
A NEW COMEDY
FROM THE GUYS THAT CREATED
SHAUN OF THE DEAD
POLICE
THEY'RE BAD BOYS.
THEY'RE DIE HARDS.
THEY'RE LETHAL WEAPONS.
THEY ARE...
HOT FUZZ
SIMON PEGG NICK FROST
ROGUE PICTURES PRESENTS IN ASSOCIATION WITH STUDIOCANAL A WORKING TITLE PRODUCTION IN ASSOCIATION WITH BIG TALK PRODUCTIONS
"HOT FUZZ" SIMON PEGG NICK FROST JIM BROADBENT CASTING DIRECTOR NINA GOLD MUSIC BY DAVID ARNOLD COSTUME DESIGNER ANNIE HARDINGE
EDITOR CHRIS DICKENS PRODUCTION DESIGNER MARCUS ROWLAND DIRECTOR OF PHOTOGRAPHY JESS HALL LINE PRODUCER RONALDO VASCONCELLOS EXECUTIVE PRODUCER NATASCHA WHARTON
WRITTEN BY EDGAR WRIGHT & SIMON PEGG PRODUCED BY NIRA PARK TIM BEVAN ERIC FELLNER DIRECTED BY EDGAR WRIGHT
R RESTRICTED
UNDER 17 REQUIRES ACCOMPANYING PARENT OR ADULT GUARDIAN
VIOLENT CONTENT INCLUDING SOME GRAPHIC IMAGES, AND LANGUAGE.
WORKING TITLE
ROGUE PICTURES
www.jointhefuzz.com

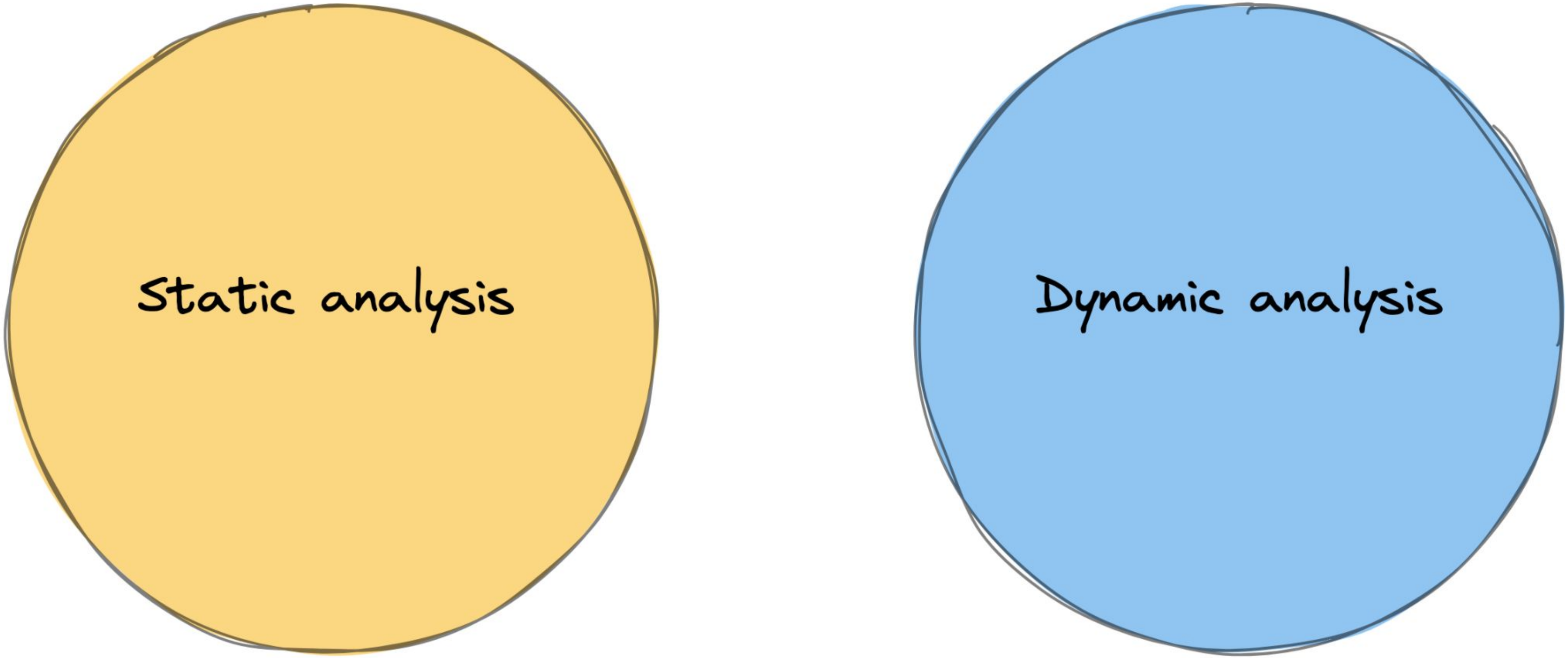
A NEW COMEDY
FROM THE GUYS THAT CREATED
SHAUN OF THE DEAD
POLICE
SIMON
ROGUE PICTURES PRESENTS IN ASSOCIATION WITH
"HOT FUZZ" SIMON PEGG NICK FROST JIM BROADBENT
EDITOR CHRIS DICKENS PRODUCTION DESIGNER MARCUS ROWLAND DIRECTOR OF PHOTOGRAPHY JESS HALL
WRITTEN BY EDGAR WRIGHT & SIMON PEGG PRODUCED BY NIRA PARK
R RESTRICTED
VIOLENT CONTENT INCLUDING SOME GRAPHIC IMAGES, AND LANGUAGE.
www.jointhefuzz.com


