The Hitchhiker’s Guide to Fuzzer Coverage Metrics

$ whoami

- > 10 years as a security researcher @ DSTG

- Principal vulnerability researcher @ Interrupt Labs

- Just submitted my PhD @ ANU 🎉

`$ whoami`

- > 10 years as a security researcher @ DSTG
- Principal vulnerability researcher @ Interrupt Labs
- Just submitted my PhD @ ANU 🎉

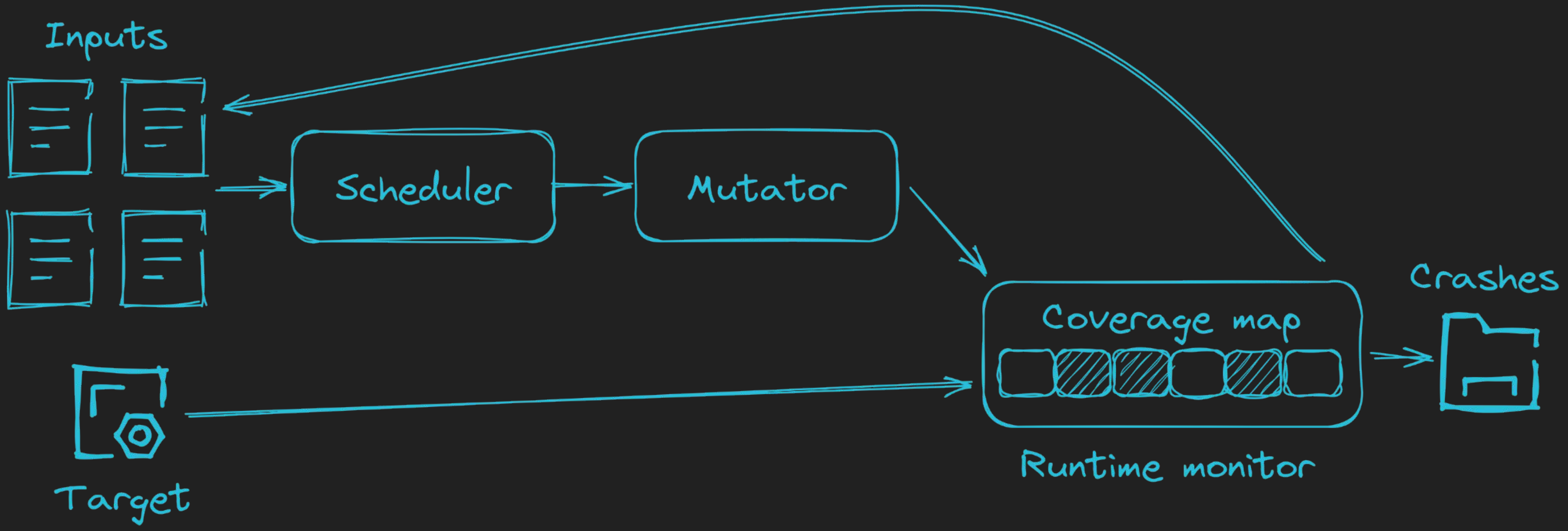
Me as a grad student
(sometimes)

Let’s talk (more) about fuzzing

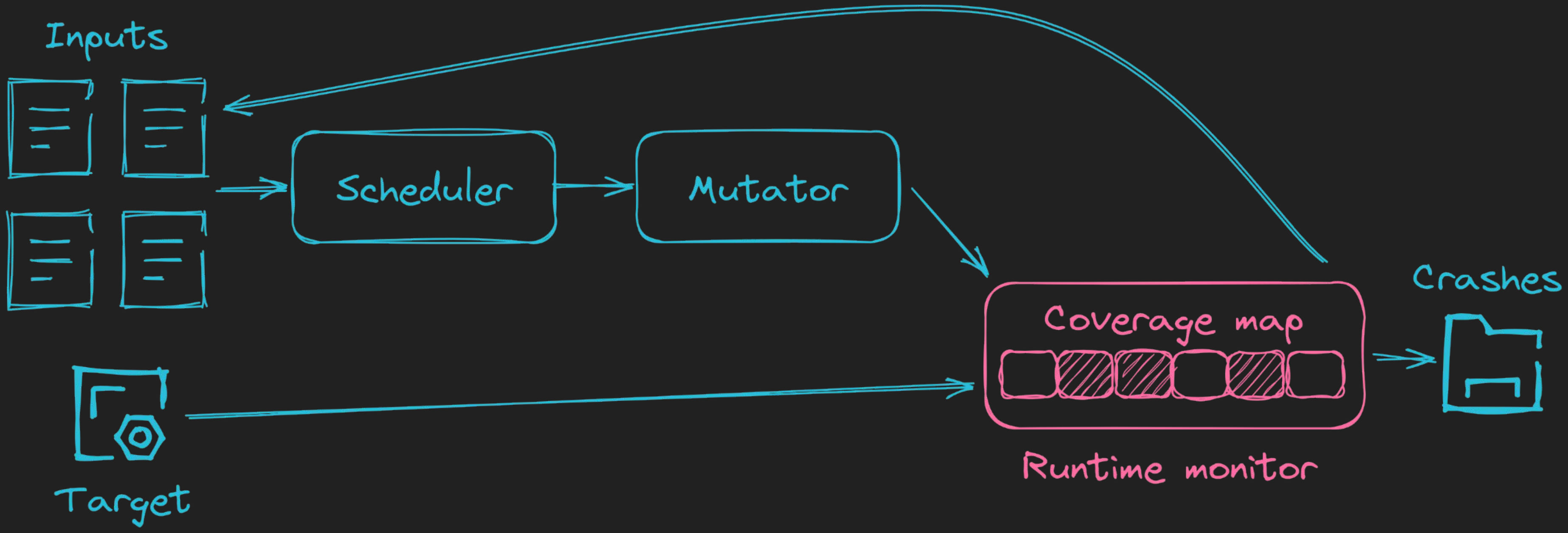
What is Fuzzing?

1. Generate random/invalid data
2. Execute target with said data
3. See if target breaks
4. Return to 1.

Fuzzing 101


Inputs
Scheduler
Mutator
Coverage map
Runtime monitor
Crashes
Target

Fuzzing 101

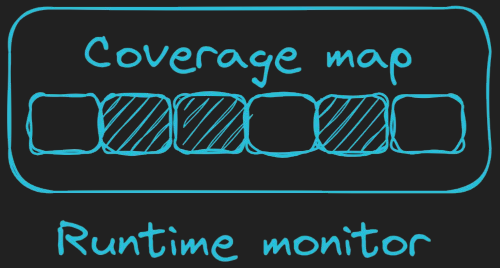

Inputs
Scheduler
Mutator
Coverage map
Runtime monitor
Crashes
Target

Fuzzing 101

Fuzzers find bugs by exploring a target's *state space*

Approximate the target's state space and track at runtime

- Must be lightweight!



