
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 NINA GOLD DAVID ARNOLD ANNIE HARDINGE
CHRIS DICKENS MARCUS ROWLAND JESS HALL RONALDO VASCONCELLOS NATASCHA WHARTON
EDGAR WRIGHT & SIMON PEGG NIRA PARK TIM BEVAN ERIC FELLNER EDGAR WRIGHT
R
VIOLENT CONTENT INCLUDING SOME GRAPHIC IMAGES, AND LANGUAGE
ROGUE
www.jointhefuzz.com


A NEW COMEDY
FROM THE GUYS THAT CREATED
SHAUN OF THE DEAD
POLICE
SIMON
www.jointhefuzz.com


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

whoami

- PhD student @ ANU
- Team lead @ Defence Science and Technology Group
- Interests in fuzzing, (binary) program analysis

whoami

- PhD student @ ANU
- Team lead @ Defence Science and Technology Group
- Interests in **fuzzing**, (binary) program analysis

Outline

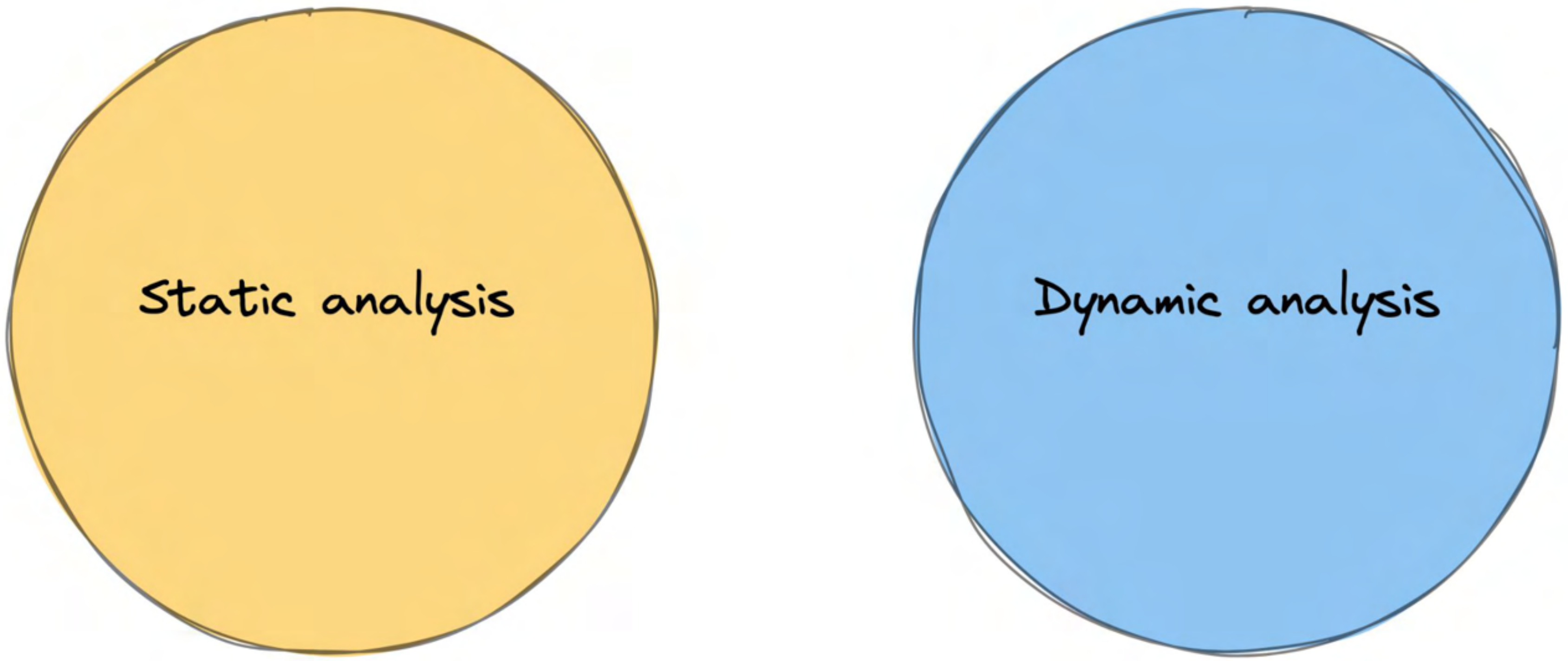
1. What is fuzzing?