YOU CAN'T FIND BUGS WITH SUPER SIMPLE TECHNIQUES
FUZZER GO BRRRRRRRRRR
imgflip.com

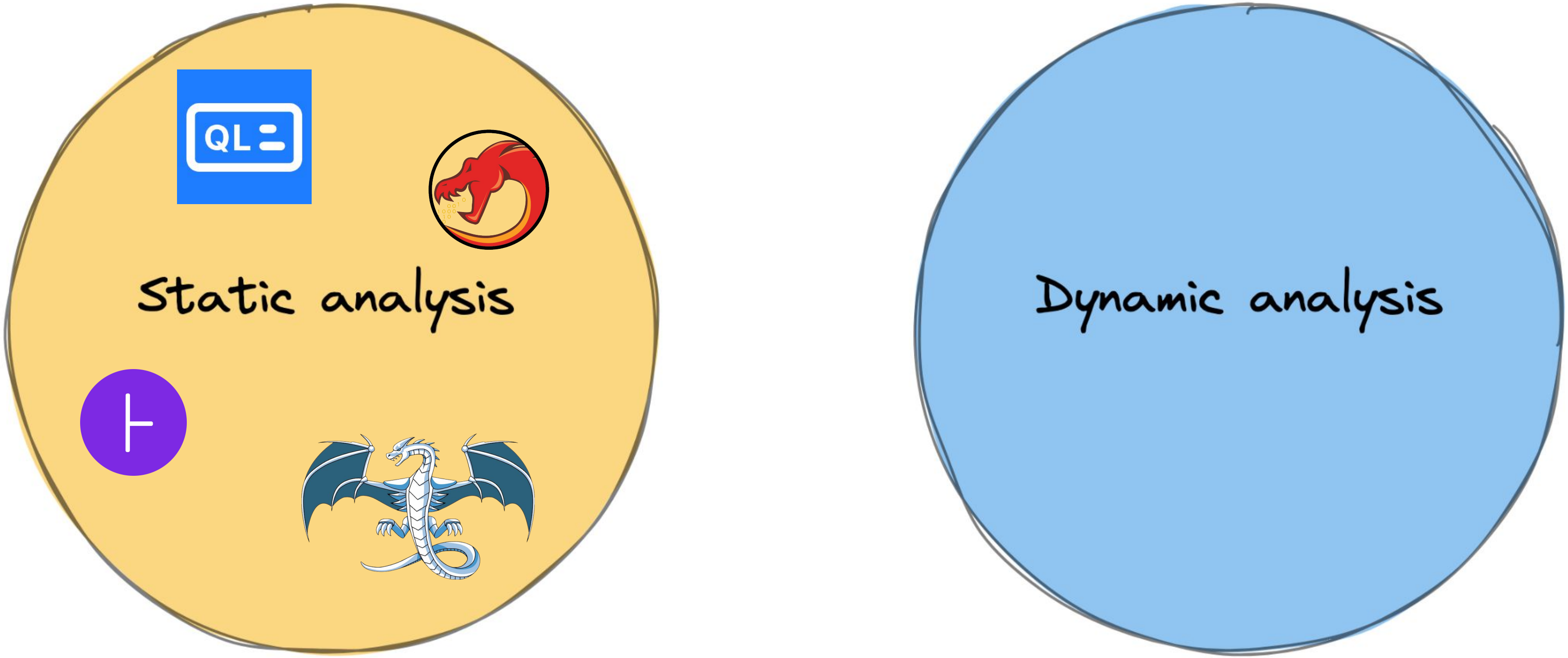
Outline

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

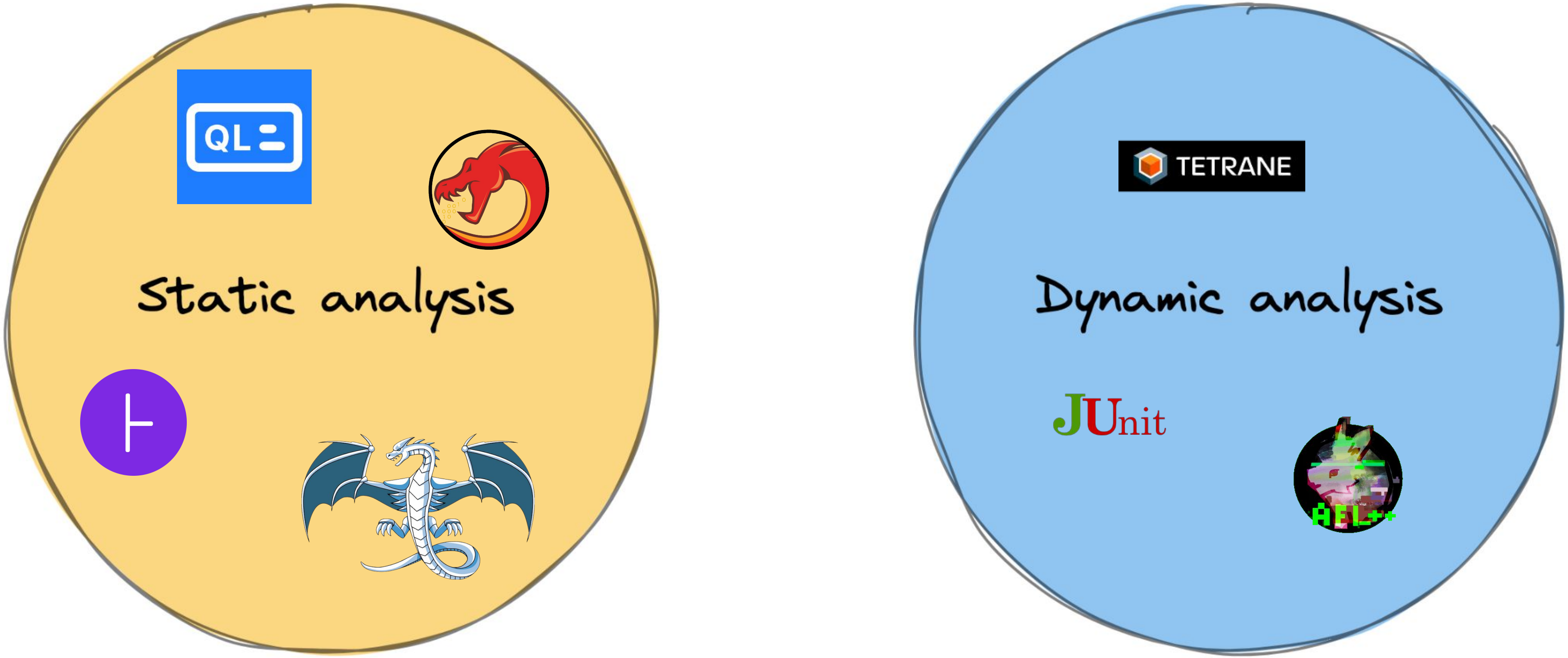
What is fuzzing?



What is fuzzing?



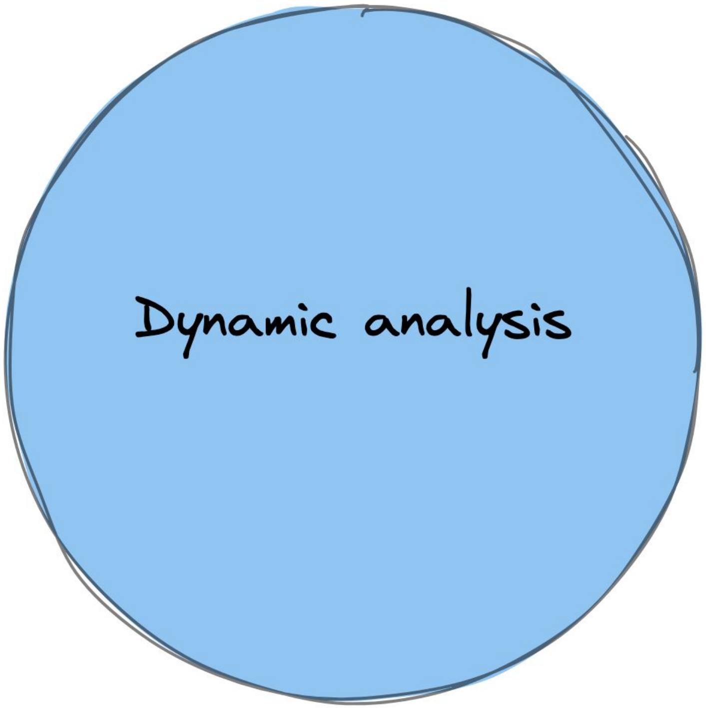
What is fuzzing?



What is fuzzing?

Pros

- No false positives
- Produces PoC
- Scalable



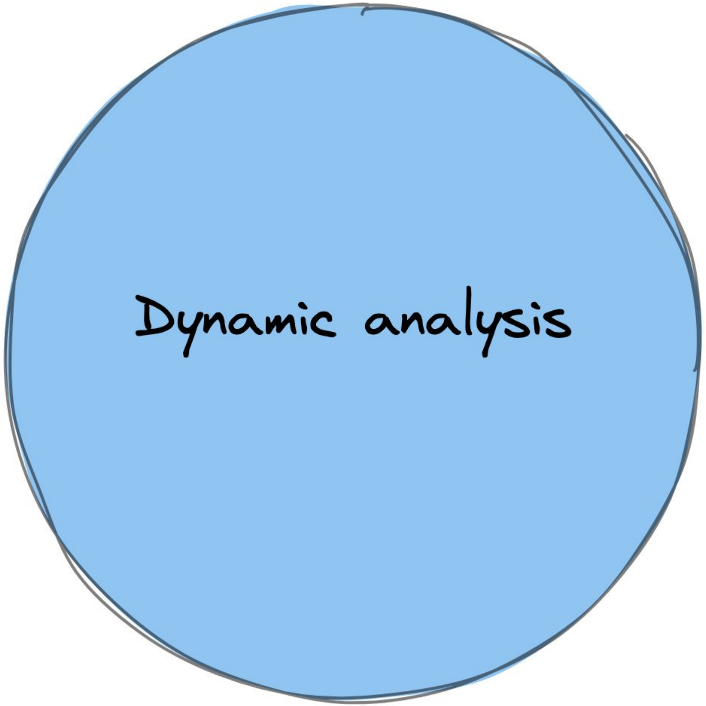
What is fuzzing?

Pros

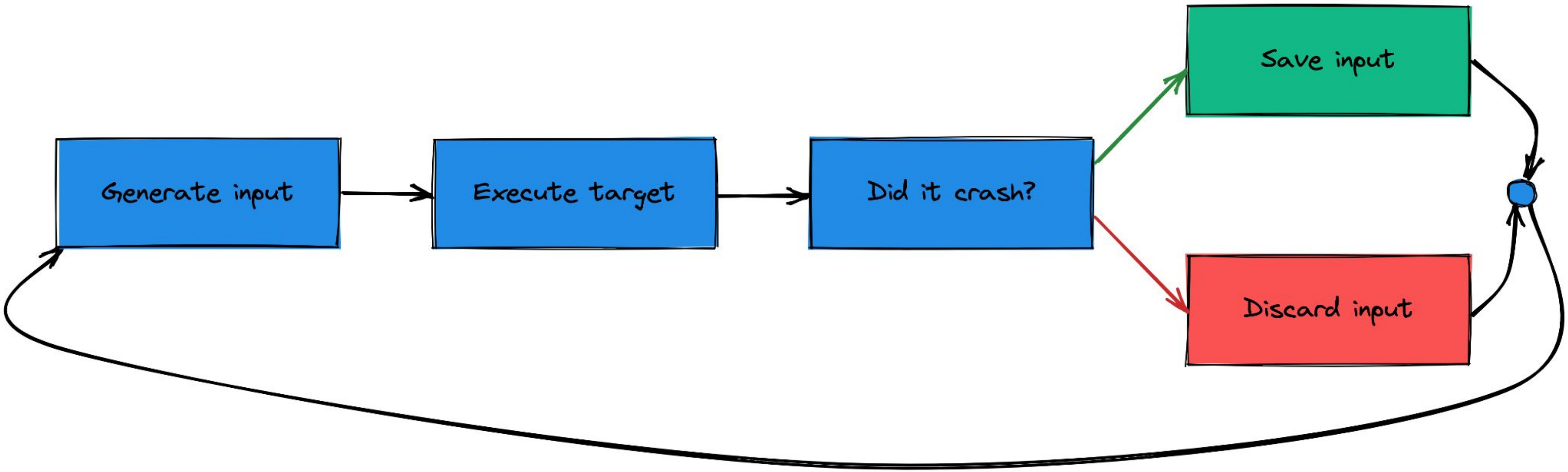
- No false positives
- Produces PoC
- Scalable

Cons

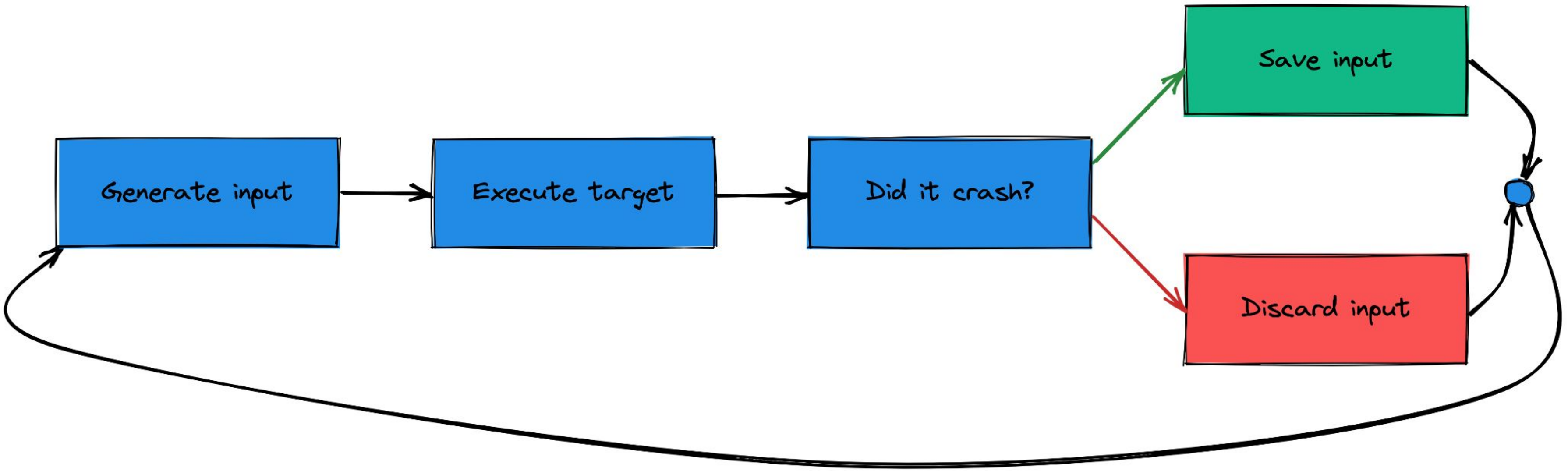
- Incomplete
- Requires buildable target
- Scalability



