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.* 1, 1, Article 1 (January 2023), 31 pages. https://doi.org/10.1145/3587156

1 INTRODUCTION

Fuzzers are an indispensable item in the software-testing toolbox. The idea of fuzzing—to test a target program by subjecting it to a large number of inputs—can be traced back to an assignment in a graduate Advanced Operating Systems class [49]. 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 only observed

*Also with Defence Science and Technology Group.

Authors' addresses: Adrian Herrera, adrian.herrera@anu.edu.au, School of Computing, Australian National University, Canberra, ACT, Australia; Mathias Payer, mathias.payer@nebelwelt.net, School of Computer and Communication Sciences, École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland; Antony L. Hosking, antony.hosking@anu.edu.au, School of Computing, Australian National University, Canberra, ACT, Australia.

1049-331X/2023/1-ART1
https://doi.org/10.1145/3587156

the input/output behavior of the target. In contrast, modern fuzzers use sophisticated program analysis to model program and input structure, and continuously gather dynamic information about the target.

Exploiting this 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 that reach new code. Intuitively, a fuzzer can only find bugs in code it executes, so maximizing the amount of code covered should implicitly maximize the number of bugs found. Code coverage also approximates program behavior: expanding code coverage implies exploring new or different program behaviors.

Coverage-guided greybox fuzzers are now pervasive. Their success [56] is attributable to one greybox fuzzer in particular: American Fuzzy Lop (AFL) [73]. AFL uses lightweight instrumentation to track edges covered in the target's control-flow graph (CFG). A large body of research has built on AFL [3, 7, 8, 16, 21, 24, 33, 43, 70], and 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`) [71]. Such targets require *data-flow* coverage [11, 22, 27, 34, 55, 62, 71] to accurately capture program behavior. 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 [55]; indeed, there may be no control dependence between definition and use sites (see Section 3 for details).

In fuzzing, data flow typically takes the form of *dynamic taint analysis* (DTA), in which the target's input data is *tainted* at its definition site and tracked as it is accessed and used at run time. Unfortunately, accurate DTA is difficult to achieve and expensive to compute (e.g., prior work has found DTA is expensive [23, 60] and its accuracy highly variable across implementations [15, 60]). Moreover, several real-world programs fail to compile under DTA, increasing deployability concerns. Thus, most widely-deployed greybox fuzzers (e.g., AFL [73], libFuzzer [42], and honggfuzz [65]) eschew DTA in favor of higher execution rates.

While lightweight alternatives to DTA exist (e.g., Redqueen [5], GreyOne [23]), the full potential of control- vs. data-flow fuzzer coverage metrics remains to be thoroughly explored. To support this exploration, we present *datAFLow*, a greybox fuzzer that tracks a program's data flow (rather than control flow) without requiring DTA. Notably, our work performs data-flow analysis inline with the execution, directly guiding the fuzzer. This is in contrast to prior work (e.g., GreyOne), which performed *post hoc* trace analysis in an attempt to infer or approximate data flow. Unlike DTA, which strives for accuracy, we take inspiration from popular greybox fuzzers (e.g., AFL) and embrace some imprecision to reduce overhead and thus maximize fuzzing throughput.

We perform a large-scale evaluation (> 3 CPU-yr) of datAFLow's effectiveness, comparing it against three state-of-the-art fuzzers. Our evaluation on the Magma benchmark [26] shows that, while generally outperformed by control-flow-guided fuzzers, datAFLow uncovers bugs that these fuzzers fail to find. This is because data-flow coverage revealed states in the target not visible in the CFG. Curiously, this is despite the control-flow-guided fuzzers achieving more control-*and* data-flow coverage (on targets previously identified as being amenable to data-flow-guided fuzzing [47]). We determined the run-time costs of data flow tracking to be the root cause of this result; intuitively, the cost of data-flow-guided fuzzing is not recoverable in targets where data flow

[1]The original fuzzer of Miller et al. [49] is now known as a *blackbox* fuzzer (because it has no knowledge of the target's internals).

mostly follows control flow. We encourage the community to continue exploring data-flow-guided fuzzing to maximize bug discovery.

Summary of Contributions

We contribute the following, making our work available at https://github.com/HexHive/datAFLow:

(1) A framework for reasoning about and constructing data-flow coverage metrics for greybox fuzzing (Section 4).
(2) A new data-flow-guided fuzzer, datAFLow, to explore data flow in a target program with low overhead (Section 5).
(3) An extensive evaluation and comparison of representative fuzzers guided by control flow, taint analysis, and data flow (Section 6).

2 BACKGROUND & RELATED WORK

2.1 Fuzzing

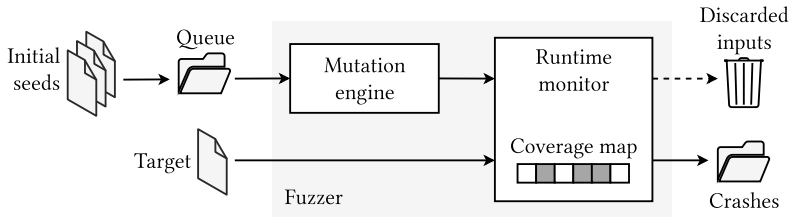


Fig. 1. High-level overview of a typical greybox fuzzer.

Fuzzing is a dynamic analysis for finding bugs in a target program by subjecting it to random inputs. Coverage-guided greybox fuzzers—the most popular class of fuzzer—do not just blindly feed these random inputs into the target. Instead, they use a feedback loop based on a *coverage metric*. This feedback loop guides the fuzzer toward generating inputs that explore new behaviors of the target (as determined by the coverage metric).

Fig. 1 illustrates the architecture of a typical coverage-guided greybox fuzzer. The user provides (a) an instrumented program called the fuzzing *target*, and (b) an optional set of starting inputs called *seeds* (an *empty seed* is used if not provided [29]).

The fuzzer places the seeds into a queue and then: (i) selects a seed from the queue; (ii) mutates the seed (via bit-flipping, value substitution, etc.); (iii) executes the target with the mutated seed, storing coverage (or an approximation thereof) in a coverage map; and (iv) detects crashes and newly-discovered coverage in the target (saving the former for offline analysis and discarding the seed, or in the latter returning the seed into the queue for further exploration by mutation). This process repeats until the *residual risk* of a missed bug falls beneath a suitable threshold [6].

2.2 Data-flow Analysis

Data-flow analysis typically refers to a collection of techniques for reasoning about the run-time *flow* of values in a program. These techniques can be static—such as those used by compilers for liveness analysis, constant propagation, and reaching definition analysis—or dynamic. Dynamic data-flow analysis is an approach adopted in software testing for reasoning about the sequence of actions performed on data (i.e., program variables) at run time [13, 31, 32]. These actions are typically analyzed in terms of the interactions between a variable's *definition*—or *def* site—and

Table 1. Survey of related fuzzers. CS = context sensitive. IJON requires manual analysis to identify variables to track. CONFETTI uses exact and approximate DTA to provide both global and local hints, respectively.

	Angora	GREYONE	CONFETTI	IJON	INVSCOV	DDFUZZ	GraphFuzz	DATAFLOW
Feedback	CS edge	Edge	Edge	Edge + value	Edge + value	Edge + DDG	Edge	*def-use*
Manual analysis	✗	✗	✗	✓	✗	✗	✗	✓
"Exact" DTA	✓	✗	✓	✗	✗	✗	✗	✗
"Appx." DTA	✗	✓	✓	✗	✗	✗	✗	✗

how that variable is *used* at one or more *use* sites [55, 62]. Data flows between these definition and usage sites are known as *def-use chains*.

Empirical studies have shown the effectiveness of data-flow coverage metrics over control-flow metrics when developing software tests [22, 27, 34, 55, 62] and comparing program executions [64]. However, to the best of our knowledge, these data-flow techniques have not yet been explored by the fuzzing community.

2.3 Related Work

Fuzzing is an active area of research. Consequently, we focus on recent work related to *coverage metrics* for fuzzing. We summarize the fuzzers discussed below in Table 1, comparing them to our DATAFLOW fuzzer (described further in Sections 4 and 5).

The most popular fuzzers are those guided by code coverage [44]. Typically, this code coverage is based on a target's control-flow graph (CFG) and is measured at either basic block or edge granularities. While edge coverage is typically considered more sensitive than basic-block coverage, as we shall see in Section 3, it is not without its issues. Indeed, TortoiseFuzz showed that basic-block coverage is effective when paired with other coverage metrics that increase sensitivity (e.g., function call and loop coverage) [70].

To improve mutation precision, some fuzzers use dynamic taint analysis (DTA) to track input bytes. The fuzzer uses this information to infer which bytes to mutate. Unfortunately, DTA suffers from accuracy and performance issues [15, 36, 60], limiting deployment. To overcome performance issues, Angora [12] amortizes DTA cost by limiting its application to once per input (over many mutations) [12]. Other fuzzers avoid DTA in favor of *approximate taint tracking*; e.g., REDQUEEN [5] uses input-to-state correspondence, based on the idea that "*parts of the input directly correspond to the memory or registers at run time*". Similarly, GREYONE [23] infers taint by monitoring the value of variables as input bytes are mutated, while CONFETTI [39] uses concolic execution to overcome missing data-flow relationships and implicit flows (see Section 3).

Alternatives to code coverage metrics are also being explored. MEMFUZZ [16] and AFL-Sensitive [69] augment edge coverage with memory access information. In theory, this approach allows the fuzzer to distinguish between executions that cannot be distinguished by control flow alone. In practice, this approach leads to saturation of the fuzzer's coverage map.

To give more say to the human analyst (e.g., to prevent coverage map saturation), IJON [4] introduced an annotation mechanism for tracking key state variables in the coverage map (e.g., Mario's x and y coordinates in the game *Super Mario Bros.*). This approach overcame fuzzer roadblocks that automated approaches could not.

INVSCOV [20] augments code coverage with the value of and relationships between key program variables. These variables are based on *likely invariants* (i.e., invariants that hold for a set of dynamic traces but may not hold for all inputs); the violation of a likely invariant indicates "interesting" program behavior (and is recorded in the coverage map).

```
1 size_t max; // User specified
2 unsigned int i = 0, j = 0;
3 char *prime = malloc(max);
4 memset(prime, 1, sizeof(char) * max);
5
6 for (i = 2; i < max; ++i) {
7   if (prime[i]) {
8     for (j = i; i * j < max; ++j) {
9       prime[i * j] = 0;
10    }
11  }
12 }
```

Fig. 2. Motivating example. The Sieve of Eratosthenes for finding all prime numbers up to `max` value.

`DDFuzz` [47] also augments code coverage with data flows between program variables. Here, data flows are derived from the target's *data dependency graph* (DDG). DDGs describe the data flows between instructions in a program and are traditionally used by optimizing compilers [19]. Like INVSCOV, `DDFuzz` only considers a subset of program variables (to prevent state explosion and coverage-map saturation): variable *def* sites are restricted to `load` and `alloca` instructions in the LLVM intermediate representation (IR), while variable *use*s are restricted to `store` and `call` instructions. Further filtering is applied to discard data flows subsumed by edge coverage.

GraphFuzz [25] fuzzes library APIs by modeling sequences of executed functions as a data flow graph. Using a data flow graph and control-flow-based coverage feedback, GraphFuzz generates fuzzing harnesses that explore a greater range of API combinations.

Despite the body of work on fuzzer coverage metrics, *pure data flow* coverage remains an under-explored metric. This is likely due to the perceived run-time cost of measuring data flow [20, 69]. Nevertheless, we hypothesize lightweight data-flow tracking is possible. To this end, we introduce DATAFLOW, a data-flow-guided greybox fuzzer with a tunable sensitivity range.

3 MOTIVATING DATA-FLOW COVERAGE

A fuzzer's coverage metric should accurately capture/approximate program behavior with minimal run-time overheads. Here we discuss why control-flow-based metrics are insufficient to accurately capture program behavior, using Fig. 2 as a running example.

While basic block and edge coverage (the most pervasive coverage metrics in greybox fuzzers) are performant, they often provide a poor approximation of program behavior. This is because code coverage ultimately represents a static view of the target, whereas data-flow coverage more closely captures the target's run-time computations; i.e., *how input is consumed by the target*.

Fuzzers using *basic-block* coverage cannot differentiate between different orderings of the same blocks. This can be improved by using *edge* coverage, which allows the fuzzer to differentiate between a loop's forward and backward edges (such as the loops at Lines 6 and 8 in Fig. 2).

Unfortunately, edge coverage still loses important information about program behavior (e.g., greybox fuzzers rely on coverage information to decide which input mutations lead to new program behaviors). However, uncovering new behaviors can be highly inefficient because a fuzzer guided by code coverage alone cannot identify *which* mutated input bytes led to new program behavior.

Some fuzzers address this issue (i.e., determining which input bytes to mutate) by applying dynamic taint analysis (DTA). DTA improves mutation accuracy by tracking the subset of program values used as arguments to comparison operations. However, the effectiveness of DTA depends on its *taint policy*, which specifies the taint relationship between an instruction's input and output.

In Fig. 2, `max` is user-controlled (i.e., the user selects the maximum prime number) and is therefore the *taint source*. While `max` is read directly on Lines 3, 4, 6 and 8, it is `prime` accesses that most accurately captures the program behavior. From a bug-finding perspective, `prime` accesses are also the most likely source of memory-safety vulnerabilities.

Given `max` determines the size of `prime` (via `malloc`, Line 3), taint may propagate to `prime`. However, this is an *implicit flow* that the taint policy may not capture. For example, compiler-based DTA—e.g., LLVM's DataFlowSanitizer (DFSan) [66]—cannot track taint outside uninstrumented code (e.g., through functions provided by external libraries, such as `malloc`). Ensuring taint is accurately tracked in uninstrumented code requires significant manual effort. Moreover, prior work has shown this accuracy to be highly variable and dependent on the DTA implementation (e.g., due to incorrect taint policies and unsupported instructions) [15].

