protocols or markup languages. Typically, such inputs are consumed by the program in two stages: parsing and semantic analysis. If parsing of the input fails, deeper parts of the target program—containing the actual application logic—fail to execute; hence, bugs hidden “deep” in the code cannot be reached. Even advanced feedback fuzzers—such as AFL—are typically unable to produce diverse sets of syntactically valid inputs. This leads to an imbalance, as these programs are part of the most relevant attack surface in practice, yet are currently unable to be fuzzed effectively. A prominent example are browsers, as they parse a multitude of highly-structured inputs, ranging from XML or CSS to JavaScript and SQL queries.

Previous approaches to address this problem are typically based on manually provided grammars or seed corpora [2, 14, 45, 52]. On the downside, such methods require human experts to (often manually) specify the grammar or suitable seed corpora, which becomes next to impossible for applications with undocumented or proprietary input specifications. An orthogonal line of work tries to utilize advanced program analysis techniques to automatically infer grammars

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

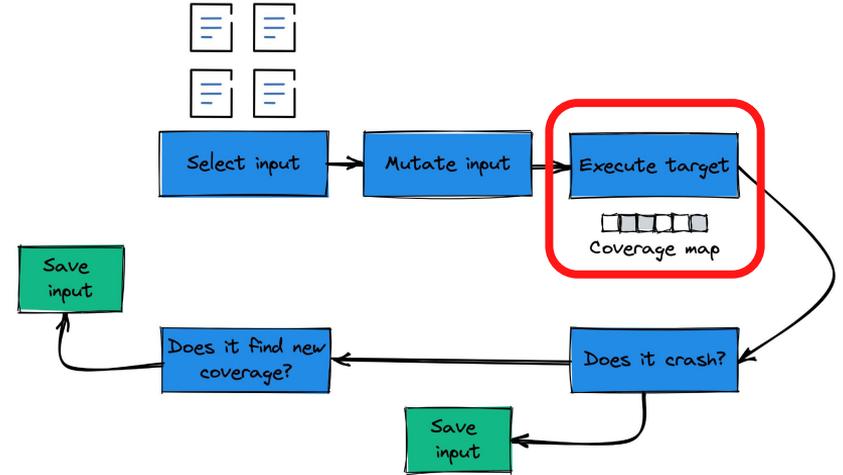
- Require a priori knowledge of input format

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

Abstract—Recent advances in fuzz testing several forms of feedback mechanisms, fact that for a large range of programs a coverage alone is insufficient to reveal con- spired by this line of research, we examine representations looking for a match betw of the structure and adaptability to th testing. In particular, we believe that data d (DDCs) represent a good candidate for this information embedded by this data struc useful to find vulnerable constructs by s tions of def-use pairs that would be difficu graph overlap with the control flow of t possible to reduce the additional instrun only “interesting” data-flow dependency the fuzzer to visit the code in a distinct standard methodologies.

To test these observations, in this p DDFuzz, a new approach that rewards tl with code coverage information, but also in the data dependency graph are hit. that the adoption of data dependency in coverage-guided fuzzing is a promising sol to discover bugs that would otherwise rem standard coverage approaches. This is der 72 different vulnerabilities that our data-d approach can identify when executed on 3 from three different datasets.

1. Introduction

In a society that makes software appl tral core of many every-day activities is such software as secure as possible bef to the public. This has led to a large an focused on the development of increasr techniques to discover vulnerabilities, st ware testing [36], [60], [77], symboli [62], [71] and dynamic analysis [73].

In the context of dynamic analysis, proposed many approaches to measure it certain input produces in the software u of the possible metrics is *path coverage* all independent paths present in a progr its software testing, the community has coverage for tests generation [64], [7] of automatically producing inputs that code locations. The main limitation of

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

Coverage-guided g most common techniq metric, which decide essential parameter of results. While there an ness of different cove ally affect the fuzzing it is unclear whether is superior to all the the first systematic s age metrics in fuzzin discuss the concept o retically compare diffi several coverage met study on these metri LAVA-M dataset, and of 221 binaries). We l has limited resources metric has its unique of branches (this vul grand slam coverage also explore combini cross-seeding, and th fuzzing based appro of binaries in the CGC that combines fuzzin time, our approach u

Abstract—Coverage-guided greybox fuzzers rely on feedback derived from control-flow coverage to explore a target program and uncover bugs. This is despite control-flow feedback offering only a coarse-grained 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 when heavyweight program analyses (e.g., taint analysis, symbolic execution) are used. Unfortunately, these more accurate analyses incur a high run-time penalty, impeding fuzzing throughput. Lightweight data-flow alternatives to control-flow fuzzing remain unexplored.

We present DATAFLOW, a greybox fuzzer driven by lightweight data-flow profiling. 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: indeed, there may be no control dependence between those operations. As such, 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, allowing the 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 uncovering bugs in most targets we fuzzed (72% of them). 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 nique that has been w panies such as Googl tups (e.g., Trail of Bi Challenge (CGC), gr to be more effective symbolic execution a

USENIX Association

I. 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—can be traced back to an assignment in a graduate Advanced Operating Systems class [1]. These fuzzers were relatively primitive (compared to a modern fuzzer): they simply fed a randomly-generated input to the target, failing the test if the target crashed or hung. 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

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.¹ This allows the fuzzer to focus its mutations on inputs reaching new code. Intuitively, a fuzzer cannot find bugs 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’s control-flow graph (CFG). A large body of research has built on AFL [4–12]. While improvements have been made, most fuzzers still default to edge coverage as an approximation of program behavior. *Is this the best we can do?*

In some targets, control flow offers only a coarse-grained approximation of program behavior. This includes targets whose control structure is decoupled from its semantics (e.g., LR parsers generated by yacc) [13]. Such targets require *data flow* coverage [13–17]. Whereas control flow focuses on the order of operations in a program (i.e., branch and loop structures), data flow instead focuses on how variables (i.e., data) are defined and used [14]; indeed, there may be no control dependence between variable definition and use sites (see §III for details).

