datAFLow: Towards a Data-Flow-Guided Fuzzer

Adrian Herrera, Mathias Payer, Antony L. Hosking

Australian National University
NATURAM PRIMUM COGNOSCERE RERUM
DST GROUP
EPFL

whoami

- PhD student at ANU
- Interests in fuzzing, binary analysis, program analysis

Australian National University DST GROUP EPFL

Accepted Registered Reports

Dissecting American Fuzzy Lop - A FuzzBench Evaluation
Andrea Fioraldi, Alessandro Mantovani (EURECOM), Dominik Maier (TU Berlin), Davide Balzarotti (EURECOM)

NSFuzz: Towards Efficient and State-Aware Network Service Fuzzing
Shisong Qin (Tsinghua University), Fan Hu (State Key Laboratory of Mathematical Engineering and Advanced Computing), Bodong Zhao, Tingting Yin, Chao Zhang (Tsinghua University)

Fuzzing Configurations of Program Options
Zenong Zhang (University of Texas at Dallas), George Klees (University of Maryland), Eric Wang (Poolesville High School), Michael Hicks (University of Maryland), Shiyi Wei (University of Texas at Dallas)

Generating Test Suites for GPU Instruction Sets through Mutation and Equivalence Checking
Shoham Shitrit, Sreepathi Pai (University of Rochester)

First, Fuzz the Mutants
Alex Groce, Goutamkumar Kalburgi (Northern Arizona Univeristy), Claire Le Goues, Kush Jain (Carnegie Mellon University), Rahul Gopinath (Saarland University)

Fine-Grained Coverage-Based Fuzzing
Bernard Nongpoh, Marwan Nour, Michaël Marcozzi, Sébastien Bardin (Université Paris Saclay)

datAFLow: Towards a Data-Flow-Guided Fuzzer
Adrian Herrera (Australian National University), Mathias Payer (EPFL), Antony Hosking (Australian National University)

Registered Report: Dissecting American Fuzzy Lop

A FuzzBench Evaluation

Other research directions instead explored different instrumentation techniques to study better forms of feedback. A popular form of feedback, usually considered the de-facto standard in the fuzzing community, is *code coverage*. This approach rewards the fuzzer when a new target execution results in a different coverage value, computed over the control flow graph (CFG) of the target application. In general, we refer to this family of approaches as *coverage-guided* fuzzing techniques.

Australian National University DST GROUP EPFL

Registered Report: Dissecting American Fuzzy Lop
A FuzzBench Evaluation

Other research directions instead explored different instrumentation techniques to study better forms of feedback. A popular form of feedback, usually considered the de-facto standard in the fuzzing community, is *code coverage*. This approach rewards the fuzzer when a new target execution results in a different coverage value, computed over the control flow graph (CFG) of the target application. In general, we refer to this family of approaches as *coverage-guided* fuzzing techniques.

Registered Report: NSFuzz: Towards Efficient and State-Aware Network Service Fuzzing

recent years, grey box fuzzing solutions that combine genetic algorithms and code coverage feedbacks have become more and more popular [8], [9], [10]. For instance, the representative fuzzer AFL [8] has greatly improved the code coverage and overall fuzzing effectiveness.

Registered Report: Dissecting American Fuzzy Lop

A FuzzBench Evaluation

Other research directions instead explored different instrumentation techniques to study better forms of feedback. A popular form of feedback, usually considered the de-facto standard in the fuzzing community, is *code coverage*. This approach rewards the fuzzer when a new target execution results in a different coverage value, computed over the control flow graph (CFG) of the target application. In general, we refer to this family of approaches as *coverage-guided* fuzzing techniques.

Registered Report: NSFuzz: Towards Efficient and State-Aware Network Service F

recent years, grey box fuzzing solutions that co
algorithms and code coverage feedbacks have become
and more popular [8], [9], [10]. For instance, the represen
fuzzer AFL [8] has greatly improved the code coverag
overall fuzzing effectiveness.

Registered Report: Fuzzing Configurations of Program Options

While it is expected that different configurations could result in different part of code being executed, there is no prior study that focuses on understanding how tuning a program's configurations would affect a fuzzer's results in terms of code coverage. The answer to this question can be used to motivate the design of a fuzzer that fuzzes configurations.

Australian National University DST GROUP EPFL

Registered Report: Dissecting American Fuzzy Lop

A FuzzBench Evaluation

Other research directions instead explored different instrumentation techniques to study better form… ular form of feedback, usually considered… in the fuzzing community, is *code cove*… rewards the fuzzer when a new target e… different coverage value, computed over t… (CFG) of the target application. In gene…, family of approaches as *coverage-guided* fu…

Registered Report: Generating Test Suites for GPU Instruction Sets through Mutation and Equivalence Checking

Coverage-guided fuzzing is used to construct test inputs in [21] where mutation is used to increase code coverage in an instruction set simulator. In constrast to these works, we mutate a stand-alone semantics which is not embedded in a simulator. We mutate the semantics to deliberately introduce bugs and use equivalence checking to surface inputs that trigger those bugs. Coverage-based techniques would complement our method.

Registered Report: NSFuzz: Towards… State-Aware Network Service…

…recent years, grey box fuzzing solutions that combine genetic algorithms and code coverage feedbacks have become more and more popular [8], [9], [10]. For instance, the representative fuzzer AFL [8] has greatly improved the code coverage and overall fuzzing effectiveness.

…Configurations of Program Options

While it is expected that different configurations could result in different part of code being executed, there is no prior study that focuses on understanding how tuning a program's configurations would affect a fuzzer's results in terms of code coverage. The answer to this question can be used to motivate the design of a fuzzer that fuzzes configurations.

Australian National University DST GROUP EPFL

Registered Report: Dissecting American Fuzzy Lop

A FuzzBench Evaluation

Other research directions instead explored different instrumentation techniques to study better forms of feedback. A pop-

Registered Report: Generating Test Suites for ...uction Sets through ...d Equivalence Checking

...uzzing is used to construct test inputs in ... is used to increase code coverage in an ...or. In constrast to these works, we mutate ...cs which is not embedded in a simulator. We mutate the semantics to deliberately introduce bugs and use equivalence checking to surface inputs that trigger those bugs. Coverage-based techniques would complement our method.

Registered Report: First, Fuzz the Mutants

RQ1 is the overall question of whether any variant of fuzzing using mutants increases standard fuzzing evaluation metrics (unique faults detected and code coverage). **RQ2-RQ4** consider some of the primary choices to be made in implementing fuzzing mutants.