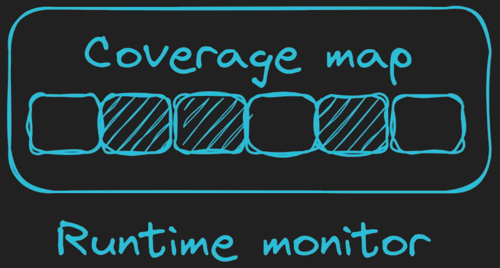
Retain inputs uncovering new states

Fuzzing 101

Fuzzers find bugs by exploring a target's *state space*

Approximate the target's state space and track at runtime

- Must be lightweight!



Retain inputs uncovering new states

You can't find bugs in states never covered

How do we measure coverage?

Abstraction!

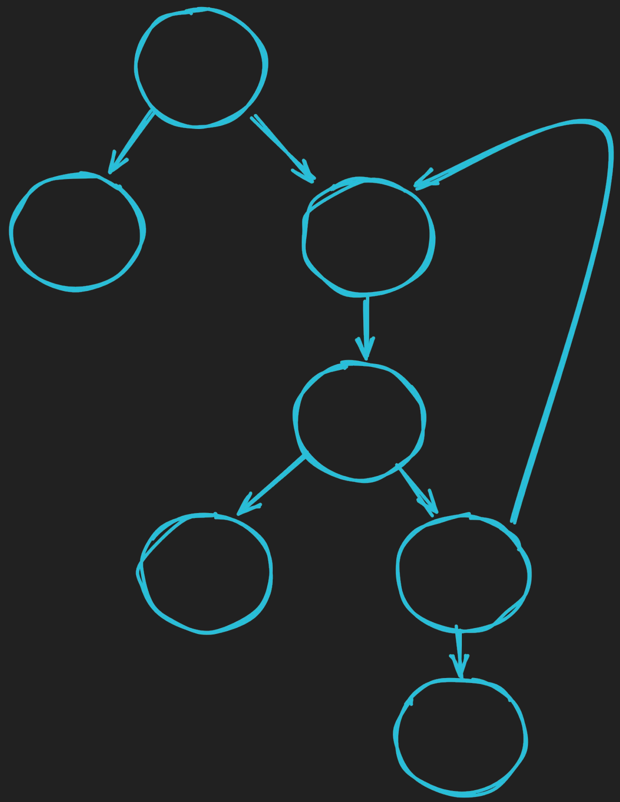
Approximate program states

- Control flow

- Data flow

Control Flow

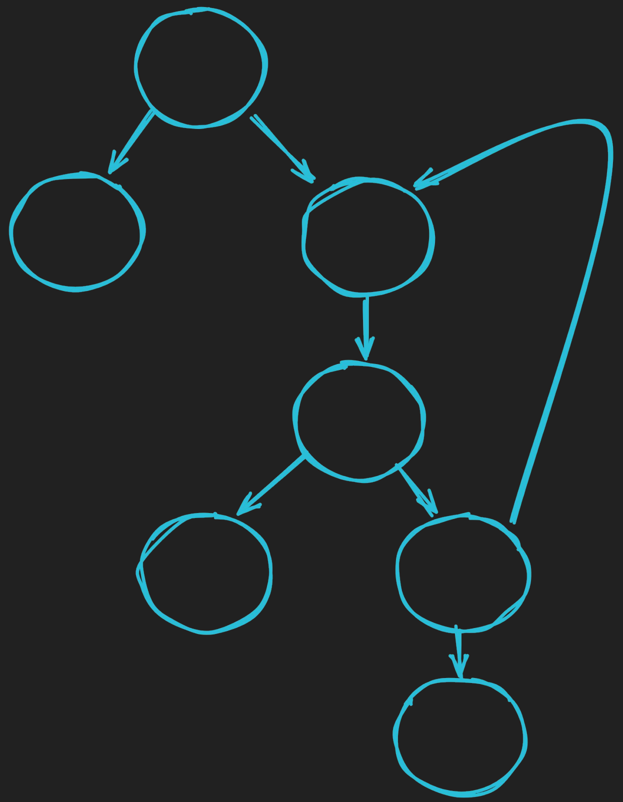
Decompose a function into a **control-flow graph**