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 at runtime. Unfortunately, accurate DTA is difficult to achieve and expensive to compute (e.g., prior work has found DTA is expensive [18, 19] and its accuracy highly variable across implementations [18, 20]). Moreover, several real-world programs fail to compile under DTA, increasing deployability concerns. Thus, most widely-deployed greybox fuzzers (e.g., AFL [3], libFuzzer [21], and honggfuzz [22]) eschew DTA in favor of higher fuzzing throughput.

While lightweight alternatives to DTA exist (e.g., RETROQUEER [23], GREYONE [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 Fuzzor [1] is now known as a *Madbox* fuzzer, because it has no knowledge of the target’s internals.

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

Abstract—Recent advances in fuzz testing several forms of feedback mechanisms, fact that for a large range of programs a coverage alone is insufficient to reveal con- spired by this line of research, we examine representations looking for a match betw of the structure and adaptability to th testing. In particular, we believe that data d (DDCs) represent a good candidate for this information embedded by this data struc useful to find vulnerable constructs by s tions of def-use pairs that would be difficu fuzzer to trigger. Since some portions o graph overlap with the control flow of t possible to reduce the additional instrun only “interesting” data-flow dependency the fuzzer to visit the code in a distinct standard methodologies.

To test these observations, in this p DDFuzz, a new approach that rewards tl with code coverage information, but also in the data dependency graph are hit. that the adoption of data dependency i coverage-guided fuzzing is a promising sol to discover bugs that would otherwise rem standard coverage approaches. This is det 72 different vulnerabilities that our data-d approach can identify when executed on 3 from three different datasets.

1. Introduction

In a society that makes software appl tral core of many every-day activities is such software as secure as possible bef to the public. This has led to a large an focused on the development of increasr techniques to discover vulnerabilities, st ware testing [36], [60], [77], symboli [62], [71] and dynamic analysis [73].

In the context of dynamic analysis, proposed many approaches to measure it certain input produces in the software u of the possible metrics is *path coverage* all independent paths present in a progr its software testing, the community has coverage for tests generation [64], [7] of automatically producing inputs that code locations. The main limitation of

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@ehelwelt.net

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

Coverage-guided g most common techniq metric, which decide essential parameter of results. While there an ness of different cove ally affect the fuzzing it is unclear whether is superior to all the the first systematic s age metrics in fuzzin discuss the concept o ritically compare diffi several coverage met study on these metri LAVA-M dataset, and of 221 binaries). We l has limited resourc metric has its unique of branches (this vul grand slam coverage also explore combi cross-seeding, and th fuzzing based on CGC of binaries in the CCP that combines fuzzin time, our approach u

Abstract—Coverage-guided greybox fuzzers rely on feedback derived from control-flow coverage to explore a target program and uncover bugs. This is despite control-flow feedback offering only a coarse-grained approximation of program behavior. *Data flow* intuitively more-accurately characterizes program behavior. Despite this advantage, fuzzers driven by data-flow edges have received comparatively little attention, appearing mainly when heavyweight program analyses (e.g., taint analysis, symbolic execution) are used. Unfortunately, these more accurate analyses incur a high run-time penalty, impeding fuzzing throughput. Lightweight data-flow alternatives to control-flow fuzzing remain unexplored.

We present DATAFLOW, a greybox fuzzer driven by lightweight data-flow profiling. 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: indeed, there may be no control dependence between those operations. As such, 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, allowing the 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 uncovering bugs in most targets we fuzzed (72% of them). 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 nique that has been v panies such as Googl tups (e.g., Trail of Bi Challenge (CGC), gr to be more effective symbolic execution a

USENIX Association

I. 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—can be traced back to an assignment in a graduate Advanced Operating Systems class [1]. These fuzzers were relatively primitive (compared to a modern fuzzer): they simply fed a randomly-generated input to the target, failing the test if the target crashed or hung. 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

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.¹ This allows the fuzzer to focus its mutations on inputs reaching new code. Intuitively, a fuzzer cannot find bugs 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’s control-flow graph (CFG). A large body of research has built on AFL [4–12]. While improvements have been made, most fuzzers still default to edge coverage as an approximation of program behavior. *Is this the best we can do?*

In some targets, control flow offers only a coarse-grained approximation of program behavior. This includes targets whose control structure is decoupled from its semantics (e.g., LR parsers generated by yacc) [13]. Such targets require *data flow* coverage [13–17]. Whereas control flow focuses on the order of operations in a program (i.e., branch and loop structures), data flow instead focuses on how variables (i.e., data) are defined and used [14]; indeed, there may be no control dependence between variable definition and use sites (see §III for details).

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 at runtime. Unfortunately, accurate DTA is difficult to achieve and expensive to compute (e.g., prior work has found DTA is expensive [18, 19] and its accuracy highly variable across implementations [18, 20]). Moreover, several real-world programs fail to compile under DTA, increasing deployability concerns. Thus, most widely-deployed greybox fuzzers (e.g., AFL [3], libFuzzer [21], and honggfuzz [22]) eschew DTA in favor of higher fuzzing throughput.