Registered Report: NSFuzz: Towards Efficient and State-Aware Network Service Fuzzing

...recent years, grey box fuzzing solutions that combine genetic algorithms and code coverage feedbacks have become more and more popular [8], [9], [10]. For instance, the representative fuzzer AFL [8] has greatly improved the code coverage and overall fuzzing effectiveness.

Registered Report: Fuzzing Configurations of Program Options

While it is expected that different configurations could result in different part of code being executed, there is no prior study that focuses on understanding how tuning a program's configurations would affect a fuzzer's results in terms of code coverage. The answer to this question can be used to motivate the design of a fuzzer that fuzzes configurations.

Australian National University DST GROUP EPFL

Registered Report: Dissecting American Fuzzy Lop

A FuzzBench Evaluation

Other research directions instead explored different instrumentation techniques to study better forms of feedback. A popular form of feedback, usually considered the de facto standard in the fuzzing community, is edge coverage. This approach rewards the fuzzer when a new target execution results in a different coverage value computed over the control flow graph (CFG) of the target application. In general, we refer to this family of approaches as *coverage-guided* fuzzing.

Registered Report: First, Fuzz the Mutants

RQ1 is the overall question of whether any variant of fuzzing using mutants increases standard fuzzing evaluation metrics (unique faults detected and code coverage). **RQ2-RQ4** consider some of the primary choices to be made in implementing fuzzing mutants.

Registered Report: Generating Test Suites for Instruction Sets through Mutation and Equivalence Checking

Coverage-guided fuzzing is used to construct test inputs in [21] where mutation is used to increase code coverage in an instruction set simulator. In constrast to these works, we mutate a stand-alone semantics which is not embedded in a simulator. We mutate the semantics to deliberately introduce bugs and use equivalence checking to surface inputs that trigger those bugs.

...ur method.

Registered Report: NSFuzz: Towards Efficient and

State-A...

recent years, gr... algorithms and... and more popul... fuzzer AFL [8]... overall fuzzing eff...

Registered Report: ... ram Options

...tions could result ... there is no prior ... ning a program's ... ults in terms of code ... be used to motivate ... urations.

Registered Report: Fine-Grained Coverage-Based Fuzzing

Problem. In order to select which of the generated inputs will be saved for subsequent mutation, current fuzzers run the PUT with these inputs and measure some form of branch coverage.

coverage = control-flow coverage

This is changing…

Fuzzing with Data Dependency Information

Alessandro Mantovani
EURECOM
mantovan@eurecom.fr

Andrea Fioraldi
EURECOM
fioraldi@eurecom.fr

Davide Balzarotti
EURECOM
balzarot@eurecom.fr

The Use of Likely Invariants as Feedback for Fuzzers

Andrea Fioraldi, *EURECOM;* Daniele Cono D'Elia, *Sapienza University of Rome;* Davide Balzarotti, *EURECOM*

https://www.usenix.org/conference/usenixsecurity21/presentation/fioraldi

This is changing…

Fuzzing with Data Dependency Information

Alessandro Mantovani
EURECOM
mantovan@eurecom.fr

Andrea Fioraldi
EURECOM
fioraldi@eurecom.fr

Davide Balzarotti
EURECOM
balzarot@eurecom.fr

Data flow is becoming a "first-class citizen"

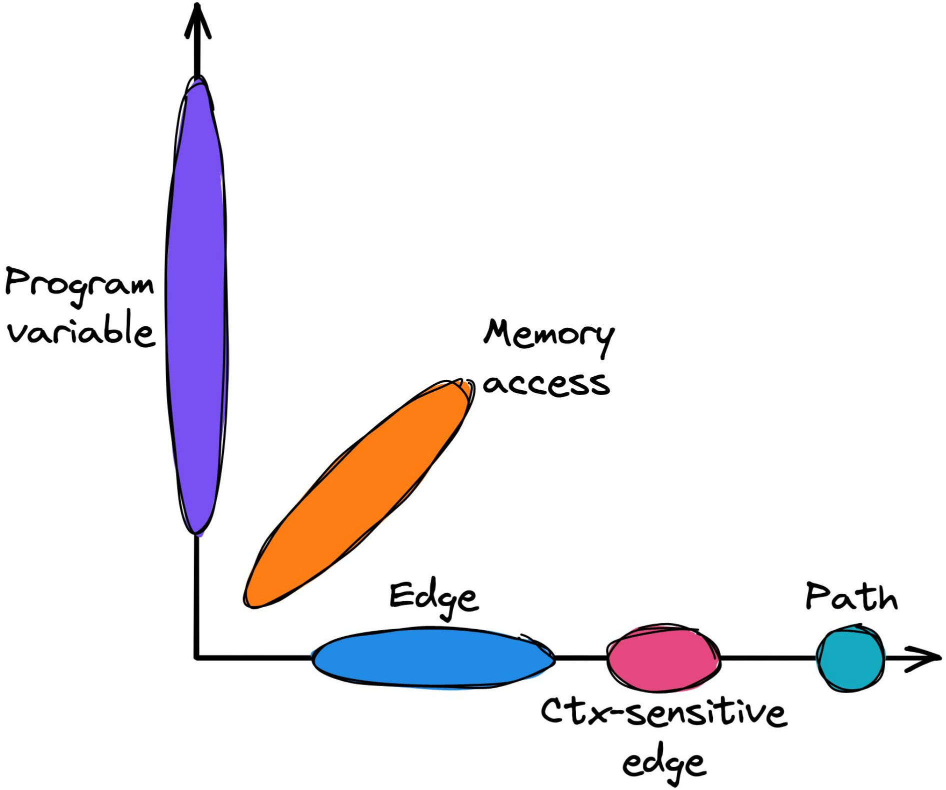
The Use of Likely Invariants as Feedback for Fuzzers