DTA is also expensive. She et al. [60] found none of their targets completed within a 24 h period when run with the Triton DTA tool. We also found that Angora's compiler-based DTA (built on DFSan) exhibited a run-time overhead of 32.79× over the same uninstrumented code from the SPEC CPU2006 benchmark suite (see Section 6.2). This is notable because prior work has found DFSan to be one of the more performant DTA frameworks (due to compile time—rather than run time—instrumentation) [60].

Given the disadvantages of DTA (low accuracy and high cost), we propose an alternative approach: tracking data flows between `prime`'s *def* (Line 3) and *use* sites (Lines 7 and 9). The following section describes our data-flow tracking approach.

4 DESIGN

A greybox fuzzer should maintain *accurate* coverage information without negatively impacting *performance*. These requirements exist irrespective of the coverage metric used. With this in mind, we describe: (i) a theoretical foundation for constructing data-flow-based coverage metrics; (ii) how datAFLow incorporates these observations; and (iii) the implementation of a datAFLow prototype.

4.1 Coverage Sensitivity

Based on Section 2.2, we define data-flow coverage as follows:

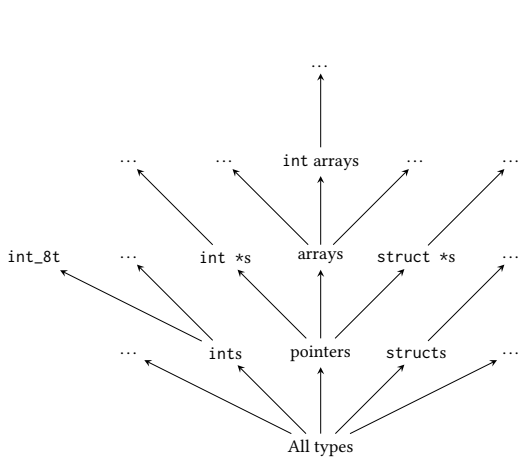
> *Data-flow coverage is the tracking of def-use chains executed at run time.*

This definition allows us to explore data-flow-based coverage metrics with different *sensitivities* [57, 69]. We follow the program analysis literature and define sensitivity as a coverage metric's ability to discriminate between a set of program behaviors [37]. In fuzzing, a coverage metric's sensitivity is its ability to preserve a chain of mutated test cases until they trigger a bug [69]. Different sensitivities allow us to balance efficacy and performance: more sensitive metrics incur higher performance penalties (e.g., edge coverage sensitivity is increased by incorporating function call context [12]. However, this requires additional instrumentation, increasing run-time overhead [57]).

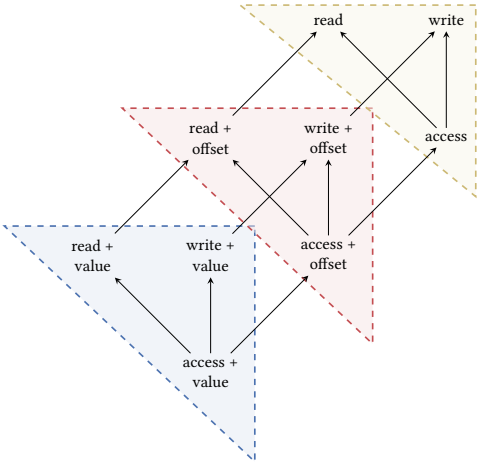
Like traditional data-flow analysis (Section 2.2), our data-flow coverage metric requires identifying variable *def* and *use* sites. Following Horgan and London [31], we define a data-flow variable *def* site as a name referring to storage allocated statically (e.g., storage class static, global) or automatically (i.e., local to a procedure). We deviate from this definition by: (i) including calls to dynamic memory allocation routines (e.g., `malloc`); and (ii) excluding reallocations/reassignments that would traditionally *kill* a definition. Instead, *def*s are only killed when they (a) go out of scope (e.g., a local variable in a returning procedure), or (b) are explicitly deallocated (e.g., via `free`). Consequently, a *use* site includes both reads/writes from/to a *def* site. We deviate from the classic definition to ensure scalability: the difficulties of scaling data-flow analyses on real-world

programs are well known [11, 27, 62]. We believe reducing precision by not killing definitions (when assigning a new value to a variable) is a suitable trade-off to maintain scalability.

Once we identify *def* and *use* sites, DATAFLOW instruments these sites (using compiler-based instrumentation, discussed in Section 5) so *def-use* chains can be tracked at run time. However, exactly which *def-use* sites are instrumented (and hence which are tracked) depends on the required sensitivity. Inspired by Wang et al. [69], this leads us to define a pair of sensitivity lattices—one for *def* sites and another for *use* sites, in Fig. 3—that can be composed to achieve the desired overall sensitivity (we discuss related threats to validity in Section 5.4).



(a) *Def* site sensitivity lattice. Variables are partitioned based on their type.



(b) *Use* site sensitivity lattice. Variables are partitioned based on their access (e.g., read, write) and what information about the access is recorded (e.g., offset, value).

Fig. 3. *Def* and *use* site sensitivity lattices. The sensitivity of coverage metrics increases toward the bottom.

4.1.1 Def Site Sensitivity. Complete data-flow coverage requires identifying and instrumenting *all* variable *def* sites. Unfortunately, the overhead to achieve this level of sensitivity is prohibitively expensive [10]. Therefore, a method for identifying (and hence instrumenting) a subset of important program variables is required. Ideally, this would be an (almost entirely) automated process, reducing the developer burden on the user.

One approach is to partition *def* sites by *type* and restrict instrumentation to *def* sites of a given type (or type set). Figure 3a shows the sensitivity lattice for this type-based partitioning.

Partitioning *def* sites by type has several advantages. For example, instrumenting array variables focuses the fuzzer on memory-safety vulnerabilities. Similarly, tracking the data flow of structs may allow for the discovery of type confusion vulnerabilities [35, 61]. Type-based partitioning requires some upfront knowledge of the target to ensure meaningful variables are tracked at run time. For example, the fuzzer may miss important program behaviors (and hence bugs) if "uninteresting" variables are tracked (e.g., `max` in Fig. 2).

> Tracking *all* data flows is prohibitively expensive. Identification (and instrumentation) of only important variables is required.

4.1.2 Use Site Sensitivity. Figure 3b shows the *use* site sensitivity lattice. Variables are either read from or written to (i.e., "accessed"). Variable accesses are strictly more sensitive than just writes or reads on their own. The simplest and least sensitive metrics only track when a variable is accessed (shown at the top of the lattice).

Conversely, the most sensitive data-flow coverage metrics are ones that track not only *when* a particular variable is accessed, but the *value* of that variable when accessed. For example, considering Line 9 in Fig. 6, this is the difference between writing to `prime` and assigning it the value `0`. The latter is akin to traditional data-flow testing, which focuses on the values that variables take at run time [55, 62], and is similar to GreyOne, which monitors (a subset of) program variables and their values to infer taint [23]. Depending on the *def* site sensitivity, this approach will quickly saturate the fuzzer's coverage map (due to the path collision problem [24]); a middle ground between this overly sensitive approach and simple accesses is required.

We achieve this middle ground by incorporating more fine-grained spatial information into a variable's *use*. This is particularly useful when *def* sites include arrays and/or structs (e.g., Line 9 in Fig. 2), as *def-use* chains are now differentiated by the *offset* at which an array/struct is accessed (analogous to a field-sensitive static analysis).

> Information at different granularity is recorded at *use* sites. Care is required when recording more precise information to ensure the coverage map does not saturate, clogging the fuzzing queue.

4.1.3 Composing Sensitivity Lattices. Different *def-use* sensitivities can be composed to track data flow at different granularities. We reuse the code in Fig. 2 to illustrate this. Given the *def* sensitivity lattice in Fig. 3a, either: (i) all three variables (`prime`, `i`, and `j`); (ii) the indices `i` and `j`; or (iii) only the `prime` array are instrumented (and hence tracked). Here we restrict *def* site instrumentation to array variables. Consequently, only `prime` is tracked. This leads to varying *def-use* chains depending on the *use* site sensitivity.

Simple access. The yellow region in Fig. 3b. Tracks when `prime` is accessed (Lines 7 and 9 in Fig. 2). This results in two *def-use* chains: Line 3 $\rightsquigarrow$ Line 7 and Line 3 $\rightsquigarrow$ Line 9. This is equivalent to basic block coverage (per Section 2.1): to reach the *use* at Line 9 requires the execution of all basic blocks in the CFG. Like block coverage, this provides a poor approximation of program behavior (as information about the loop and how it affects data is lost).

Access with offset. The red region in Fig. 3b. Tracks when `prime` is accessed along with the offsets where `prime` is accessed (indices `i` and `j`). This provides a more complete view of how `prime` is used with negligible overhead. This is similar to MemFuzz's approach, which incorporates memory accesses into code coverage [16]. This results in $2 \times (\texttt{max} - 2)$ *def-use* chains: one for every read/write at each index where `prime` is read from/written to.

Access with value. The blue region in Fig. 3b. Tracks when `prime` is accessed along with the values (being read/written) during these accesses. This is the most sensitive *use* site coverage metric and achieves the goal of traditional data-flow coverage: associate values with variables, and how these associations can affect the execution of the target [55]. This is similar to GreyOne's "taint inference", which looks at the value of variables used in path constraints [23].

Again, this level of sensitivity results in $2 \times (\texttt{max} - 2)$ *def-use* chains. Here, `prime`'s value range is fully deterministic. However, these values will typically depend on user input, resulting in rapid saturation of the fuzzer's coverage map.

Sensitivity lattice composition must balance efficacy and performance: too precise, and the fuzzer's coverage map will saturate, reducing throughput.

By composing *def* and *use* sensitivity lattices, we realize a variety of data-flow-based coverage metrics. We do this in our fuzzer, datAFLow, described in the following sections.

5 IMPLEMENTATION

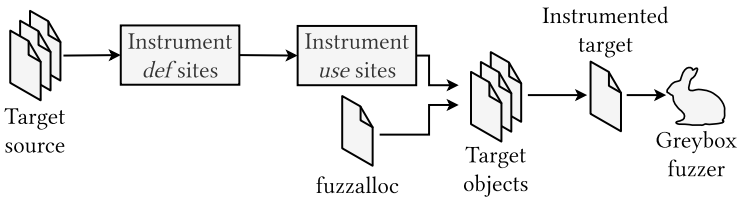


Fig. 4. High-level overview of datAFLow.

Figure 4 depicts datAFLow's high-level architecture, including: (i) compiler instrumentation (built on LLVM v12) for capturing *def*-*use* sites at the desired sensitivity (Sections 5.1 and 5.2); and (ii) a run-time library for feeding data-flow information to the fuzzing engine (Section 5.3).

Our architecture is agnostic to the underlying fuzzer; the instrumented target produced by the compiler (and linked with the `fuzzalloc` run-time library) can be executed by any AFL-based fuzzer (i.e., any fuzzer using an AFL-style coverage map). However, instead of recording and tracking control-flow coverage, the fuzzer's coverage map tracks data-flow coverage.

5.1 *Def*-*Use* Site Identification

We must first identify *def* and *use* sites so that data flows between these sites can be tracked. Per Section 4.1, *def* site selection impacts coverage sensitivity: more instrumented *def* sites leads to more complete data-flow coverage. We implement several *def* site instrumentation schemes based on the type-based partitioning described in Section 4.1.1.

We make the following assumptions during *def*-*use* site identification. First, we assume debug metadata is available in the LLVM IR. We use this metadata to identify and limit variable *def* sites to source-level variables. Second, we assume tracked variables are accessed via memory references (i.e., `load`/`store` instructions), rather than registers. This is automatic for most composite types (e.g., arrays). For primitive types (e.g., integers), this requires demoting registers to memory references (via LLVM's `reg2mem` pass).

The first assumption reduces the number of potential data flows and is adopted from prior work [20, 47]. The second assumption limits *use* sites to memory access instructions, simplifying instrumentation. We apply existing LLVM transforms to limit *use* sites to two instructions: `load`s and `store`s.[2] Exactly which instructions we instrument depends on the *use* sensitivity required (configured at compile time). We describe our instrumentation in Section 5.2.

5.2 *Def*-*Use* Tracking

We reduce the run-time tracking of *def*-*use* chains to a metadata management problem. Here, *def* site identifiers are the metadata requiring efficient retrieval at *use* sites. Inspired by AFL's

[2]We lower atomic memory intrinsics and expand `llvm.mem*` intrinsics so we can focus on `load`/`store` instructions (both of which are trivial to identify and hence instrument).

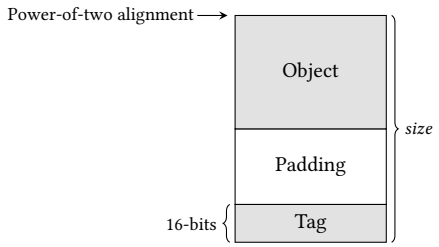


Fig. 5. PAMD's approach for inline metadata. "Object" is aligned to a power-of-two boundary and "padding" is inserted to ensure *size* is a power-of-two.

approach for tracking edge coverage—where basic blocks (in the LLVM IR) are statically assigned a random 16-bit integer—we statically "tag" *def* sites (again, in the LLVM IR) with a random 16-bit integer (Section 5.2.1). This tag is then propagated to *use* sites, where it is retrieved and used to construct a *def-use* chain (Section 5.3).

5.2.1 Def Site Instrumentation. We adopt *Padding Area MetaData* (PAMD) [41] for tracking *def-use* chains. PAMD extends *baggy bounds checking*, a technique proposed by Ding et al. [17] for protecting C and C++ code against buffer overruns. PAMD attaches inline metadata to memory objects (hence our assumption that tracked variables are accessed via memory references; Section 5.1) and provides constant-time lookup of this metadata. This lookup occurs via the "baggy bounds table", which stores the binary logarithm of an object's size and alignment (denoted e). Once e is retrieved from the baggy bounds table, the *base* and *size* of an object pointed to by p is computed using:

$$base = p \;\&\; \sim(2^e - 1) \tag{1}$$

$$size = 2^e \tag{2}$$

Equations (1) and (2) require an object's size and alignment to be a power-of-two. To meet this requirement, PAMD pads static objects (i.e., stack and global variables) before attaching the *def* site tag. Figure 5 illustrates this process. For example, given a 4-byte object, then $size = 8$, $e = 3$ (the binary logarithm of *size*), and two bytes of padding is inserted before the tag.