Our first fuzzer

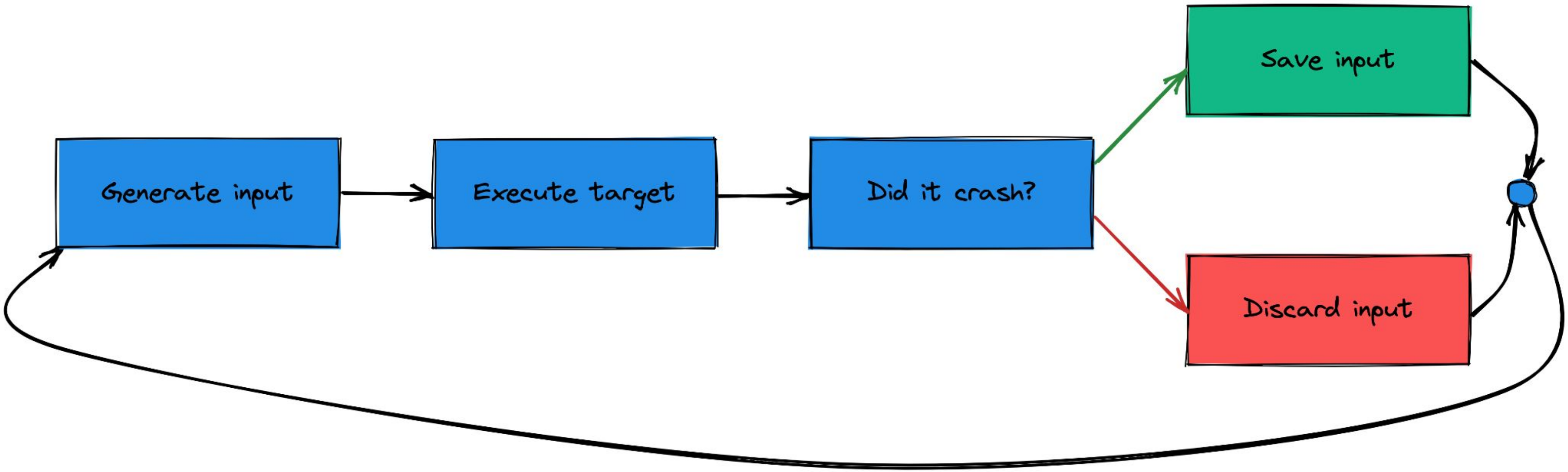


Our first fuzzer



How is this different to dynamic testing?

Our first fuzzer

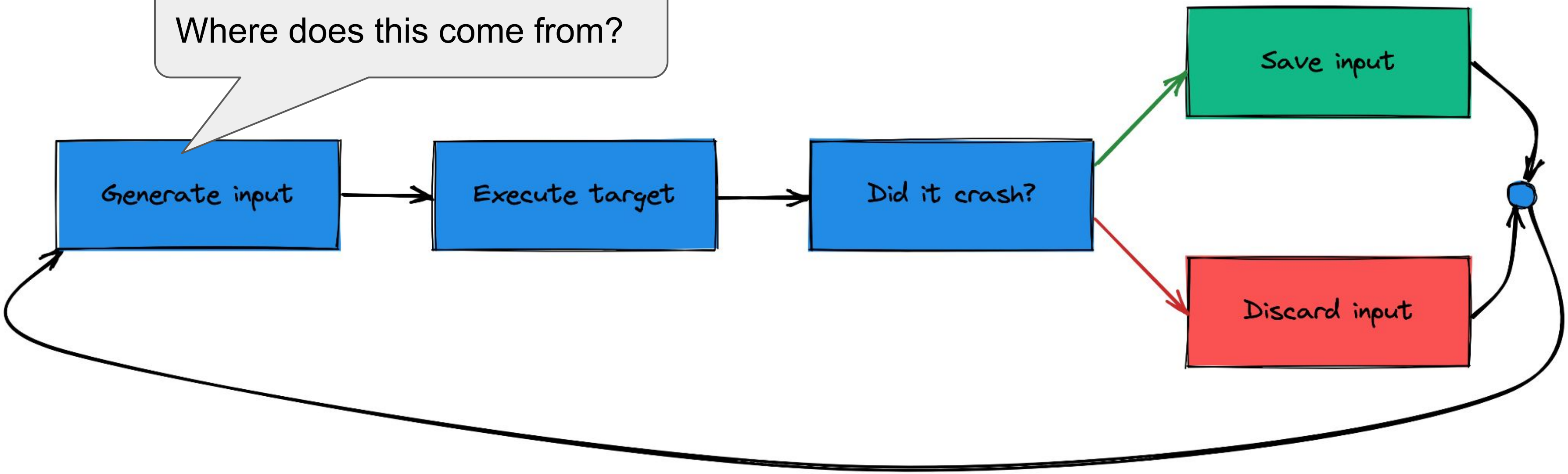


How is this different to dynamic testing?

Or regression testing?

Our first fuzzer

Where does this come from?

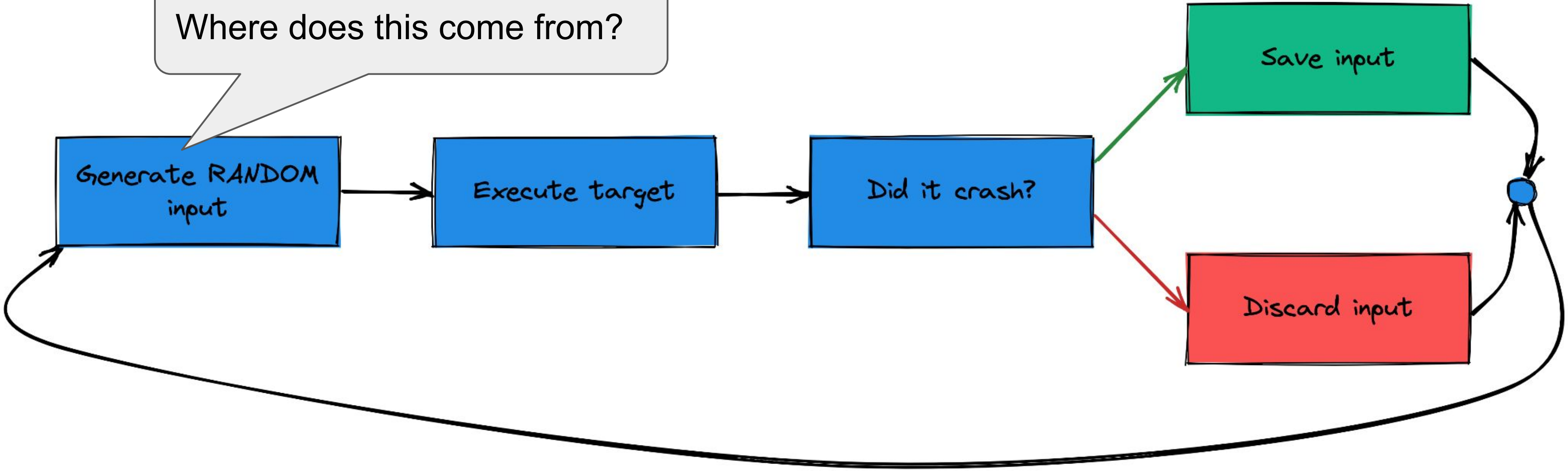


How is this different to dynamic testing?

Or regression testing?

Our first fuzzer

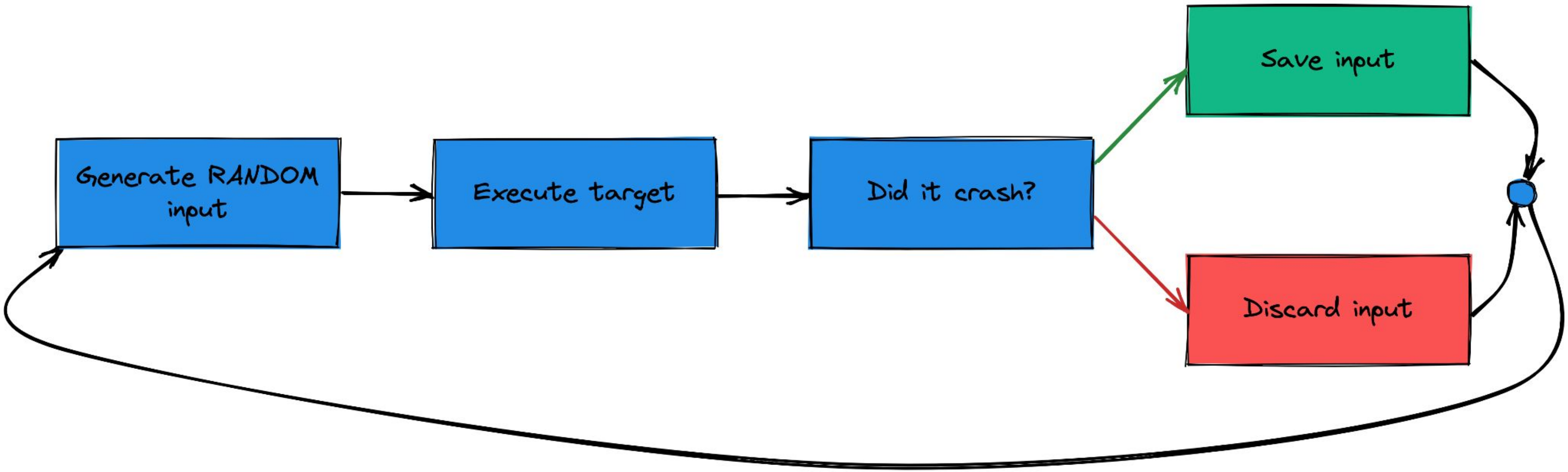
Where does this come from?