Andrea Fioraldi, *EURECOM;* Daniele Cono D'Elia, *Sapienza University of Rome;* Davide Balzarotti, *EURECOM*

https://www.usenix.org/conference/usenixsecurity21/presentation/fioraldi

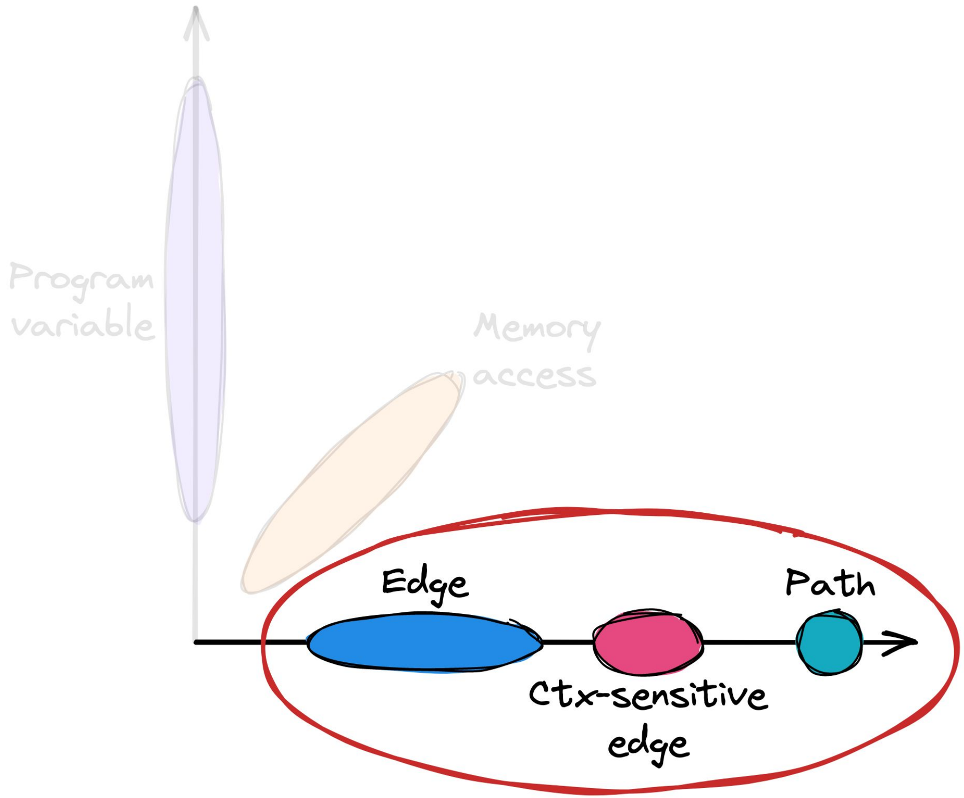
Australian National University DST GROUP EPFL

The “coverage spectrum”*

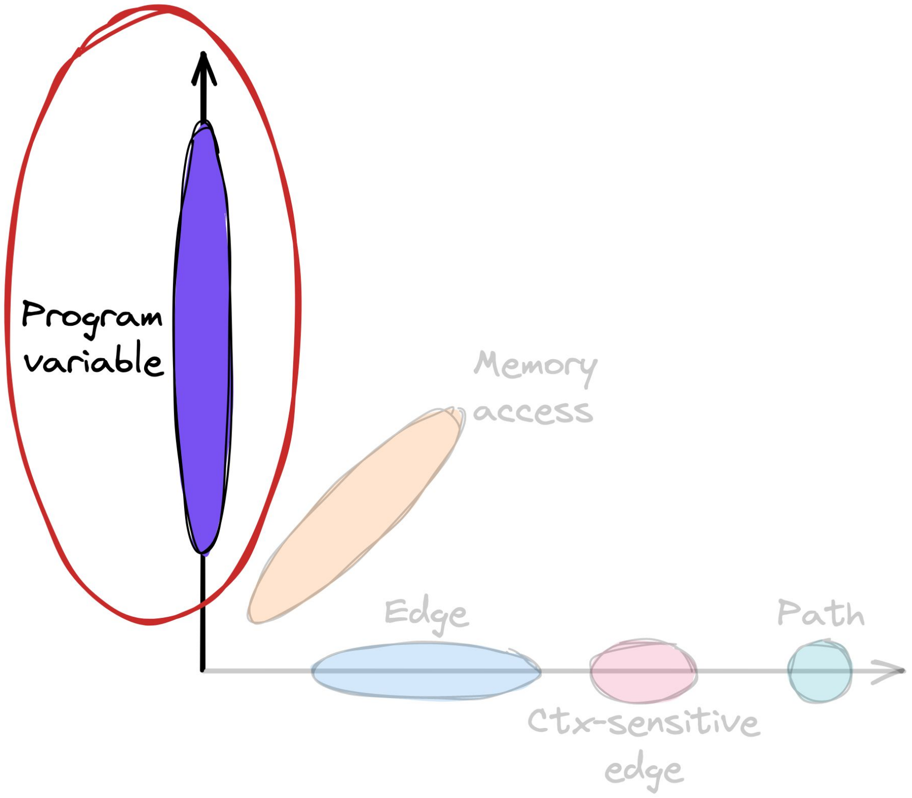


* Not to scale

The "coverage spectrum"



The "coverage spectrum"



Requirements

1. Define “data-flow coverage”
2. Efficiently track data flows
3. Data flows → fuzzer coverage
4. Evaluate!

Australian National University DST GROUP EPFL

Requirements

1. Define "data-flow coverage"

2. Efficiently track data flows

3. Data flows → fuzzer coverage

4. Evaluate!

Australian National University DST GROUP EPFL

1. Defining “data-flow coverage”

1. Defining "data-flow coverage"

ARTICLE

Data flow analysis techniques for test data selection

Authors: Sandra Rapps, Elaine J. Weyuker Authors Info & Claims

ICSE '82: Proceedings of the 6th international conference on Software engineering • September 1982 • Pages 272–278

Online: 13 September 1982 Publication History

Australian National University DST GROUP EPFL

1. 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

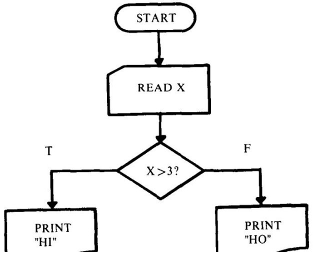
This paper examines a family of program test data selection criteria derived from data flow analysis techniques similar to those used in compiler optimization. It is argued that currently used path selection criteria which examine only the control flow of a program are inadequate. Our procedure associates 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 %", &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*

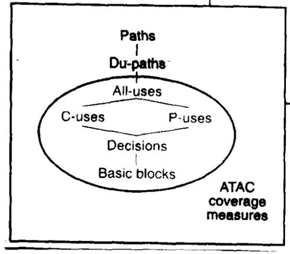


Figure 2. A hierarchy of control and dataflow coverage measures.

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

1. Defining “data-flow coverage”

Def site: Variable allocation site (static and dynamic)

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

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

1. Defining “data-flow coverage”

Def site: Variable allocation site (static and dynamic)

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

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

How to efficiently implement this?