Objects whose padding or overall size becomes too large for static allocation are "*heapified*" (i.e., move to the heap). We adopt CCured's [51] approach to heapify objects. For heap-allocated objects (including heapified objects), calls to `malloc`, `calloc`, and `realloc` are replaced with tagged versions (e.g., `__bb_malloc`) accepting the 16-bit tag as an additional argument. Figures 6 and 7 demonstrate our *def-use* instrumentation.

Figure 6 shows the (un)instrumented LLVM IR for the Sieve of Eratosthenes (Fig. 2). We focus our *def* site instrumentation on the dynamically-allocated `prime` array (Line 2 in Fig. 6a). DATAFLOW tags this *def* site with the identifier `1337` (Line 2 in Fig. 6b). This tagging occurs by replacing `malloc` with `__bb_malloc`, which also registers the allocation in the baggy bounds table.

In comparison, Fig. 7 demonstrates the instrumentation of a stack *def* site. The original code (Fig. 7a) statically allocates an 8-byte buffer `buf`, filling it via a call to `read`. The buffer's second element is later accessed. During compilation, DATAFLOW resizes `buf` to meet PAMD's object size requirement. Here, six bytes of padding is inserted before the two-byte tag (Line 2 in Fig. 7b). This *def* site is tagged with the identifier `1102` (Line 4) and registered in the baggy bounds table (Line 8).

5.2.2 Use Site Instrumentation. Per Section 5.1, *use* sites are limited to `load` and `store` instructions in the LLVM IR (e.g., Line 7 in Fig. 6a and Line 10 in Fig. 7a). We instrument these instructions with a call to `__hash_def_use`, which retrieves the object's size from the baggy bounds table and uses

```
; char *prime = malloc(max)
%prime = call i8* @malloc(i64 %max)

; prime[i * j] = 0
%idx = mul i32 %i, i32 %j
%prime_idx = getelementptr inbounds i8, i8* %prime, i64 %idx
store i8 0, %prime_idx
```

(a) Original code.

```
; Dynamically allocate prime and register the allocation in the baggy bounds
    table
%prime = call i8* @__bb_malloc(i16 1337, i64 %max)

; Instrument the use of prime with a call to __hash_def_use
%idx = mul i32 %i, i32 %j
%prime_idx = getelementptr inbounds i8, i8* %prime, i64 %idx
call @__hash_def_use(i16 4242, i8* %prime_idx, i64 1)
store i8 0, %idx
```

(b) Instrumented code.

Fig. 6. The Sieve of Eratosthenes. The array `prime` is dynamically allocated. DatAFLow replaces this allocation with a call to `__bb_malloc` and registers this allocation in the baggy bounds table. The *use* site is instrumented with a call to `__hash_def_use`.

this size to retrieve the *def* tag. The size is also used to determine the offset at which an object is accessed (enabling the *access with offset* sensitivity described in Section 4.1.3). Like *def* sites, *use* sites are tagged at compile time with a randomly-generated identifier (e.g., `4242` at Line 7 in Fig. 6b and `1234` at Line 17 in Fig. 7b). Finally, we leverage several techniques from AddressSanitizer (ASan) [59] to limit the number of *use* instrumentation sites, thereby reducing overhead without sacrificing precision. We describe the internals of `__hash_def_use`, and how it integrates with the fuzzer, in the following section.

5.3 Fuzzer Integration

The `__hash_def_use` function constructs a *def-use* chain by hashing together the *def* and *use* sites. This hash is used as a lookup into the fuzzer's coverage map to guide the fuzzer toward discovering new data flows. This is analogous to AFL tracing edges to discover new control flow paths. Consequently, we leverage techniques used by traditional greybox fuzzers (e.g., compact bitmaps) to efficiently record data-flow coverage [44].

In particular, we use *coarse* data-flow coverage metrics—*def-use* chain hit counts stored in a compact bitmap—to achieve efficient fuzzing. While these techniques result in path collisions [24], we are willing to tolerate such imprecision to limit overhead costs. Coarse coverage metrics also lower implementation costs, enabling the reuse of existing fuzzing engines (here, AFL++ [21]).

```
; char buf[8]
%buf = alloca [8 x i8], align 1

; read(0, buf, 8)
%1 = getelementptr [8 x i8]* %buf, i64 0, i64 0
call i64 @read(i32 0, i8* %1, i64 8)

; a = buf[1]
%idx = getelementptr [8 x i8]* %buf, i64 0, i64 1
%a = load i8, i8* %idx
```

(a) Original code.

```
; Allocate buf and assign it the tag 1102 (after 6 bytes of padding)
%buf = alloca <{ [8 x i8], [6 x i8], i16 }>, align 16
%tag_idx = getelementptr <{ [8 x i8], [6 x i8], i16 }>* %buf, i32 0, i32 2
store 1102, i16* %tag_idx

; Register the allocation in the baggy bounds table
%buf_cast = bitcast %buf to i8*
call void @__bb_register(i8* %buf_cast, i64 16)

; Ignore the padding and tag when calling read
%1 = getelementptr <{ [8 x i8], [6 x i8], i16 }>* %buf, i32 0, i32 0
%2 = getelementptr [8 x i8]* %1, i64 0, i64 0
call i64 @read(i32 0, i8* %2, i64 8)

; Instrument the use site with __hash_def_use
%idx = getelementptr [8 x i8], [8 x i8]* %1, i64 0, i64 1
call i64 @__hash_def_use(i16 1234, i8* %idx, i64 1)
%a = load i8, i8* %idx
```

(b) Instrumented code.

Fig. 7. Example instrumentation of a stack variable *def*. An 8-byte buffer `buf` is allocated on the stack and filled by a call to `read`. The second byte in `buf` is later read.

We adopt AFL's hashing process for looking up data flows in the fuzzer's coverage map. By default, AFL represents a control-flow edge using the following hash algorithm:

$$\begin{aligned} i &\leftarrow l \oplus l_{\text{prev}} \\ l_{\text{prev}} &\leftarrow l \gg 1 \end{aligned} \tag{3}$$

Where l is a randomly-generated basic-block identifier (assigned at compile time) and l_{prev} is the identifier of the previously-executed block. AFL uses the result i as an index into the coverage map. Right-shifting l allows AFL to differentiate between different orders of two blocks. Our hash algorithm varies depending on the desired data-flow sensitivity (Section 4.1.3):

Simple access. Xor of the *def* and *use* site tags:

$$i \leftarrow def \oplus use \tag{4}$$

Access with offset. The *def* site tag, *use* site tag, and the offset being accessed. The offset (e.g., array index, struct offset) is computed by subtracting the base address—found using Eq. (1)—from pointer p. We compute the hash as:

$$i \leftarrow def \oplus (use + \text{offset}) \tag{5}$$

Access with value. The *def* site tag, *use* site tag, the offset, and the value accessed. The *def*/*use* tags and offset are left-shifted to allow room for the value hash and reduce collisions. The accessed value is divided into single-byte chunks $\{v_0, v_1, \ldots\}$ that are hashed into the *def*-*use* chain:

$$i \leftarrow (def \oplus (use + \text{offset}) \ll 2) \oplus (v_0 \oplus v_1 \oplus \ldots) \tag{6}$$

This is implemented as a loop, resulting in a double load of the accessed object (in addition to the `load` in the original code). We implement the `__hash_def_use` function so that uninstrumented data flows (i.e., those without an entry in the baggy bounds table) are bucketed in their own coverage map entry.

5.4 Threats to Validity

5.4.1 Def Site Selection. Our *def* site selection approach (Section 4.1.1) is incomplete: important data flows may be missed if the appropriate *def* sites are not instrumented. Per our *def* site sensitivity lattice, our prototype focuses on composite types (i.e., arrays and structs) and eschews instrumenting primitive types (e.g., integers). While this approach may miss important data flows, we accept this trade-off, given (a) memory safety remains a key concern [50], and (b) the prohibitive run-time overheads when tracking all *def* sites.

5.4.2 Custom Memory Allocators. Identifying *def* sites is complicated because many applications do not directly call the standard allocation routines (e.g., `malloc`), but indirectly through a custom memory allocator. For example, standard memory allocation routines may be wrapped in other functions. These functions may then be indirectly called via global variables/aliases, stored and passed around in structs, or used as function arguments.

To address the challenge imposed by custom memory allocators and memory allocation patterns, DATAFLOW allows the user to specify wrapper functions to tag (in addition to the standard allocation routines). While DATAFLOW requires the user to find these wrappers manually, existing techniques [14] could assist in this process. We wrap these memory allocation routines within *trampoline* functions when their address is taken (e.g., stored in a global variable). Rather than a compile-time *def* site tag (which may not be statically computable), these trampolines revert to using the lower 16-bits of the PC as the *def* site tag. This approach avoids the need for expensive and imprecise static analysis (e.g., to track the access of memory allocators through global variables).

5.4.3 C++ Dynamic Memory Allocation. To simplify our instrumentation, we rewrite C++ `new` calls as `malloc` calls. However, this prevents us from handling any `std::bad_alloc` exceptions, meaning any failed allocations will cause a program crash (irrespective of any exception handlers in place). Such false negatives are removed by replaying crashing inputs through the original target.

5.4.4 Coverage Imprecision. Storing coarse coverage information in a compact bitmap is inherently inaccurate and incomplete [24]. While this may limit DATAFLOW's ability to discover and explore data flows, this limitation is not unique to DATAFLOW and affects many greybox fuzzers [3, 4, 12, 16, 20, 23, 33, 43, 47, 69, 70, 73].

6 EVALUATION

We perform an extensive evaluation (> 3 CPU-yr of fuzzing) to test the following hypothesis:

Table 2. Evaluated fuzzer configurations. Angora and DDFuzz use their default map sizes. AFL++'s LTO instrumentation does not require a fixed-size map.

Name	Map size (KB)	Description
A_{LTO}	–	AFL++ with LTO instrumentation
A_{CL}	–	AFL++ with LTO and CmpLog instrumentation
An	1,024	Angora
DD	64	DDFuzz
$D_{A/A}$	1,024	datAFLow with array *def*s and accessed *use*s
$D_{A/A+O}$	1,024	datAFLow with array *def*s and accessed offset *use*s
$D_{A/A+V}$	1,024	datAFLow with array *def*s and accessed value *use*s
$D_{A+S/A}$	1,024	datAFLow with array + struct *def*s and accessed *use*s
$D_{A+S/A+O}$	1,024	datAFLow with array + struct *def*s and accessed offset *use*s
$D_{A+S/A+V}$	1,024	datAFLow with array + struct *def*s and accessed value *use*s

Data-flow-guided fuzzing offers superior performance (over control-flow-guided fuzzers) on targets where control flow is decoupled from semantics.

Specifically, we answer the following research questions:

RQ 1 Is data-flow-guided fuzzing viable with minimal run-time overheads? (Section 6.2)
RQ 2 Does data-flow-guided fuzzing find more or different bugs? (Section 6.3)
RQ 3 Does data-flow-guided fuzzing expand more coverage? (Section 6.4)
RQ 4 Can we predict *a priori* the targets most amenable to data-flow-guided fuzzing? (Section 6.5)

6.1 Methodology

6.1.1 Fuzzer Selection. Our evaluation compares the performance of fuzzers using: (i) pure control-flow coverage; (ii) pure data-flow coverage; and (iii) exact and approximate DTA, combining control-flow coverage with data-flow tracking.

We select AFL++ as the pure control-flow-guided fuzzer because it is the current state-of-the-art coverage-guided greybox fuzzer. We configure AFL++ with: (i) link-time optimization (LTO) instrumentation, eliminating hash collisions; and (ii) with and without "CmpLog" instrumentation. CmpLog—inspired by Redqueen's input-to-state correspondence [5]—approximates DTA by capturing comparison operands. Similarly, we select Angora as an alternative control-flow-guided fuzzer (using context-sensitive edge coverage) that also incorporates exact DTA. Finally, we select DDFuzz as an alternative data-flow-guided fuzzer.

We configure datAFLow with: (i) two *def* site sensitivities: arrays only ("A") and arrays + structs ("A+S"); and (ii) three *use* site sensitivities: simple access ("A"), accessed offset ("O"), and accessed value ("V"). We use the notation "X/Y to refer to the composition of X *def* and Y *use* site sensitivities; e.g., "A/A" refers to array *def* and access *use* sites; "A+S/O" refers to arrays + structs *def* and accessed offset *use* sites. The evaluated fuzzers are summarized in Table 2.

6.1.2 Target Selection. We evaluate the ten fuzzers in Table 2 on the following targets. We fuzz 20 target programs in total.

SPEC CPU2006. The SPEC CPU benchmark suite [28] is an industry-standardized, CPU-intensive benchmark suite for stress-testing a system's processor, memory subsystem, and compiler. We use SPEC CPU2006 to answer RQ 1.

Magma. Unlike other fuzzing benchmarks (e.g., UniFuzz [40]), Magma [26] contains ground-truth bug knowledge. We exclude the *php* target because it failed to build with AFL++'s CmpLog instrumentation (failing with a segmentation fault). We use 15 Magma targets to answer RQ 2.

Table 3. DDFuzz target dataset.

Target	Driver	Command line	Commit hash
bison	`bison`	`@@ -o /dev/null`	`5555f4d`
pcre2	`pcre2test`	`@@ /dev/null`	`db53e40`
mir	`c2m`	`@@`	`852b1f2`
qbe	`qbe`	`@@`	`c8cd282`
faust	`faust`	`@@`	`13def69`

DDFuzz *dataset.* Mantovani et al. [47] select five targets—*bison*, *pcre2*, *mir*, *qbe*, and *faust*—they believe to contain a large number of data dependencies, and hence are amenable to data-flow-guided fuzzing. We use newer versions of these targets (because some did not compile on Ubuntu 20.04), shown in Table 3. We use these targets to answer RQ 3.