Control Flow: Basic Blocks

Decompose a function into a **control-flow graph**

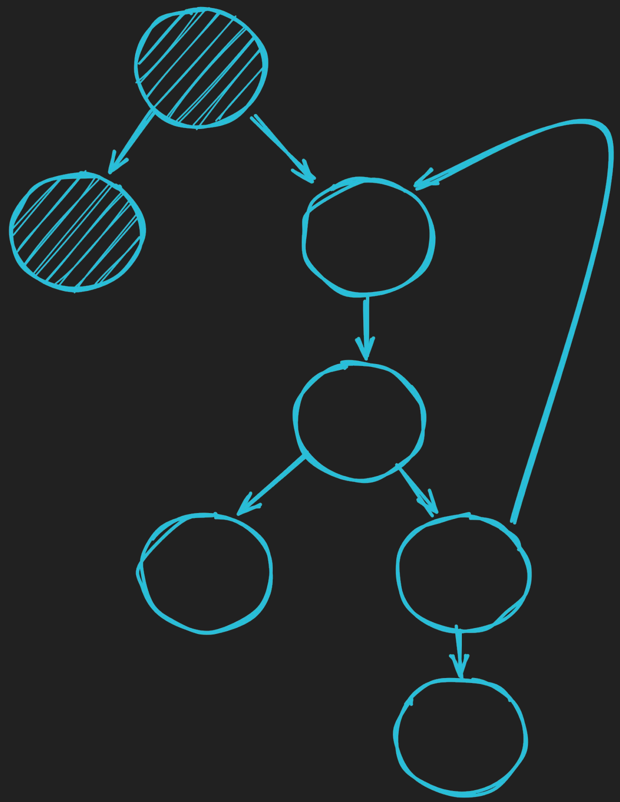
Record when nodes are covered



Control Flow: Basic Blocks

Decompose a function into a **control-flow graph**

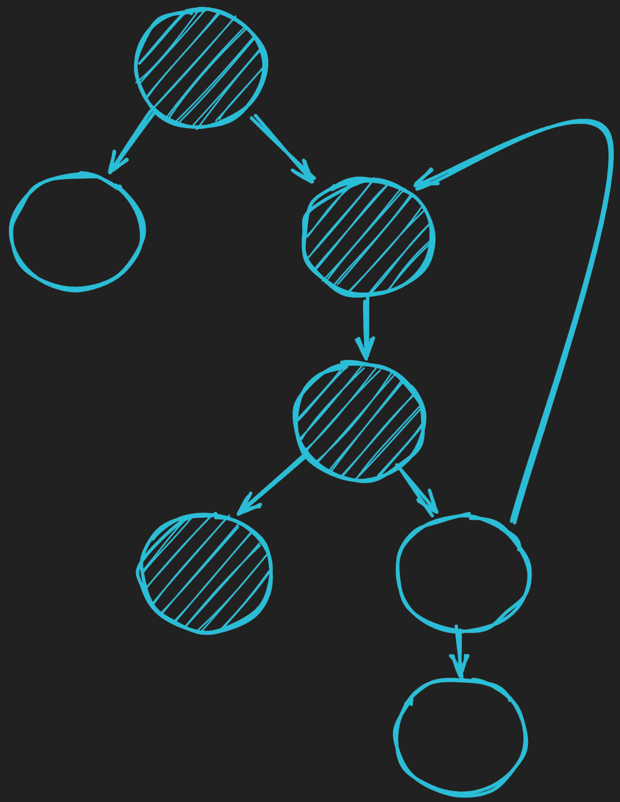
Record when nodes are covered



Control Flow: Basic Blocks

Decompose a function into a **control-flow graph**

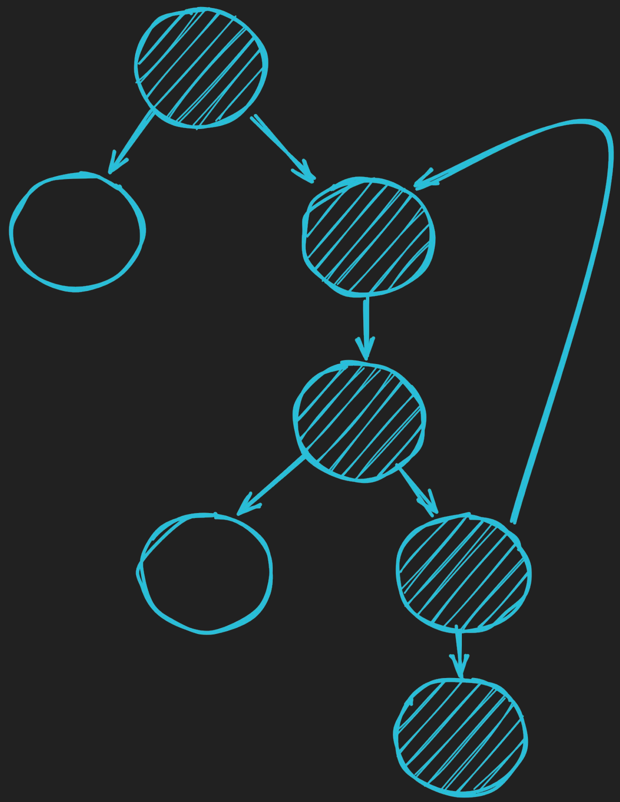
Record when nodes are covered



Control Flow: Basic Blocks

Decompose a function into a **control-flow graph**

Record when nodes are covered

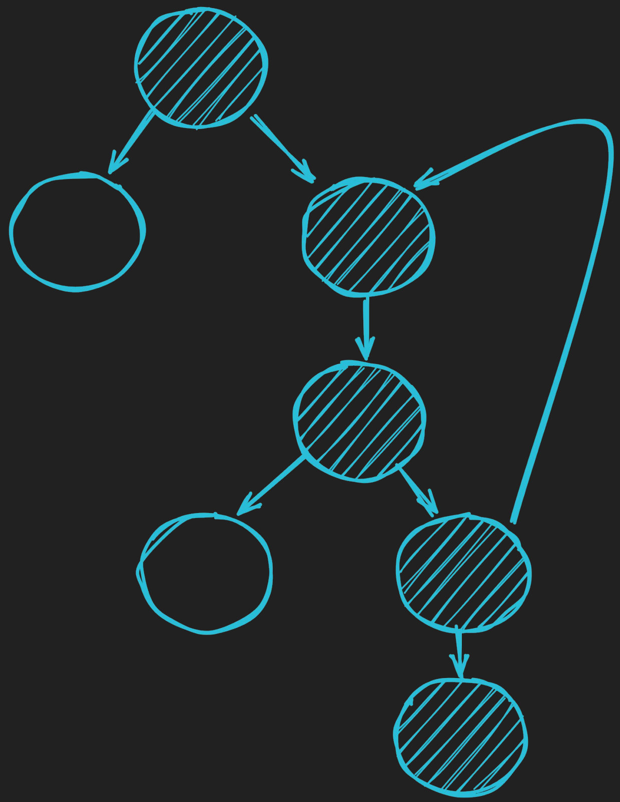


Control Flow: Basic Blocks

Decompose a function into a **control-flow graph**

Record when **nodes** are covered

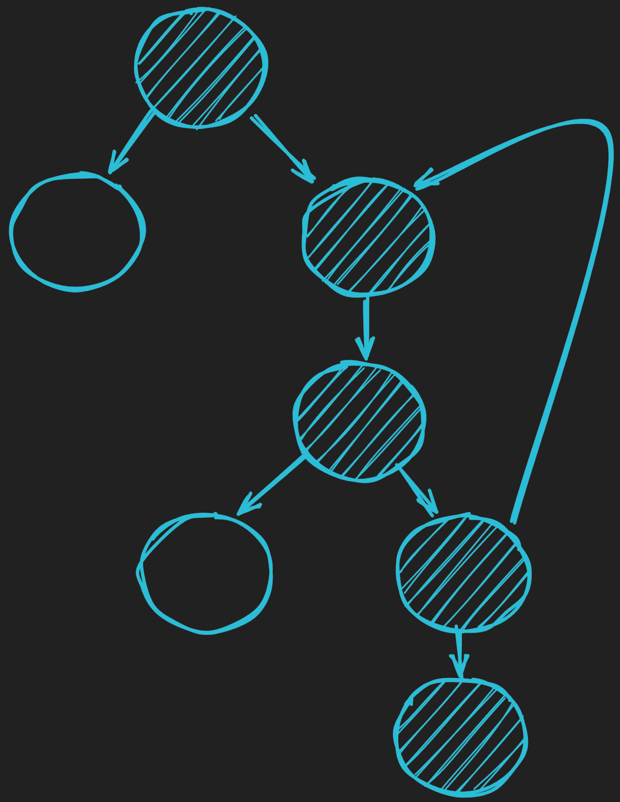
What's the problem?