2. Shades of fuzzers
    - Black, grey, white

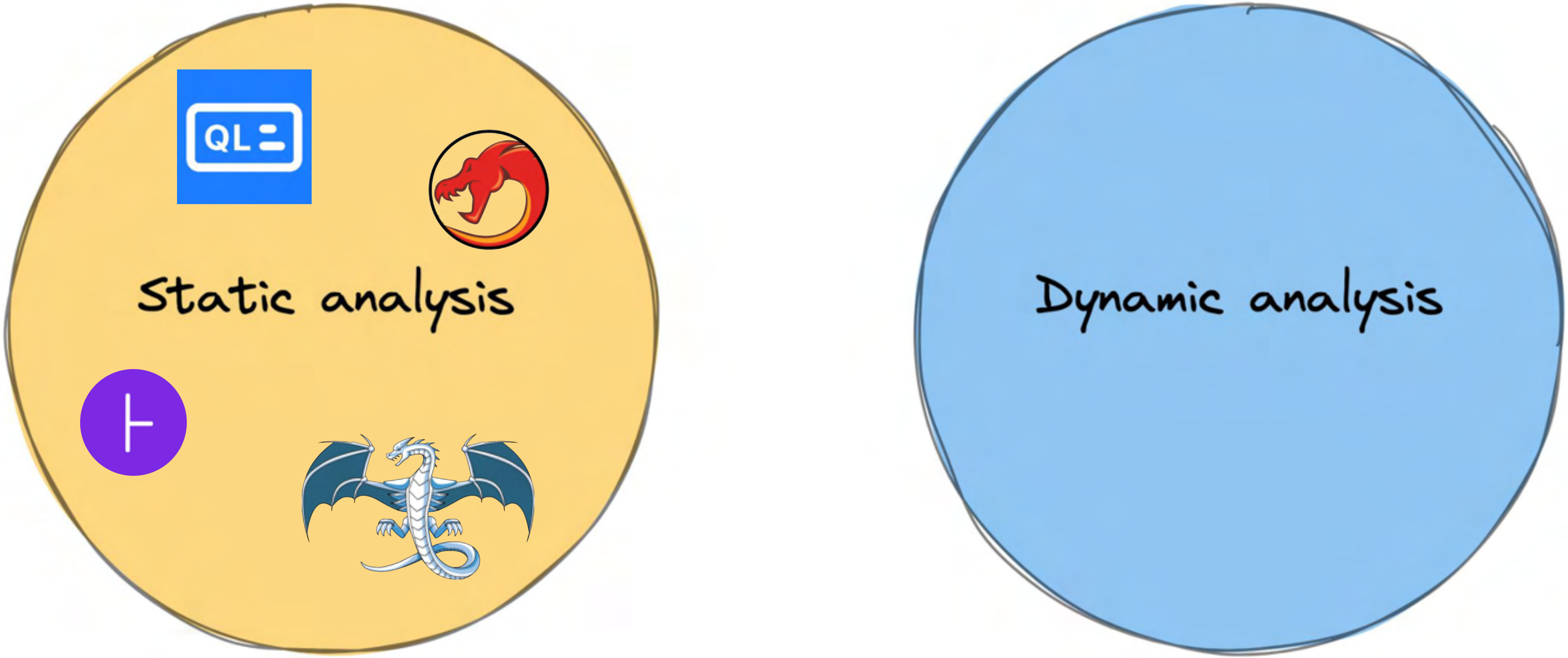
3. Fuzzing research state-of-the-art

4. Future directions

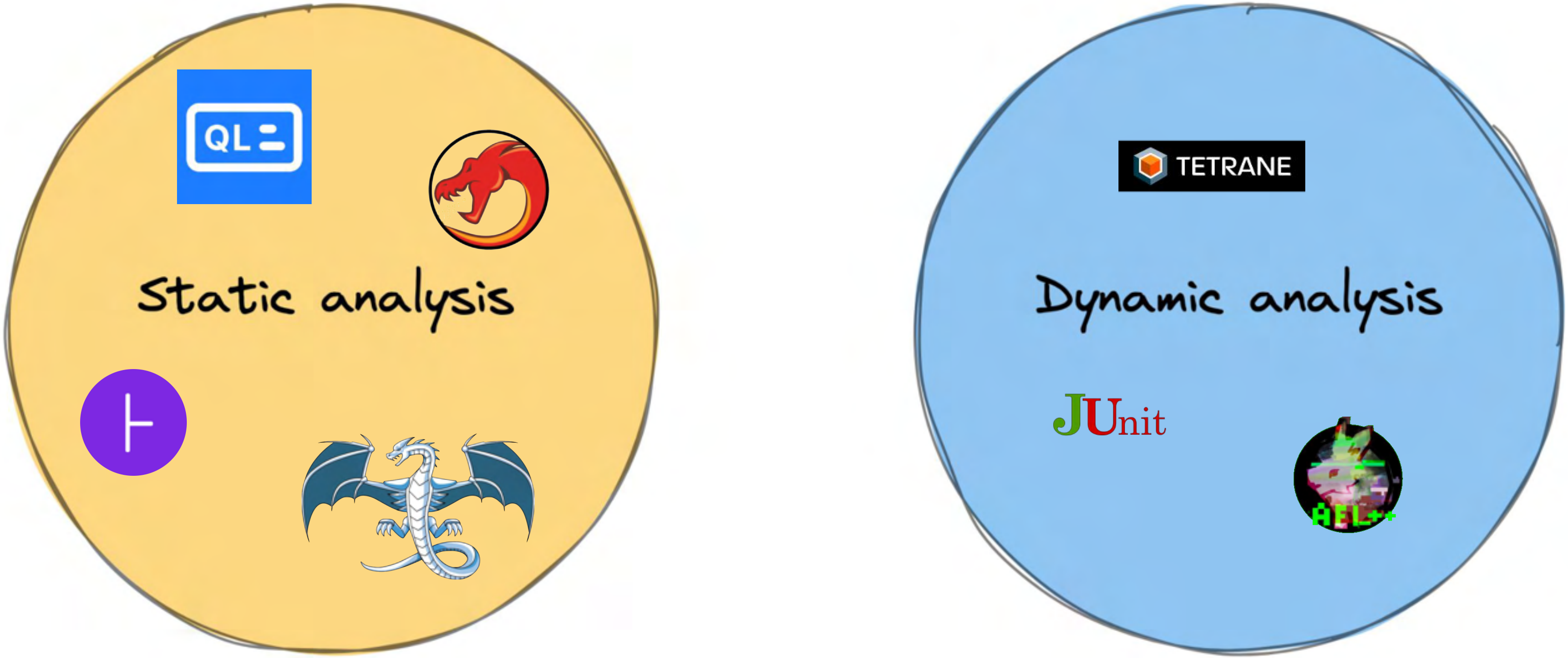
What is fuzzing?



What is fuzzing?



What is fuzzing?



What is fuzzing?

Pros

- No false positives
- Produces PoC
- Scalable

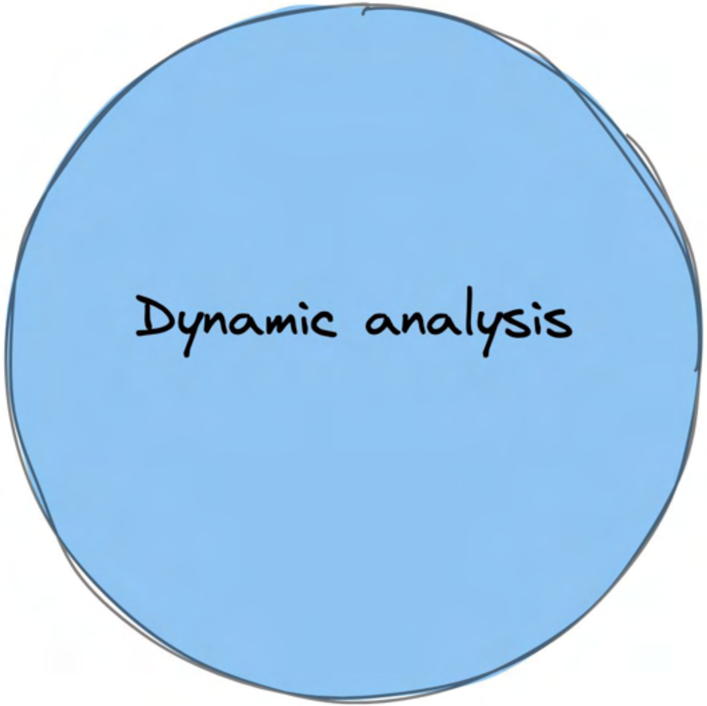
What is fuzzing?

Pros

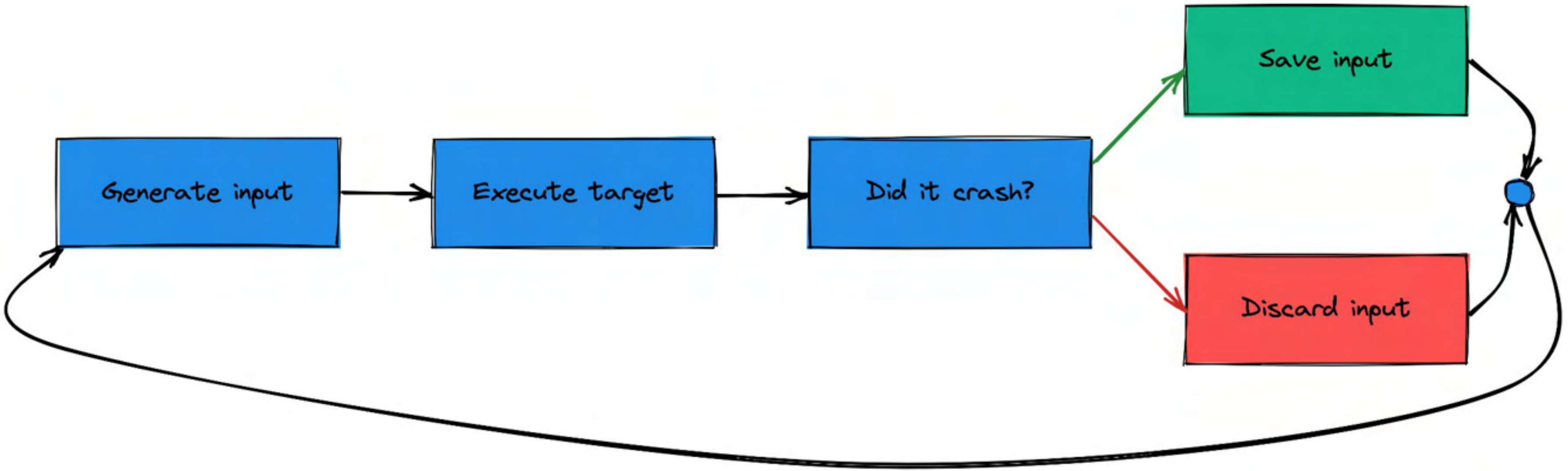
- No false positives
- Produces PoC
- Scalable

Cons

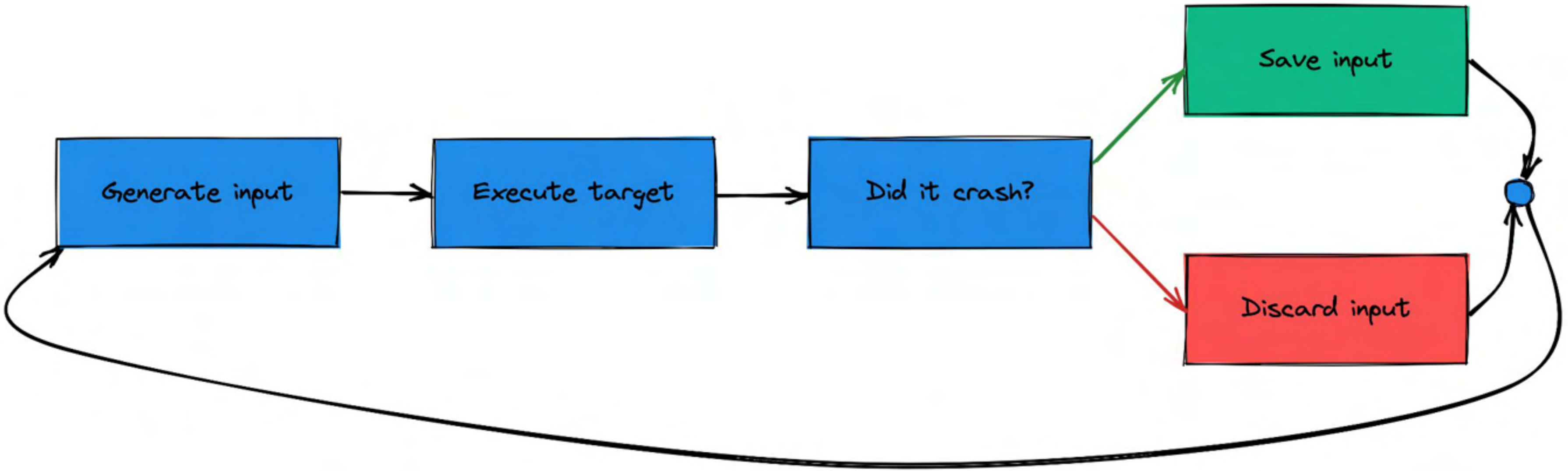
- Incomplete
- Requires buildable target
- Scalability