6.1.3 Experimental Setup. We conduct all experiments on an Ubuntu 20.04 AWS EC2 instance with a 48-core Intel® Xeon® Platinum 8275CL 3.0 GHz CPU and 92 GiB of RAM. Each fuzz run was conducted for 24 h and repeated five times (ensuring statistically sound results). All targets were bootstrapped with their provided seeds.[3] Finally, we (a) manually located and specified memory allocation functions for datAFLow to tag, and (b) used Angora's default behavior to discard taint when calling an external library.

6.2 Run-time Overheads (RQ 1)

Conventional wisdom assumes data-flow-based coverage metrics are too heavyweight, adversely affecting a fuzzer's performance by reducing its execution rate. We investigate the extent to which this assumption is true by isolating the effects of instrumentation overhead *outside* of a fuzzing environment. Per Section 6.1.2, we measure performance overheads on SPEC CPU2006.

Table 4 shows the overhead of all ten evaluated fuzzers on all 19 C and C++ targets in the SPEC CPU2006 v1.0 benchmark suite. We compare these measurements against a baseline without instrumentation (`clang` v12), calculating the geometric mean ("geomean") and 95 % bootstrap confidence intervals (CI) over three repeated iterations. The following results are omitted because they failed to build or run: AFL++ (LTO) *445.gobmk* triggered a run-time assertion; datAFLow (all configurations) *429.mcf* crashed with a run-time segmentation fault; and Angora *447.dealII*, *471.omentpp*, *473.astar*, and *483.xalancbmk* failed to link with DFSan's run-time library.

Per Section 3, Angora has a geomean overhead of 32.79×. This is particularly notable because previous work has found DFSan—the framework upon which Angora's taint tracking mode is built—to be one of the more performant DTA frameworks [60]. However, while this overhead is significantly higher than AFL++ (LTO) and AFL++ (CmpLog)—which have geomean overheads of 1.19× and 2.80×, respectively—it is important to recall Angora amortizes this cost over the lifetime of a fuzzing campaign by only tracking taint once on a given input over many mutations.

Of the six datAFLow configurations, A/A has the lowest overhead (10.69×), while A + S/V has the highest (15.01×). This is unsurprising, given the rolling hash approach used for the "access with value" *use* sensitivity (Section 5.3). Performance improvements are possible by specializing the hash function based on the *type* of value accessed (e.g., hashing a `uint64_t` or `float` value directly, rather than dividing it into single-byte chunks). Increasing the *def* site sensitivity to include structs added minimal overhead. However, this is target specific: the median number of tracked arrays (across the 12 SPEC CPU2006 targets) is 51, compared to 33 structs. This result may not generalize across targets where structs outnumber arrays.

[3]We contacted Mantovani et al. [47] to obtain their initial seed sets.

Table 4. SPEC CPU2006 overhead. Computed as the geomean (over three repeated iterations) relative to an uninstrumented benchmark (compiled with `clang` v12). The 95 % bootstrap CI is reported for the geomean across all targets (for a given fuzzer). The bootstrap CI is zero for individual targets and hence is omitted.

Target	Fuzzer (×)									
	A_{LTO}	A_{CL}	An	DD	$DF_{A/A}$	$DF_{A/O}$	$DF_{A/V}$	$DF_{A+S/A}$	$DF_{A+S/O}$	$DF_{A+S/V}$
400.perlbench	1.27	3.86	141.85	21.81	12.04	12.75	16.34	12.51	13.21	16.79
401.bzip2	1.26	2.17	25.83	2.54	7.75	8.53	11.12	7.69	8.49	11.09
403.gcc	1.30	3.40	21.19	3.45	19.53	21.22	26.07	19.58	21.18	26.20
429.mcf	1.12	2.46	12.08	1.52	✗	✗	✗	✗	✗	✗
445.gobmk	✗	2.48	23.41	5.26	6.99	7.51	9.48	6.92	7.44	9.73
456.hmmer	1.12	3.08	60.41	1.56	13.47	15.07	21.61	13.60	14.95	21.62
458.sjeng	1.21	4.36	29.69	4.44	7.57	8.13	10.05	7.54	8.01	10.32
462.libquantum	1.20	2.40	27.09	1.61	3.81	4.04	6.97	3.70	4.14	6.97
464.h264ref	1.19	1.82	41.01	1.88	100.63	109.40	134.10	100.96	109.68	134.58
471.omnetpp	1.06	2.02	✗	2.05	6.58	6.34	6.82	6.15	6.34	7.56
473.astar	1.13	2.19	✗	1.59	5.53	5.83	6.80	5.60	5.96	7.28
483.xalancbmk	1.29	5.04	✗	3.48	11.44	12.25	15.67	11.66	12.43	15.93
Geomean	1.19 ± 0.00	2.80 ± 0.01	32.79 ± 0.34	2.91 ± 0.03	10.69 ± 0.13	11.41 ± 0.18	14.64 ± 0.22	10.65 ± 0.16	11.47 ± 0.21	15.01 ± 0.21

Our results reflect those presented by Liu and Criswell [41] (e.g., *464.h264ref* has the highest run-time overhead in both the original and our works). However, there is a significant increase in our run-time overheads compared to the original PAMD implementation [41]. To validate our PAMD (re)implementation we evaluated a version of DATAFLOW that *only* performed metadata lookup in the baggy bounds table (i.e., it did not construct *def-use* chains nor update the fuzzer's coverage map). This version of DATAFLOW has a geomean overhead of 3.97×. *Def-Use* chain construction is a simple xor operation (Section 5.3), so we attribute this dramatic increase in run-time overhead to the interaction of the baggy bounds table and coverage map. In particular, cache effects associated with reading from/writing to these two tables.

> Despite building on PAMD—an efficient metadata encoding scheme—DATAFLOW remains impaired by high run-time overheads. Maximizing fuzzer execution rates (e.g., by lowering run-time instrumentation costs) is crucial to maximizing fuzzing outcomes.

6.3 Bug Finding (RQ 2)

Following prior work [2, 26, 29, 68], we use survival analysis to summarize our bug-finding results. Table 5 shows the restricted mean survival time (RMST), measuring the mean time for a bug to "survive" (i.e., remain undiscovered) five repeated 24 h fuzz runs. Lower RMSTs imply a fuzzer finds a bug "faster", while a smaller CI implies the fuzzer finds the given bug (at a given time) more consistently. We use the log-rank test [46]—computed under the null hypothesis that two fuzzers share the same survival function—to statistically compare bug survival times. Thus, we consider two fuzzers to have statistically equivalent bug survival times if the log-rank test's p-value > 0.05.

We present our bug-finding results in Table 5. Based on raw bug counts, AFL++ was the best-performing fuzzer, triggering 60 bugs. The two data-flow-driven fuzzers followed this; `DDFuzz` (44 bugs) and DATAFLOW (41 bugs). Angora was the worst-performing fuzzer, triggering only 24 bugs.

DATAFLOW with "simple access" *use* sensitivity ($DF_{A/A}$ and $DF_{A+S/A}$) was the best performing version of DATAFLOW (39 bugs). This was followed by $DF_{A+S/O}$ (31 bugs). DATAFLOW was "accessed

Table 5. Magma bugs, presented as the RMST (in hours) with 95 % bootstrap CI. Bugs never found by a particular fuzzer have an RMST of ⊤ (to distinguish bugs with a 24 h RMST). We only report the RMST for bugs triggered; bugs not triggered by any fuzzer are omitted. The best performing fuzzer (fuzzers if the bug survival times are statistically equivalent per the log-rank test) for each bug is highlighted in green (smaller is better).

Target	Driver	Bug	Fuzzer									
			A_{LTO}	A_{CL}	An	DD	$DF_{A/A}$	$DF_{A/O}$	$DF_{A/V}$	$DF_{A+S/A}$	$DF_{A+S/O}$	$DF_{A+S/V}$
libpng	`read_fuzzer`	PNG003	0.01 ± 0.01	0.01 ± 0.01	0.01 ± 0.01	0.05 ± 0.01	0.03 ± 0.02	0.06 ± 0.01	0.24 ± 0.15	0.04 ± 0.02	0.03 ± 0.03	0.99 ± 0.44
		PNG006	⊤	0.02 ± 0.02	0.07 ± 0.03	⊤	⊤	⊤	⊤	⊤	⊤	⊤
		PNG007	7.47 ± 6.93	17.54 ± 4.15	⊤	19.49 ± 7.37	23.38 ± 1.39	⊤	⊤	19.49 ± 10.21	19.94 ± 9.19	⊤
libsndfile	`sndfile_fuzzer`	SND001	0.43 ± 0.27	0.72 ± 0.29	–	1.75 ± 0.55	0.78 ± 0.20	19.97 ± 6.74	⊤	0.77 ± 0.32	20.44 ± 8.07	⊤
		SND005	0.55 ± 0.21	0.62 ± 0.43	–	2.05 ± 0.79	5.63 ± 2.25	23.25 ± 1.69	⊤	5.63 ± 1.55	15.32 ± 7.81	20.49 ± 7.96
		SND006	0.36 ± 0.23	0.26 ± 0.16	–	1.06 ± 0.21	6.54 ± 5.97	⊤	19.52 ± 10.15	3.05 ± 2.43	⊤	18.96 ± 8.45
		SND007	0.64 ± 0.27	0.27 ± 0.16	–	2.53 ± 0.54	1.39 ± 0.73	⊤	⊤	3.33 ± 2.21	21.12 ± 6.52	22.57 ± 3.24
		SND017	0.43 ± 0.28	0.06 ± 0.02	–	0.00 ± 0.01	0.01 ± 0.02	0.74 ± 1.21	0.04 ± 0.04	0.02 ± 0.02	0.01 ± 0.01	14.42 ± 11.50
		SND020	0.71 ± 0.30	0.49 ± 0.14	–	0.60 ± 0.11	0.30 ± 0.20	4.76 ± 2.72	1.71 ± 0.57	0.61 ± 0.36	5.53 ± 7.39	15.15 ± 10.63
		SND024	0.31 ± 0.24	0.26 ± 0.16	–	1.03 ± 0.21	0.50 ± 0.30	21.61 ± 3.20	19.38 ± 10.45	0.60 ± 0.26	21.03 ± 6.73	15.07 ± 10.72
libtiff	`read_rgba_fuzzer`	TIF002	18.44 ± 6.92	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤
		TIF007	0.01 ± 0.01	0.02 ± 0.01	0.81 ± 0.40	0.27 ± 0.11	0.17 ± 0.07	0.95 ± 0.63	19.62 ± 9.92	0.31 ± 0.18	2.20 ± 1.85	14.93 ± 10.89
		TIF008	19.97 ± 5.16	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤
		TIF012	0.16 ± 0.05	0.99 ± 0.47	6.09 ± 7.19	10.32 ± 6.45	3.71 ± 1.45	14.83 ± 11.01	⊤	4.56 ± 3.05	19.50 ± 10.17	⊤
		TIF014	0.60 ± 0.20	1.26 ± 0.37	⊤	15.46 ± 10.33	14.15 ± 7.31	20.12 ± 8.79	⊤	20.07 ± 8.90	19.45 ± 10.29	⊤
	`tiffcp`	TIF005	⊤	⊤	14.91 ± 10.91	⊤	⊤	⊤	⊤	⊤	⊤	⊤
		TIF006	7.52 ± 4.11	15.40 ± 8.28	10.14 ± 6.14	⊤	22.86 ± 2.59	⊤	⊤	18.12 ± 7.14	⊤	⊤
		TIF007	0.03 ± 0.02	0.02 ± 0.02	0.73 ± 0.66	0.29 ± 0.09	0.26 ± 0.08	6.26 ± 7.22	1.68 ± 0.45	0.38 ± 0.20	0.66 ± 0.16	10.57 ± 9.61
		TIF008	⊤	22.17 ± 4.13	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤
		TIF009	10.33 ± 6.86	13.30 ± 7.97	⊤	⊤	17.70 ± 7.79	⊤	⊤	⊤	⊤	⊤
		TIF012	0.26 ± 0.13	0.71 ± 0.27	11.57 ± 9.09	11.14 ± 7.90	4.99 ± 1.97	20.42 ± 8.11	⊤	7.02 ± 6.80	⊤	⊤
		TIF014	0.50 ± 0.21	1.60 ± 0.70	14.23 ± 7.56	8.79 ± 6.65	12.87 ± 8.05	20.31 ± 8.35	⊤	13.78 ± 7.91	14.22 ± 6.59	⊤

Table 5. Magma bugs (continued).