Control Flow: Edges

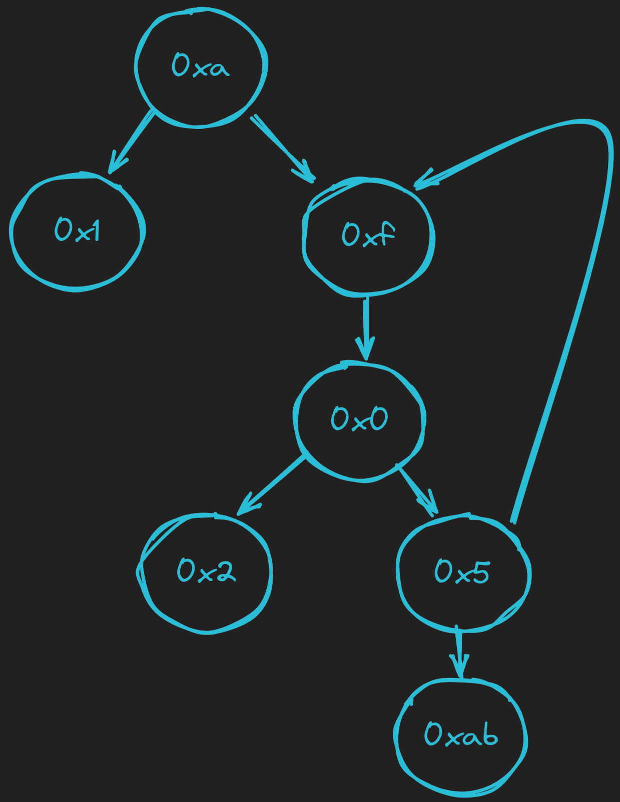
Decompose a function into a **control-flow graph**

Record when ~~nodes~~ **edges** are covered



Control Flow: Edges

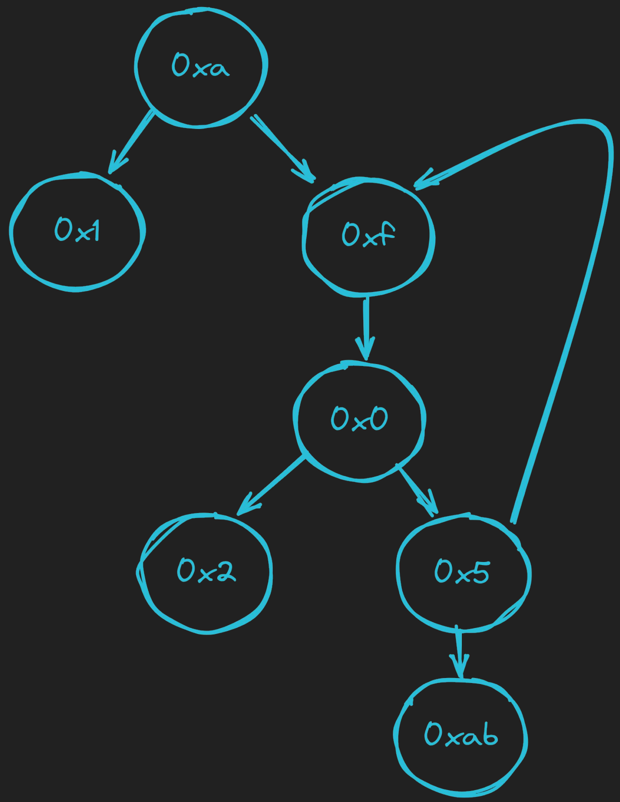
Label nodes (at compile time)



Control Flow: Edges

At start of each block (at runtime):

1. Edge ID = Prev block ^ Curr block
2. Prev block = Right-shift Curr block

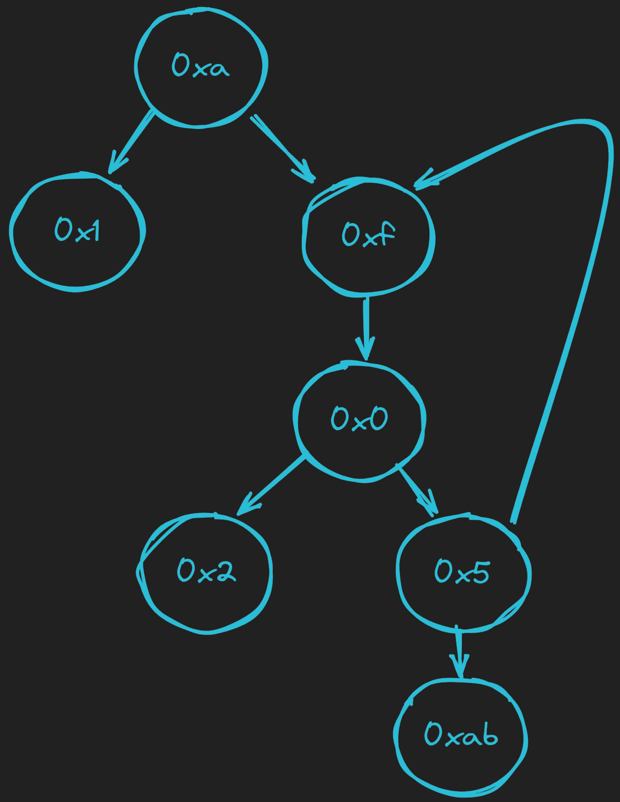


Control Flow: Edges

At start of each block (at runtime):

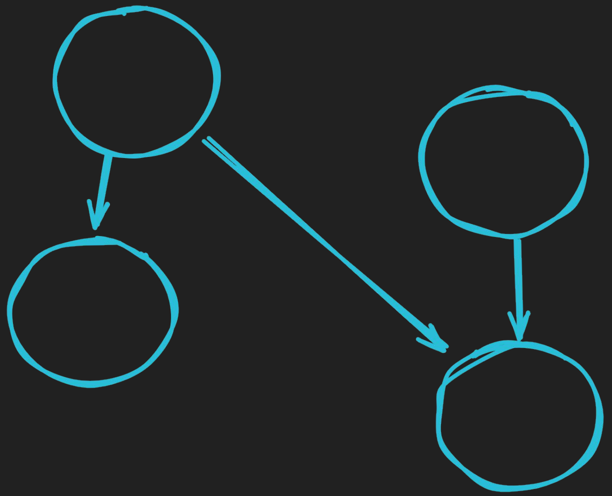
1. Edge ID = Prev block ^ Curr block
2. Prev block = Right-shift Curr block

What's the problem?