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

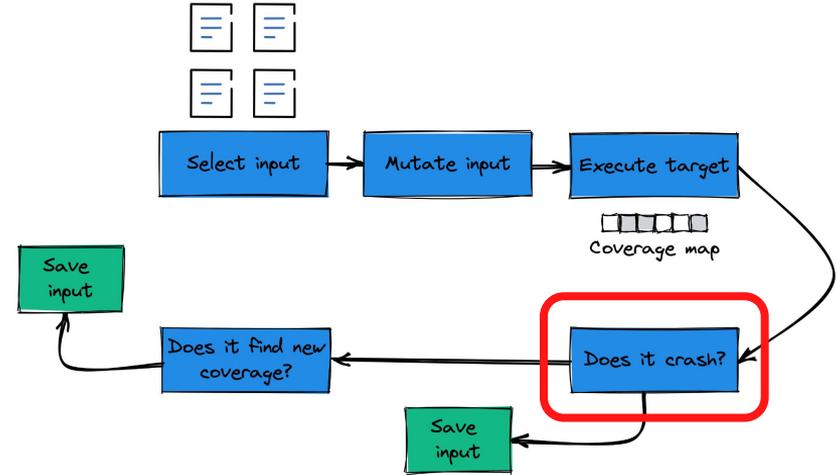
International Fuzzing Workshop (FUZZING) 2022
24 April 2022, San Diego, CA, USA
ISBN 1-891562-77-0
https://dx.doi.org/10.14722/fuzzing.2022.23001
www.doss-symposium.org

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

Greybox fuzzing

Does it crash?

- Classic memory-safety violation
 - SIGSEGV

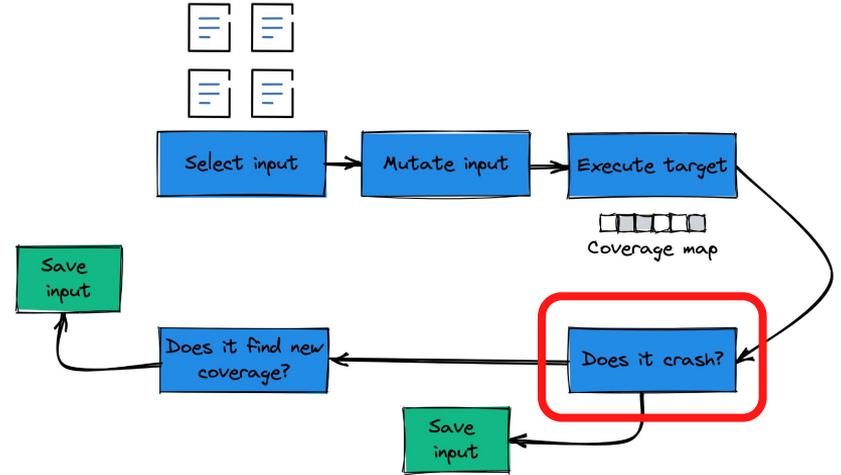


Greybox fuzzing

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 Franz
University of California, Irvine
{dokyungs, jlettner, rajasekp, yeouln, stijnv, perl, franz}@uci.edu

RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization

Abstract—The C and C++ binaries insecure yet remain resort to a multi-pronged in adversaries. These include analysis. Dynamic bug fine can find bugs that side observe the actual execut directly observe incorrect p A vast number of sanit demics and refined by pr overview of sanitizers with security issues. Specifically, the security vulnerabilities t and compatibility proprie

Abstract—Analyzing the security currently incorrect for end-users, on third-party libraries. Such analysis ability discovery techniques, most no enabled. The current state of the sanitization to binaries is dynamic b prohibitive performance overhead. TI binary rewriting, cannot fully recove and hence has difficulty modifying bi for fuzzing or to add security check The ideal solution for binary secur rewriter that can intelligently add if as if it were inserted at compile t requires an analysis to statically disa and scalars, a problem known to be case. We show that recovering this practice for the most common clas 64-bit, position independent code. I we develop RetroWrite, a binar; to support American Fuzzy Lop (A (AFuLoP)), and show that it can ad nance while retaining precision. Bin guided fuzzing using Retrowrite i to compiler-instrumented binaries i QEMU-based instrumentation by bugs. Our implementation of binar; faster than Valgrind's memcheck, the memory checker, and detects 80% n

Abstract—Analyzing the security currently incorrect for end-users, on third-party libraries. Such analysis ability discovery techniques, most no enabled. The current state of the sanitization to binaries is dynamic b prohibitive performance overhead. TI binary rewriting, cannot fully recove and hence has difficulty modifying bi for fuzzing or to add security check The ideal solution for binary secur rewriter that can intelligently add if as if it were inserted at compile t requires an analysis to statically disa and scalars, a problem known to be case. We show that recovering this practice for the most common clas 64-bit, position independent code. I we develop RetroWrite, a binar; to support American Fuzzy Lop (A (AFuLoP)), and show that it can ad nance while retaining precision. Bin guided fuzzing using Retrowrite i to compiler-instrumented binaries i QEMU-based instrumentation by bugs. Our implementation of binar; faster than Valgrind's memcheck, the memory checker, and detects 80% n