Requirements

1. Define “data-flow coverage”
2. Efficiently track data flows
3. Data flows → fuzzer coverage
4. Evaluate!

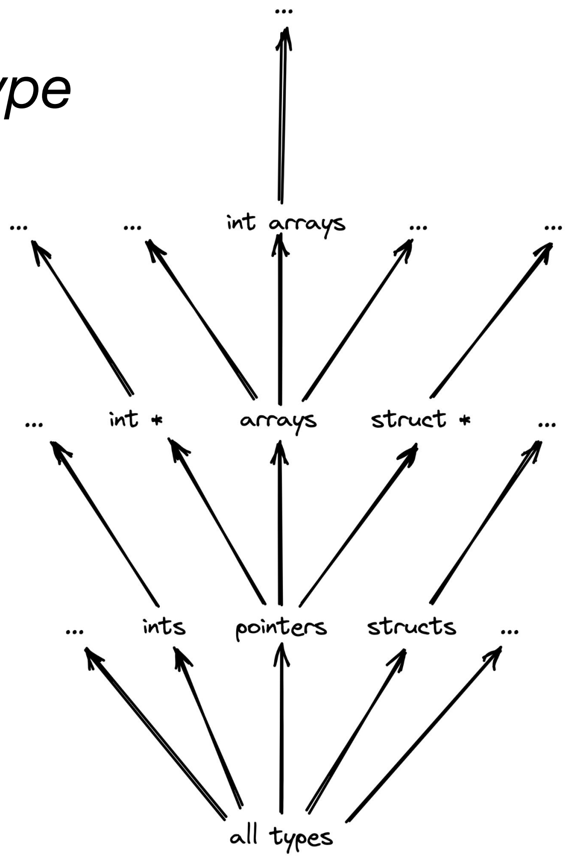
Australian National University DST GROUP EPFL