Control Flow: “Better” Edges

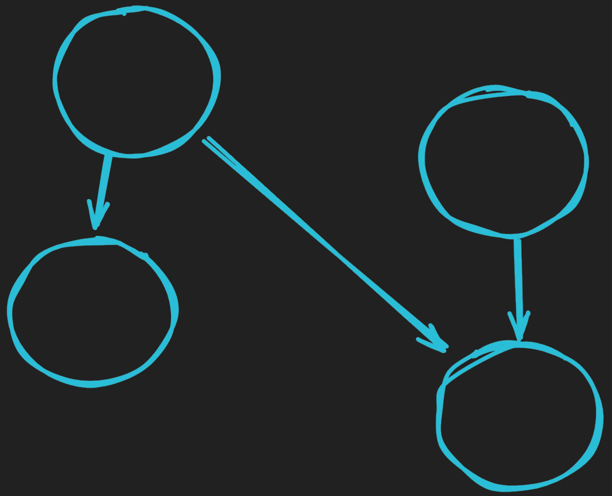
Transform the CFG and split **critical edges**



Control Flow: “Better” Edges

Transform the CFG and split **critical edges**

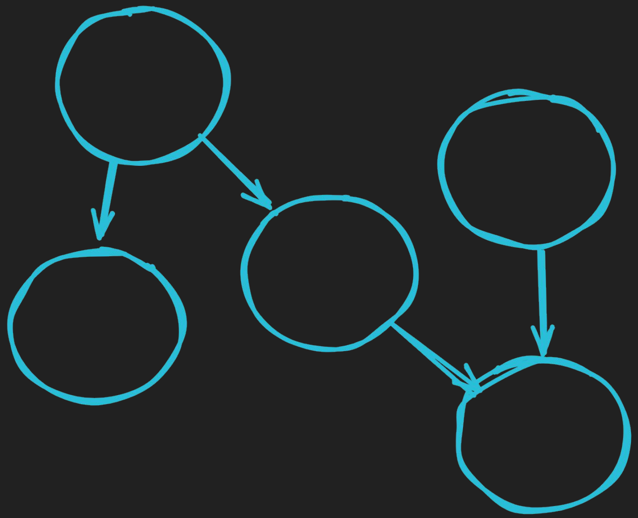
An edge whose destination has multiple predecessors and source has multiple successors



Control Flow: “Better” Edges

Transform the CFG and split **critical edges**

An edge whose destination has multiple predecessors and source has multiple successors



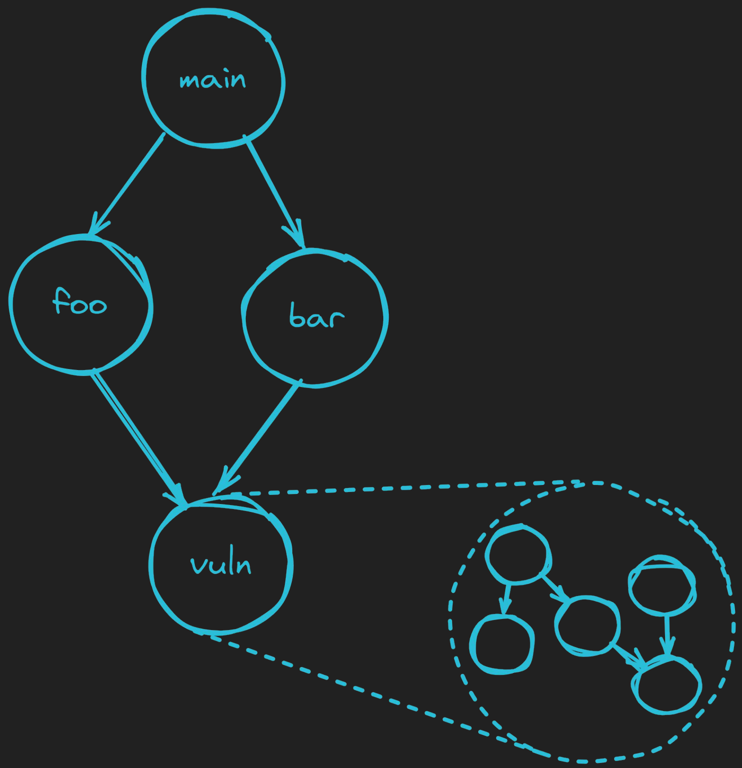
Insert a “dummy” block. Now, block coverage => edge coverage

What else can we do with control flow?

Context Sensitivity

Consider the calling context

I.e., the chain of function calls leading to current location



Context Sensitivity

Label nodes and functions (at compile time)

At function call and return (at runtime):

1. Call ctx = Call ctx ^ Function ID

At start of each block (at runtime):

1. Edge ID = Prev block ^ Curr block ^ Call ctx
2. Prev block = Right-shift Curr block

Context Sensitivity, Issues

Return of collisions

- Requires increasing coverage map size => slowdown

“Queue explosion”

- Retain useless seeds

Predictive Context Sensitivity

Function cloning

- Turn a context-insensitive analysis to a context-sensitive analysis
- No more collisions!

Predictive Context-sensitive Fuzzing

Pietro Borrello*, Andrea Fioraldi†, Daniele Cono D'Elia*,
Davide Balzarotti†, Leonardo Querzoni* and Cristiano Giuffrida‡
*Sapienza University of Rome
†EURECOM
‡Vrije Universiteit Amsterdam
{borrello, delia, querzoni}@diag.uniroma1.it, {fioraldi, balzarot}@eurecom.fr, giuffrida@cs.vu.nl

***Abstract*—Coverage-guided fuzzers expose bugs by progressively mutating testcases to drive execution to new program locations. Code coverage is currently the most effective and popular exploration feedback. For several bugs, though, also how execution reaches a buggy program location may matter: for those, only tracking what code a testcase exercises may lead fuzzers to overlook interesting program states. Unfortunately, context-sensitive coverage tracking comes with an inherent state explosion problem. Existing attempts to implement context-sensitive coverage-guided fuzzers struggle with it, experiencing non-trivial issues for precision (due to coverage collisions) and performance (due to context tracking and queue/map explosion).**

In this paper, we show that a much more effective approach to context-sensitive fuzzing is possible. First, we propose function cloning as a backward-compatible instrumentation primitive to enable precise (i.e., collision-free) context-sensitive coverage tracking. Then, to tame the state explosion problem, we argue to account for contextual information only when a fuzzer explores contexts selected as promising. We propose a prediction scheme to identify one pool of such contexts: we analyze the data-flow diversity of the incoming argument values at call sites, exposing to the fuzzer a contextually refined clone of the callee if the latter sees incoming abstract objects that its uses at other sites do not.