Our first fuzzer

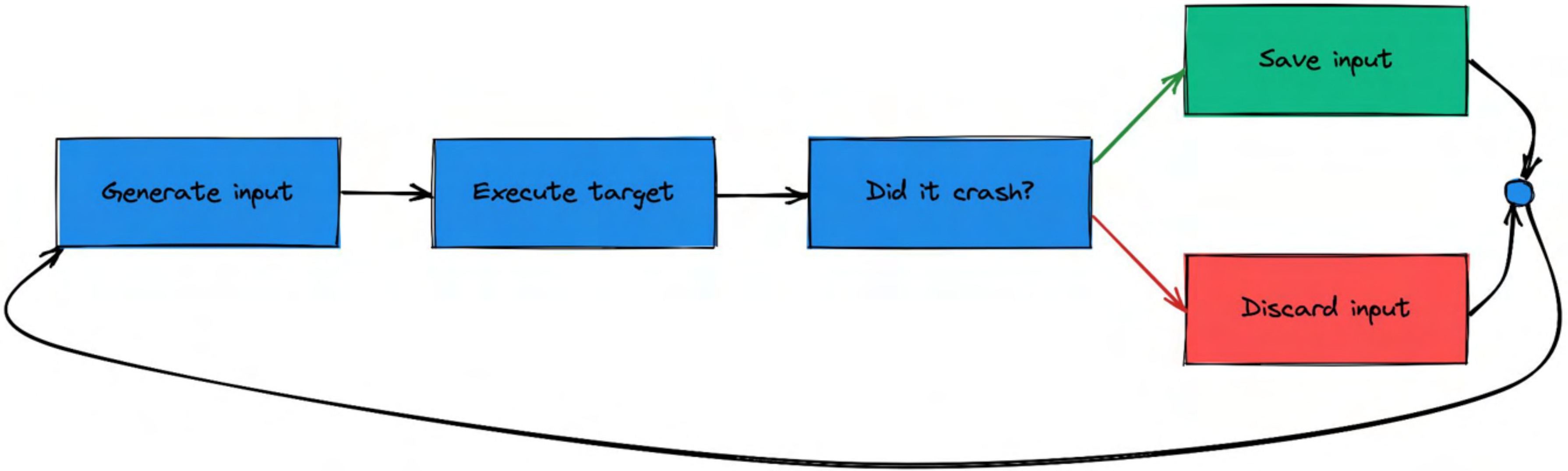


Our first fuzzer



How is this different to dynamic testing?

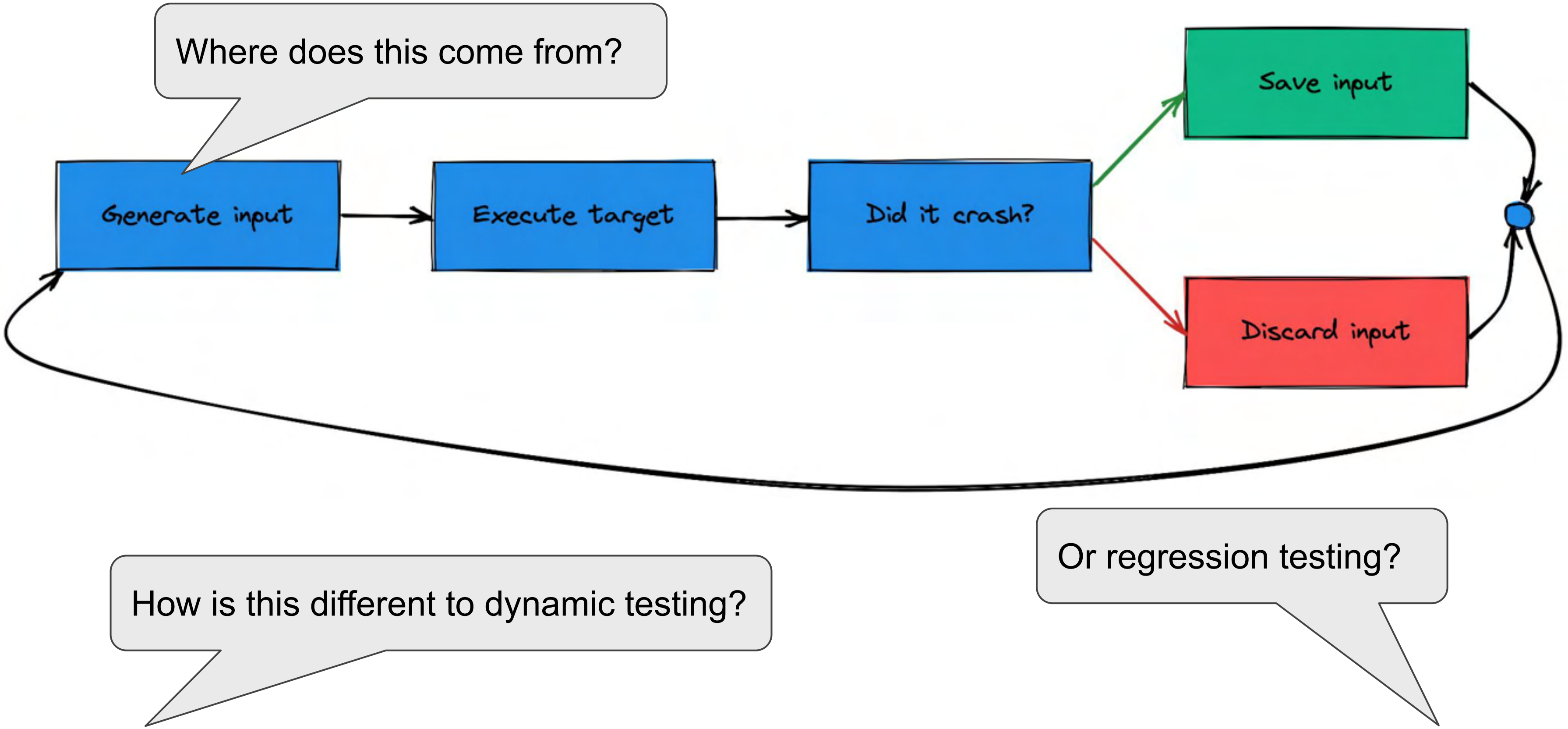
Our first fuzzer



How is this different to dynamic testing?

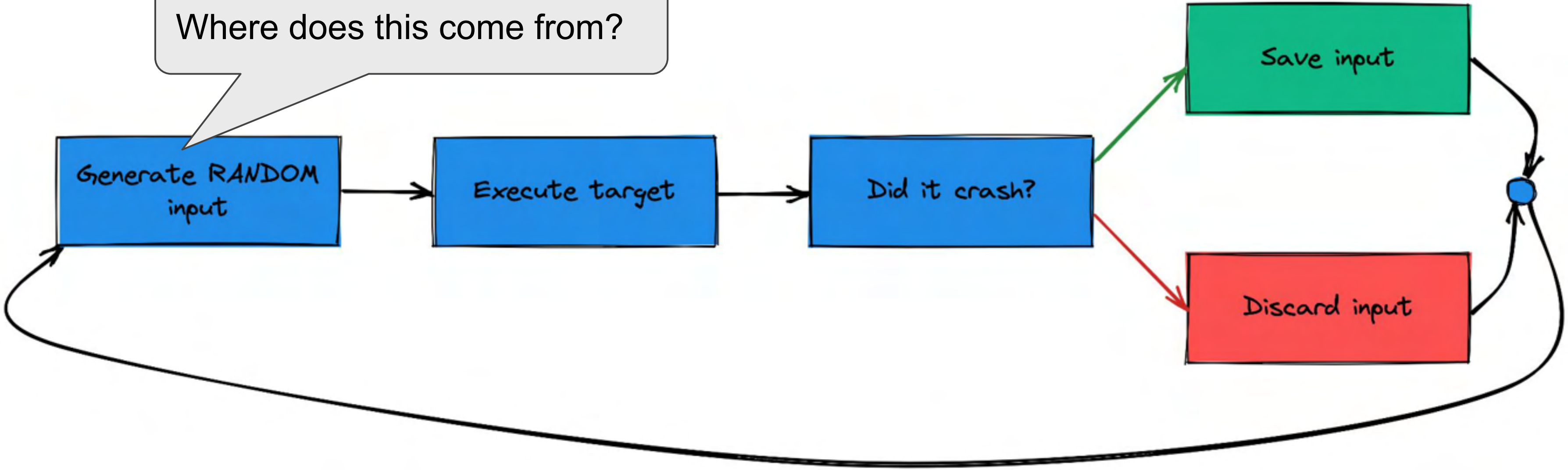
Or regression testing?

Our first fuzzer



Our first fuzzer

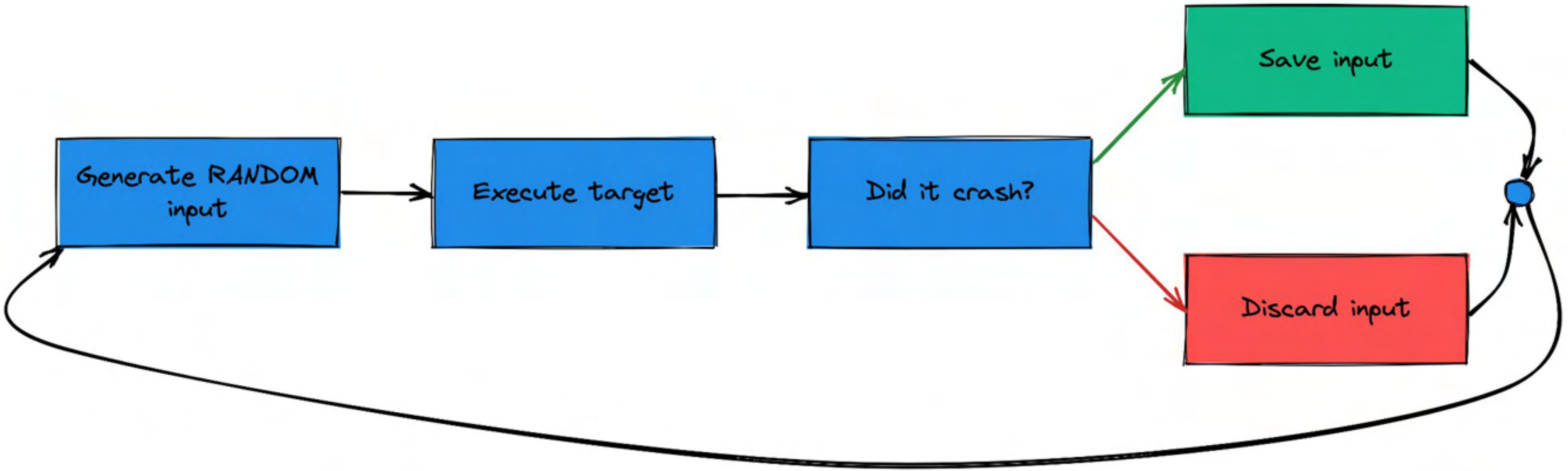
Where does this come from?