Problem: Tracking *all* data flows is infeasible

Solution: Track data flows at varying *sensitivities*

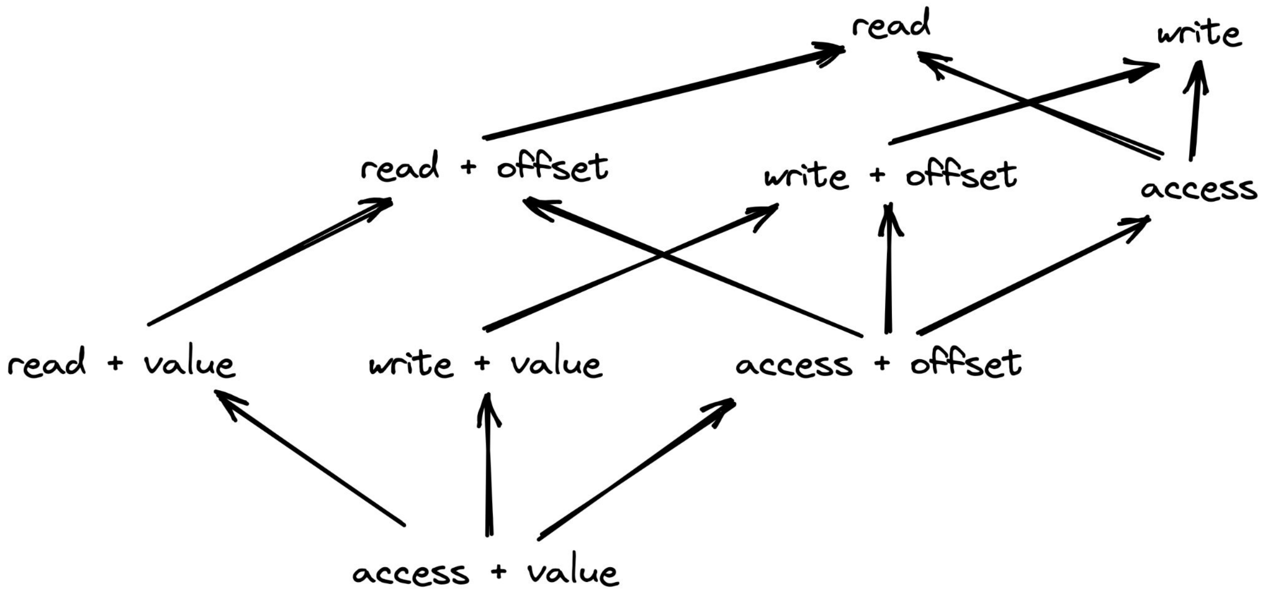
2. Efficiently track data flows

Partition def sites by *type*



2. Efficiently track data flows

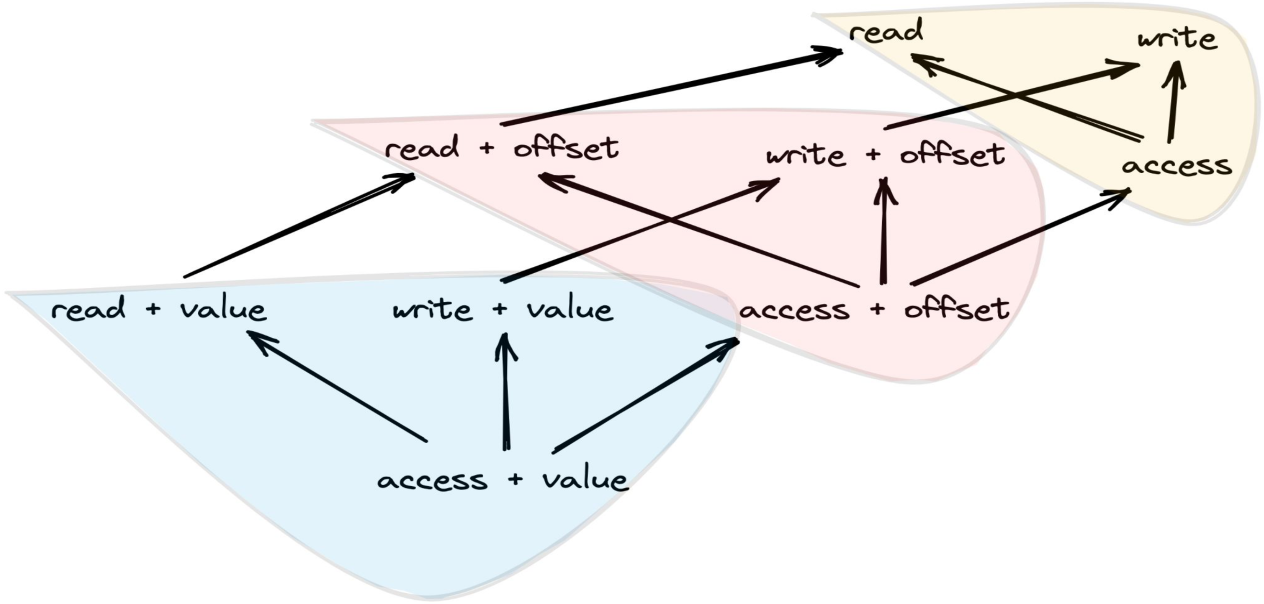
Partition use sites by *access*



Australian National University

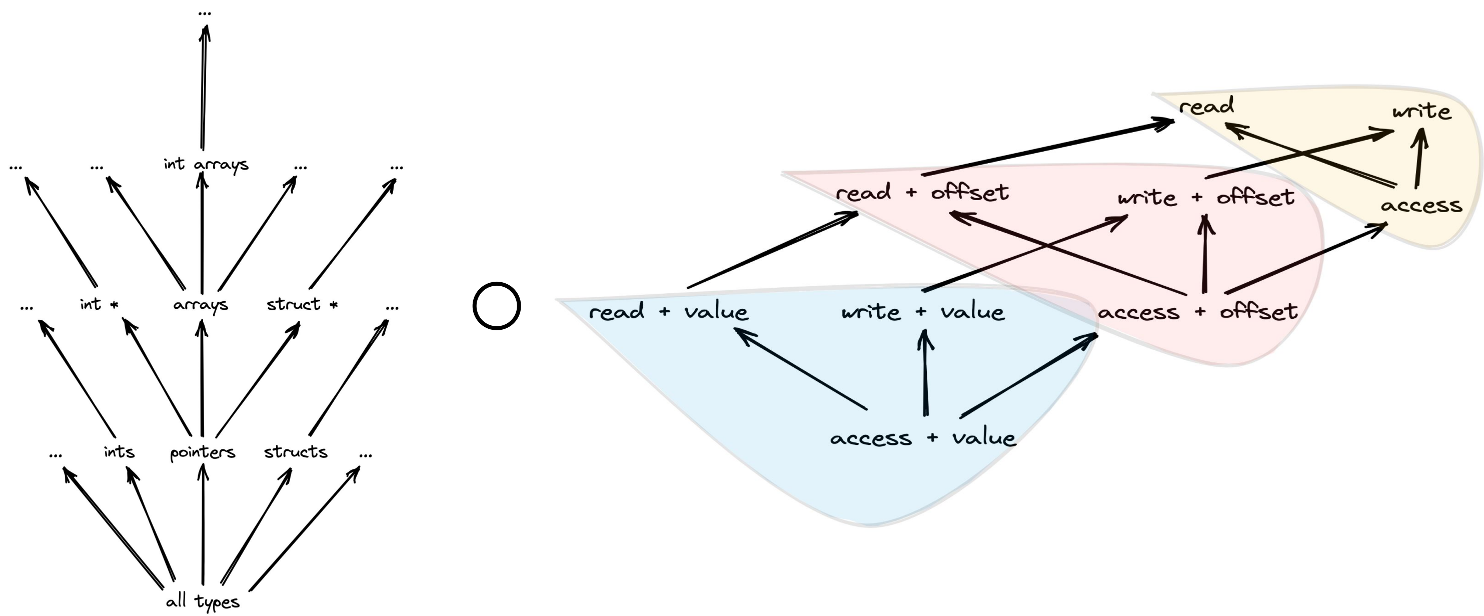
2. Efficiently track data flows

Partition use sites by *access*



2. Efficiently track data flows

Compose def/use lattices to realize desired sensitivity

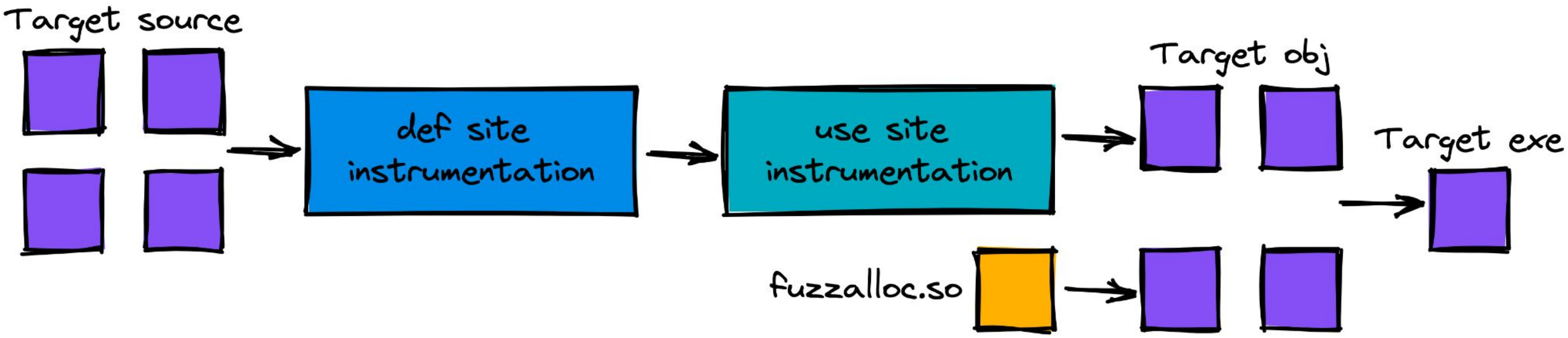


Do efficient implementations exist?