Our work shows that, by applying function cloning to program regions that we predict to benefit from context-sensitivity, we can overcome the aforementioned issues while preserving, and even improving, fuzzing effectiveness. On the FuzzBench suite, our approach largely outperforms state-of-the-art coverage-guided fuzzing embodiments, unveiling more and different bugs without incurring explosion or other apparent inefficiencies. On these heavily tested subjects, we also found 8 enduring security issues in 5 of them, with 6 CVE identifiers issued.

I. INTRODUCTION

Fuzz testing (or fuzzing for short) techniques earned a prominent place in the software security research landscape over the last decade. Their efficacy in generating unexpected or invalid inputs that make a program crash helps developers catch bugs early, even before they turn into vulnerabilities [1]. As an example, their deployment at scale in the OSS-Fuzz [2] initiative has led so far to the discovery of over 30 000 bugs in the daily testing of hundreds of open-source projects.

Network and Distributed System Security (NDSS) Symposium 2024
26 February - 1 March 2024, San Diego, CA, USA
ISBN 1-891562-93-2
https://dx.doi.org/10.14722/ndss.2024.24113
www.ndss-symposium.org

The most popular and researched form of fuzzing is coverage-guided fuzzing (CGF), which uses code or other coverage information from program execution to deem whether the current testing input led to *interesting* (for example, previously unseen) portions of a program. The main intuition behind much CGF research is that code coverage is strongly correlated with bug coverage [3] and no dynamic testing technique can detect a bug if execution does not reach the corresponding program point at least once. A flourishing topic of research is to enlarge the covered code by improving the effectiveness of the input generation process, e.g., by guiding input mutations to meet complex control-flow conditions in the program [4], [5], [6].

However, for software testing, coverage is only one part of the equation [7], and the ultimate metric for the effectiveness of fuzzing remains the ability to discover bugs. As recently observed in [8], successful CGF embodiments balance between exploration and exploitation. While exploration aims to increase coverage, exploitation tries to trigger bugs in already-covered program regions by varying the inputs used to reach them before. As there is no immediate feedback for exploitation, fuzzers have to count on input mutations to execute such code "sufficiently well" to trigger bugs in it [8].

Therefore, other efforts focus on retaining for further mutation inputs that, while being equivalent to prior executions in terms of covered program points, exercise new valuable execution paths and/or internal states of the program [9]. Intuitively, these inputs offer alternative (and possibly more profitable) "starting points" for the above-said mutations to trigger some bugs. For example, most state-of-the-art CGF systems track edge coverage information to distinguish visits to the same basic block from different predecessor blocks [10].

Edge coverage and other *function-local* metrics track and summarize program execution for its effects on entities (e.g., code blocks, variable values) involving individual functions. A limitation of this strategy is that it may lead a fuzzer to overlook internal program states for which also *how* an entity is reached matters. In program analysis, this concept goes under the name of *context-sensitivity* and has seen many applications, such as refining the precision of pointer analyses [11] and developing compiler optimizations [12].

ANGORA [1] showcases the benefits of context-sensitivity for fuzzing by augmenting edge coverage with *calling-context* information, which captures the sequence of active function calls on the stack leading to the currently executing function [13]. In principle, such a *fully context-sensitive* approach can differentiate the coverage of each testcase in a fine-grained manner and lead to the discovery of more bugs [1], [10].

Predictive Context Sensitivity

Can't clone everything

- Use static analysis to inform context-sensitivity
- Favor call sites that see a higher diversity of for incoming **data flow** in function arguments
- Use points-to analysis to determine diversity

Predictive Context-sensitive Fuzzing

Pietro Borrello*, Andrea Fioraldi†, Daniele Cono D'Elia*,
Davide Balzarotti†, Leonardo Querzoni* and Cristiano Giuffrida‡
*Sapienza University of Rome
†EURECOM
‡Vrije Universiteit Amsterdam
{borrello, delia, querzoni}@diag.uniroma1.it, {fioraldi, balzarot}@eurecom.fr, giuffrida@cs.vu.nl

***Abstract*—Coverage-guided fuzzers expose bugs by progressively mutating testcases to drive execution to new program locations. Code coverage is currently the most effective and popular exploration feedback. For several bugs, though, also how execution reaches a buggy program location may matter: for those, only tracking what code a testcase exercises may lead fuzzers to overlook interesting program states. Unfortunately, context-sensitive coverage tracking comes with an inherent state explosion problem. Existing attempts to implement context-sensitive coverage-guided fuzzers struggle with it, experiencing non-trivial issues for precision (due to coverage collisions) and performance (due to context tracking and queue/map explosion).**

In this paper, we show that a much more effective approach to context-sensitive fuzzing is possible. First, we propose function cloning as a backward-compatible instrumentation primitive to enable precise (i.e., collision-free) context-sensitive coverage tracking. Then, to tame the state explosion problem, we argue to account for contextual information only when a fuzzer explores contexts selected as promising. We propose a prediction scheme to identify one pool of such contexts: we analyze the data-flow diversity of the incoming argument values at call sites, exposing to the fuzzer a contextually refined clone of the callee if the latter sees incoming abstract objects that its uses at other sites do not.

Our work shows that, by applying function cloning to program regions that we predict to benefit from context-sensitivity, we can overcome the aforementioned issues while preserving, and even improving, fuzzing effectiveness. On the FuzzBench suite, our approach largely outperforms state-of-the-art coverage-guided fuzzing embodiments, unveiling more and different bugs without incurring explosion or other apparent inefficiencies. On these heavily tested subjects, we also found 8 enduring security issues in 5 of them, with 6 CVE identifiers issued.

I. Introduction

Fuzz testing (or fuzzing for short) techniques earned a prominent place in the software security research landscape over the last decade. Their efficacy in generating unexpected or invalid inputs that make a program crash helps developers catch bugs early, even before they turn into vulnerabilities [1]. As an example, their deployment at scale in the OSS-Fuzz [2] initiative has led so far to the discovery of over 30 000 bugs in the daily testing of hundreds of open-source projects.

Network and Distributed System Security (NDSS) Symposium 2024
26 February - 1 March 2024, San Diego, CA, USA
ISBN 1-891562-93-2
https://dx.doi.org/10.14722/ndss.2024.24113
www.ndss-symposium.org

The most popular and researched form of fuzzing is coverage-guided fuzzing (CGF), which uses code or other coverage information from program execution to deem whether the current testing input led to *interesting* (for example, previously unseen) portions of a program. The main intuition behind much CGF research is that code coverage is strongly correlated with bug coverage [3] and no dynamic testing technique can detect a bug if execution does not reach the corresponding program point at least once. A flourishing topic of research is to enlarge the covered code by improving the effectiveness of the input generation process, e.g., by guiding input mutations to meet complex control-flow conditions in the program [4], [5], [6].