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

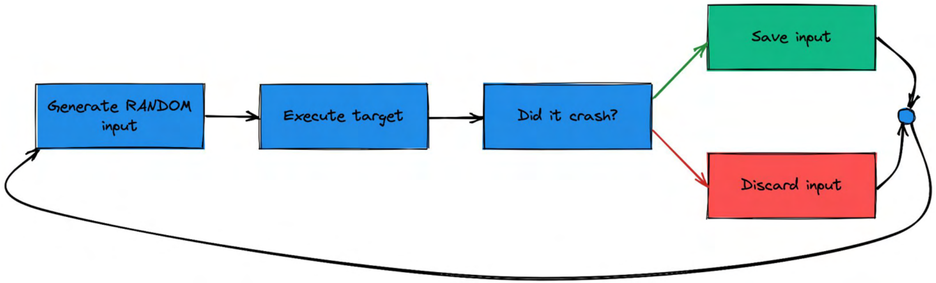
Blackbox fuzzers

- Radamsa
  - General-purpose file mutator
- Domato 🍅
  - Google ProjectZero DOM fuzzer
- Peach 🍑
  - Web API fuzzing (REST, SOAP, GraphQL)

Blackbox fuzzing

Pros

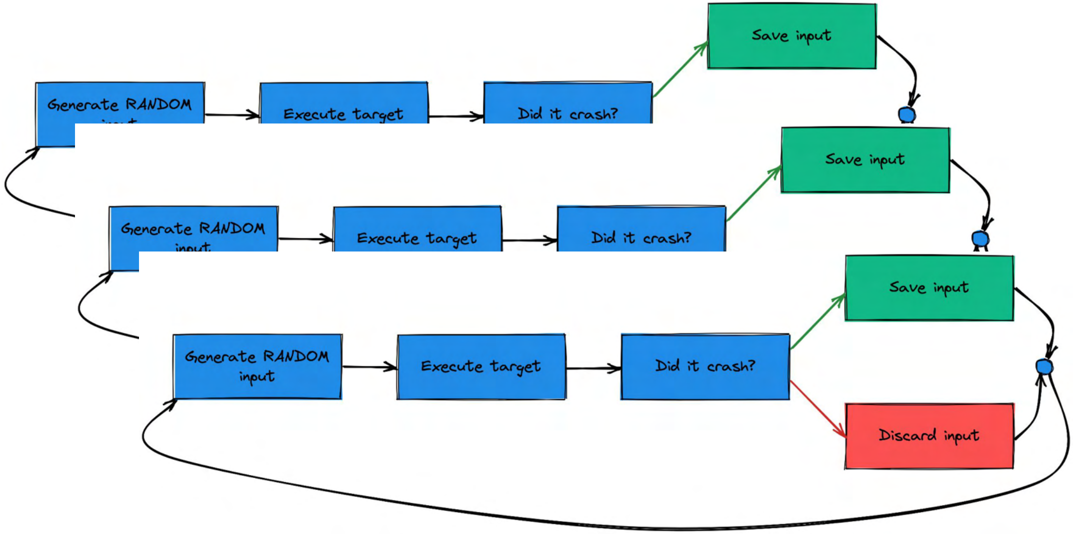
- Simple
- Fast
- Embarrassingly parallel



Blackbox fuzzing

Pros

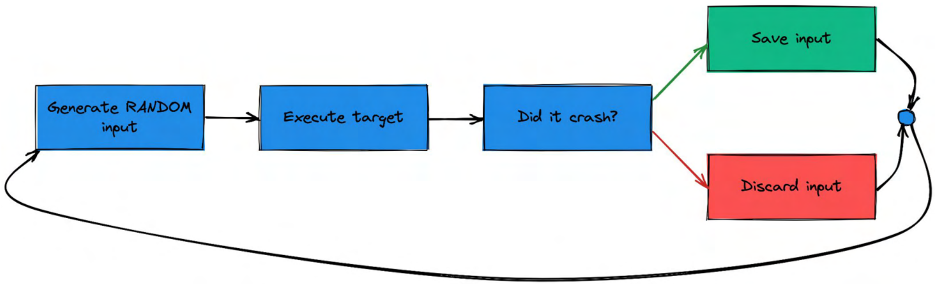
- Simple
- Fast
- Embarrassingly parallel



Blackbox fuzzing

Cons

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

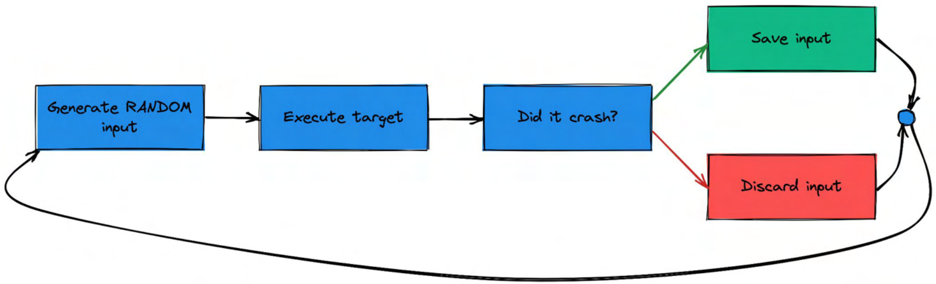


Can we do better?

Blackbox fuzzing

Cons

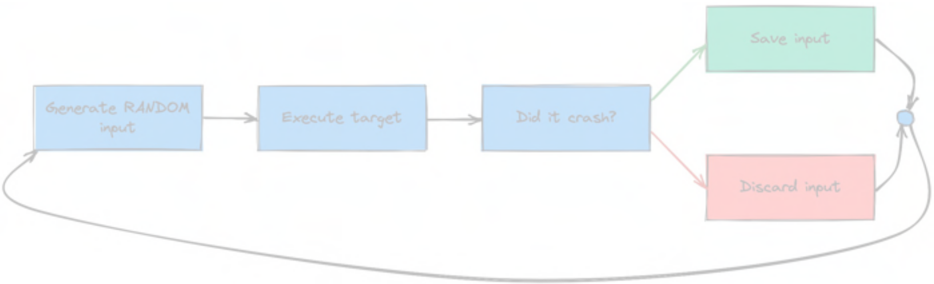
- Generate mostly rubbish
- No notion of “progress”
- Only detect `SIGSEGV`



Blackbox fuzzing

Cons

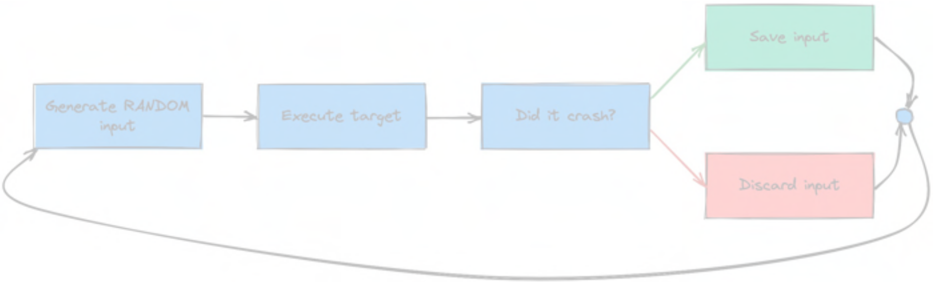
- Generate mostly rubbish
    - ~~Generate~~ **mutate**
- No notion of “progress”
- Only detect `SIGSEGV`