Target	Driver	Bug	Fuzzer									
			A_{LTO}	A_{CL}	An	DD	$DF_{A/A}$	$DF_{A/O}$	$DF_{A/V}$	$DF_{A+S/A}$	$DF_{A+S/O}$	$DF_{A+S/V}$
libxml2	`read_memory_fuzzer`	XML001	⊤	⊤	10.16 ± 3.03	13.11 ± 6.48	23.23 ± 1.74	⊤	⊤	19.99 ± 4.84	⊤	⊤
		XML002	19.51 ± 10.15	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤
		XML003	4.00 ± 3.45	0.78 ± 0.61	0.01 ± 0.01	0.03 ± 0.03	0.04 ± 0.05	0.05 ± 0.00	0.05 ± 0.04	0.05 ± 0.05	0.04 ± 0.05	0.05 ± 0.06
		XML009	1.07 ± 0.35	0.95 ± 0.29	⊤	14.59 ± 6.94	3.38 ± 3.11	15.31 ± 10.48	⊤	6.70 ± 6.33	15.72 ± 10.08	⊤
		XML017	0.01 ± 0.01	0.01 ± 0.01	0.01 ± 0.01	0.01 ± 0.01	0.07 ± 0.00	0.08 ± 0.00	0.07 ± 0.00	0.07 ± 0.00	0.07 ± 0.00	0.07 ± 0.00
	`xmllint`	XML001	22.88 ± 2.53	⊤	⊤	0.01 ± 0.01	0.06 ± 0.00	0.08 ± 0.00	0.07 ± 0.00	0.06 ± 0.00	0.07 ± 0.00	0.07 ± 0.00
		XML002	22.83 ± 2.64	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤
		XML009	0.59 ± 0.24	1.14 ± 0.61	4.29 ± 0.12	7.16 ± 3.70	1.57 ± 0.74	18.53 ± 8.70	⊤	1.18 ± 0.65	15.65 ± 10.03	⊤
		XML012	20.61 ± 4.99	22.30 ± 3.84	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤
		XML017	0.02 ± 0.02	0.02 ± 0.02	0.01 ± 0.01	0.02 ± 0.02	0.05 ± 0.04	0.06 ± 0.04	0.05 ± 0.00	0.05 ± 0.05	0.05 ± 0.00	0.05 ± 0.00
lua	`lua`	LUA003	⊤	⊤	⊤	⊤	⊤	⊤	⊤	19.81 ± 9.47	⊤	⊤
		LUA004	6.06 ± 3.88	10.38 ± 5.67	⊤	0.22 ± 0.37	0.02 ± 0.02	0.01 ± 0.02	0.02 ± 0.01	0.01 ± 0.02	0.21 ± 0.37	0.01 ± 0.02
openssl	`asn1`	SSL001	3.53 ± 3.38	8.34 ± 4.86	⊤	21.83 ± 3.63	⊤	⊤	⊤	⊤	⊤	⊤
		SSL003	0.05 ± 0.07	0.03 ± 0.03	0.00 ± 0.01	0.01 ± 0.01	0.01 ± 0.02	0.01 ± 0.02	0.02 ± 0.03	0.01 ± 0.02	0.01 ± 0.01	0.01 ± 0.02
	`client`	SSL002	0.06 ± 0.00	0.06 ± 0.00	0.01 ± 0.01	0.05 ± 0.05	0.02 ± 0.02	0.03 ± 0.03	0.03 ± 0.03	0.02 ± 0.03	4.82 ± 10.85	0.03 ± 0.02
	`server`	SSL002	0.08 ± 0.00	0.09 ± 0.01	0.18 ± 0.02	5.00 ± 7.60	0.52 ± 0.00	0.69 ± 0.16	0.69 ± 0.27	5.22 ± 7.51	0.39 ± 0.04	0.58 ± 0.02
		SSL020	20.78 ± 6.00	⊤	6.13 ± 0.14	21.90 ± 4.76	⊤	⊤	⊤	⊤	⊤	⊤
	`x509`	SSL009	⊤	⊤	0.01 ± 0.02	0.11 ± 0.00	0.02 ± 0.02	0.02 ± 0.02	0.02 ± 0.02	0.02 ± 0.02	0.01 ± 0.02	0.02 ± 0.02
poppler	`pdf_fuzzer`	PDF010	1.08 ± 0.54	1.47 ± 0.96	⊤	2.23 ± 1.33	11.01 ± 4.15	⊤	16.14 ± 9.47	7.91 ± 6.84	16.90 ± 5.17	22.22 ± 4.03
		PDF016	0.03 ± 0.01	0.02 ± 0.02	0.23 ± 0.00	0.16 ± 0.03	0.37 ± 0.09	0.31 ± 0.01	0.27 ± 0.01	0.31 ± 0.02	0.25 ± 0.01	0.49 ± 0.03
		PDF018	17.60 ± 7.22	15.86 ± 4.14	⊤	17.35 ± 8.27	20.09 ± 5.16	⊤	⊤	⊤	⊤	⊤
		PDF019	23.14 ± 1.94	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤
		PDF021	23.52 ± 1.09	⊤	⊤	22.30 ± 3.86	⊤	⊤	⊤	⊤	⊤	⊤

value" *use* sensitivity was the worst performer. This suggests incorporating variable values at *use* sites is not worth the increased run-time cost; simply tracking the existence of *def-use* chains is "good enough" (for discovering bugs).

Table 5. Magma bugs (continued).

Target	Driver	Bug	Fuzzer									
			A_{LTO}	A_{CL}	An	DD	$DF_{A/A}$	$DF_{A/O}$	$DF_{A/V}$	$DF_{A+S/A}$	$DF_{A+S/O}$	$DF_{A+S/V}$
poppler	pdfimages	PDF002	⊤	21.31 ± 6.08	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤
		PDF003	6.89 ± 3.26	13.29 ± 3.79	0.01 ± 0.01	0.00 ± 0.01	0.00 ± 0.01	0.00 ± 0.01	0.00 ± 0.01	0.00 ± 0.01	4.80 ± 10.86	0.00 ± 0.01
		PDF011	⊤	19.88 ± 9.33	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤
		PDF016	0.02 ± 0.01	0.01 ± 0.01	0.20 ± 0.06	0.11 ± 0.04	0.20 ± 0.08	0.13 ± 0.03	0.19 ± 0.05	0.15 ± 0.03	4.91 ± 7.64	0.16 ± 0.02
		PDF018	3.68 ± 1.61	9.13 ± 6.32	⊤	14.63 ± 6.86	20.66 ± 7.57	⊤	⊤	6.56 ± 7.09	⊤	⊤
		PDF019	21.99 ± 4.54	14.37 ± 7.73	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤
		PDF021	⊤	22.58 ± 3.22	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤
	pdftoppm	PDF006	20.72 ± 7.43	⊤	⊤	22.86 ± 2.57	⊤	⊤	⊤	⊤	⊤	⊤
		PDF008	⊤	⊤	⊤	23.49 ± 1.16	⊤	⊤	⊤	⊤	⊤	⊤
		PDF010	2.20 ± 1.15	1.78 ± 0.56	12.52 ± 7.98	13.13 ± 7.80	3.70 ± 0.71	⊤	19.50 ± 10.18	3.66 ± 1.30	19.54 ± 10.10	17.32 ± 8.99
		PDF011	21.29 ± 6.14	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤
		PDF016	0.05 ± 0.05	0.02 ± 0.02	1.29 ± 0.46	0.19 ± 0.07	0.30 ± 0.06	0.23 ± 0.02	0.30 ± 0.06	0.42 ± 0.14	0.17 ± 0.02	0.37 ± 0.02
		PDF018	15.87 ± 5.87	15.34 ± 4.72	⊤	16.95 ± 6.77	15.80 ± 9.85	⊤	⊤	18.77 ± 6.28	⊤	⊤
		PDF021	⊤	17.10 ± 8.38	⊤	20.74 ± 7.38	⊤	⊤	⊤	⊤	⊤	⊤
sqlite3	sqlite3_fuzz	SQL002	0.72 ± 0.21	1.59 ± 0.84	⊤	11.67 ± 4.01	15.07 ± 10.75	21.30 ± 6.12	⊤	19.40 ± 10.40	⊤	⊤
		SQL012	22.93 ± 2.41	22.74 ± 2.86	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤
		SQL013	⊤	23.49 ± 1.15	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤
		SQL014	4.38 ± 2.57	5.60 ± 4.45	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤
		SQL018	1.74 ± 0.87	2.74 ± 1.37	⊤	17.17 ± 8.20	⊤	21.49 ± 5.68	⊤	20.03 ± 9.00	23.75 ± 0.56	⊤
		SQL020	17.70 ± 8.30	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤

AFL++ remains the best-performing fuzzer when accounting for RMSTs (i.e., it triggers bugs fastest), outperforming the data-flow-guided fuzzers for the majority of bugs triggered (60 %). However, this result is reversed (i.e., the data-flow-guided fuzzers outperform AFL++) for 14 % of the triggered bugs. Notably, DatAFLow was the only fuzzer to trigger LUA003 (not previously triggered by any fuzzer in any prior Magma evaluation), while DatAFLow and `DDFuzz` triggered XML001 (`xmllint`) and LUA004 orders-of-magnitude faster than AFL++. `DDFuzz` was the only fuzzer to trigger PDF008. However, this bug was only triggered once (over five trials) and towards

the end of the trial (after 20 h). This suggests that the bug is difficult to find and DDFuzz may have just "gotten lucky". Finally, AFL++ either failed to trigger or was orders-of-magnitude slower at triggering SSL009 (x509) and PDF003 (pdfimages). *Do these bugs share properties that make them amenable to discovery via dat -flow-guided fuzzing?* To answer this question, we examine the two lua bugs in greater depth.

```
1 #define l_checkmodep(m) ((m[0] == 'r' || m[0] == 'w') && m[1] == '\0')
```

Fig. 8. LUA003 missing popen check.

LUA003. This bug is caused by a missing check of the "mode" argument to popen. The check is shown in Fig. 8. While the check is quickly reached by DDFuzz (after ~4 h) and all six datAFLow variations (on average, after ~60 s), the exact trigger conditions were only met once by $DF_{A+S/A}$. Upon examining the compiled binary, we found the second check (`m[1] == '\0'`) was optimized to a *branchless* operation (i.e., it did not contain conditional control flow). This effectively makes the program state where `m[1] != '\0'` invisible to a control-flow-guided fuzzer (in particular, there is no explicit edge for AFL++ to instrument). This state is explicitly visible to datAFLow, which reaches it after ~19 h of fuzzing.

LUA004. This is a logic bug, caused by a missing update to the interpreter's "old" program counter (occurring under particular conditions when tracing the execution of a Lua function). Again, there is no explicit "state" in the target's CFG for the fuzzer to reach. Instead, the bug is triggered when the `oldpc` field in the `lua_State` struct is not updated. This only happens under particular conditions, again depending on specific data values.

> The control-flow-guided fuzzers (AFL++ and Angora) outperform the data-flow-guided fuzzers (DDFuzz and datAFLow) on 60 % of the triggered Magma bugs. However, the data-flow-guided fuzzers significantly outperform the control-flow-guided fuzzers (by orders-of-magnitude) on 11 % of the triggered bugs. These results suggest that fuzzers guided by control flow and data flow should be combined to maximize bug-finding potential.

6.4 Coverage Expansion (RQ 3)

Control-flow coverage is typically quantified by reasoning over the target's CFG (e.g., basic blocks, edges, lines of code). For example, FuzzBench replays the fuzzer's queue through an independent and precise (i.e., collision-free) coverage metric; specifically, Clang's source-based coverage [48, 67]. However, the equivalent process for quantifying *data-flow* coverage does not exist.

We quantify coverage expansion using both control-flow and data-flow metrics, using (a) static analyses to approximate an upper bound, and (b) dynamic analyses to quantify coverage expansion against this upper bound. The usual limitations of static analysis (e.g., undecidability) mean this upper bound may be larger than the set of executable coverage elements (e.g., a code region may not be reachable from the target's driver, or a pointer's points-to set may be over-approximated). We accept this imprecision for both metrics. We use the Mann-Whitney U-test [45] to statistically compare dynamic coverage across fuzzers: two fuzzers cover the same number of coverage elements if the Mann-Whitney U-test's p-value > 0.05.

Control-flow coverage. We use Clang's existing source-based coverage metric [67]. Specifically, we use *region coverage* (as used by FuzzBench), Clang's version of statement coverage. Like classic

statement coverage, region coverage is more granular than function and line coverage [32]. Region information is embedded into the target during compilation and can be statically extracted using existing LLVM tooling (to obtain the upper bound).

Data-flow coverage. We develop an SVF-based [63] static analysis to compute the set of *def-use* chains in a target (for the set of tracked variables, as determined by the chosen *def* site sensitivity). This analysis leverages a flow- and context-insensitive interprocedural pointer analysis based on the Andersen algorithm [1].[4] For the dynamic analysis, we modify the PAMD metadata stored at each *def* site (Section 5.2.1) to store a tuple of ⟨*variable name*, *location*⟩, where *location* is another tuple ⟨*source filename*, *function name*, *line*, *column*⟩. Both tuples are constructed by extracting source-level information from the target's debug information. A *use* site (Section 5.2.2) is similarly labeled with a *location* tuple. Unlike the 16-bit tags used by datAFLow, this approach does not result in hash collisions and is precise (albeit with a higher run-time cost). Importantly, neither the static nor dynamic analysis take into account *def-use* chain *values*. We also exclude dynamic memory allocations from these analyses (to simplify run-time *def-use* tracking when faced with custom memory allocators, per Section 5.4.2).

Table 6 and Figs. 9 and 10 summarize our coverage expansion results. Two targets, `bison` and `faust`, failed to build with AFL++'s CmpLog (again, due to a segmentation fault) and are excluded from our results.

AFL++ is again the best-performing fuzzer, achieving the highest control-flow (i.e., code region) coverage. CmpLog improves AFL++'s already-strong coverage expansion capabilities. These results are unsurprising, given control-flow coverage (specifically, edge coverage) guides AFL++. Similarly, Angora again performs poorly, outperformed by both `DDFuzz` and datAFLow in maximizing both control- and data-flow coverage. Curiously, however, AFL++ also achieves the highest *data-flow* (i.e., *def-use* chain) coverage. This is despite datAFLow's data-flow guidance. We attribute this (surprising) result to the differences in *fuzzer execution rates* (i.e., the number of inputs executed by the fuzzer per unit of time).

6.4.1 Accounting for Execution Rates. AFL++ (LTO) achieves a mean execution rate of 1,172 execs/s (median 347 execs/s). In contrast, `DDFuzz`, Angora, and datAFLow achieve mean execution rates of 974, 616 and 270 execs/s, respectively (median 442, 249 and 144 execs/s). This dramatic decrease in execution rates reflects our overhead results in Section 6.2.

To account for differences in execution rates, rather than comparing coverage at the *end* of each fuzz run (i.e., after 24 h of fuzzing), we compare coverage at a given *execution* ("exec"). Specifically, we compare coverage at the last exec of the slowest fuzzer (i.e., with the lowest execution rate). Intuitively, this places a "ceiling" on the coverage achieved by faster fuzzers (i.e., those able to execute more inputs within a single 24 h fuzz run). For example, $DF_{A+S/V}$ is the slowest fuzzer on `bison` (85 execs/s). Thus, we compare the coverage achieved at the last execution of $DF_{A+S/V}$ (exec = 7,358,231), implicitly ignoring any additional coverage expanded after this exec. Unfortunately, Angora does not provide the necessary information to map coverage to a particular exec, so we exclude it from our analysis (despite it being the slowest fuzzer on two targets: `bison` and `faust`).