Abstract—Analyzing the security currently incorrect for end-users, on third-party libraries. Such analysis ability discovery techniques, most no enabled. The current state of the sanitization to binaries is dynamic b prohibitive performance overhead. TI binary rewriting, cannot fully recove and hence has difficulty modifying bi for fuzzing or to add security check The ideal solution for binary secur rewriter that can intelligently add if as if it were inserted at compile t requires an analysis to statically disa and scalars, a problem known to be case. We show that recovering this practice for the most common clas 64-bit, position independent code. I we develop RetroWrite, a binar; to support American Fuzzy Lop (A (AFuLoP)), and show that it can ad nance while retaining precision. Bin guided fuzzing using Retrowrite i to compiler-instrumented binaries i QEMU-based instrumentation by bugs. Our implementation of binar; faster than Valgrind's memcheck, the memory checker, and detects 80% n

Sushant Dinesh
Purdue University

Nathan Burrow
Purdue University

Dongyan Xu
Purdue University

Mathias Payer
EPFL

HexType: Efficient Detection of Type Confusion Errors for C++

Yuseok Jeon
Purdue University
jeon41@purdue.edu

Priyam Biswas
Purdue University
biswas12@purdue.edu

Scott Carr
Purdue University
carr27@purdue.edu

Byoungyoung Lee
Purdue University
byoungyoung@purdue.edu

Mathias Payer
Purdue University
mathias.payer@epfl.ch

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++, but 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 runtime type 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 software products including Adobe Flash, PHP, Google Chrome, or Firefox.

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.

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++.

CCS CONCEPTS

Security and privacy → Systems security; Software and application security;

KEYWORDS

Type confusion; Bad casting; Type safety; Typing; Static_cast; Dynamic_cast; Reinterpret_cast

1 INTRODUCTION

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 citation 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 of the publisher. For more information, contact the publisher at 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.
ISSN 978-1-4503-4966-6/17/11...\$15.00
DOI: <http://dx.doi.org/10.1145/3133956.3134602>

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 citation 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 of the publisher. For more information, contact the publisher at 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.
ISSN 978-1-4503-4966-6/17/11...\$15.00
DOI: <http://dx.doi.org/10.1145/3133956.3134602>

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 of the fields or virtual functions of the descendant class. When the program subsequently uses the fields or functions, it may use data, say, as a regular field in one context and as a virtual function table (vtable) pointer in another. Such type confusion vulnerabilities are not only wide-spread (e.g., many are found in a wide range of software products, such as Google Chrome (CVE-2017-5023), Adobe Flash (CVE-2017-2095), WebKit (CVE-2017-2415), Microsoft Internet Explorer (CVE-2015-6184) and PHP (CVE-2016-3185)), but also security critical (e.g., many are demonstrated to be easily exploitable due to deterministic runtime behaviors).

Previous research efforts tried to address the problem through runtime checks for static casts. Existing mechanisms can be categorized into two types: (i) mechanisms that identify objects through existing fields embedded in the objects (such as vtable pointers) [6, 14, 29, 38], and (ii) mechanisms that leverage disjoint metadata [15, 21]. 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 vtable 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 unexpectedly 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?

SoK: Sanitizing for Security

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

RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization

Abstract—The C and C++ binaries insecure yet remain resort to a multi-pronged in adversaries. These include analysis. Dynamic bug fine can find bugs that code observe the actual execution directly observe incorrect p A vast number of sanitizers currently instrumented for end-users, on third-party libraries. Such analysis ability discovery techniques, most no enabled. The current state of the sanitization to binaries is dynamic b prohibitive performance overhead. TI binary rewriting, cannot fully recover and hence has difficulty modifying bit as if it were inserted at compile I. IN C and C++ remain the systems software such as libraries, and browsers. A and leave the programmatic hardware. On the flip side every memory access is a undefined behavior, etc. In short of meeting these req make the code vulnerable At the same time, mem more sophisticated [1]–[4] tions such as Address Spr and Data Execution Preven as Return-Oriented Progra such as function pointers, the control-flow of the progr Data-Oriented Programmi can be invoked on legal c program by corrupting on As a first line of defen analysis tools to identify se is deployed in production program analysis, dynamic Static tools analyze the I results that are conservative of the code [5]–[9]. In c often called “sanitizers” and output a precise analy. Sanitizers are now in v many vulnerability discove and critical role in finding not well-understood, whic

Sushant Dinesh
Purdue University

Nathan Burrow
Purdue University

Dongyan Xu
Purdue University

Mathias Payer
EPFL

Abstract—Analyzing the security currently instrumented for end-users, on third-party libraries. Such analysis ability discovery techniques, most no enabled. The current state of the sanitization to binaries is dynamic b prohibitive performance overhead. TI binary rewriting, cannot fully recover and hence has difficulty modifying bit as if it were inserted at compile I. IN C and C++ remain the systems software such as libraries, and browsers. A and leave the programmatic hardware. On the flip side every memory access is a undefined behavior, etc. In short of meeting these req make the code vulnerable At the same time, mem more sophisticated [1]–[4] tions such as Address Spr and Data Execution Preven as Return-Oriented Progra such as function pointers, the control-flow of the progr Data-Oriented Programmi can be invoked on legal c program by corrupting on As a first line of defen analysis tools to identify se is deployed in production program analysis, dynamic Static tools analyze the I results that are conservative of the code [5]–[9]. In c often called “sanitizers” and output a precise analy. Sanitizers are now in v many vulnerability discove and critical role in finding not well-understood, whic

HexType: Efficient Detection of Type Confusion Errors for C++

Yuseok Jeon
Purdue University
jeon41@purdue.edu

Priyam Biswas
Purdue University
biswas12@purdue.edu

Scott Carr
Purdue University
carr27@purdue.edu

Byoungyoung Lee
Purdue University
byoungyoung@purdue.edu

Mathias Payer
Purdue University
mathias.payer@nebelwelt.net

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 runtime type 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 software products including Adobe Flash, PHP, Google Chrome, or Firefox.