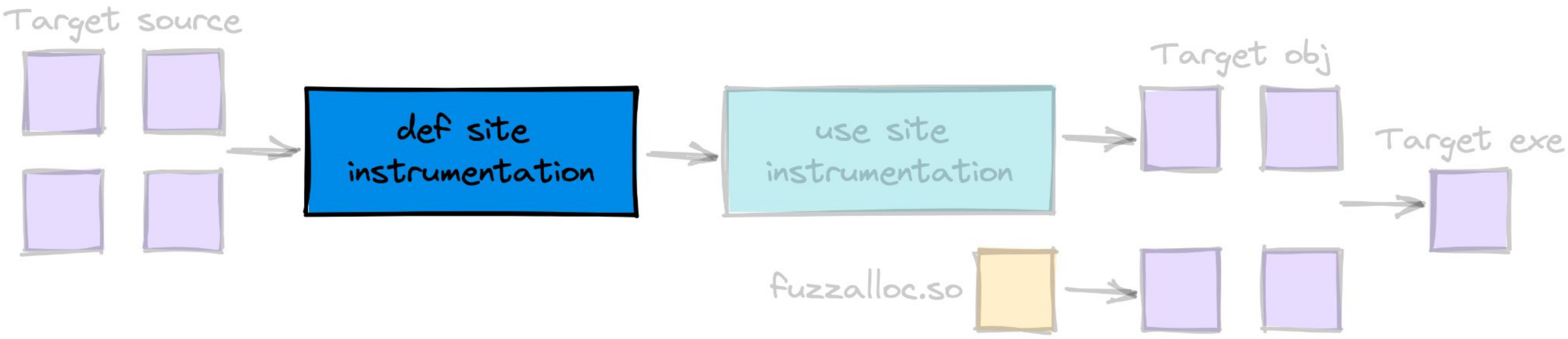
Requirements

1. Define “data-flow coverage”
2. Efficiently track data flows
3. Data flows → fuzzer coverage
4. Evaluate!

3. Data flows → fuzzer coverage



3. Data flows → fuzzer coverage



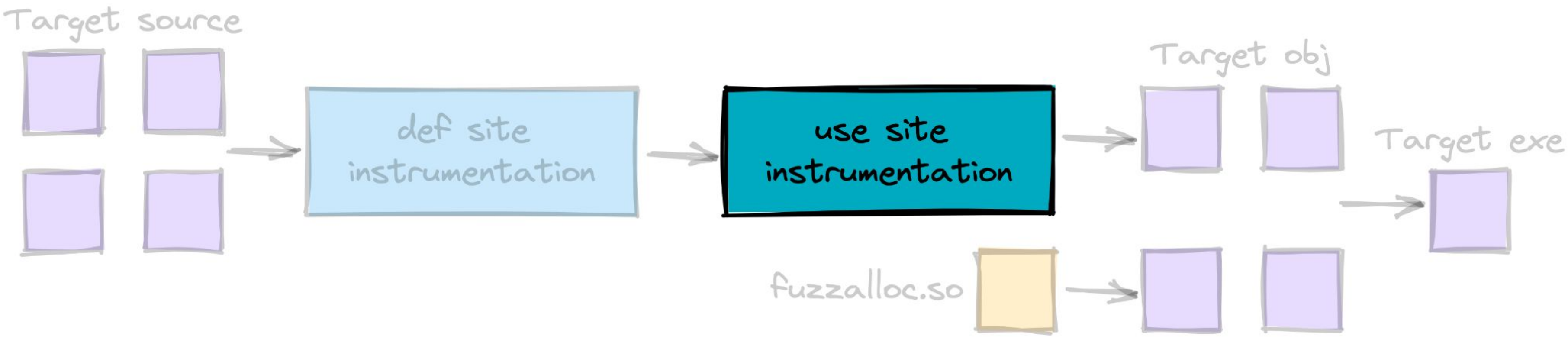
Def site instrumentation

1. Identify allocation sites (static and dynamic) based on desired sensitivity

2. Replace dynamic allocations with tagged allocation

3. “Heapify” static allocations (and tag)

3. Data flows → fuzzer coverage



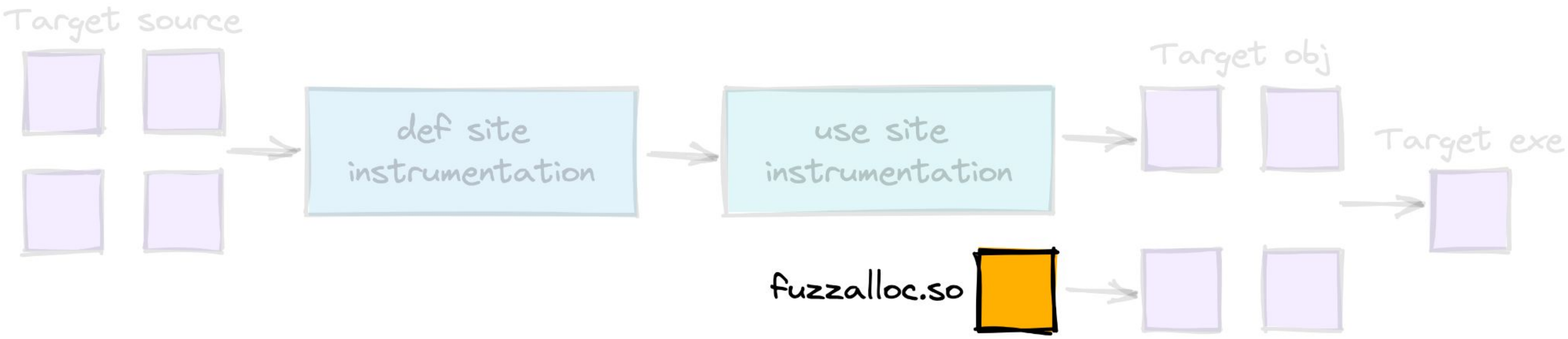
Australian National University
DST GROUP
EPFL