However, for software testing, coverage is only one part of the equation [7], and the ultimate metric for the effectiveness of fuzzing remains the ability to discover bugs. As recently observed in [8], successful CGF embodiments balance between exploration and exploitation. While exploration aims to increase coverage, exploitation tries to trigger bugs in already-covered program regions by varying the inputs used to reach them before. As there is no immediate feedback for exploitation, fuzzers have to count on input mutations to execute such code "sufficiently well" to trigger bugs in it [8].

Therefore, other efforts focus on retaining for further mutation inputs that, while being equivalent to prior executions in terms of covered program points, exercise new valuable execution paths and/or internal states of the program [9]. Intuitively, these inputs offer alternative (and possibly more profitable) "starting points" for the above-said mutations to trigger some bugs. For example, most state-of-the-art CGF systems track edge coverage information to distinguish visits to the same basic block from different predecessor blocks [10].

Edge coverage and other *function-local* metrics track and summarize program execution for its effects on entities (e.g., code blocks, variable values) involving individual functions. A limitation of this strategy is that it may lead a fuzzer to overlook internal program states for which also *how* an entity is reached matters. In program analysis, this concept goes under the name of *context-sensitivity* and has seen many applications, such as refining the precision of pointer analyses [11] and developing compiler optimizations [12].

ANGORA [1] showcases the benefits of context-sensitivity for fuzzing by augmenting edge coverage with *calling-context* information, which captures the sequence of active function calls on the stack leading to the currently executing function [13]. In principle, such a *fully context-sensitive* approach can differentiate the coverage of each testcase in a fine-grained manner and lead to the discovery of more bugs [1], [10].

There’s that “data flow” thing again…

Data Flow Analysis

Process of collecting information about the ways variables are defined and used in a program

In compilers:

- Enables optimizations

In testing:

- Useful technique for measuring coverage

Defining Data Flow Coverage

Data Flow Analysis Techniques for Test Data Selection

Sandra Rapps* and Elaine J. Weyuker

Department of Computer Science, Courant Institute of Mathematical Sciences,
New York University, 251 Mercer Street, N.Y., N.Y. 10012

*also, YOURDON inc., 1133 Ave. of the Americas, N.Y., N.Y. 10036

Abstract

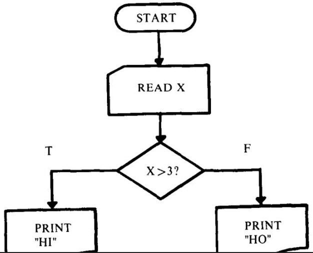
This paper examines a family of program test data selection criteria derived from data flow analysis techniques similar to those used in compiler optimization. It is argued that currently used path selection criteria which examine only the control flow of a program are inadequate. Our procedure associates with each point in a program at which a variable is defined, those points at which the value is used. Several related path criteria, which differ in the number of these associations needed to adequately test the program, are defined and compared.

Introduction

Program testing is the most commonly used method for demonstrating that a program actually accomplishes its intended purpose. The testing procedure consists of selecting elements from the program's input domain, executing the program on these test cases, and comparing the actual output with the expected output (in this discussion, we assume the existence of an "oracle", that is, some method to correctly determine the expected output). While exhaustive testing of all possible input values would provide the most complete picture of a program's performance, the size of the input domain is usually too large for this to be feasible. Instead, the usual procedure is to select a relatively small subset of the input domain which is, in some sense, representative of the entire input domain. An evaluation of the performance of the program on this test data is then used to predict its performance in general. Ideally, the test data should be chosen so that executing the program on this set will uncover all errors, thus guaranteeing that any program which produces correct results for the test data will produce correct results for any data in the input domain. However, discovering such a perfect set of test data is a difficult, if not impossible task [1,2]. In practice, test data is selected to give the tester a feeling of confidence that most errors will be discovered, without actually guaranteeing that the tested and debugged program is correct. This feeling of confidence is

select paths through the program whose elements fulfill the chosen criterion, and then to find the input data which would cause each of the chosen paths to be selected.

Using path selection criteria as test data selection criteria has a distinct weakness. Consider the strongest path selection criterion which requires that *all* program paths $p_1, p_2, \ldots$ be selected. This effectively partitions the input domain D into a set of classes $D = \cup D[j]$ such that for every $x \in D$, $x \in D[j]$ if and only if executing the program with input x causes path p_j to be traversed. Then a test $T = \{t_1, t_2, \ldots\}$, where $t_j \in D[j]$ would seem to be a reasonably rigorous test of the program. However, this still does not guarantee program correctness. If one of the D[j] is not revealing [2], that is for some $x_1 \in D[j]$ the program works correctly, but for some other $x_2 \in D[j]$ the program is incorrect, then if x_1 is selected as t_j the error will not be discovered. In figure 1 we see an example of this.



based on the dataflow coverage criteria. We have adapted these dataflow coverage definitions to define realistic dataflow coverage measures for C programs. A coverage measure associates a value with a set of tests for a given program. This value indicates the completeness of the set of tests for that program. We define the following dataflow coverage measures for C programs based on Rapps and Weyuker's[7] definitions: block, decision, c-use, p-use, all-uses, path, and du-path.

Precisely defining these concepts for the C language requires some care, but the basic ideas can be illustrated by the example in Figure 1. We define the measures to be intraprocedural, so they apply equally well to individual procedures (functions), sets of procedures, or whole programs.

Block. The simplest example of a coverage measure is basic block coverage. The body of a C procedure may be considered as a sequence of basic blocks. These are portions of code that nor-

Figure 1. Sum.c computes the sum and product of numbers from 0 to *N*.

```
#include <stdio.h>
main()
{
        int n, i, k, sum, prod;

        printf("Enter an integer and 0 for +, 1 for *: ");
        scanf("%d %d", &n, &k);
        sum = 0;
        prod = 1;
        i = 1;
        while (i <= n)
        {
            sum += i;
            prod *= i;
            i++;
        }
        if(k == 0)
            printf("n = %d, sum = %d\n", n, sum);
        if(k == 1)
            printf("n = %d, prod = %d\n", n, prod);
```

A definition of variable *i*

Uses of variable *i*

A basic block

A decision predicate involving variable *k*

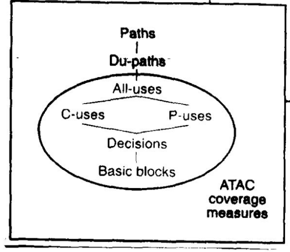


Figure 2. A hierarchy of control and dataflow coverage measures.

gram behavior, presumably due to one or more faults in the code.)