Or regression testing?

How is this different to dynamic 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

Barton P. Miller, Lars Fredriksen and Bryan So

An Empirical Study of the Reliability of UNIX Utilities

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 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 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 program crashes were identified and the common mistakes that cause these crashes were categorized. As a result of testing almost 90 different utility programs on seven versions of Unix™, we were able to crash more than 24% of these programs. Our testing included versions of Unix that underwent commercial product testing. A byproduct 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 crashs 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 to the Internet worm (the “gets finger” bug) [2,3] 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 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.

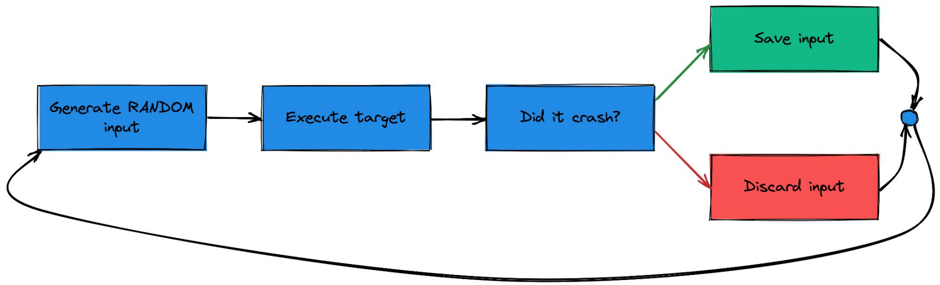
While our testing strategy sounds somewhat naive, its ability to discover fatal program bugs is impressive. If we consider a program to be a complex finite state machine, then our testing strategy can be thought of as a random walk through the state space, searching for undefined states. Similar techniques 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 cases.

Unix is a trademark of AT&T Bell Laboratories.

Blackbox fuzzing

Pros

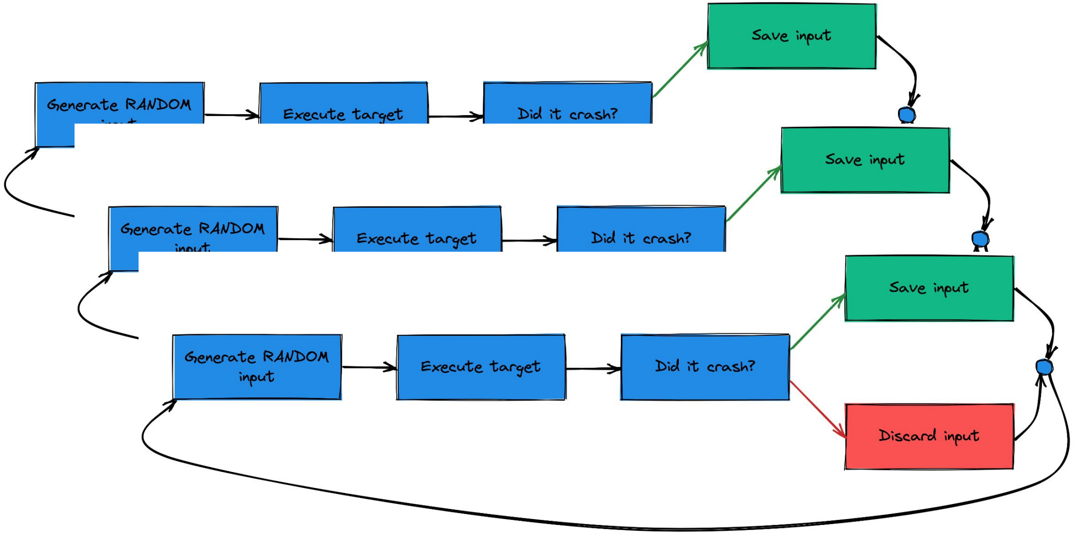
- Simple
- Fast
- Embarrassingly parallel



Blackbox fuzzing

Pros

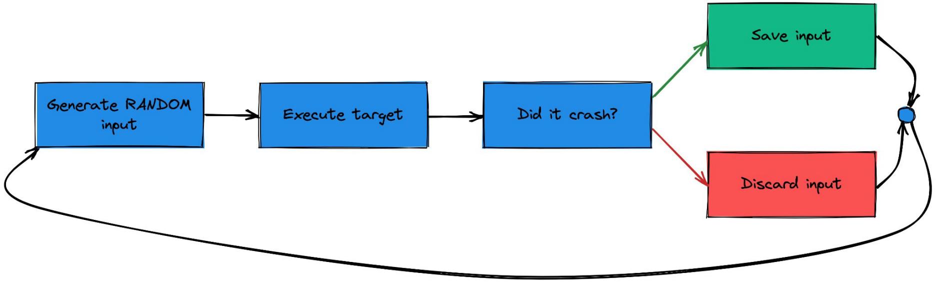
- Simple
- Fast
- Embarrassingly parallel



Blackbox fuzzing

Cons

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

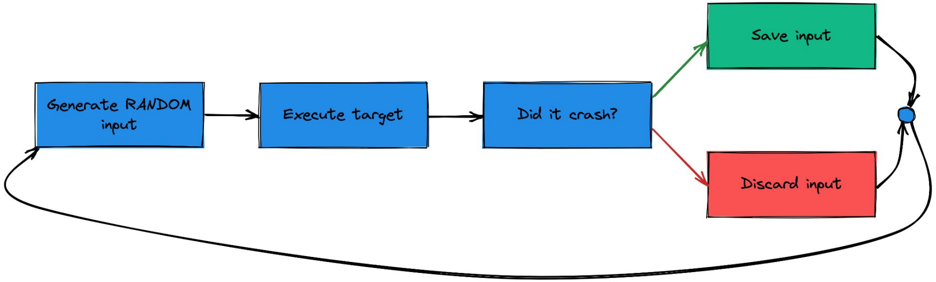


Can we do better?

Blackbox fuzzing

Cons

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



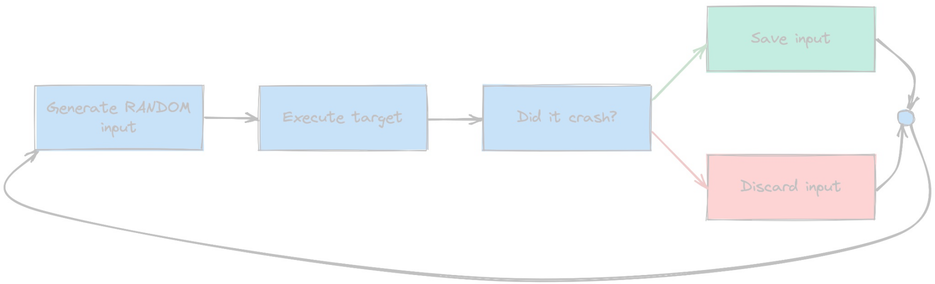
Blackbox fuzzing

Cons

- Generate mostly rubbish
  - ~~Generate~~ **mutate**

- No notion of "progress"

- Only detect `SIGSEGV`



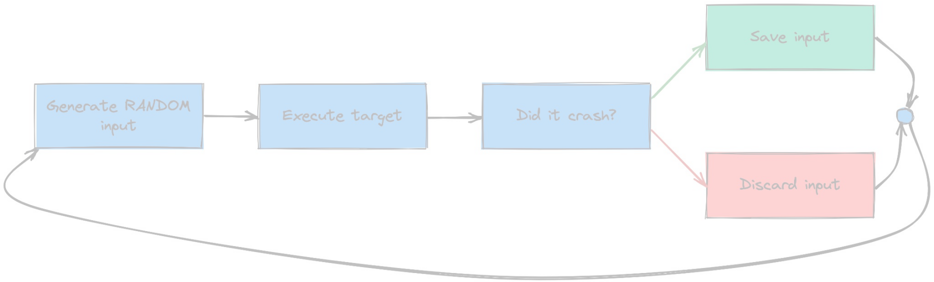
Blackbox fuzzing

Cons

- Generate mostly rubbish
    - ~~Generate~~ **mutate**

- No notion of “progress”
    - Add a **feedback loop**

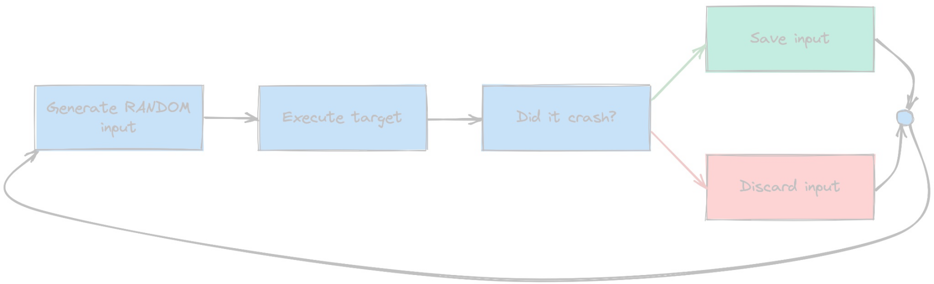
- Only detect `SIGSEGV`



Blackbox fuzzing

Cons

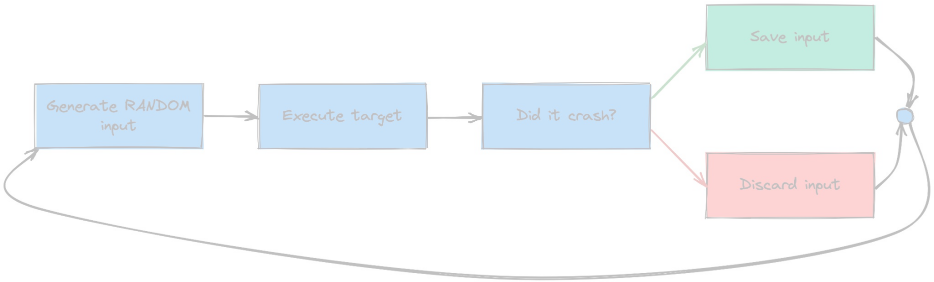
- Generate mostly rubbish
    - ~~Generate~~ **mutate**
- No notion of “progress”
    - Add a **feedback loop**
- Only detect `SIGSEGV`
    - Add a **sanitizer**