Blackbox fuzzing

Cons

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

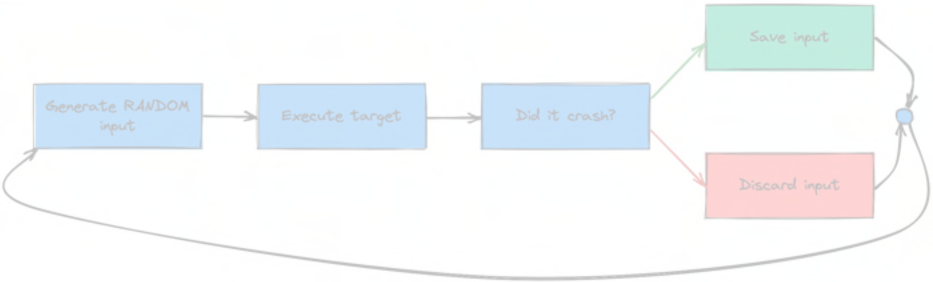


- No notion of “progress”
  - Add a **feedback loop**

- Only detect `SIGSEGV`

Blackbox fuzzing

Cons

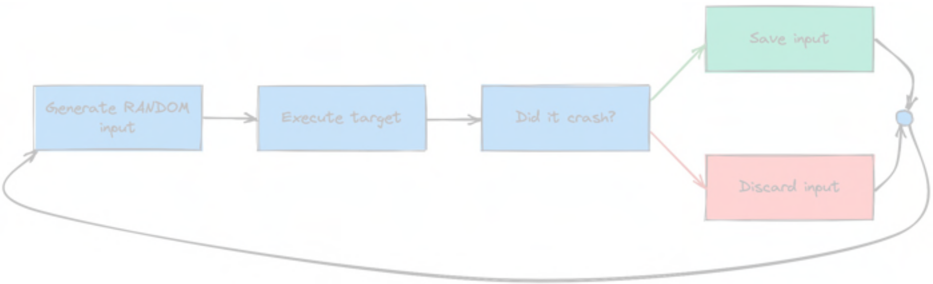


- Generate mostly rubbish
    - ~~Generate~~ **mutate**
- No notion of "progress"
    - Add a **feedback loop**
- Only detect `SIGSEGV`
    - Add a **sanitizer**

Blackbox fuzzing

Cons

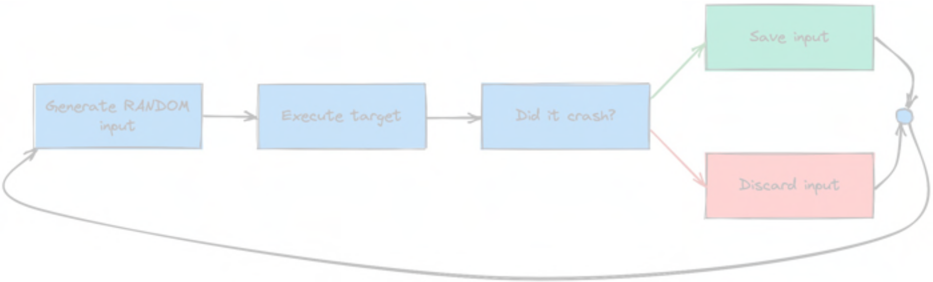
- Generate mostly rubbish
  - ~~Generate~~ **mutate**
- No notion of “progress”
  - Add a **feedback loop**
- Only detect `SIGSEGV`
  - Add a **sanitizer**