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.

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++.

CSS CONCEPTS

Security and privacy → Systems security; Software and application security;

KEYWORDS

Type confusion; Bad casting; Type safety; Typing; Static_cast; Dynamic_cast; Reinterpret_cast

1 INTRODUCTION

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 citation 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 of the publisher. For more information, contact the publisher at permissions@acm.org.
CST'17, Oct. 30–Nov. 3, 2017, Dallas, TX, USA.
© 2017 Copyright held by the owner/authors(s). Publication rights licensed to ACM.
ISSN 978-1-4503-4566-6/17/11...\$15.00
DOI: <http://dx.doi.org/10.1145/3133956.3134662>

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 citation 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 of the publisher. For more information, contact the publisher at permissions@acm.org.
CST'17, Oct. 30–Nov. 3, 2017, Dallas, TX, USA.
© 2017 Copyright held by the owner/authors(s). Publication rights licensed to ACM.
ISSN 978-1-4503-4566-6/17/11...\$15.00
DOI: <http://dx.doi.org/10.1145/3133956.3134662>

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 security: 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 mistake 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 of the fields or virtual functions of the descendant class. When the program subsequently uses the fields or functions, it may use data, say, as a regular field in one context and as a virtual function table (vtable) pointer in another. Such type confusion vulnerabilities are not only wide-spread (e.g., many are found in a wide range of software products, such as Google Chrome (CVE-2017-5023), Adobe Flash (CVE-2017-2095), WebKit (CVE-2017-2415), Microsoft Internet Explorer (CVE-2015-6184) and PHP (CVE-2016-3185)), but also security critical (e.g., many are demonstrated to be easily exploitable due to deterministic runtime behaviors).

Previous research efforts tried to address the problem through runtime checks for static casts. Existing mechanisms can be categorized into two types: (i) mechanisms that identify objects through existing fields embedded in the objects (such as vtable pointers) [6, 14, 29, 38], and (ii) mechanisms that leverage disjoint metadata [15, 21]. 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 vtable 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 unexpectedly 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

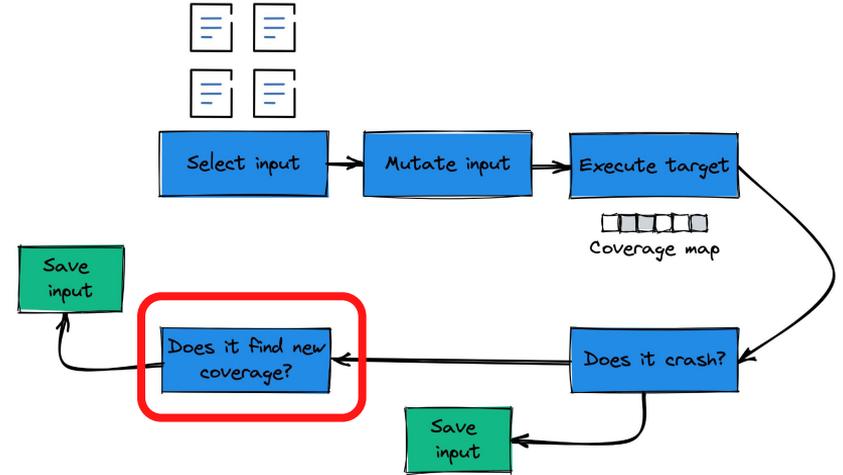
Anything we can encode as an **invariant**

- 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 Dav
¹ Beijing K
Ins

³ School of Cyber
[zhengyaowen,zhubongso

Abstract

Cyber attacks against IoT devices an attacks exploit software vulnerabil Fuzzing is an effective software tes nerability discovery. In this work, we first high-throughput greybox fuzzer AFL addresses two fundamental pr First, it addresses compatibility issues POSIX-compatible firmware that can emulator. Second, it addresses the f caused by system-mode emulation - called augmented process emulation mode emulation and user-mode em augmented process emulation provid system-mode emulation and high th emulation. Our evaluation results sho fully functional and capable of findin ties in IoT programs: (2) the through average 8.2 times higher than system fuzzing; and (3) FIRM-AFL is able t ries much faster than system-mode e and is able to find 0-day vulnerabilities

1 Introduction

The security impact of IoT devices o By 2020, the number of connected IoT number of people [10]. This creates i surface leaving almost everybody w the hackers leverage the lack of sec create large botnets (e.g., Mirai, VPMI malware attacks exploit the vulnerl to penetrate into the IoT devices. As defenders to discover vulnerabilities then before attacks.

*The work was done while visiting Univer
¹Corresponding author

USENIX Association

MoonShine: Optimizing OS Fuzzer Seed Selection with Trace Distillation

Shankara Palloor, Andrew Aday, and Suman Jana
Columbia University

Abstract

OS fuzzers primarily tween the OS kernel a rity vulnerabilities. TI lutionary OS fuzzers diversity of their seed generating good seed as the behavior of ea the OS kernel state c system calls. Therefo often rely on hand-c sequences of system c process. Unfortunately the diversity of the se fore limits the effecti

In this paper, we c In this paper, we egy for distilling see traces of real-world dependencies across i ages light-weight stat dependencies across c We designed and extension to Syszkal fuzzer for the Linux taining 2.8 million s real-world programs, over 14,000 calls w code coverage. Usin sequences, MoonShir achieved code covera average. MoonShine in the Linux kernel th