Blackbox fuzzing

Cons

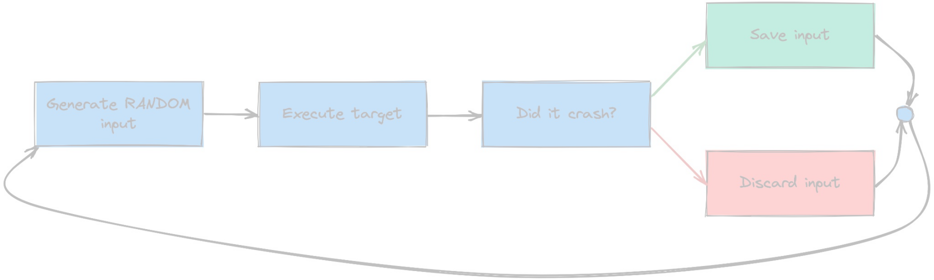
- Generate mostly rubbish
  - ~~Generate~~ **mutate**
- No notion of “progress”
  - Add a **feedback loop**
- Only detect `SIGSEGV`
  - Add a **sanitizer**



Blackbox fuzzing

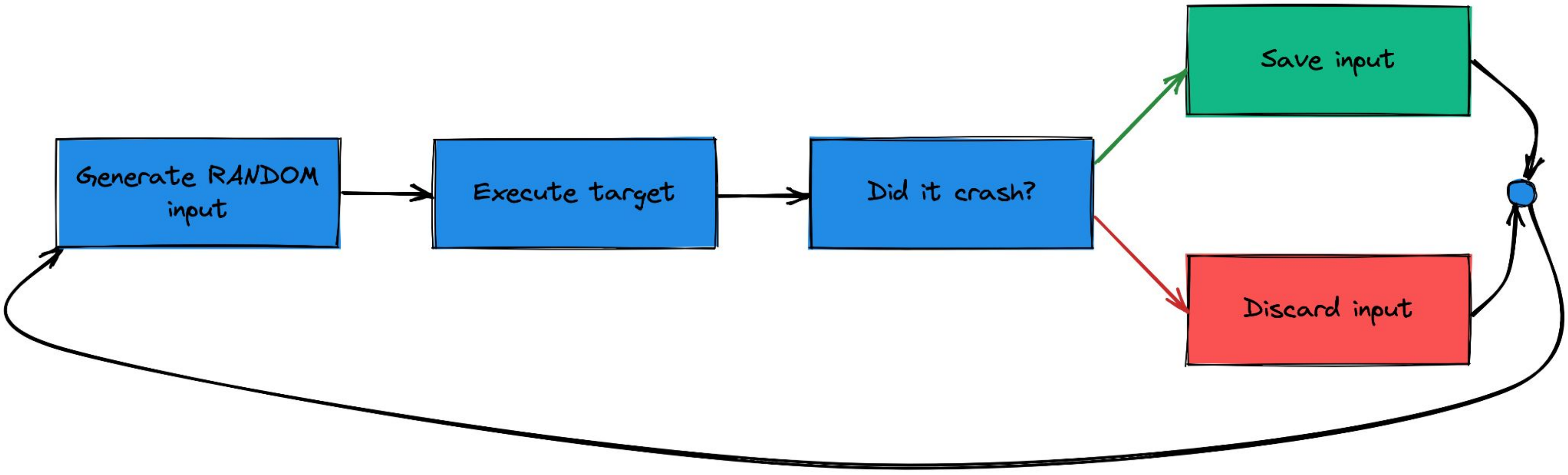
Cons

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

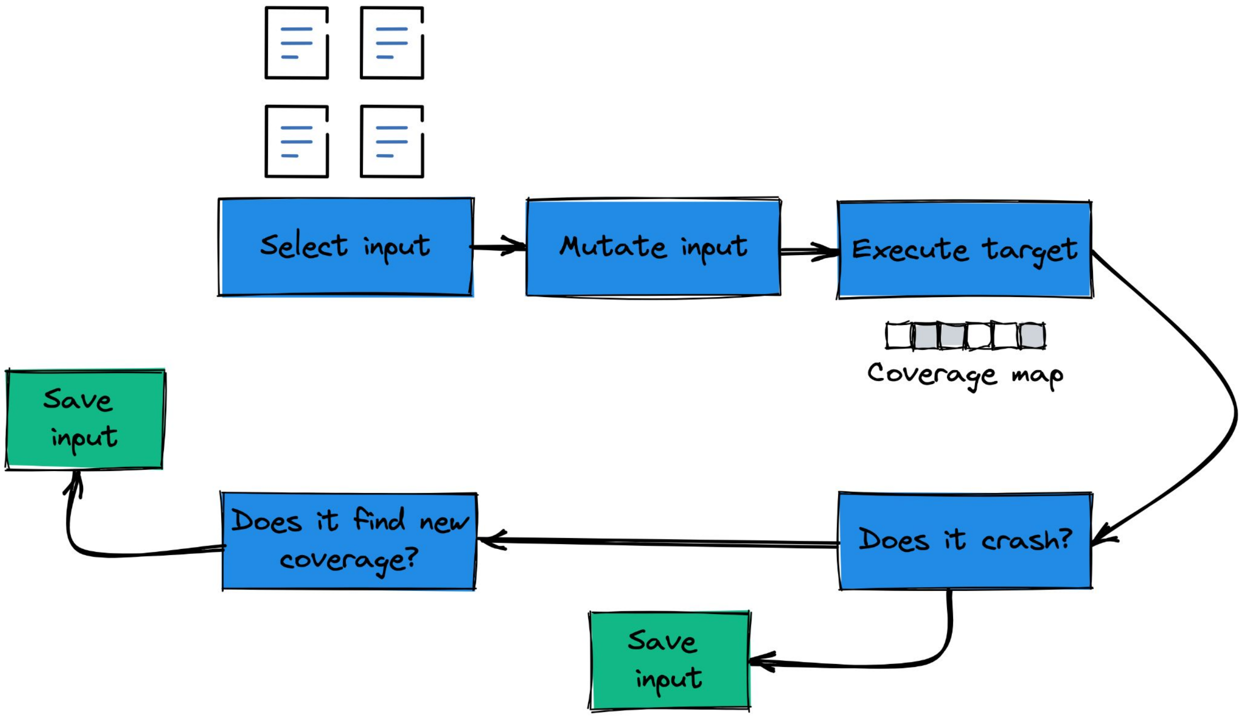


Mutational coverage-guided fuzzer
aka
greybox fuzzer

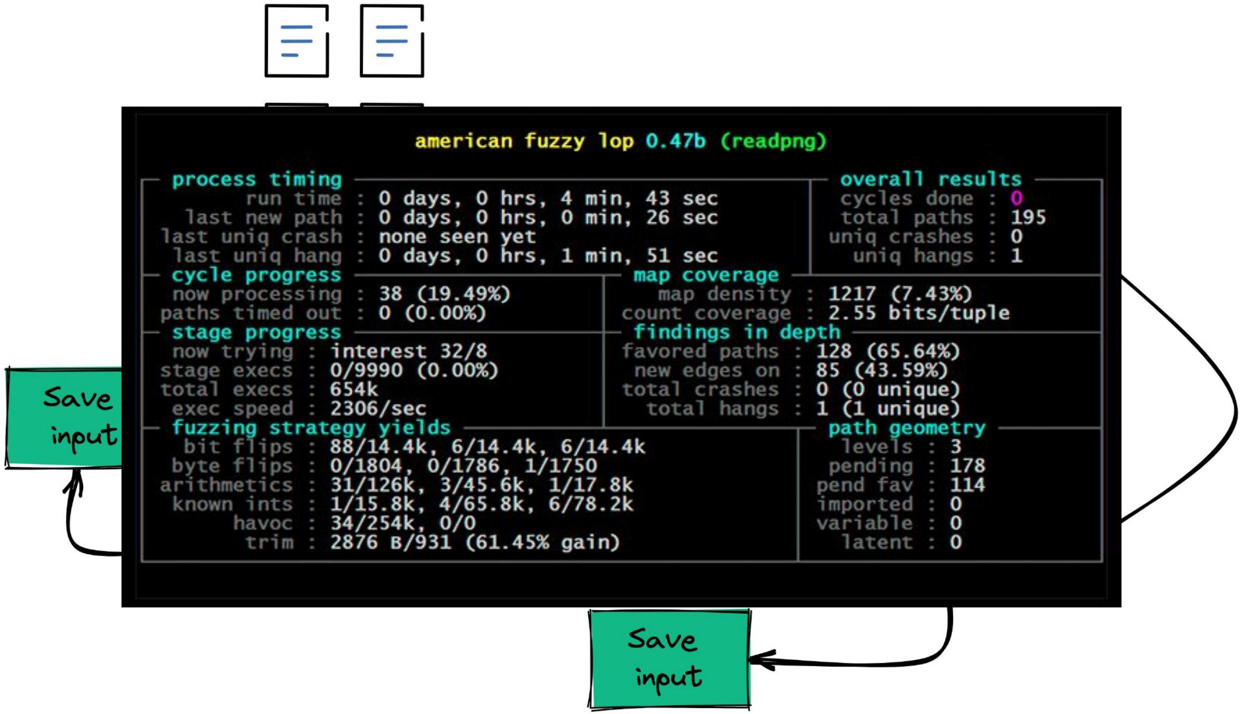
Blackbox fuzzing



Greybox fuzzing



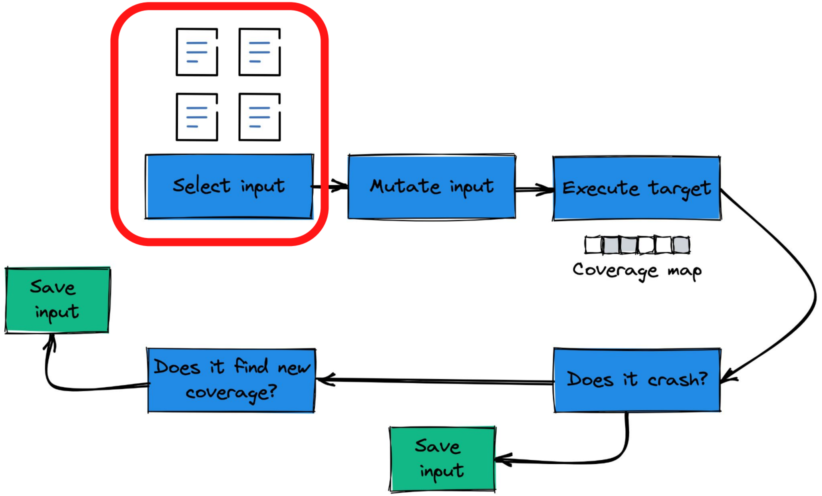
Greybox fuzzing



Greybox fuzzing

Select input

- Rather than generating random data, mutate existing data

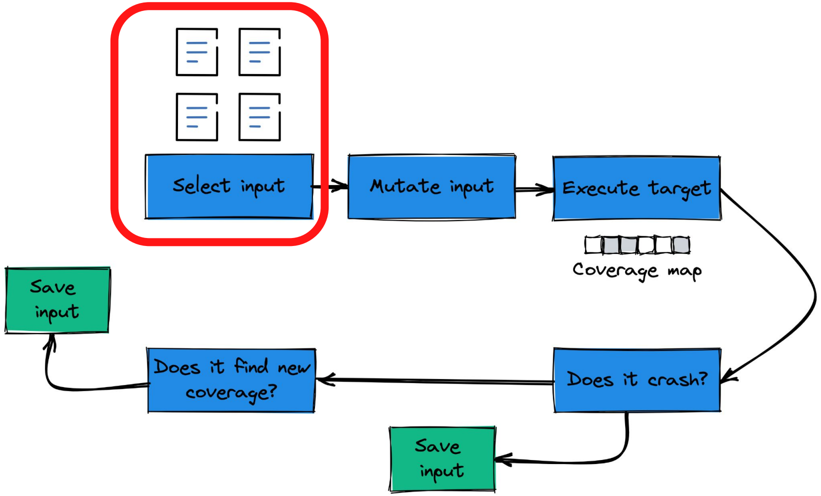


Greybox fuzzing

Select input

- Rather than generating random data, mutate existing data

Where do these initial inputs come from?



Seed selection

- In academic evaluations: “empty seed” common

- In practice: large corpora

Seed selection

- In academic evaluations: “empty seed” common

- In practice: large corpora

Which is better?

Seed selection

Optimizing Seed Selection for Fuzzing

Alexandre Rebert[‡,$] alex@forallsecure.com
Sang Kil (sangkilc@cı
David Warren[†] dwarren@cert.org
G gg@c
[‡] Carnegie Mellon University
[†] Software I

Abstract

Randomly mutating well-formed program inputs
ply *fuzzing*, is a highly effective and widely used s
to find bugs in software. Other than showing fuzz
bugs, there has been little systematic effort in und
ing the science of how to fuzz properly. In this
we focus on how to mathematically formulate and
about one critical aspect in fuzzing: how best to pi
files to maximize the total number of bugs found
a fuzz campaign. We design and evaluate six di
algorithms using over 650 CPU days on Amazo
tic Compute Cloud (EC2) to provide ground tru
Overall, we find 240 bugs in 8 applications and sh
the choice of algorithm can greatly increase the r
of bugs found. We also show that current seed se
strategies as found in Peach may fare no better tha
ing seeds at random. We make our data set an
publicly available.

1 Introduction