Blackbox fuzzing

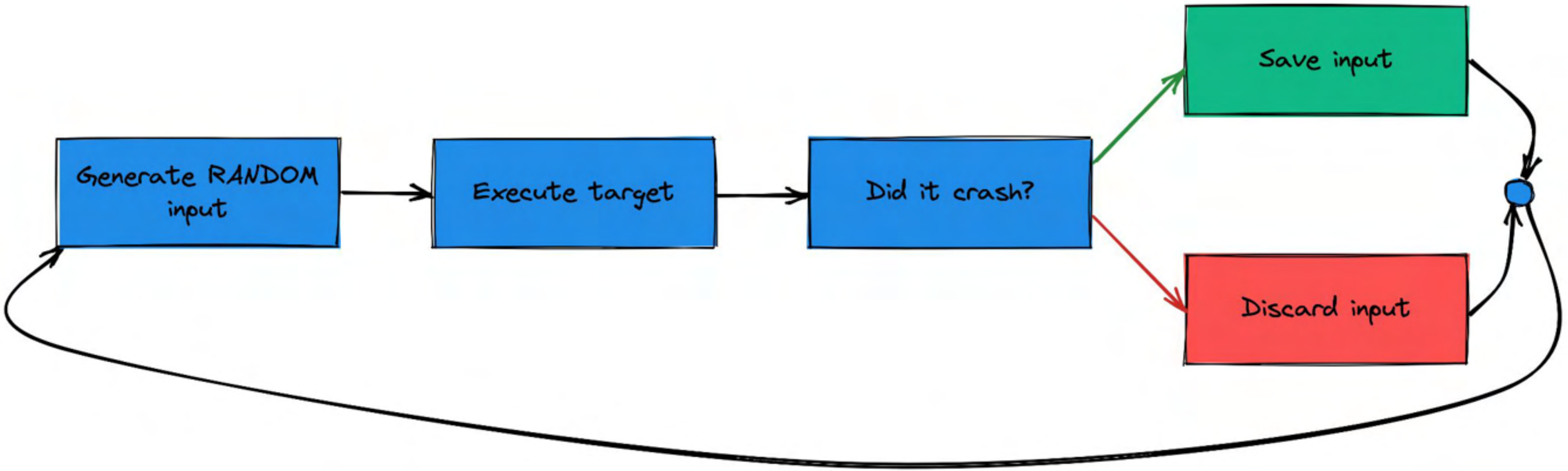
Cons

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

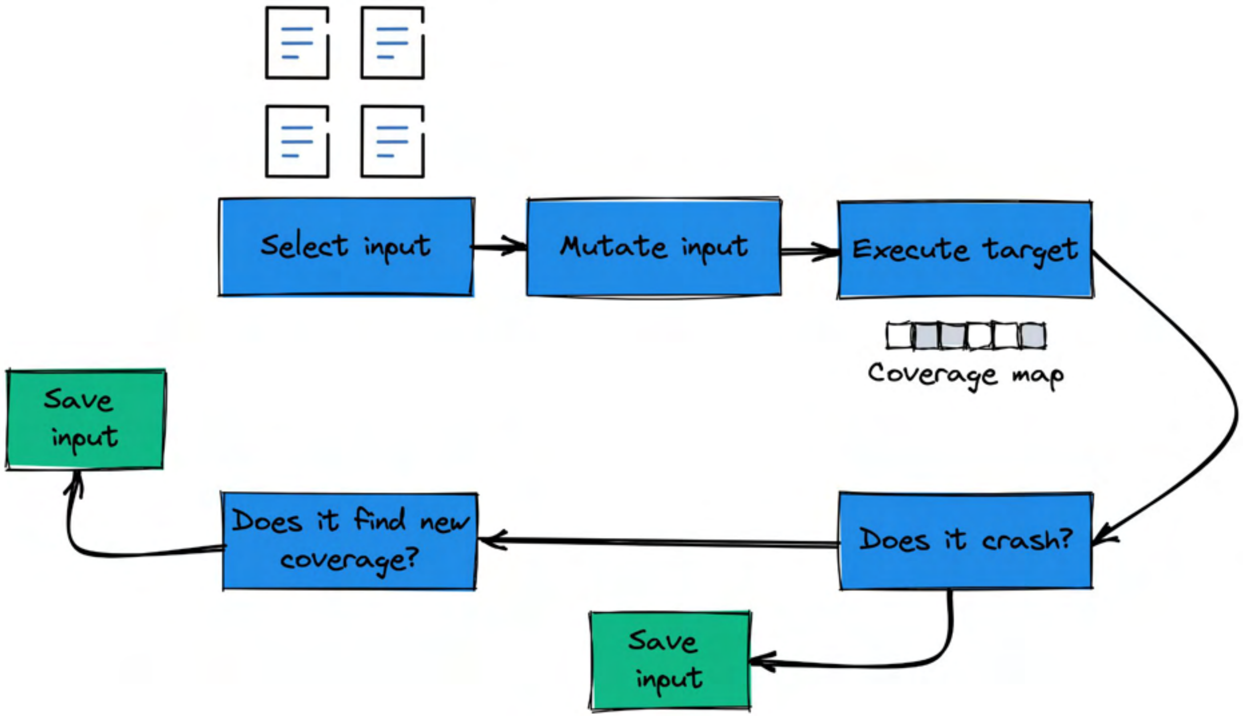


Mutational coverage-guided fuzzer
aka
greybox fuzzer

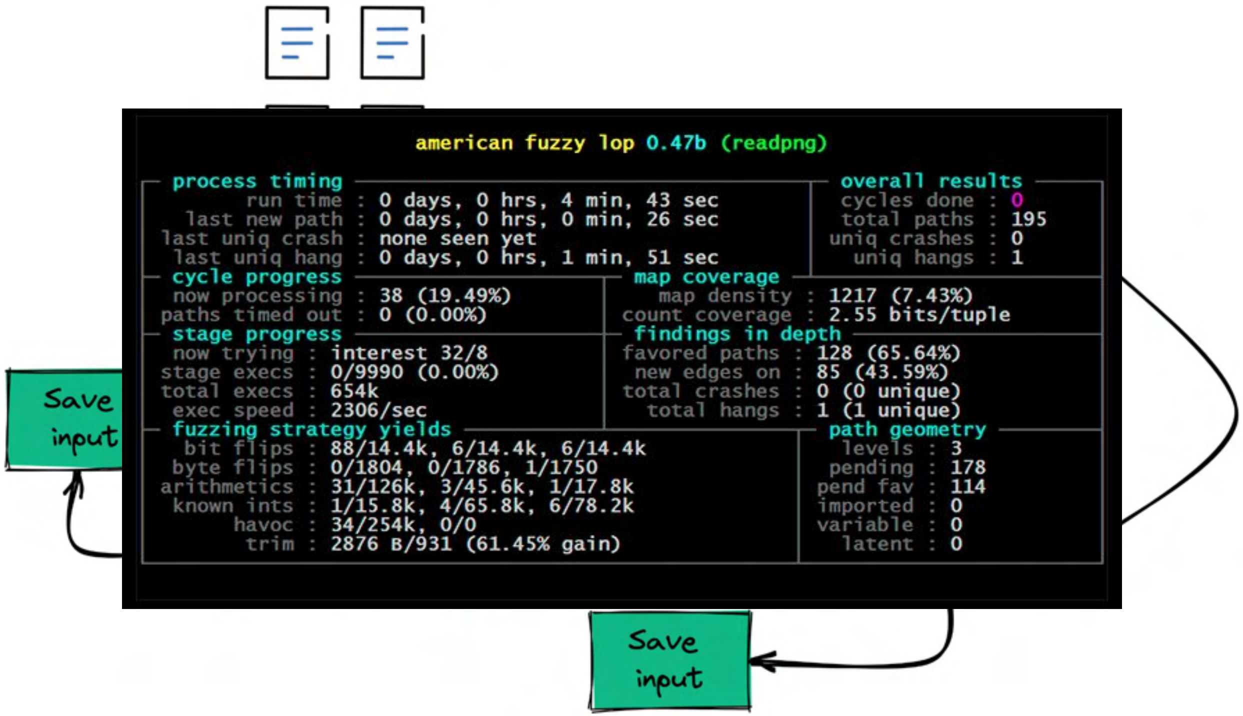
Blackbox fuzzing



Greybox fuzzing



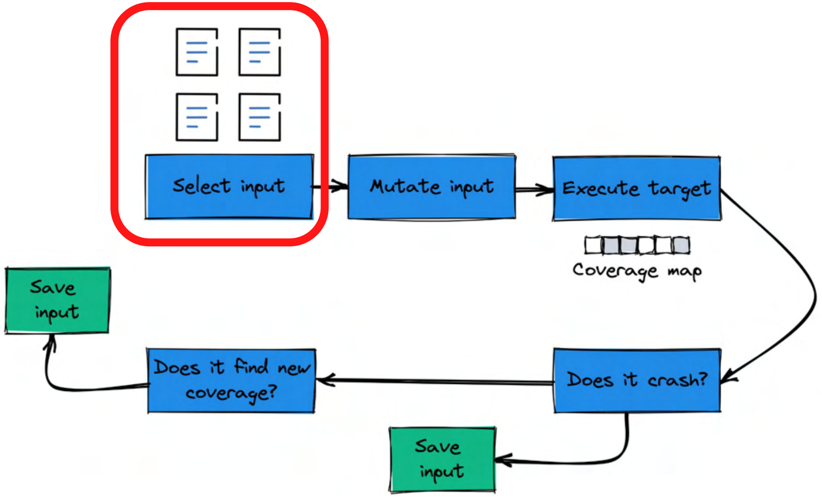
Greybox fuzzing



Greybox fuzzing

Select input

- Rather than generating random data, mutate existing data

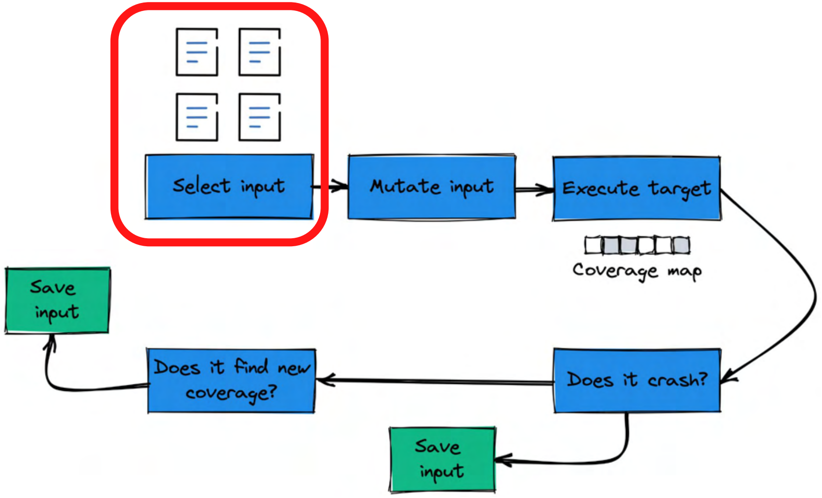


Greybox fuzzing

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@c…

David Warren[†] dwarren@cert.org

G… gg@c…

[‡] Carnegie Mellon University

[†] Software …

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 unde… 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 … 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 … 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 … part of the overall quality checking process emplo… Adobe [28], Microsoft [14], and Google [27], as …

USENIX Association

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

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.

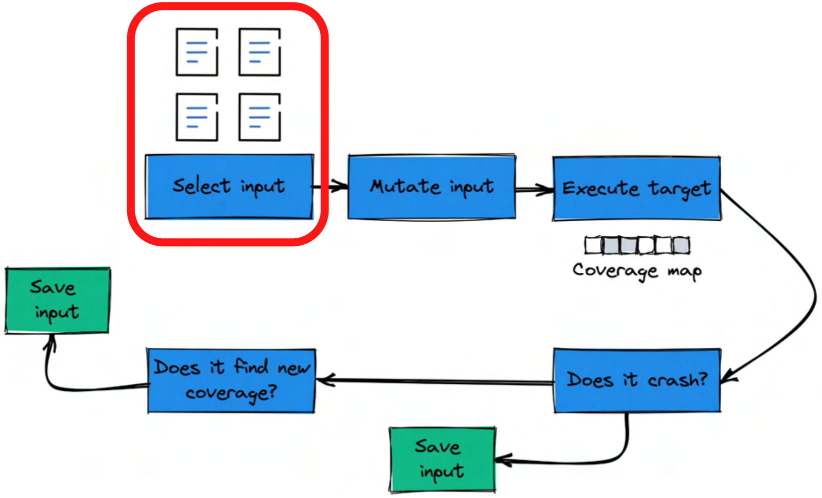
Under these assumptions, simply gathering as many input files as possible is not a reasonable approach for constructing a fuzzing corpus. Conversely, these assumptions also suggest that beginning with the "empty corpus" (e.g., consisting of one zero-length file) may be less than ideal. And yet, as we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

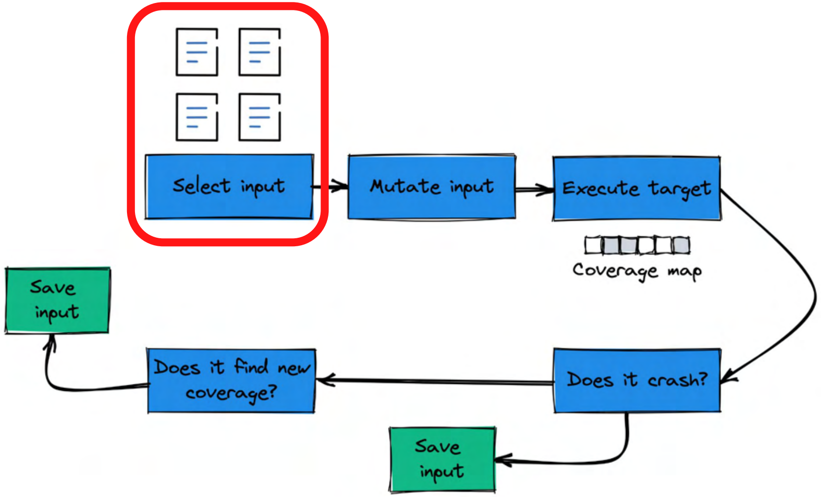
- Rather than generating random data, mutate existing data



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

Coverage-Based Greybox Fuzzing as Markov Chain

Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury

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

CEREBRO: Context-Aware Adaptive Fuzzing for Effective Vulnerability Detection

Yuekang Li, University of Science and Technology of China, China; Nanyang Technological University, Singapore
Yinxing Xue*, University of Science and Technology of China, China
Hongxu Chen, Nanyang Technological University, Singapore
Xiuheng Wu, Nanyang Technological University, Singapore
Cen Zhang, Nanyang Technological University, Singapore
Xiaofei Xie, Nanyang Technological University, Singapore
Haijun Wang, Nanyang Technological University, Singapore
Yang Liu, Nanyang Technological University, Singapore; Zhejiang Sci-Tech University, China

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 challenges. 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 introduce 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.