1 Introduction

Security vulnerability after-free inside, oper ticularly dangerous a completely comprom a popular technique fixing such critical s fuzzers focus primar face as it is one of the OS kernel and u

Abstract

Coverage-guided stream and we has area recently. Ho test network servi methods. In this plementation of f approach that can spanning servers, Process Communi of-the-art method up to 300x and co Nyx-Net is able t targets that no t Nyx-Net is to play speedups of 10-30 Nyx-Net is able t such as Lightpdc, Firefox’s IPC mec versatility of the j implementation v abling fuzzing on solving a long-sta

In this paper, we present SnapFuzz, a novel fuzzing framework for network applications. SnapFuzz offers a robust architecture 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: LightFTP, TinyDTLS, Dnsmaq, LIV5555 and Demqsp. We report impressive performance speedups of 62.8 x, 41.2 x, 30.6 x, 24.6 x, and 8.4 x, 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 r
• Software i
• Software a
• Software v

Keywords

Testin
ACM Reference F
Sergej Schumilo, C
basi, and Thorsten
Incremental Snapsh
Pater Systems (fanz
New York, NY, USA,

ACM Reference Format

Anastasio 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 (ISSTA ’22)*, July 18–22, 2022, Virtual, South Korea. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3533767.3534376>

INTRODUCTION

Fuzzing is an effective technique for testing software systems, with popular fuzzers such as AFL and L1Fuzzer having found thousands of bugs in both open-source and commercial software. For instance,

1 INTRODUCTION

Fuzzing is an effective technique for testing software systems, with popular fuzzers such as AFL and L1Fuzzer having found thousands of bugs in both open-source and commercial software. For instance,

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.

Nyx-Net: Network Fuzzing with Incremental Snapshots

Sergej Schumilo¹, Cornelius Aschermann¹, Andrea Jemmett², Ali Abbasi¹, and Thorsten Holz²
¹Ruhr-Universität Bochum, ²Vrije Universiteit Amsterdam
³CISPA Helmholtz Center for Information Security

SnapFuzz: High-Throughput Fuzzing of Network Applications

Anastasio 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 present SnapFuzz, a novel fuzzing framework for network applications. SnapFuzz offers a robust architecture 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: LightFTP, TinyDTLS, Dnsmaq, LIV5555 and Demqsp. We report impressive performance speedups of 62.8 x, 41.2 x, 30.6 x, 24.6 x, and 8.4 x, 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

Anastasio 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 (ISSTA ’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 L1Fuzzer having found thousands of bugs in both open-source and commercial software. For instance,

ISSTA ’22, July 18–22, 2022, Virtual, South Korea
© 2022 Copyright held by the owner/authors. 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 Record was published in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA ’22)*, July 18–22, 2022, Virtual, South Korea, <https://doi.org/10.1145/3533767.3534376>.

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 protocol they implement (e.g., FTP, DICOM, SIP), and the fact that they have side effects, such as writing data to the file system or exchanging messages over the network.

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 an actual server instance interacts with actual clients.

A second approach, used by AFLNet [30], performs system-level testing by starting actual server and client processes, and generating 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 add or configure various time delays in order to make sure 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 communication into fast synchronous communication, speeds up file operations and removes the need for clean-up scripts via an in-memory filesystem, and improves other aspects such as delaying and automating the forker/server placement, correctly handling signal propagation and eliminating developer-added delays.

These improvements significantly simplify the construction of fuzzing harnesses for network applications and dramatically improve fuzzing throughput in the range of 8.4 x to 62.8 x (mean 30.6 x) for a set of five popular server benchmarks.

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 Davutyan¹, Beijing King Ins

³ School of Cyber [zhengyaowen,zhubongso]

Abstract

Cyber attacks against IoT devices an attacks exploit software vulnerability AFL addresses two fundamental pr First, it addresses compatibility issues POSIX-compatible firmware that can emulator. Second, it addresses the f caused by system-mode emulation- called augmented process emulation mode emulation and user-mode em augmented process emulation provid system-mode emulation and high th emulation. Our evaluation results sho fully functional and capable of findi ties much faster than system-mode e and is able to find 0-day vulnerabilities

1 Introduction

The security impact of IoT devices o By 2020, the number of connected IoT number of people [10]. This creates i surface leaving almost everybody in the hackers leverage the lack of sec create large botnets (e.g., Mirai, VPMI malware attacks exploit the vulnerl to penetrate into the IoT devices. As defenders to discover vulnerabilities then before attacks.

*The work was done while visiting Univer Corresponding author

USENIX Association

MoonShine: Optimizing OS Fuzzer Seed Selection with Trace Distillation

Shankara Palloor, Andrew Aday, and Suman Jana Columbia University

Abstract

OS fuzzers primarily tween the OS kernel a rity vulnerabilities. TI lutionary OS fuzzers diversity of their seed generating good seed as the behavior of ea the OS kernel state c system calls. Therefo often rely on hand-c sequences of system c process. Unfortunately the diversity of the se fore limits the effecti

In this paper, we c implementation of f approach that can spanning servers. Process Communi of-the-art method up to 300x and o Nyx-Net is able t targets that no t Nyx-Net to play speedups of 10-30 Nyx-Net is able t real-world programs, over 14,000 calls w code coverage. Usi sequences, MoonShir achieved code covera average. MoonShine in the Linux kernel th

1 Introduction

Security vulnerability after-free inside, oper ticularly dangerous a completely comprom a popular technique fixing such critica fuzzers focus primari face that is one of the the OS kernel and u