Use site instrumentation

1. Identify based on desired sensitivity (read/write/access)

2. Identified via runtime address

3. Data flows → fuzzer coverage



`fuzzalloc.so`

- Data-flow tracking is reduced to metadata management

- Def site IDs are the metadata to retrieve at use site

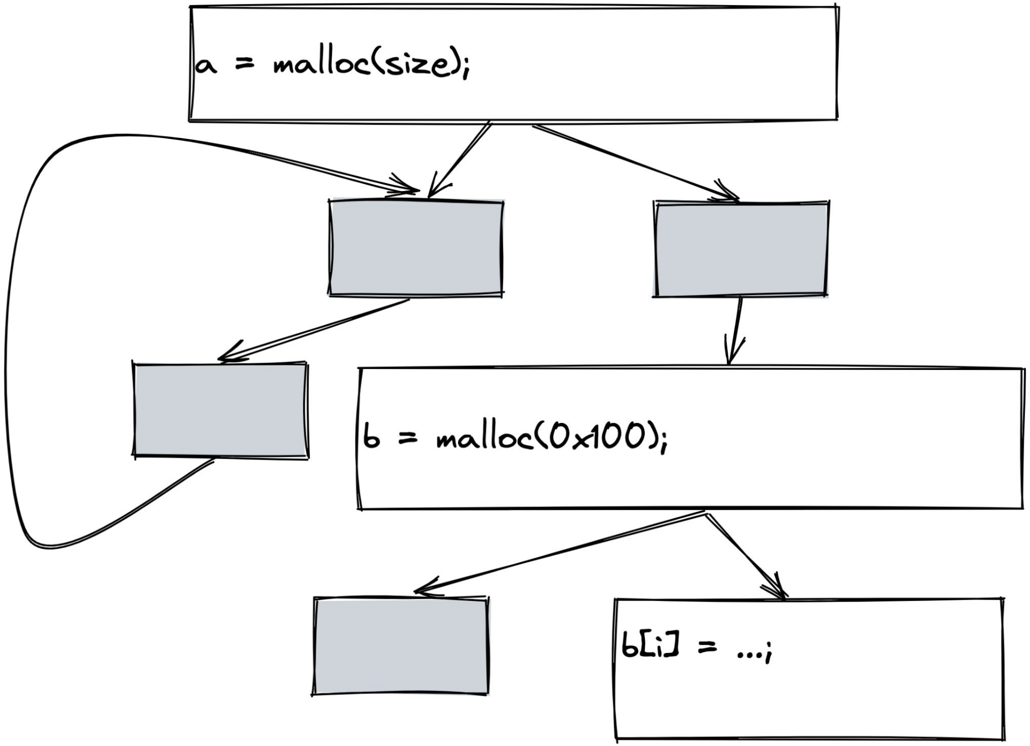
`fuzzalloc.so`

- Data-flow tracking is reduced to metadata management

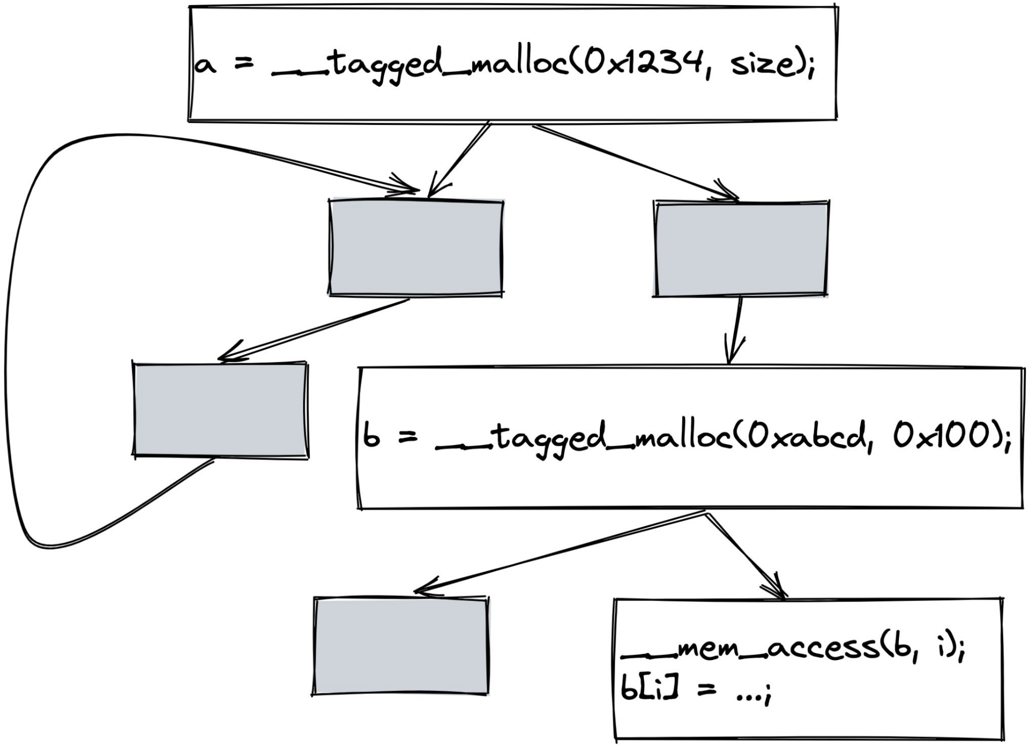
- Def site IDs are the metadata to retrieve at use site