KEYWORDS

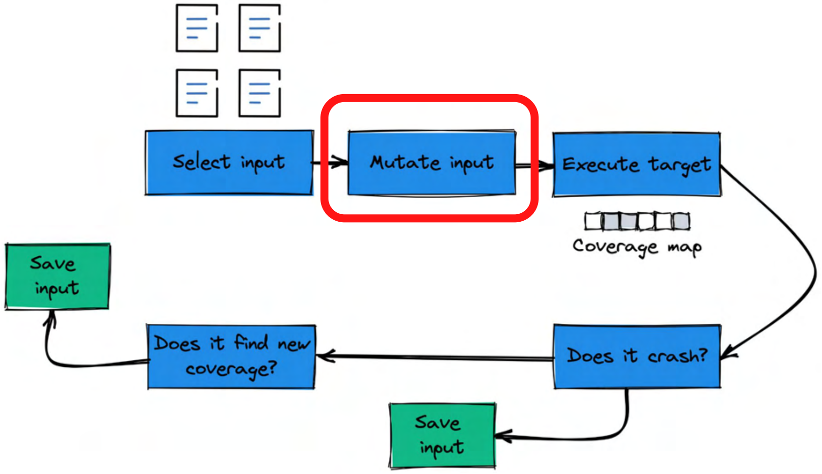
Fuzz Testing; Software Vulnerability

Greybox fuzzing

Mutate input

- Mutate enough to explore "interesting" states

- Don't mutate too much, or we'll just error out



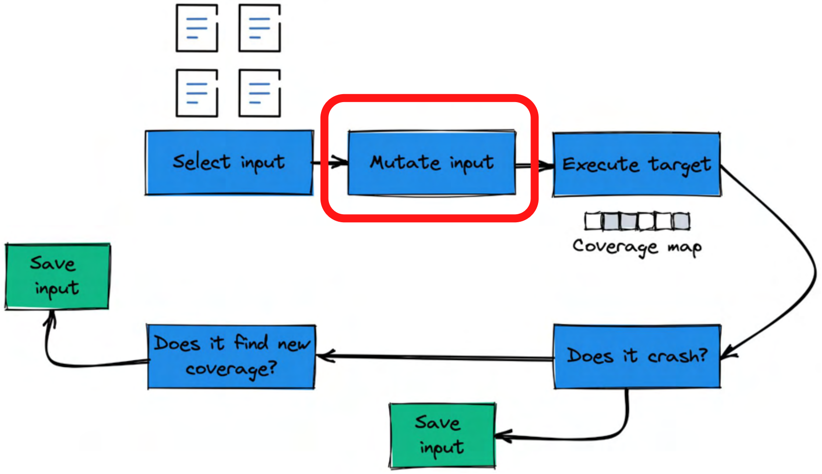
Greybox fuzzing

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

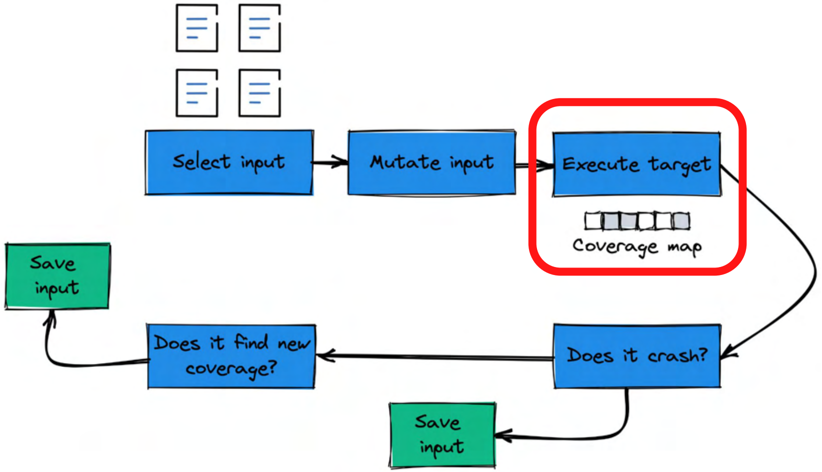
Some fuzzers try to “learn” this input format

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:

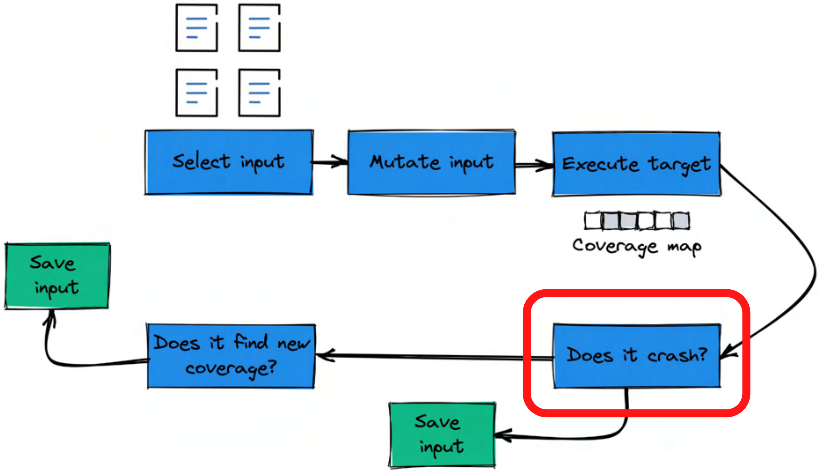
- Context-sensitive edge
- Path
- Data flow

Accuracy vs performance

Greybox fuzzing

Does it crash?

- Classic memory-safety violation
  - `SIGSEGV`

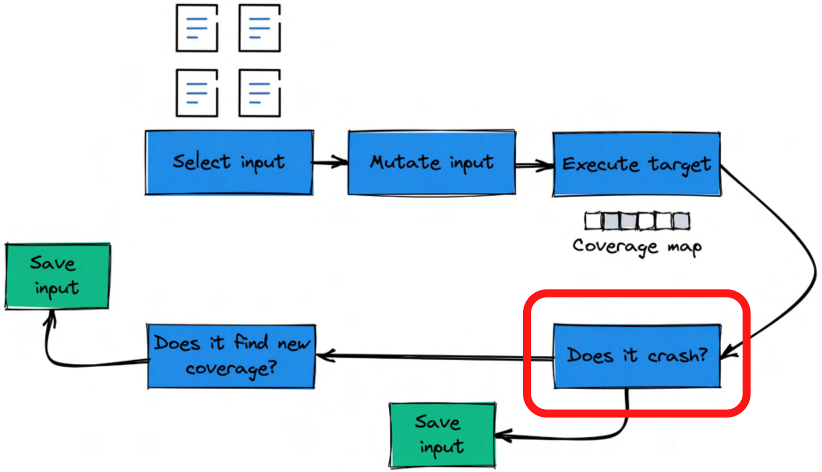


Greybox fuzzing

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

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.

Sanitization

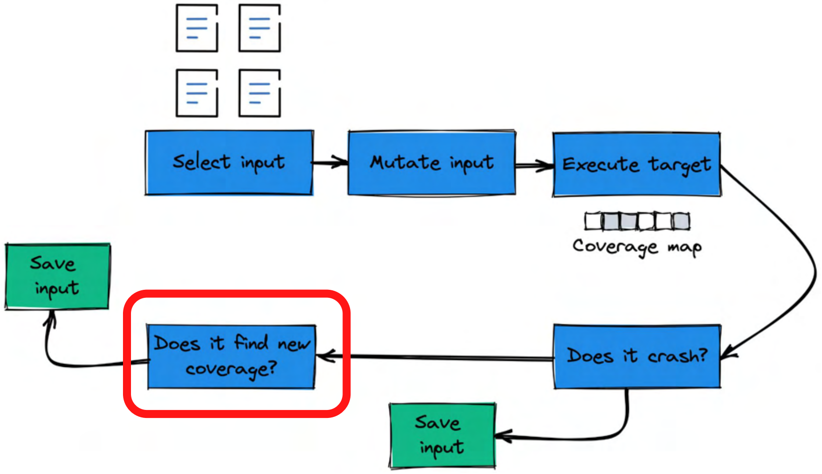
Anything we can encode as an **invariant**

- 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, …
- Writing a harness
  - The fuzzer has to start somewhere
- Overcoming “roadblocks”
  - E.g., complex conditionals

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

IoT

Challenges

- Measuring coverage
- Performance
- Seeds?

Solutions

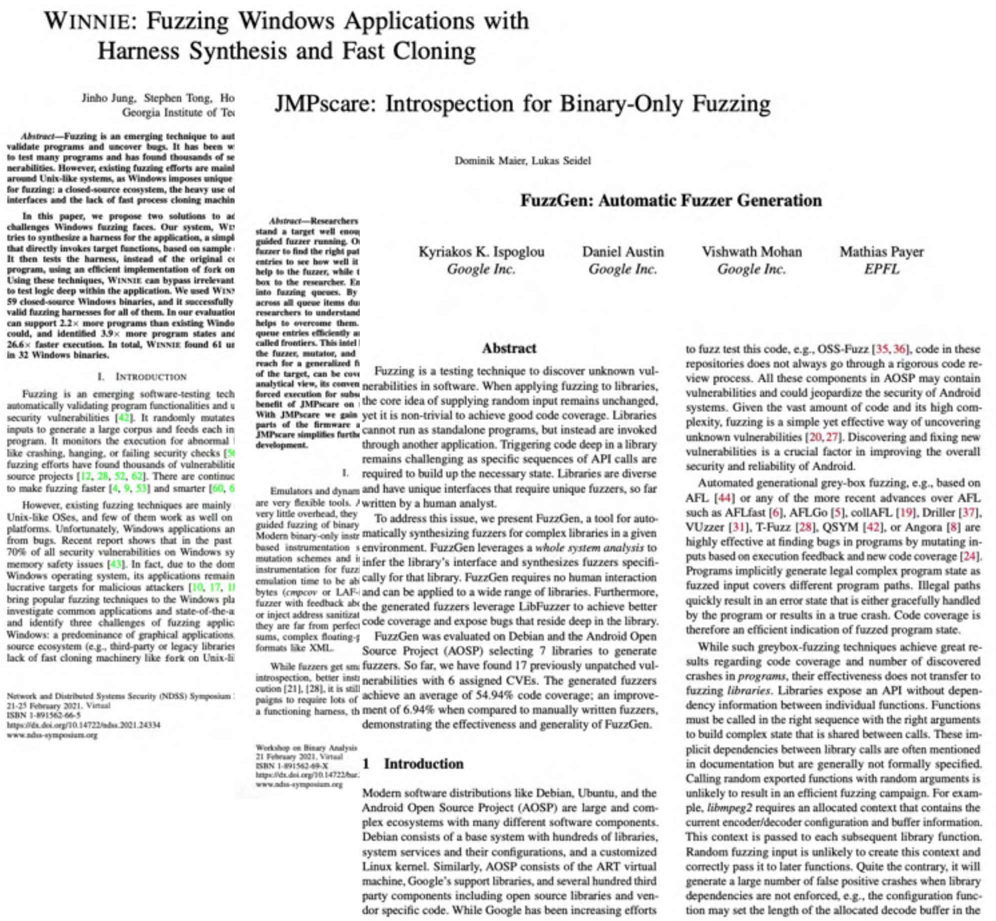
- QEMU (slow / incomplete)
- Avatar^2 orchestration