Software bugs are expensive. A single softwa
is enough to take down spacecrafts [2], make r
centrifuges spin out of control [17], or recall 100,(
faulty cars resulting in billions of dollars in dama
In 2012, the software security market was estim
$19.2 billion [12], and recent forecasts predict a
increase in the future despite a sequestering econon
The need for finding and fixing bugs in software
they are exploited by attackers has led to the devel
of sophisticated automatic software testing tools.

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

Seed Selection for Successful Fuzzing

Adrian Herrera
ANU & DST
Australia

Hendra Gunadi
ANU
Australia

Shane Magrath
DST
Australia

Michael Norrish
CSIRO's Data61 & ANU
Australia

Mathias Payer
EPFL
Switzerland

Antony L. Hosking
ANU & CSIRO's Data61
Australia

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 privacy → Software and application security**.

KEYWORDS

fuzzing, corpus minimization, software testing

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/10.1145/3460319.3464795

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/10.1145/3460319.3464795

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 [39, 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 2× 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.

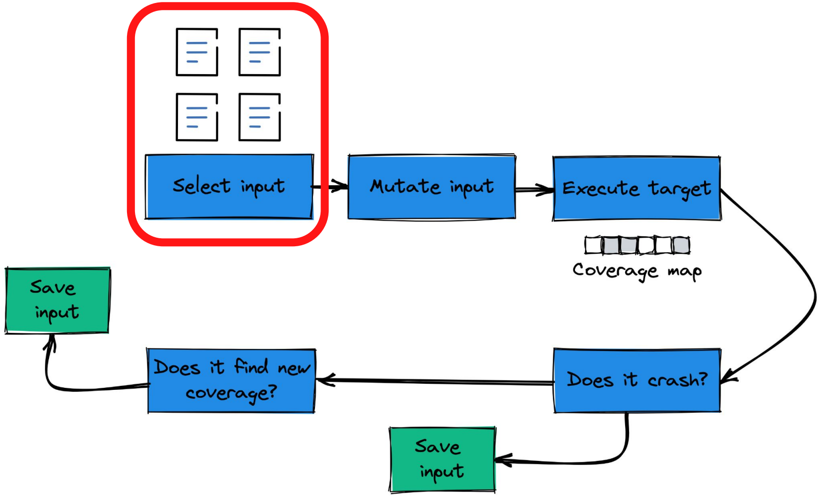
Under these assumptions, simply gathering as many input files as possible is not a reasonable approach for constructing a fuzzing corpus. Conversely, these assumptions also suggest that beginning with the "empty corpus" (e.g., consisting of one zero-length file) may be less than ideal. And yet, as we 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

Greybox fuzzing

Select input

- Rather than generating random data, mutate existing data



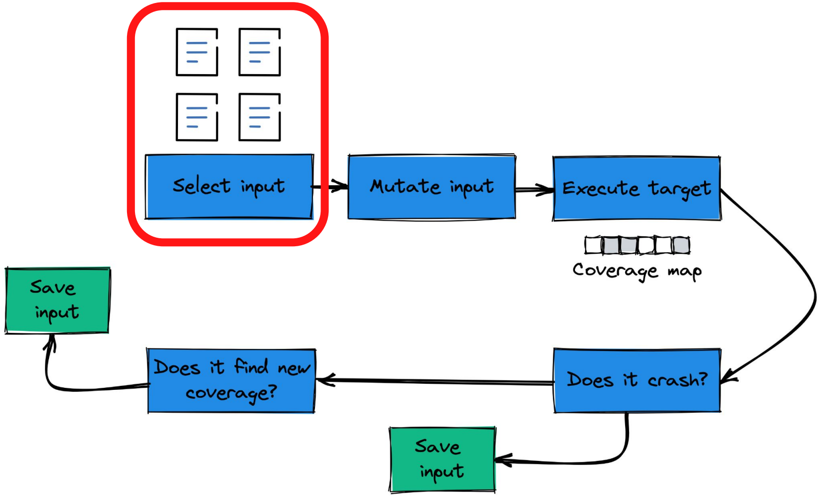
Greybox fuzzing

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

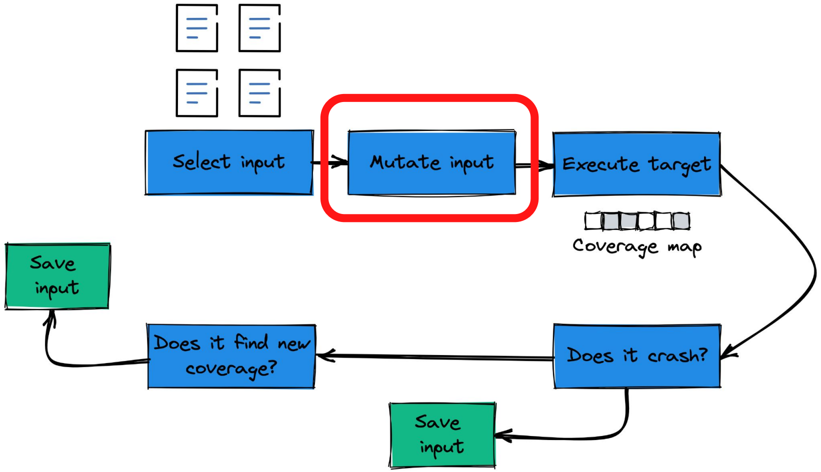
- Power schedule = amount of *energy* assigned to an input
  - Decrease energy each execution
  - When energy = 0, change inputs