Nyx-Net: Network Fuzzing with Incremental Snapshots

Sergej Schumilo¹, Cornelius Aschermann¹, Andrea Jemmett², Ali Abbasi¹, and Thorsten Holz²
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 improved 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 fuzzing framework for network applications. SnapFuzz offers a robust architecture 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: LightFTP, TinyDTLS, Dnsmasq, LIVENESS and Dempsp. We report impressive performance speedups of 62.8 x, 41.2 x, 30.6 x, 24.6 x, and 8.4 x, 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 (ISSTA '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 L1Fuzzer having found thousands of bugs in both open-source and commercial software. For instance,

ISSTA '22, July 18–22, 2022, Virtual, South Korea
© 2022 Copyright held by the owner/authors. 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 Record was published in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '22)*, July 18–22, 2022, Virtual, South Korea, <https://doi.org/10.1145/3533767.3534376>.

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 protocol they implement (e.g., FTP, DICOM, SIP), and the fact that they have side effects, such as writing data to the file system or exchanging messages over the network.

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 an actual server instance interacts with actual clients.

A second approach, used by AFLNet [30], performs system-level testing by starting actual server and client processes, and generating 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 add or configure various time delays in order to make sure 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 communication into fast synchronous communication, speeds up file operations and removes the need for clean-up scripts via an in-memory filesystem, and improves other aspects such as delaying and automating the forker-server placement, correctly handling signal propagation and eliminating developer-added delays.

These improvements significantly simplify the construction of fuzzing harnesses for network applications and dramatically improve fuzzing throughput in the range of 8.4 x to 62.8 x (mean 30.6 x) for a set of five popular server benchmarks.

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.

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 Davutyan¹,
Beijing King Ins

³ School of Cyber
[zhengyaowen,zhubongso]

Abstract

Cyber attacks against IoT devices an attacks exploit software vulnerability. Fuzzing is an effective software testability discovery. In this work, we first high-throughput greybox fuzzer AFL addresses two fundamental problems. First, it addresses compatibility issues POSIX-compatible firmware that can emulate. Second, it addresses the fragility caused by system-mode emulation called augmented process emulation mode emulation and user-mode emulated process emulation provide system-mode emulation and high throughput. Our evaluation results show fully functional and capable of finding bugs in IoT programs: (2) the throughput average 8.2 times higher than system fuzzer; and (3) FIRM-AFL is able to run 8 times faster than system-mode emulator and is able to find 0-day vulnerabilities.

1 Introduction

The security impact of IoT devices by 2020, the number of connected IoT devices [10]. This creates a surface for almost everybody with the hackers leverage the lack of secure create large botnets (e.g., Mirai, VPN malware attacks exploit the vulnerability to penetrate into the IoT devices. As defenders to discover vulnerabilities them before attackers.

*The work was done while visiting University of Cambridge

UNIVERSITY OF CAMBRIDGE

MoonShine: Optimizing OS Fuzzer Seed Selection with Trace Distillation

Shankara Palloor, Andrew Aday, and Suman Jana
Columbia University

Abstract

OS fuzzers primarily generate good seeds. In this paper, we propose MoonShine, a novel fuzzer that uses trace distillation to generate seeds. MoonShine is able to find 0-day vulnerabilities in the OS kernel state calls. Therefore, often rely on hand-crafted sequences of system calls. Unfortunately, the diversity of the system calls before limits the effectiveness of this approach that can span servers. Process Community of the-art method up to 300x and can Nyx-Net is able to targets that not Nyx-Net to play speedups of 10-30x. Nyx-Net is able to run such as Lighttpd, Firefox's IPC mechanism, the implementation of solving a long-standing problem.

In this paper, we present SnapFuzz, a novel fuzzer for network applications. SnapFuzz offers a robust architecture 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: LightFTP, TinyDTLS, Dnsmaq, LIVES55 and Demqsp. We report impressive performance speedups of 62.8 x, 41.2 x, 30.6 x, and 8.4 x, 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.

1 Introduction

Security vulnerabilities after-free inside operating systems are particularly dangerous as they completely compromise a popular technique for fixing such critical bugs. Fuzzers focus primarily on the OS kernel and user

Nyx-Net: Network Fuzzing with Incremental Snapshots

Sergej Schumilo¹, Cornelius Aschermann¹, Andrea Jemmett², Ali Abbasi¹, and Thorsten Holz²
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 present SnapFuzz, a novel fuzzer for network applications. SnapFuzz offers a robust architecture 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: LightFTP, TinyDTLS, Dnsmaq, LIVES55 and Demqsp. We report impressive performance speedups of 62.8 x, 41.2 x, 30.6 x, and 8.4 x, 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 (ISSTA '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 L1Fuzzer having found thousands of bugs in both open-source and commercial software. For instance,

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 protocol they implement (e.g., FTP, DICOM, SIP), and the fact that they have side effects, such as writing data to the file system or exchanging messages over the network.

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 an actual server instance interacts with external clients.

A second approach, used by AFLNet [30], performs system-level testing by starting actual server and client processes, and generating 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 add or configure various time delays in order to make sure 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.

SnapFuzz addresses both of these challenges through a robust architecture that transforms slow asynchronous network communication into fast synchronous communication, speeds up file operations and removes the need for clean-up scripts via an in-memory filesystem, and improves other aspects such as delaying and automating the forker-server placement, correctly handling signal propagation and eliminating developer-added delays.

These improvements significantly simplify the construction of fuzzing harnesses for network applications and dramatically improve fuzzing throughput in the range of 8.4 x to 62.8 x (mean 30.6 x) for a set of five popular server benchmarks.