Achieved via mid-fat pointers + custom memory allocator

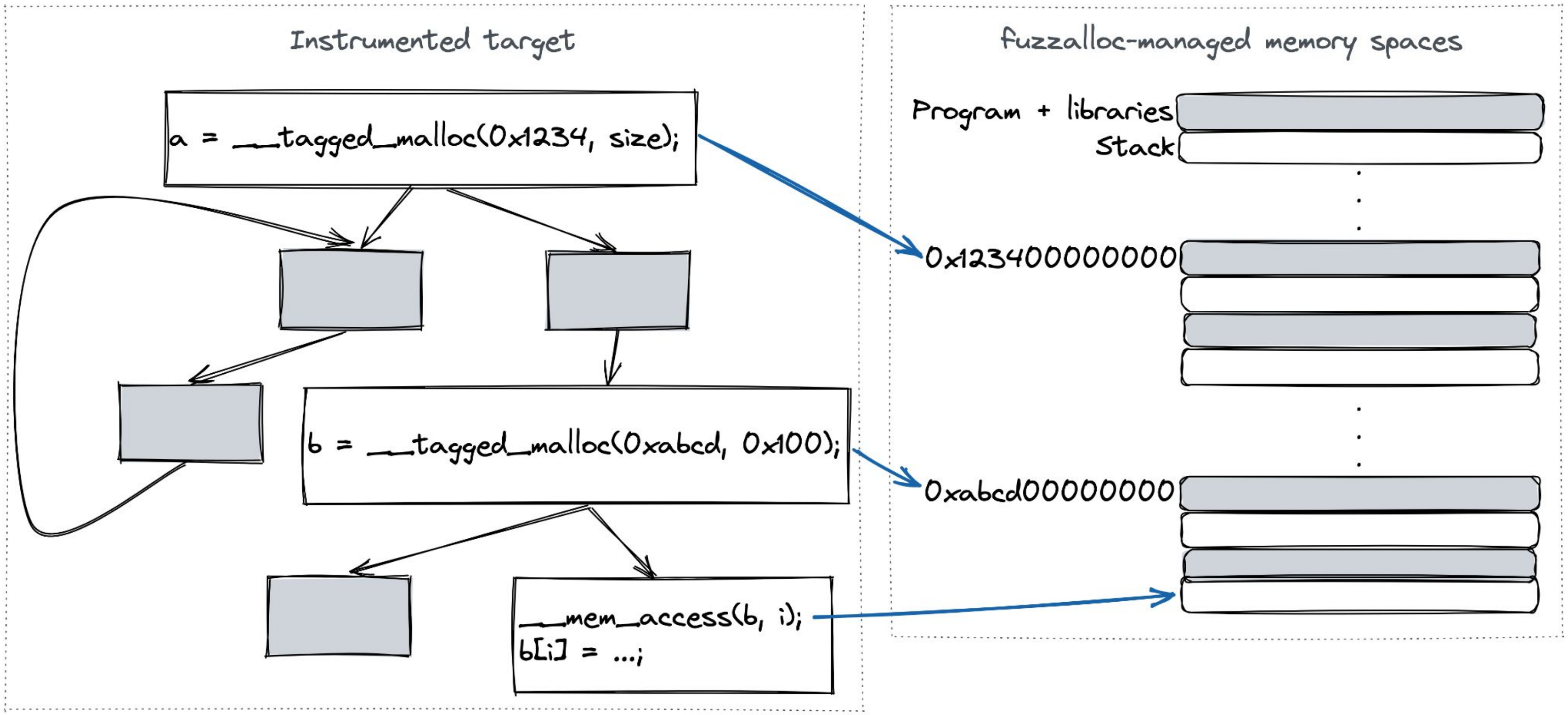
fuzzalloc.so



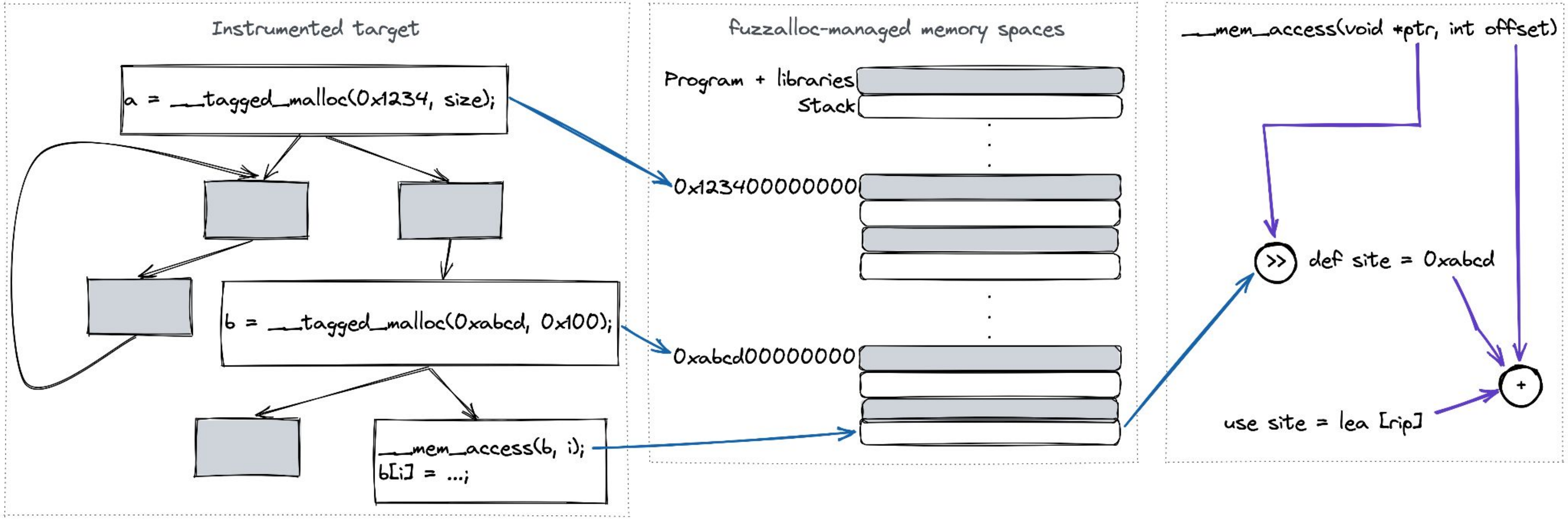
fuzzalloc.so



fuzzalloc.so



fuzzalloc.so



Requirements

1. Define “data-flow coverage”
2. Efficiently track data flows
3. Data flows → fuzzer coverage
4. Evaluate!

4. Evaluate!

Targets

- Magma benchmark suite (🐛)
- `jq` JSON parser (📈)

Fuzzers

- datAFLow (with different use site sensitivities)
- AFL++ (with/out cmplog)
- Angora

Australian National University DST GROUP EPFL

4. Evaluate!

Bug-finding results

- datAFLow found less bugs than other fuzzers

- Found two previously-undiscovered bugs
  - In Lua interpreter

4. Evaluate!

Code-coverage results

- AFL++ subsumed least-sensitive def/use coverage
- datAFLow performed slightly-better when more-sensitive metric used

4. Evaluate!

Evaluation plan

- Improve performance
- Characterizing target programs
- Quantifying data-flow coverage
- Fuzz!

4. Evaluate!

Research Qs

- RQ1: Can we characterize target programs for control- vs. data-flow coverage?
- RQ2: How can we quantify data-flow coverage?
- RQ3: Is def-use chain fuzzing effective?

FIN

- Paper @ https://www.ndss-symposium.org/wp-content/uploads/fuzzing2022_23001_paper.pdf
- Code @ https://github.com/HexHive/datAFLow

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

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

Mathias Payer
EPFL
mathias.payer@nebelwelt.net

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

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

We present DATAFLOW, a greybox fuzzer driven by lightweight data-flow profiling. Whereas control-flow edges represent the order of operations in a program, data-flow edges capture the dependencies between operations that produce data values and the operations that consume them: indeed, there may be no control dependence between those operations. As such, data-flow coverage captures behaviors not visible as control flow and intuitively discovers more or different bugs. Moreover, we establish a framework for reasoning about data-flow coverage, allowing the computational cost of exploration to be balanced with precision.

We perform a preliminary evaluation of DATAFLOW, comparing fuzzers driven by control flow, taint analysis (both approximate and exact), and data flow. Our initial results suggest that, so far, pure coverage remains the best coverage metric for uncovering bugs in most targets we fuzzed (72 % of them). However, data-flow coverage does show promise in targets where control flow is decoupled from semantics (e.g., parsers). Further evaluation and analysis on a wider range of targets is required.

I. INTRODUCTION

Fuzzers are an indispensable tool in the software-testing toolbox. The idea of fuzzing—to test a target program by subjecting it to a large number of randomly-generated inputs—can be traced back to an assignment in a graduate Advanced Operating Systems class [1]. These fuzzers were relatively primitive (compared to a modern fuzzer): they simply fed a randomly-generated input to the target, failing the test if the target crashed or hung. They did not model program or input structure, and could only observe the input/output behavior of the target. In contrast, modern fuzzers use sophisticated program analyses to model program and input structure, and continuously gather dynamic information about the target.

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

Leveraging dynamic information drives fuzzer efficiency. For example, *coverage-guided greybox fuzzers*—perhaps the most widely-used class of fuzzer—track code paths executed by the target.[1] This allows the fuzzer to focus its mutations on inputs reaching new code. Intuitively, a fuzzer cannot find bugs in code never executed, so maximizing the amount of code executed should maximize the number of bugs found. Code coverage serves as an approximation of program behavior, and expanding code coverage implies exploring program behaviors.

Coverage-guided greybox fuzzers are now pervasive. Their success [2] can be attributed to one fuzzer in particular: American Fuzzy Lop (AFL) [3]. AFL is a greybox fuzzer that uses lightweight instrumentation to track edges covered in the target's control-flow graph (CFG). A large body of research has built on AFL [4–12]. While improvements have been made, most fuzzers still default to edge coverage as an approximation of program behavior. *Is this the best we can do?*

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

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

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

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