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

Greybox fuzzing

Mutate input

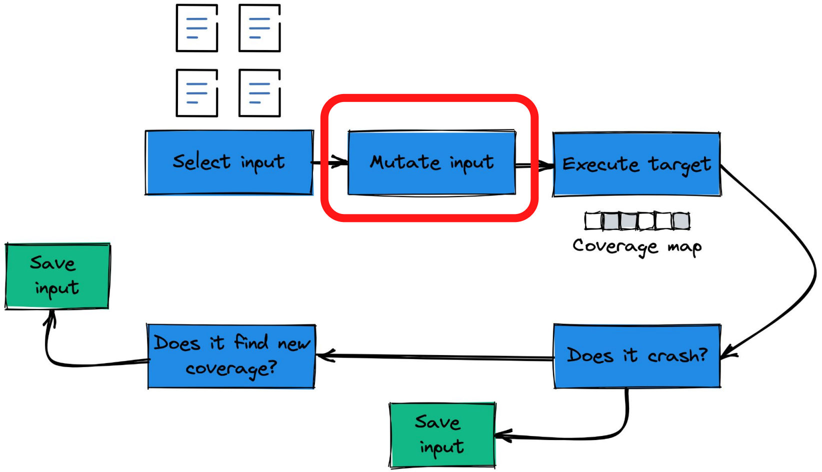
- Mutate enough to explore "interesting" states
- Don't mutate too much, or we'll just error out



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

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

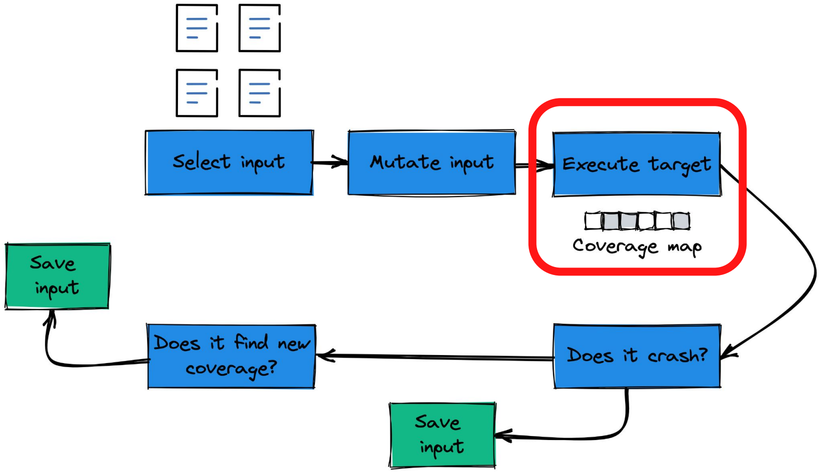
Some fuzzers try to “learn” this input format

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

Coverage map

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

Alternatives:

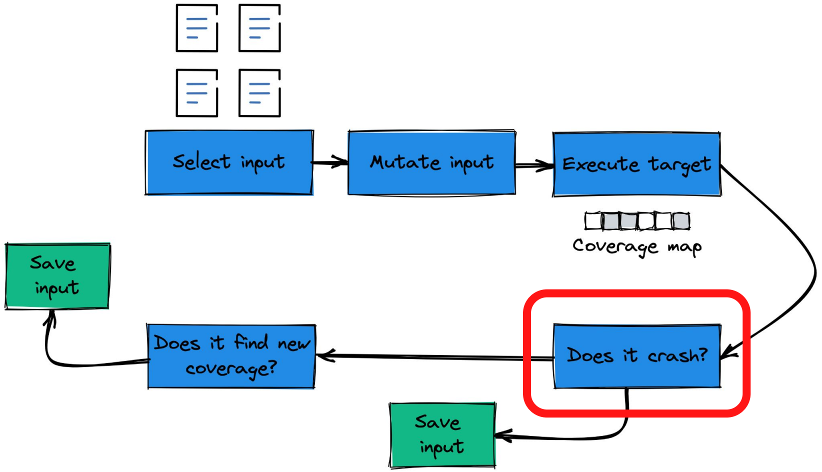
- Context-sensitive edge
- Path
- Data flow

Accuracy vs performance

Greybox fuzzing

Does it crash?

- Classic memory-safety violation
    - `SIGSEGV`

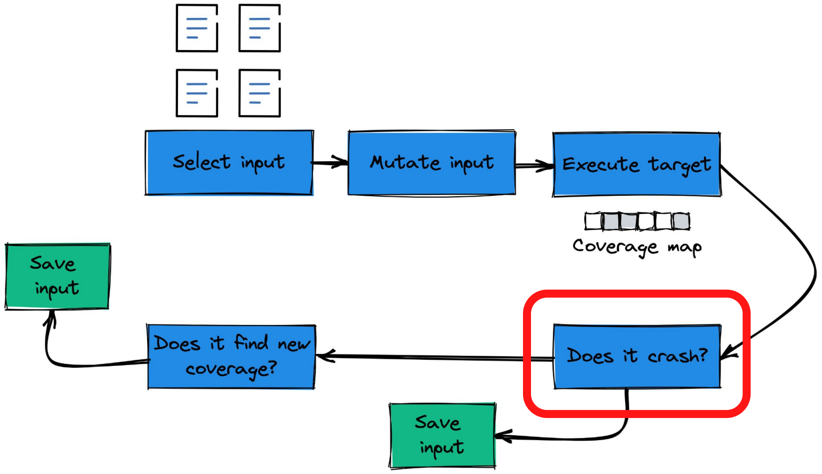


Greybox fuzzing

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

Sushant Dinesh, Purdue University
Nathan Burow, 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@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 – 33.4 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; Typecasting; 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 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 unexpected crashes at runtime. Therefore, recent state-of-the-art detectors leverage disjoint metadata for type information. Upon object allocation, the runtime system records the true type of the object in a disjoint metadata table. This approach indeed does not

CCS'17, Oct. 30–Nov. 3, 2017, Dallas, TX, USA.

ISBN 978-1-4503-4946-8/17/10…$15.00
DOI: http://dx.doi.org/10.1145/3133956.3134062

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

Sushant Dinesh, Purdue University
Nathan Burow, 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@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 – 33.4 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; Typecasting; 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 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 unexpected crashes at runtime. Therefore, recent state-of-the-art detectors leverage disjoint metadata for type information. Upon object allocation, the runtime system records the true type of the object in a disjoint metadata table. This approach indeed does not

CCS'17, Oct. 30–Nov. 3, 2017, Dallas, TX, USA.
© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ISBN 978-1-4503-4946-8/17/10…$15.00
DOI: http://dx.doi.org/10.1145/3133956.3134062

Sanitization

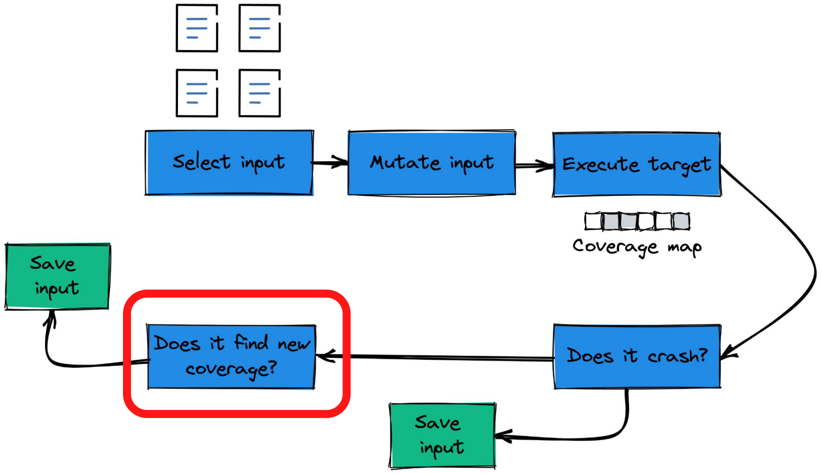
Anything we can encode as an **invariant**