We present coverage "normalized" against execution rates in Table 7. DatAFLow is now more competitive (against AFL++) in expanding data-flow coverage. It achieves the highest *def-use* chain coverage on `bison` and `faust`, and is only ~4 % behind the number of *def-use* chains expanded by AFL++ on `qbe`. Again, increasing datAFLow's *use* sensitivity to include variable values fails to improve fuzzing outcomes. These results reinforce our belief that fuzzer execution rates have a significant impact on fuzzing outcomes.

[4]We experimented with SVF's flow-sensitive interprocedural analysis but found the run-time overheads prohibitively large.

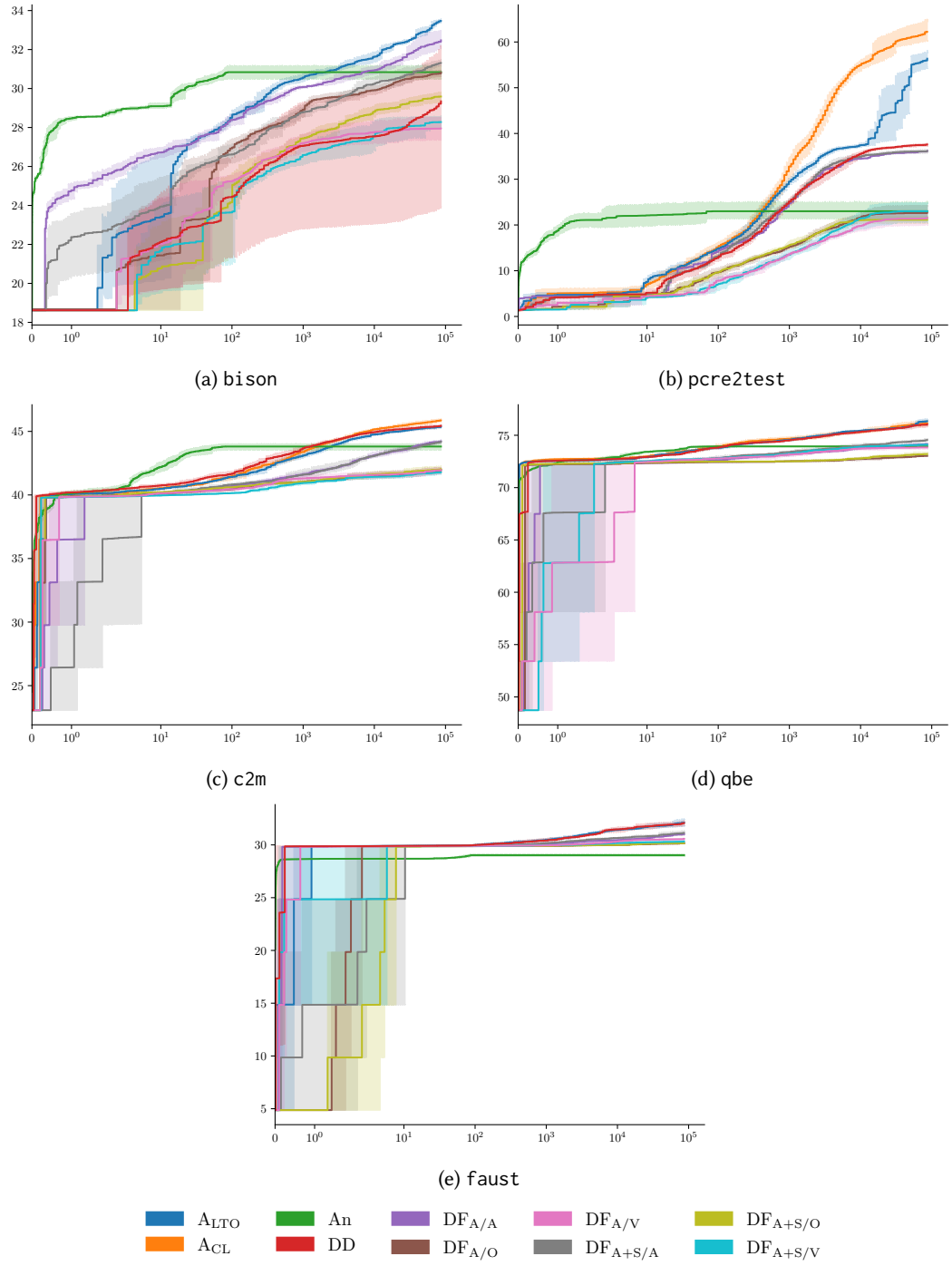


Fig. 9. Control-flow coverage expansion over time. The x-axis is time in seconds (log scale), and the y-axis is the percentage of code regions expanded (against the static upper bound in Table 6a). The mean coverage (over five repeated trials) and 95 % bootstrap CI is shown.

Table 6. Coverage expansion (control and data flow) on DDFuzz targets, reported as the mean over five repeated trials with 95 % bootstrap CI. The "static" results give an approximate upper bound, while the "dynamic" results give the percentage of coverage elements covered at run time. The best performing fuzzer (fuzzers if the coverages are statistically equivalent per the Mann-Whitney U-test) for each target is highlighted in green (larger is better).

(a) Control-flow coverage. Quantified in terms of Clang's code region coverage.

Target	Static (#)	Dynamic (%)									
		A_{LTO}	A_{CL}	An	DD	$DF_{A/A}$	$DF_{A/O}$	$DF_{A/V}$	$DF_{A+S/A}$	$DF_{A+S/O}$	$DF_{A+S/V}$
bison	35,476	33.47 ± 0.08	✗	30.84 ± 0.31	29.95 ± 3.21	32.47 ± 0.39	30.83 ± 0.11	27.95 ± 0.56	31.31 ± 0.12	29.59 ± 0.23	28.28 ± 0.27
pcre2test	36,973	56.35 ± 1.78	62.22 ± 2.25	23.01 ± 1.68	37.56 ± 0.30	36.21 ± 0.00	22.63 ± 1.31	21.51 ± 1.25	36.13 ± 0.32	21.13 ± 0.66	23.01 ± 1.98
c2m	43,765	45.35 ± 0.07	45.86 ± 0.08	43.81 ± 0.22	45.45 ± 0.07	44.22 ± 0.09	41.99 ± 0.15	41.95 ± 0.22	44.18 ± 0.08	42.02 ± 0.21	41.78 ± 0.17
qbe	5,400	76.15 ± 0.19	76.34 ± 0.21	73.94 ± 0.03	76.03 ± 0.15	74.17 ± 0.09	73.03 ± 0.03	73.84 ± 0.12	74.54 ± 0.11	73.20 ± 0.14	74.11 ± 0.07
faust	26,872	32.11 ± 0.32	✗	29.02 ± 0.09	32.07 ± 0.20	31.02 ± 0.08	30.17 ± 0.04	30.57 ± 0.04	31.13 ± 0.17	30.16 ± 0.03	30.31 ± 0.12

(b) Data-flow coverage. Quantified in terms of interprocedural *def-use* chains (scaled by $\times 10^{-3}$, due to the small percentage of *def-use* chains covered across all targets). The c2m target is excluded because it unexpectedly crashed with our precise data-flow tracking instrumentation.

Target	Static (#)	Dynamic ($\times 10^{-3}$ %)									
		A_{LTO}	A_{CL}	An	DD	$DF_{A/A}$	$DF_{A/O}$	$DF_{A/V}$	$DF_{A+S/A}$	$DF_{A+S/O}$	$DF_{A+S/V}$
bison	9,923,196	11.60 ± 0.10	✗	9.94 ± 0.17	10.40 ± 0.17	11.27 ± 0.26	10.71 ± 0.10	9.57 ± 0.33	11.26 ± 0.16	10.60 ± 0.03	9.84 ± 0.28
pcre2test	1,401,778	181.03 ± 14.78	205.78 ± 16.14	78.24 ± 7.23	125.53 ± 1.53	122.72 ± 1.30	76.12 ± 4.87	69.51 ± 5.71	124.83 ± 1.65	69.73 ± 2.86	73.82 ± 5.99
c2m	25,462,192	–	–	–	–	–	–	–	–	–	–
qbe	450,407	381.97 ± 3.06	378.68 ± 4.46	178.28 ± 0.47	376.64 ± 3.62	363.01 ± 1.71	352.61 ± 0.36	356.21 ± 1.38	366.20 ± 0.40	353.24 ± 0.62	355.77 ± 4.22
faust	159,515,187	5.46 ± 0.06	✗	4.88 ± 0.07	5.55 ± 0.06	5.19 ± 0.01	4.81 ± 0.03	5.01 ± 0.05	5.34 ± 0.03	4.78 ± 0.02	4.89 ± 0.07

The control-flow-guided fuzzers (specifically, AFL++) achieve the highest control- *and* data-flow coverage. Data-flow-guided fuzzers require more complex instrumentation (compared to control-flow-guided fuzzers), impairing the fuzzers' execution rates.

6.5 Characterizing Data-Flow (RQ 4)

The fuzzing community has largely settled on control-flow-based coverage metrics—in particular, edge coverage—to drive a fuzzer's exploration. While prior successes have largely validated this approach [18, 56, 58, 65, 73], we wish to understand what (if any) program characteristics lend themselves to data-flow-based coverage.

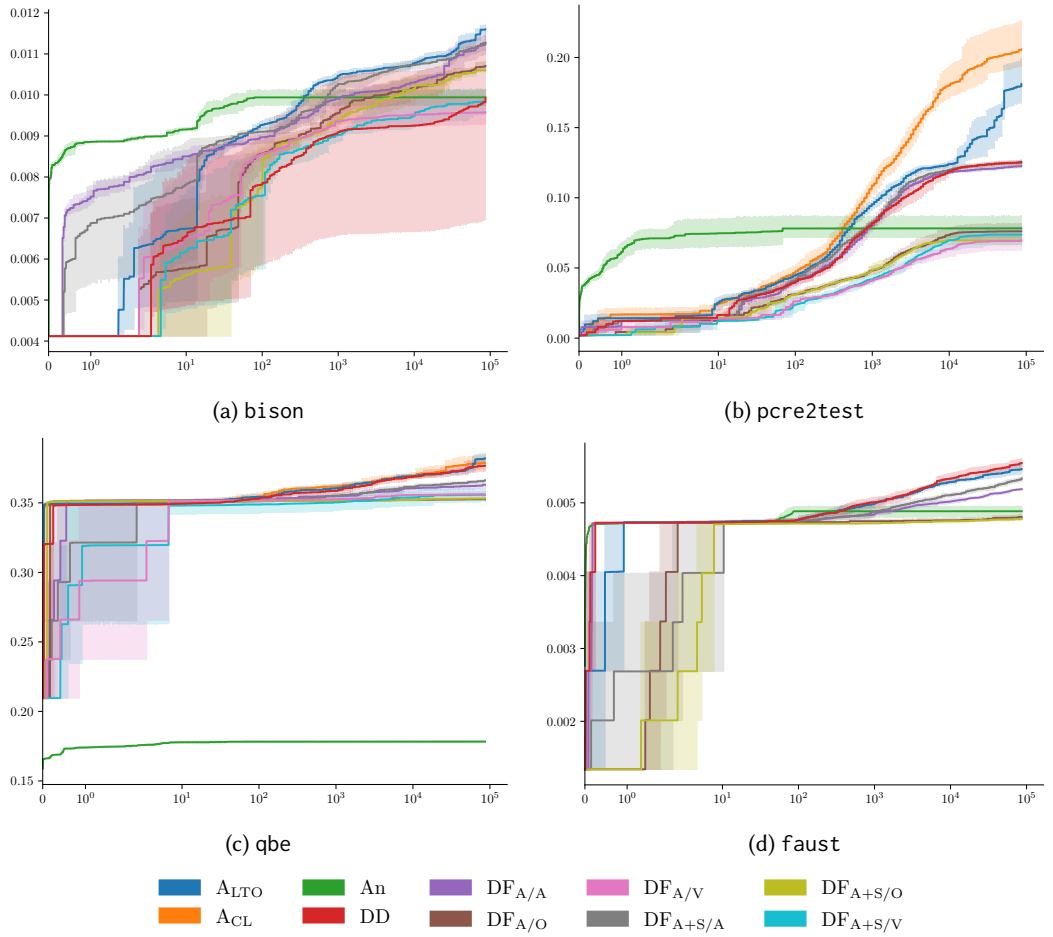


Fig. 10. Data-flow coverage expansion over time. The x-axis is time in seconds (log scale), and the y-axis is the percentage of *def-use* chains expanded (against the static upper bound in Table 6b). The mean coverage (over five repeated trials) and 95 % bootstrap CI is shown.

Mantovani et al. [47] propose the *DD ratio*—defined as the ratio between the number of basic blocks instrumented with data-dependency information over the total number of basic blocks in the target—to determine whether data-flow-based coverage (derived from the target's DDG) adds value (e.g., over edge coverage). A higher DD ratio suggests the target is more amenable to data-flow-guided fuzzing; a target with a DD ratio above 10 % is considered *strongly* data dependent.

Table 8 summarizes the DD ratio of our 20 target programs.[5] Thirteen of these targets (65 %) have DD ratios $\geq$ 10 %, indicating their suitability for data-flow-guided fuzzing. However, we found little correlation between a target's DD ratio and fuzzing outcomes (both bug finding and coverage expansion). For example, `png_read_fuzzer` had the highest DD ratio among the Magma targets (13.40 %), closely followed by `xmllint` (13.03 %). However, AFL++ outperformed the data-flow-guided fuzzers (DDFuzz and DATAFLOW) on both targets (across bug counts and survival times).

[5]These values differ from the original DDFuzz evaluation [47] because we used newer versions of the targets (per Section 6.1.2).

Table 7. Coverage expansion—control and data flow ("CF" and "DF", respectively)—on DDFuzz targets. In contrast to Table 6, which reports the (mean) coverage achieved at the *end of a 24 h fuzz run*, here we report the (mean) coverage at the *last exec of the slowest fuzzer* ("Exec" occurring at the given "Time"). Dynamic coverage is quantified in terms of the static analysis results in Table 6 (the DF results are again scaled by $\times 10^{-3}$). The best performing fuzzer (fuzzers if the coverage is statistically equivalent per the Mann-Whitney U-test) for each target is highlighted in green (larger is better).