Figure 2 suggests an ordering of the coverage criteria. In this hierarchy, block coverage is weaker than decision coverage, which in turn is dominated by p-use coverage. C-use coverage dominates both block and decision coverage but is independent of p-use coverage; both c-use and

Data-flow coverage
is the tracking of
def-use chains
executed at runtime

Def-Use Chain Coverage

Def site: Variable allocation site (static or dynamic)

Use site: Variable access (read and/or write)

Def-use chain: Path between a def and use site

Def-Use Chain Coverage

Def site: Variable allocation site (static or dynamic)

Use site: Variable access (read and/or write)

Def-use chain: Path between a def and use site

Need an efficient implementation

A FEW
MOMENTS
LATER

datAFLow

1. Embed def-site IDs into objects
2. Reduce data-flow tracking to a metadata management problem
3. Now, def-site IDs are the metadata to retrieve at a use site

132

DatAFLow: Toward a Data-Flow-Guided Fuzzer

ADRIAN HERRERA, Australian National University, Australia
MATHIAS PAYER, École Polytechnique Fédérale de Lausanne, Switzerland
ANTONY L. HOSKING, Australian National University, Australia

Coverage-guided greybox fuzzers rely on *control-flow* coverage feedback to explore a target program and uncover bugs. Compared to control-flow coverage, *data-flow* coverage offers a more fine-grained approximation of program behavior. Data-flow coverage captures behaviors not visible as control flow and should intuitively discover more (or different) bugs. Despite this advantage, fuzzers guided by data-flow coverage have received relatively little attention, appearing mainly in combination with heavyweight program analyses (e.g., taint analysis, symbolic execution). Unfortunately, these more accurate analyses incur a high run-time penalty, impeding fuzzer throughput. Lightweight data-flow alternatives to control-flow fuzzing remain unexplored.

We present DATAFLow, a greybox fuzzer guided by lightweight data-flow profiling. We also establish a framework for reasoning about data-flow coverage, allowing the computational cost of exploration to be balanced with precision. Using this framework, we extensively evaluate DATAFLow across different precisions, comparing it against state-of-the-art fuzzers guided by control flow, taint analysis, and data flow.

Our results suggest that the ubiquity of control-flow-guided fuzzers is well-founded. The high run-time costs of data-flow-guided fuzzing (~10× higher than control-flow-guided fuzzing) significantly reduces fuzzer iteration rates, adversely affecting bug discovery and coverage expansion. Despite this, DATAFLow uncovered bugs that state-of-the-art control-flow-guided fuzzers (notably, AFL++) failed to find. This was because data-flow coverage revealed states in the target not visible under control-flow coverage. Thus, we encourage the community to continue exploring lightweight data-flow profiling; specifically, to lower run-time costs and to combine this profiling with control-flow coverage to maximize bug-finding potential.

CCS Concepts: • **Software and its engineering → Software testing and debugging**; *Software maintenance tools*; *Empirical software validation*;

Additional Key Words and Phrases: Fuzzing, data flow, coverage

ACM Reference format:
Adrian Herrera, Mathias Payer, and Antony L. Hosking. 2023. DatAFLow: Toward a Data-Flow-Guided Fuzzer. *ACM Trans. Softw. Eng. Methodol.* 32, 5, Article 132 (July 2023), 31 pages.
https://doi.org/10.1145/3587156

A. Herrera is also with Defence Science and Technology Group.
This work was supported by the Defence Science and Technology Group Next Generation Technologies Fund (Cyber) program via the Data61 Collaborative Research Project *Advanced Program Analysis for Software Vulnerability Discovery and Mitigation*, SNSF PCEGP2_186974, and ERC H2020 StG 850868.
Authors' addresses: A. Herrera and A. L. Hosking, The Australian National University, Canberra, ACT 2600, Australia; emails: {adrian.herrera, antony.hosking}@anu.edu.au; M. Payer, École Polytechnique Fédérale de Lausanne, Rte Cantonale, 1015 Lausanne, Switzerland; email: mathias.payer@nebelwelt.net.

1049-331X/2023/07-ART132
https://doi.org/10.1145/3587156

Can we combine control + data flow?

So what actually works?


FUZZ
ALL THE THINGS

Key Findings

Speed matters

- Dumb + fast > smart + slow

Different coverage metrics find different bugs

- This occurs even when coverage of one metric is less than another

In most programs, control flow subsumes data flow

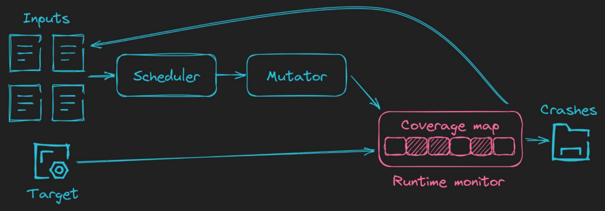
Key Questions

- What other ways can we approximate a program’s state space?
- Can we perform an initial (static?) analysis of the target to guide what coverage metric to use?
- Ensemble techniques?

The Hitchhiker's Guide to Fuzzer Coverage Metrics

Me as a grad student (sometimes)

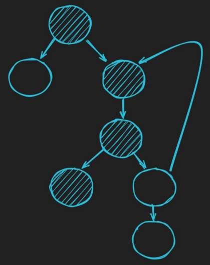
Fuzzing 101



Control Flow: Basic Blocks

Decompose a function into a **control-flow graph**

Record when nodes are covered



Predictive Context Sensitivity

Can't clone everything

- Use static analysis to inform context-sensitivity
- Favor call sites that see a higher diversity of for incoming **data flow** in function arguments
- Use points-to analysis to determine diversity

Predictive Context-sensitive Fuzzing

dataAFLow

1. Embed def-site IDs into objects
2. Reduce data-flow tracking to a metadata management problem
3. Now, def-site IDs are the metadata to retrieve at a use site

DatAFLow: Toward a Data-Flow-Guided Fuzzer

DDFuzz

- Incorporate **data dependency graph** (DDG) in coverage
- DDG represents data dependencies between instructions
- XOR into edge coverage

Fuzzing with Data Dependency Information

Key Findings

Speed matters

- Dumb + fast > smart + slow

Different coverage metrics find different bugs

- This occurs even when coverage of one metric is less than another

In most programs, control flow subsumes data flow

Key Questions

- What other ways can we approximate a program's state space?
- Can we perform an initial (static?) analysis of the target to guide what coverage metric to use?
- Ensemble techniques?