- 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? 🤔

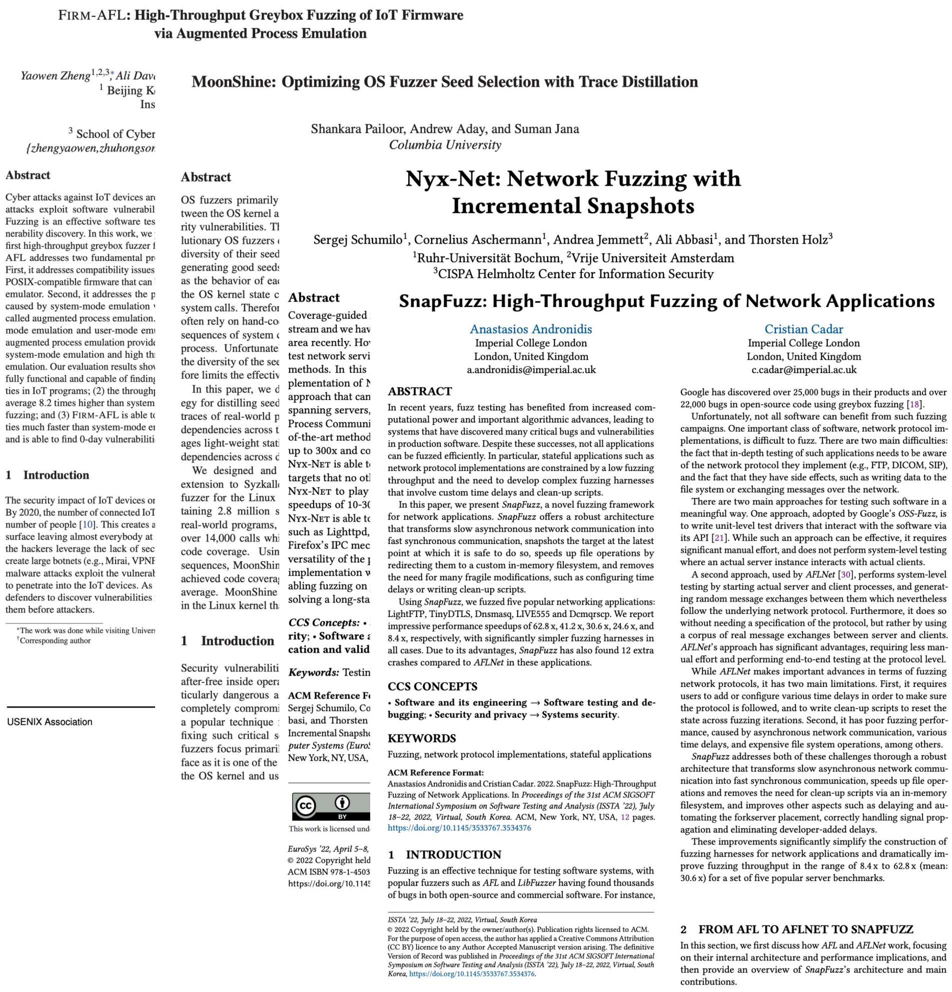
Network apps

Challenges

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

Solutions

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



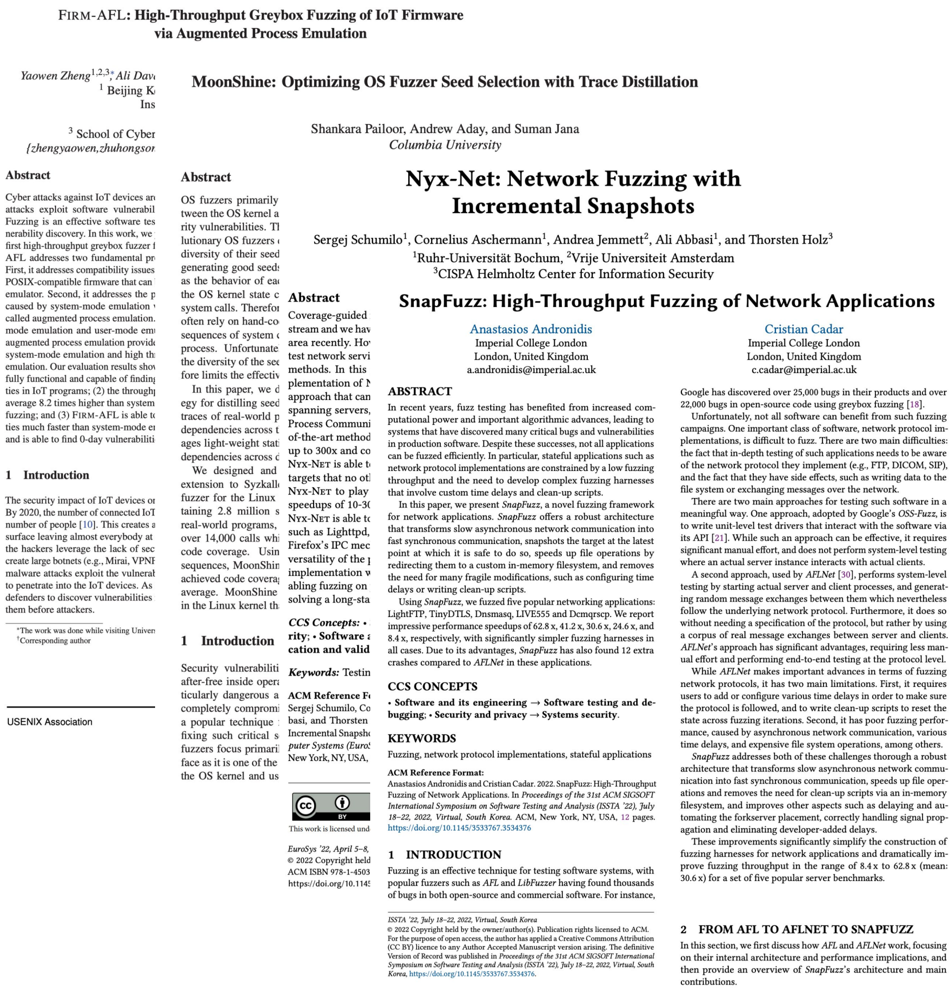
OS kernel

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

MoonShine: Optimizing OS Fuzzer Seed Selection with Trace Distillation

Shankara Pailoor, Andrew Aday, and Suman Jana
Columbia University

Nyx-Net: Network Fuzzing with Incremental Snapshots

Sergej Schumilo[1], Cornelius Aschermann[1], Andrea Jemmett[2], Ali Abbasi[1], and Thorsten Holz[3]
[1]Ruhr-Universität Bochum, [2]Vrije Universiteit Amsterdam
[3]CISPA 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 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, LIVE555 and Dcmqrscp. 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 *LibFuzzer* 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 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 thorough 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 forkserver 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.

ISSTA '22, July 18–22, 2022, Virtual, South Korea
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM. For the purpose of open access, the author has applied a Creative Commons Attribution (CC BY) licence 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.

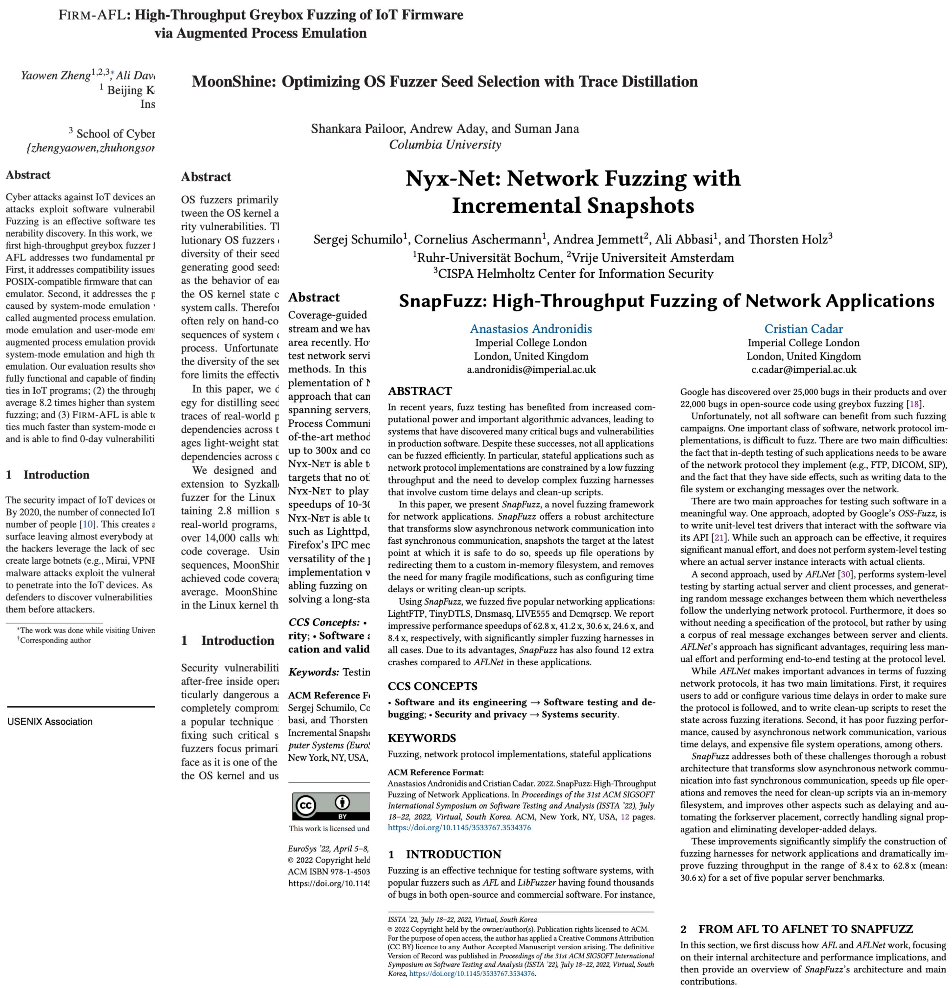
IoT

Challenges

- Measuring coverage
- Performance
- Seeds?

Solutions

- QEMU (slow / incomplete)
- Avatar^2 orchestration



Overcoming “roadblocks”

Program constraints that are hard to meet

Solutions

- Whitebox fuzzing
- Concolic execution
- Rewrite the target 🧐

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 = **conc**rete + symb**olic**

- 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?

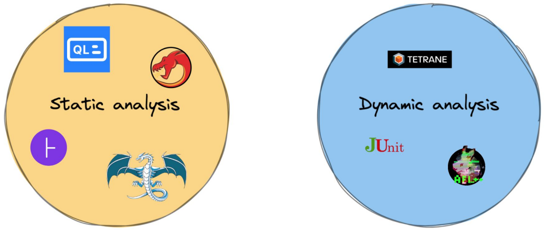
What about…

- Directed fuzzers?
- Determining whe
- Benchmarking fu

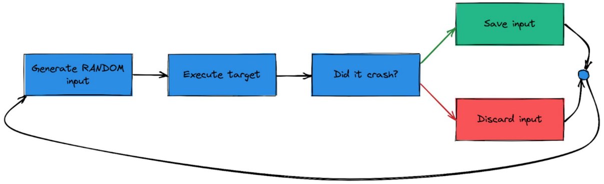
Conclusions

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

What is fuzzing?



Our first fuzzer

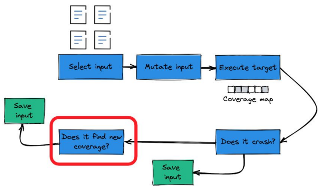


This is a classic **generational blackbox** fuzzer

Greybox fuzzing

Does it find new coverage?

- Save input
- Return to start



Grammar-based fuzzing

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

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?

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**