Target	Exec (#)	Time (hr)	Metric	Dynamic (%, $\times 10^{-3}$ %)								
				A_{LTO}	A_{CL}	DD	$DF_{A/A}$	$DF_{A/O}$	$DF_{A/V}$	$DF_{A+S/A}$	$DF_{A+S/O}$	$DF_{A+S/V}$
bison	7,358,231	13.74	CF	32.35 ± 0.09	✗	28.39 ± 3.71	32.17 ± 0.47	30.64 ± 0.11	27.89 ± 0.53	30.90 ± 0.20	29.54 ± 0.23	28.11 ± 0.51
			DF	10.95 ± 0.09	✗	9.56 ± 2.08	11.06 ± 0.24	10.59 ± 0.06	9.55 ± 0.34	11.03 ± 0.08	10.56 ± 0.08	9.79 ± 0.35
pcre2test	37,992,897	8.22	CF	37.51 ± 0.30	55.79 ± 1.09	36.66 ± 0.16	35.85 ± 0.00	22.56 ± 1.28	21.38 ± 1.15	35.82 ± 0.29	21.12 ± 0.66	23.01 ± 1.98
			DF	124.40 ± 1.69	182.84 ± 11.61	122.42 ± 0.77	121.02 ± 0.68	75.96 ± 4.72	69.08 ± 5.31	123.47 ± 1.68	69.73 ± 2.83	73.82 ± 5.99
c2m	516,654	5.27	CF	43.20 ± 0.13	43.44 ± 0.08	43.41 ± 0.27	42.84 ± 0.11	41.44 ± 0.24	41.67 ± 0.25	42.77 ± 0.22	41.39 ± 0.11	41.61 ± 0.19
			DF	–	–	–	–	–	–	–	–	–
qbe	12,589,430	15.78	CF	75.39 ± 0.07	75.27 ± 0.12	75.28 ± 0.10	74.11 ± 0.12	72.93 ± 0.06	73.82 ± 0.12	74.41 ± 0.10	73.02 ± 0.13	74.05 ± 0.09
			DF	368.69 ± 1.33	368.38 ± 1.73	368.33 ± 1.20	362.03 ± 1.89	352.53 ± 0.38	356.17 ± 1.33	364.83 ± 0.75	352.93 ± 0.67	355.68 ± 4.26
faust	1,033,846	23.97	CF	31.26 ± 0.04	✗	31.05 ± 0.08	30.80 ± 0.06	30.12 ± 0.04	30.49 ± 0.06	30.86 ± 0.08	30.15 ± 0.04	30.25 ± 0.12
			DF	5.25 ± 0.02	✗	5.22 ± 0.01	5.13 ± 0.01	4.79 ± 0.03	4.99 ± 0.05	5.22 ± 0.00	4.77 ± 0.01	4.87 ± 0.08

Table 8. Characterizing data flow using the data dependency ratio ("DD ratio") introduced by Mantovani et al. [47]. *Strongly* data-dependent targets (i.e., those with a DD ratio ≥ 10 %) are highlighted in green.

(a) Magma.

Target	DD ratio (%)
png_read_fuzzer	13.40
sndfile_fuzzer	12.14
tiff_read_rgba_fuzzer	12.01
tiffcp	11.37
xml_read_memory_fuzzer	12.86
xmllint	13.03
lua	12.73
asn1	9.89
client	9.99
server	9.98
x509	9.98
pdf_fuzzer	11.62
pdfimages	9.33
pdftoppm	11.77
sqlite3_fuzz	12.80

(b) DDFuzz targets.

Target	DD ratio (%)
bison	6.59
pcre2test	22.60
c2m	21.82
qbe	12.45
faust	7.29

Similarly, `pcre2test` and `c2m` had the highest DD ratios among the `DDFuzz` targets (22.60 and 21.82 %, respectively). Again, AFL++ outperformed the two data-flow-guided fuzzers (across both control- and data-flow coverage expansion).

Based on these results, we conclude that the DD ratio is not suitable for determining a target's suitability for data-flow-guided fuzzing. We propose an alternative approach in Section 6.7.

> The "DD ratio" is not suitable for determining whether a target is amenable to data-flow-guided fuzzing. Alternative approaches are required.

6.6 Discussion

Comparison to the registered report. DATAFLOW's implementation has evolved significantly since the initial registered report [30]. In particular, *def-use* chain tracking changed from using low-fat pointers to PAMD (Section 5.2). Consequently, heapification of *all* tracked *def* sites is no longer required (only *def* sites that cannot be statically resized to fit the PAMD metadata require heapification). Surprisingly, this resulted in *higher* run-time overheads. Despite this, our bug-finding results improved from triggering 10 bugs to triggering 41. We attribute this improved result to PAMD's robustness and its ability to work on a wider variety of targets (e.g., *openssl* failed to build with DATAFLOW in our preliminary evaluation).

Coverage sensitivity. In Section 4.1, we introduced a framework for reasoning about and constructing data-flow coverage metrics for greybox fuzzing. This framework allows the user to balance precision with performance. Our results suggest that fuzzing outcomes (i.e., bug finding and coverage expansion) fail to improve as precision increases. Notably, this finding also applies to Angora; Angora's exact DTA provided little benefit over the approximate DTA used by AFL++'s CmpLog mode. Our results reflect prior findings that demonstrate the importance of maximizing fuzzer execution rates [5, 23, 29, 54, 72].

Bugs vs. coverage. Böhme et al. [9] found the fuzzer best at maximizing coverage expansion may not be the best at finding bugs. Our results reflect this finding; despite AFL++ outperforming DATAFLOW on coverage expansion (Section 6.4), DATAFLOW triggered bugs AFL++ failed to find (Section 6.3). Ultimately, fuzzers are deployed to find bugs and vulnerabilities; our findings reinforce the need for bug-based fuzzer evaluation [26, 38, 74] (not only a comparison of coverage profiles).

Computing coverage upper bounds with static analysis. In Section 6.4 we used static analysis to approximate a coverage upper bound (for both control- and data-flow coverage). In theory, this upper bound is useful for estimating the *residual risk* of ending a fuzz run before maximizing coverage (analogous to the residual risk of missing a bug [6]). In practice, static analysis of "real-world" programs is fraught; dynamically loaded, JIT, and inline assembly code all impact precision. Even specific command-line arguments influence the reachability of particular code regions. Thus, it is difficult to determine how realistic the upper bounds in Section 6.4 are. We leave it to future work to improve estimating coverage-based residual risk.

Testing our hypothesis. We hypothesized that data-flow-guided fuzzing offers superior performance on targets where control flow is decoupled from semantics. Our results lead us to reject this hypothesis. In most cases, control-flow-guided fuzzers outperformed data-flow-guided fuzzers (across both bug-finding and coverage-expansion metrics, and on targets identified as being amenable to data-flow-guided fuzzing). However, we are not prepared to give up on data-flow-guided fuzzing; despite lower run-time costs than DTA, DATAFLOW's run-time costs remain high, negatively impacting coverage expansion. Despite this impediment, DATAFLOW discovers bugs

control-flow-guided fuzzers do not. We believe reducing the run-time costs of data-flow-guided fuzzers will improve fuzzing outcomes.

6.7 Future Work

The significant run-time overheads remain the primary impediment to the adoption of data-flow-guided fuzzing (see Section 6.2). Liu and Criswell [41] propose using interprocedural optimizations to eliminate unnecessary object (de)allocation in the baggy bounds table, improving performance. Similarly, more sophisticated pointer analyses (e.g., those provided by SVF) could be used to eliminate unnecessary *def*/*use* site instrumentation (e.g., removing redundant instrumentation when *def*-*use* chains can be statically identified).

Per Section 5.3, datAFLow is prone to hash collisions. It is well known that hash collisions cause fuzzers to miss program behaviors [24]. While AFL++'s LTO mode solves the hash collision problem for edge coverage, we did not investigate a similar technique for *def*-*use* chain coverage. A hash-collision-free datAFLow may lead to improved coverage expansion.

Finally, datAFLow exclusively uses *def*-*use* chain coverage to drive exploration. In contrast, other data-flow-guided fuzzers (e.g., InvsCov [20], `DDFuzz` [47]) combine data flow with control flow. Given our bug-finding results—i.e., those where datAFLow significantly outperformed AFL++ (e.g., LUA003, LUA004, SSL009, and PDF003)—combining datAFLow with hash-collision-free edge coverage may provide a "best of both worlds" solution (echoing the conclusions reached by Salls et al. [57]). This combination of coverage metrics could be realized by combining control- and data-flow coverage in a single coverage map, maintaining separate coverage maps, or dynamically switching between different instrumented targets.

Our results in Section 6.5 led us to conclude that the DD ratio was not suitable for determining a target's suitability for data-flow-guided fuzzing. Prior work on characterizing programs for automated test suite generation is also unsuitable; e.g., the approaches proposed by Neelofar et al. [52], Oliveira et al. [53] are specific to object-oriented software and focus on control-flow features. Instead, we propose *subsumption*.

We say that coverage metric $\mathcal{M}_1$ strictly subsumes metric $\mathcal{M}_2$, if covering all coverage elements in $\mathcal{M}_1$ also covers all elements in $\mathcal{M}_2$. For example, edge coverage strictly subsumes basic block coverage. Relaxing this definition of strict subsumption allows us to quantify the number of coverage elements in $\mathcal{M}_2$ not subsumed by $\mathcal{M}_1$. Intuitively, more elements in $\mathcal{M}_2$ *not* subsumed by $\mathcal{M}_1$ implies fuzzing with $\mathcal{M}_2$ will lead to behaviors not detectable by $\mathcal{M}_1$. Static data-flow analysis frameworks such as those proposed by Chaim et al. [11] can be used to perform this subsumption analysis. We leave the investigation of such techniques for future work.

7 CONCLUSIONS

Observing fuzzers that introduce taint tracking along with control flow, we investigate data flow as an alternate coverage metric, making *data-flow coverage* a first-class citizen. Driven by empirical results and the conventional wisdom gathered over years of software-testing research, we hypothesized data-flow-guided fuzzing to offer superior outcomes (over control-flow-guided fuzzing) in targets where control flow is decoupled from semantics.

Our results show that "classic" control-flow-guided fuzzing produces better outcomes (bug finding and coverage expansion) in most cases. The high run-time costs associated with data-flow tracking impaired the fuzzer's ability to explore a target's behavior efficiently. Despite these costs, our data-flow-guided fuzzer discovered bugs control-flow-guided fuzzers did not. These results suggest that data-flow-guided fuzzers discover *different*, not *more*, bugs. Specifically, bugs existing in program states not explicitly visible in the target's CFG. A better understanding of *bug characteristics*, rather than program characteristics, may shed light on this result. We release our data-flow sensitivity

framework and datAFLow prototype at https://github.com/HexHive/datAFLow. Our hope is to stimulate further research into data-flow-guided fuzzing.

ACKNOWLEDGMENTS

The authors are grateful to Arlen Cox, Michael Norrish, Andrew Ruef, and the anonymous reviewers for their detailed feedback and insightful suggestions for improving this work. 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.

REFERENCES

[1] Lars Ole Andersen. 1994. *Program analysis and specialization for the C programming language*. Ph. D. Dissertation. University of Copenhagen.

[2] Andrea Arcuri and Lionel Briand. 2011. A Practical Guide for Using Statistical Tests to Assess Randomized Algorithms in Software Engineering. In *International Conference on Software Engineering (ICSE)*. ACM/IEEE, 1–10. https://doi.org/10.1145/1985793.1985795

[3] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for Deep Bugs with Grammars. In *Network and Distributed Systems Security Symposium (NDSS)*. The Internet Society, 15 pages. https://doi.org/10.14722/ndss.2019.23412

[4] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. 2020. Ijon: Exploring Deep State Spaces via Fuzzing. In *IEEE Symposium on Security and Privacy (SP)*. 1597–1612. https://doi.org/10.1109/SP40000.2020.00117

[5] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *Network and Distributed Systems Security Symposium (NDSS)*. The Internet Society, 15 pages. https://doi.org/10.14722/ndss.2019.23371

[6] Marcel Böhme, Danushka Liyanage, and Valentin Wüstholz. 2021. Estimating Residual Risk in Greybox Fuzzing. In *European Software Engineering Conference and Symposium on Foundations of Software Engineering (ESEC/FSE)*. ACM, 230–241. https://doi.org/10.1145/3468264.3468570

[7] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2329–2344. https://doi.org/10.1145/3133956.3134020

[8] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based Greybox Fuzzing As Markov Chain. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 1032–1043. https://doi.org/10.1145/2976749.2978428

[9] Marcel Böhme, László Szekeres, and Jonathan Metzman. 2022. On the Reliability of Coverage-Based Fuzzer Benchmarking. In *International Conference on Software Engineering (ICSE)*. ACM/IEEE, 1621–1633. https://doi.org/10.1145/3510003.3510230

[10] Miguel Castro, Manuel Costa, and Tim Harris. 2006. Securing Software by Enforcing Data-flow Integrity. In *Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX, 147–160. https://doi.org/10.5555/1298455.1298470

[11] Marcos Lordello Chaim, Kesina Baral, Jeff Offutt, Mario Concilio, and Roberto P. A. Araujo. 2021. Efficiently Finding Data Flow Subsumptions. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 94–104. https://doi.org/10.1109/ICST49551.2021.00021

[12] Peng Chen and Hao Chen. 2018. Angora: Efficient Fuzzing by Principled Search. In *IEEE Symposium on Security and Privacy (SP)*. 711–725. https://doi.org/10.1109/SP.2018.00046

[13] T. Y. Chen and C. K. Low. 1995. Dynamic data flow analysis for C++. In *Asia Pacific Software Engineering Conference (APSEC)*. IEEE, 22–28. https://doi.org/10.1109/APSEC.1995.496950

[14] Xi Chen, Asia Slowinska, and Herbert Bos. 2016. On the Detection of Custom Memory Allocators in C Binaries. *Empirical Software Engineering* 21, 3 (2016), 753–777. https://doi.org/10.1007/s10664-015-9362-z

[15] Zheng Leong Chua, Yanhao Wang, Teodora Baluta, Prateek Saxena, Zhenkai Liang, and Purui Su. 2019. One Engine To Serve 'em All: Inferring Taint Rules Without Architectural Semantics. In *Network and Distributed Systems Security Symposium (NDSS)*. The Internet Society, 15 pages. https://doi.org/10.14722/ndss.2019.23339

[16] Nicolas Coppik, Oliver Schwahn, and Neeraj Suri. 2019. MemFuzz: Using Memory Accesses to Guide Fuzzing. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 48–58. https://doi.org/10.1109/ICST.2019.00015

[17] Baozeng Ding, Yeping He, Yanjun Wu, Alex Miller, and John Criswell. 2012. Baggy Bounds with Accurate Checking. In *IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. 195–200. https://doi.org/10.