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.



This work is licensed under

CC BY-NC
EARS'22, April 5–8,
© 2022 Copyright held
ACM ISBN 978-1-60959-160-3
<https://doi.org/10.1145/3533767.3534376>

ISSTA '22, July 18–22, 2022, Virtual, South Korea
© 2022 Copyright held by the owner/authors. 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 Record was published in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '22)*, July 18–22, 2022, Virtual, South Korea, <https://doi.org/10.1145/3533767.3534376>.

Overcoming “roadblocks”

Program constraints that are hard to meet

Solutions

- Writebox fuzzing
- Concolic execution
- Rewrite the target 

Nick Stephens,
Jacopo Corb

{stephe

Abstract—Memory corruption vu
per risk in software, which attac
unauthorized access to confidential
with access to sensitive data are bec
number of potentially exploitable s
resulting in a greater need for autom
DARPA recently funded a competitio
in prize money, to further research
vulnerability finding and patching, s
research in this area. Current techn
bugs include static, dynamic, and
which each having their own advant
common limitation of systems design
trigger vulnerabilities is that they o
struggle to exercise deeper paths in e

We present Driller, a hybrid vul
which leverages fuzzing and selecti
a complementary manner, to find
fuzzing is used to exercise *compartme*
concolic execution is used to generat
complex checks separating the compa
strengths of the two techniques, we
avoiding the path explosion inherent i
incompleteness of fuzzing. Driller uses
to explore only the paths deemed re
generate inputs for conditions that t
evaluate Driller on 126 applications
event of the DARPA Cyber Grand
efficiency by identifying the same nu
the same time, as the top-scoring tea

Permission to freely reproduce all or part
purposes is granted provided that copies bear
on the first page. Reproduction for commerci
without the prior written consent of the In
author (for reproduction of an entire paper o
if the paper was prepared within the scope of
NDSS '16, 21-24 February 2016, San Diego,
Copyright 2016 Internet Society, ISBN 1-891
http://dx.doi.org/10.14722/ndss.2016.23368

Driller: Augmenting Fuzzing Through Selective Symbolic Execution

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

Sebastian Poepflau Aurélien Francillon

T-Fuzz: fuzzing by program transformation

Hui Peng
Purdue University
peng124@purdue.edu

Yan Shoshitaishvili
Arizona State University
yans@asu.edu

Mathias Payer
Purdue University
mathias.payer@nswc.wisc.edu

Abstract

A major impediment to practical symb
especially when compared to near-nati
fuzz testing. We propose a compilati
symbolic execution that performs bett
implementations by orders of magnitud
an LLVM-based C and C++ compil
execution right into the binary. It ca
developers as a drop-in replacement fo
and we show how to add support for
little effort. In comparison with KLEE
up to three orders of magnitude and an
also outperforms QSYM, a system tha
performance improvements over othe
up to two orders of magnitude and a
Using it on real-world software, we fi
consistently achieves higher coverage
vulnerabilities in the heavily tested Op
have been confirmed by the project m
CVE identifiers.

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 fall complex *sanity checks*, e.g., checks on magic values,
checksums, or hashes.

To improve coverage, existing approaches rely on imprecise
heuristics or complex input mutation techniques (e.g., symbolic
execution or taint analysis) to bypass *sanity checks*. Our novel
method tackles coverage from 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 fail. These checks are then removed from the
target program. Fuzzing then continues on 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 chal-
lenges: (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 reproduce true bugs in the original program.

By transforming the program as well as *mutating the input*, T-
Fuzz covers more code and finds more true bugs than any existing
technique. We have evaluated T-Fuzz on the DARPA Cyber
Grand Challenge dataset, LAVA-M dataset and 4 real-world
programs (e.g., *tielfifo*, *magic* and *pedf-openssl*). For
the CGC dataset, 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.

I. 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
paper was prepared within the scope of employment.

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 = **con**crete + **sym**bo**lic**

- 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 fuz

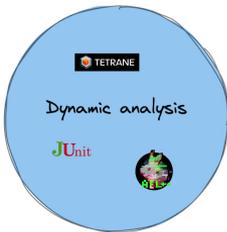
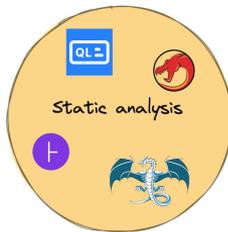


Conclusions

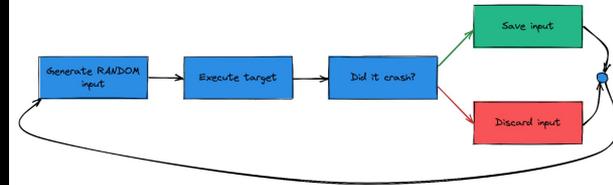
- Fuzzing research has progressed in leaps and bounds
 - No longer just “file-based + *nix-based”
- Still many open problems
- Balance between **performance** and **accuracy**



What is fuzzing?



Our first fuzzer

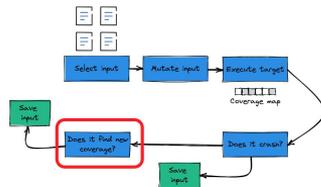


This is a classic **generational blackbox** fuzzer

Greybox fuzzing

Does it find new coverage?

- Save input
- Return to start



Grammar-based fuzzing

NAUTILUS: Fishing for Deep Bugs with Grammars



- 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

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?

SOK: Sanitizing for Security



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**