Fuzzer harnessing

Relates to previous research

Can we automatically synthesize a harness?

- Or at least help us develop one…



WINNIE: Fuzzing Windows Applications with Harness Synthesis and Fast Cloning

JMPscare: Introspection for Binary-Only Fuzzing

Dominik Maier, Lukas Seidel

FuzzGen: Automatic Fuzzer Generation

Kyriakos K. Ispoglou, Google Inc.
Daniel Austin, Google Inc.
Vishwath Mohan, Google Inc.
Mathias Payer, EPFL

Abstract

Fuzzing is a testing technique to discover unknown vulnerabilities in software. When applying fuzzing to libraries, the core idea of supplying random input remains unchanged, yet it is non-trivial to achieve good code coverage. Libraries cannot run as standalone programs, but instead are invoked through another application. Triggering code deep in a library remains challenging as specific sequences of API calls are required to build up the necessary state. Libraries are diverse and have unique interfaces that require unique fuzzers, so far written by a human analyst.

To address this issue, we present FuzzGen, a tool for automatically synthesizing fuzzers for complex libraries in a given environment. FuzzGen leverages a *whole system analysis* to infer the library's interface and synthesizes fuzzers specifically for that library. FuzzGen requires no human interaction and can be applied to a wide range of libraries. Furthermore, the generated fuzzers leverage LibFuzzer to achieve better code coverage and expose bugs that reside deep in the library.

FuzzGen was evaluated on Debian and the Android Open Source Project (AOSP) selecting 7 libraries to generate fuzzers. So far, we have found 17 previously unpatched vulnerabilities with 6 assigned CVEs. The generated fuzzers achieve an average of 54.94% code coverage; an improvement of 6.94% when compared to manually written fuzzers, demonstrating the effectiveness and generality of FuzzGen.

1 Introduction

Modern software distributions like Debian, Ubuntu, and the Android Open Source Project (AOSP) are large and complex ecosystems with many different software components. Debian consists of a base system with hundreds of libraries, system services and their configurations, and a customized Linux kernel. Similarly, AOSP consists of the ART virtual machine, Google's support libraries, and several hundred third party components including open source libraries and vendor specific code. While Google has been increasing efforts to fuzz test this code, e.g., OSS-Fuzz [35, 36], code in these repositories does not always go through a rigorous code review process. All these components in AOSP may contain vulnerabilities and could jeopardize the security of Android systems. Given the vast amount of code and its high complexity, fuzzing is a simple yet effective way of uncovering unknown vulnerabilities [20, 27]. Discovering and fixing new vulnerabilities is a crucial factor in improving the overall security and reliability of Android.

Automated generational grey-box fuzzing, e.g., based on AFL [44] or any of the more recent advances over AFL such as AFLfast [6], AFLGo [5], collAFL [19], Driller [37], VUzzer [31], T-Fuzz [28], QSYM [42], or Angora [8] are highly effective at finding bugs in programs by mutating inputs based on execution feedback and new code coverage [24]. Programs implicitly generate legal complex program state as fuzzed input covers different program paths. Illegal paths quickly result in an error state that is either gracefully handled by the program or results in a true crash. Code coverage is therefore an efficient indication of fuzzed program state.

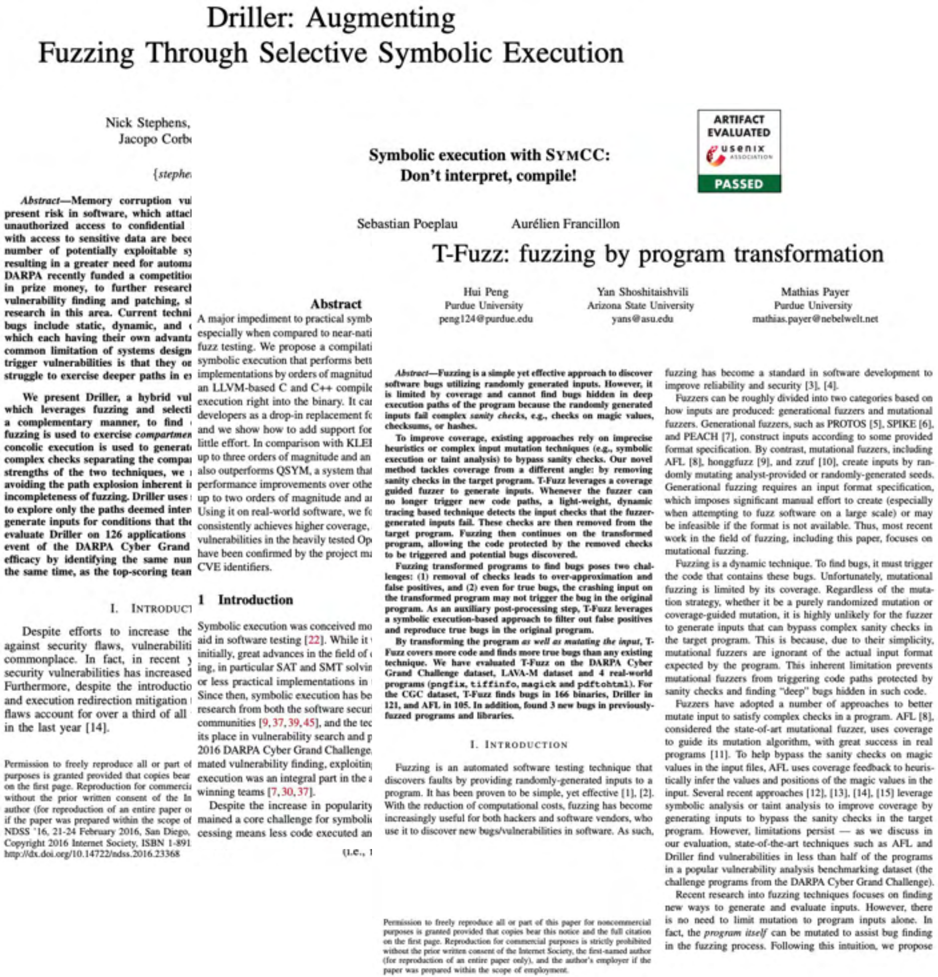
While such greybox-fuzzing techniques achieve great results regarding code coverage and number of discovered crashes in *programs*, their effectiveness does not transfer to fuzzing *libraries*. Libraries expose an API without dependency information between individual functions. Functions must be called in the right sequence with the right arguments to build complex state that is shared between calls. These implicit dependencies between library calls are often mentioned in documentation but are generally not formally specified. Calling random exported functions with random arguments is unlikely to result in an efficient fuzzing campaign. For example, *libmpeg2* requires an allocated context that contains the current encoder/decoder configuration and buffer information. This context is passed to each subsequent library function. Random fuzzing input is unlikely to create this context and correctly pass it to later functions. Quite the contrary, it will generate a large number of false positive crashes when library dependencies are not enforced, e.g., the configuration function may set the length of the allocated decode buffer in the

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?

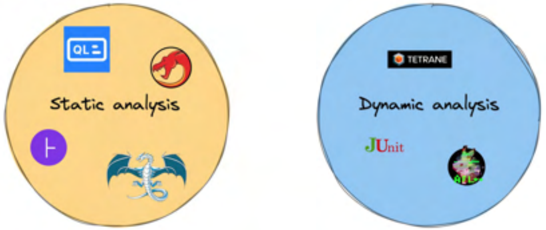
What about…

- Directed fuzzers?
- Determining wher
- Benchmarking fu

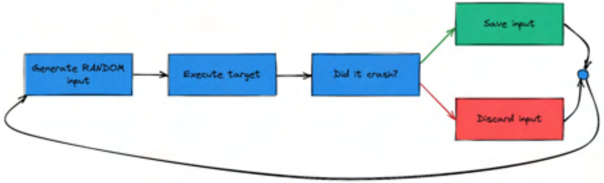
Conclusions

- 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**