1109/ISSREW.2012.24

[18] Zhen Yu Ding and Claire Le Goues. 2021. An Empirical Study of OSS-Fuzz Bugs. In *IEEE/ACM International Conference on Mining Software Repositories (MSR)*. 131–142. https://doi.org/10.1109/MSR52588.2021.00026

[19] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions of Programming Languages and Systems* 9, 3 (1987), 319–349. https://doi.org/10.1145/24039.24041

[20] Andrea Fioraldi, Daniele Cono D'Elia, and Davide Balzarotti. 2021. The Use of Likely Invariants as Feedback for Fuzzers. In *USENIX Security Symposium (SEC)*. 2829–2846. https://www.usenix.org/conference/usenixsecurity21/presentation/fioraldi

[21] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining Incremental Steps of Fuzzing Research. In *USENIX Workshop on Offensive Technologies (WOOT)*. 12 pages. https://www.usenix.org/conference/woot20/presentation/fioraldi

[22] Phyllis G. Frankl and Stewart N. Weiss. 1993. An Experimental Comparison of the Effectiveness of Branch Testing and Data Flow Testing. *IEEE Transactions on Software Engineering* 19, 8 (1993), 774–787. https://doi.org/10.1109/32.238581

[23] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. 2020. GREYONE: Data Flow Sensitive Fuzzing. In *USENIX Security Symposium (SEC)*. 2577–2594. https://www.usenix.org/conference/usenixsecurity20/presentation/gan

[24] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. 2018. CollAFL: Path Sensitive Fuzzing. In *IEEE Symposium on Security and Privacy (SP)*. 679–696. https://doi.org/10.1109/SP.2018.00040

[25] Harrison Green and Thanassis Avgerinos. 2022. GraphFuzz: Library API Fuzzing with Lifetime-Aware Dataflow Graphs. In *International Conference on Software Engineering (ICSE)*. ACM/IEEE, 1070–1081. https://doi.org/10.1145/3510003.3510228

[26] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2020. Magma: A Ground-Truth Fuzzing Benchmark. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 4, 3, Article 49 (2020), 29 pages. https://doi.org/10.1145/3428334

[27] Hadi Hemmati. 2015. How Effective Are Code Coverage Criteria?. In *IEEE International Conference on Software Quality, Reliability and Security (QRS)*. 151–156. https://doi.org/10.1109/QRS.2015.30

[28] John L. Henning. 2006. SPEC CPU2006 Benchmark Descriptions. *ACM SIGARCH Computer Architecture News* 34, 4 (2006), 1–17. https://doi.org/10.1145/1186736.1186737

[29] Adrian Herrera, Hendra Gunadi, Shane Magrath, Michael Norrish, Mathias Payer, and Antony L. Hosking. 2021. Seed Selection for Successful Fuzzing. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 230–243. https://doi.org/10.1145/3460319.3464795

[30] Adrian Herrera, Mathias Payer, and Antony L. Hosking. 2021. Registered Report: datAFLow—Towards a Data-Flow-Guided Fuzzer. In *Fuzzing Workshop (FUZZING)*. The Internet Society, 11 pages. https://doi.org/10.14722/fuzzing.2022.23001

[31] Joseph R. Horgan and Saul London. 1991. Data Flow Coverage and the C Language. In *Symposium on Testing, Analysis, and Verification (TAV)*. ACM, 87–97. https://doi.org/10.1145/120807.120815

[32] Joseph R. Horgan, Saul London, and R. Lyu. 1994. Achieving Software Quality with Testing Coverage Measures. *Computer* 27, 9 (1994), 60–69. https://doi.org/10.1109/2.312032

[33] Chin-Chia Hsu, Che-Yu Wu, Hsu-Chun Hsiao, and Shih-Kun Huang. 2018. INSTRIM: Lightweight Instrumentation for Coverage-guided Fuzzing. In *Workshop on Binary Analysis Research (BAR)*. The Internet Society, 7 pages. https://doi.org/10.14722/bar.2018.23014

[34] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. 1994. Experiments of the Effectiveness of Dataflow- and Controlflow-Based Test Adequacy Criteria. In *International Conference on Software Engineering (ICSE)*. ACM/IEEE, 191–200. https://doi.org/10.1109/ICSE.1994.296778

[35] Yuseok Jeon, Priyam Biswas, Scott Carr, Byoungyoung Lee, and Mathias Payer. 2017. HexType: Efficient Detection of Type Confusion Errors for C++. In *ACM SIGSAC Computer and Communications Security (CCS)*. 2373–2387. https://doi.org/10.1145/3133956.3134062

[36] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. 2011. DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation. In *Network and Distributed Systems Security Symposium (NDSS)*. The Internet Society, 14 pages.

[37] Se-Won Kim, Xavier Rival, and Sukyoung Ryu. 2018. A Theoretical Foundation of Sensitivity in an Abstract Interpretation Framework. *ACM Transactions on Programming Languages and Systems* 40, 3 (2018), 44 pages. https://doi.org/10.1145/3230624

[38] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2123–2138. https://doi.org/10.1145/3243734.3243804

[39] James Kukucka, Luís Pina, Paul Ammann, and Jonathan Bell. 2022. CONFETTI: Amplifying Concolic Guidance for Fuzzers. In *International Conference on Software Engineering (ICSE)*. ACM/IEEE, 438–450. https://doi.org/10.1145/3510003.3510628

[40] Yuwei Li, Shouling Ji, Yuan Chen, Sizhuang Liang, Wei-Han Lee, Yueyao Chen, Chenyang Lyu, Chunming Wu, Raheem Beyah, Peng Cheng, Kangjie Lu, and Ting Wang. 2021. UNIFUZZ: A Holistic and Pragmatic Metrics-Driven Platform for Evaluating Fuzzers. In *USENIX Security Symposium (SEC)*. 2777–2794. https://www.usenix.org/conference/usenixsecurity21/presentation/li-yuwei

[41] Zhengyang Liu and John Criswell. 2017. Flexible and Efficient Memory Object Metadata. In *ACM SIGPLAN International Symposium on Memory Management (ISMM)*. 36–46. https://doi.org/10.1145/3092255.3092268

[42] LLVM Project. 2022. libFuzzer—a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html

[43] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. MOPT: Optimized Mutation Scheduling for Fuzzers. In *USENIX Security Symposium (SEC)*. 1949–1966. https://www.usenix.org/conference/usenixsecurity19/presentation/lyu

[44] Valentin M. Manes, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. 2021. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering* 47, 11 (2021), 2312–2331. https://doi.org/10.1109/TSE.2019.2946563

[45] Henry B. Mann and Donald R. Whitney. 1947. On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *The Annals of Mathematical Statistics* 18, 1 (1947), 50–60. https://doi.org/10.1214/aoms/1177730491

[46] Nathan Mantel. 1966. Evaluation of survival data and two new rank order statistics arising in its consideration. *Cancer Chemotherapy Reports* 50, 3 (1966), 163–170.

[47] Alessandro Mantovani, Andrea Fioraldi, and Davide Balzarotti. 2022. Fuzzing with Data Dependency Information. In *IEEE European Security and Privacy (EuroSP)*. 286–302. https://doi.org/10.1109/EuroSP53844.2022.00026

[48] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. 2021. FuzzBench: An Open Fuzzer Benchmarking Platform and Service. In *European Software Engineering Conference and Symposium on Foundations of Software Engineering (ESEC/FSE)*. ACM, 1393–1403. https://doi.org/10.1145/3468264.3473932

[49] Barton P. Miller, Louis Fredriksen, and Bryan So. 1990. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM* 33, 12 (1990), 32–44. https://doi.org/10.1145/96267.96279

[50] Matt Miller. 2019. Trends and Challenges in the Vulnerability Mitigation Landscape.

[51] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. 2005. CCured: Type-safe Retrofitting of Legacy Software. *ACM Transactions on Programming Languages and Systems* 27, 3 (2005), 477–526. https://doi.org/10.1145/1065887.1065892

[52] Neelofar, Kate Smith-Miles, Mario Andrés Muñoz, and Aldeida Aleti. 2022. Instance Space Analysis of Search-Based Software Testing. *IEEE Transactions on Software Engineering* (2022), 1–20. https://doi.org/10.1109/TSE.2022.3228334

[53] Carlos Oliveira, Aldeida Aleti, Lars Grunske, and Kate Smith-Miles. 2018. Mapping the Effectiveness of Automated Test Suite Generation Techniques. *IEEE Transactions on Reliability* 67, 3 (2018), 771–785. https://doi.org/10.1109/TR.2018.2832072

[54] Sebastian Poeplau and Aurélien Francillon. 2020. Symbolic execution with SymCC: Don't interpret, compile!. In *USENIX Security Symposium (SEC)*. 181–198. https://www.usenix.org/conference/usenixsecurity20/presentation/poeplau

[55] Sandra Rapps and Elaine J. Weyuker. 1985. Selecting Software Test Data Using Data Flow Information. *IEEE Transactions on Software Engineering* 11, 4 (1985), 367–375. https://doi.org/10.1109/TSE.1985.232226

[56] Matt Ruhstaller and Oliver Chang. 2018. A New Chapter for OSS-Fuzz. https://security.googleblog.com/2018/11/a-new-chapter-for-oss-fuzz.html

[57] Christopher Salls, Aravind Machiry, Adam Doupe, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2020. Exploring Abstraction Functions in Fuzzing. In *IEEE Conference on Communications and Network Security (CNS)*. 1–9. https://doi.org/10.1109/CNS48642.2020.9162273

[58] Kostya Serebryany. 2017. OSS-Fuzz—Google's continuous fuzzing service for open source software. In *USENIX Security Symposium (SEC)*. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/serebryany

[59] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX Annual Technical Conference (ATC)*. 309–318. https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

[60] Dongdong She, Yizheng Chen, Baishakhi Ray, and Suman Jana. 2020. Neutaint: Efficient Dynamic Taint Analysis with Neural Networks. In *IEEE Symposium on Security and Privacy (SP)*. 1527–1543. https://doi.org/10.1109/SP40000.2020.00022

[61] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. 2019. SoK: Sanitizing for Security. In *IEEE Security and Privacy (SP)*. 1275–1295. https://doi.org/10.1109/SP.2019.00010

[62] Ting Su, Ke Wu, Weikai Miao, Geguang Pu, Jifeng He, Yuting Chen, and Zhendong Su. 2017. A Survey on Data-Flow Testing. *Comput. Surveys* 50, 1, Article 5 (2017), 35 pages. https://doi.org/10.1145/3020266

[63] Yulei Sui and Jingling Xue. 2016. SVF: Interprocedural Static Value-Flow Analysis in LLVM. In *Compiler Construction (CC)*. ACM, 265–266. https://doi.org/10.1145/2892208.2892235

[64] William N. Sumner and Xiangyu Zhang. 2010. Memory Indexing: Canonicalizing Addresses across Executions. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. 217–226. https://doi.org/10.1145/1882291.1882324

[65] Robert Swiecki. 2016. honggfuzz. http://honggfuzz.com/

[66] The Clang Team. 2022. DataFlowSanitizer. https://clang.llvm.org/docs/DataFlowSanitizer.html

[67] The Clang Team. 2022. Source-based Code Coverage. https://clang.llvm.org/docs/SourceBasedCodeCoverage.html

[68] Jonas Benedict Wagner. 2017. *Elastic Program Transformations: Automatically Optimizing the Reliability/Performance Trade-off in Systems Software.* Ph. D. Dissertation. EPFL. https://doi.org/10.5075/epfl-thesis-7745

[69] Jinghan Wang, Yue Duan, Wei Song, Heng Yin, and Chengyu Song. 2019. Be Sensitive and Collaborative: Analyzing Impact of Coverage Metrics in Greybox Fuzzing. In *USENIX International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. 1–15. https://www.usenix.org/conference/raid2019/presentation/wang

[70] Yanhao Wang, Xiangkun Jia, Yuwei Liu, Kyle Zeng, Tiffany Bao, Dinghao Wu, and Purui Su. 2020. Not All Coverage Measurements Are Equal: Fuzzing by Coverage Accounting for Input Prioritization. In *Network and Distributed Systems Security Symposium (NDSS)*. The Internet Society, 17 pages. https://doi.org/10.14722/ndss.2020.24422

[71] Bin Xin, William N. Sumner, and Xiangyu Zhang. 2008. Efficient Program Execution Indexing. In *ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. 238–248. https://doi.org/10.1145/1375581.1375611

[72] Wen Xu, Sanidhya Kashyap, Changwook Min, and Taesoo Kim. 2017. Designing New Operating Primitives to Improve Fuzzing Performance. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2313–2328. https://doi.org/10.1145/3133956.3134046

[73] Michał Zalewski. 2015. American Fuzzy Lop (AFL). http://lcamtuf.coredump.cx/afl/

[74] Zenong Zhang, Zach Patterson, Michael Hicks, and Shiyi Wei. 2022. FIXREVERTER: A Realistic Bug Injection Methodology for Benchmarking Fuzz Testing. In *USENIX Security Symposium (SEC)*. 3699–3715. https://www.usenix.org/conference/usenixsecurity22/presentation/zhang-zenong