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

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

Mathias Payer
EPFL
mathias.payer@nebelwelt.net

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

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

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

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

I. INTRODUCTION

Fuzzers are an indispensable tool in the software-testing
toolbox. The idea of fuzzing—to test a target program by
posing inputs reaching new code. Intuitively, a fuzzer cannot find bugs
because it has no knowledge of the target’s internals.

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

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

1Miller et al.’s original fuzzer [1] is now known as a blackbox fuzzer,
because it has no knowledge of the target’s internals.
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 performs 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 in an effort to reduce overhead and thus maximize fuzzing throughput.

We perform a preliminary evaluation of DATAFLOW’s effectiveness. So far, our results indicate data-flow-driven fuzzing provides little advantage over control-flow-driven fuzzing for most targets we evaluated. However, data-flow-driven fuzzing appears to have a niche, showing promise on targets where control flow and semantics are decoupled. We will continue this evaluation on acceptance of this paper.

Our contributions can be summarized as follows:

1) A framework for reasoning about and constructing data flow-based coverage metrics for greybox fuzzing;
2) A data-flow-driven fuzzer, DATAFLOW, to explore data flows in a target program at low overhead; and
3) A preliminary evaluation and comparison of control-flow, taint-analysis, and data-flow-driven fuzzers.

We make our material available at [https://github.com/HexHive/datAFLow](https://github.com/HexHive/datAFLow).

II. BACKGROUND & RELATED WORK

A. Fuzzing

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. Rather, they use a feedback loop based on a coverage metric. This feedback loop guides the fuzzer towards generating inputs that explore new parts 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 (the “target”), and (b) an optional set of starting inputs (an “empty seed” is used if not provided).

The fuzzer places the inputs 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 the latter back into the queue). This process repeats until the “residual risk” of a missed bug falls beneath a suitable threshold.

B. Data-flow Analysis

Data-flow analysis typically refers to a collection of techniques for reasoning about the runtime 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 runtime. These actions are typically analyzed in terms of the interactions between a variable’s definition—or def site—and how that variable is used at one or more use sites. 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 and comparing program executions. However, to the best of our knowledge, these data-flow techniques have not yet been explored by the fuzzing community.

C. Related Work

Fuzzing is an active research area. Consequently, we focus on recent fuzzing research related to coverage metrics.

The most popular fuzzers are those guided by code coverage. Typically, this code coverage 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 §III it is not without its own issues. Indeed, TortoiseFuzz showed that basic-block coverage can be effective when paired with other coverage metrics that increase sensitivity (e.g., function call and loop coverage).

To improve mutation precision, some fuzzers use dynamic taint analysis (DTA) to track input bytes. This information is used to infer which bytes to mutate. Unfortunately, DTA suffers from accuracy and performance issues, limiting deployment. To overcome performance issues, Angora amortizes DTA cost by limiting its application to once per input (over many mutations). Other fuzzers avoid DTA in favor of approximate taint tracking; e.g., REDQUEEN uses input-to-state correspondence, based on the idea that “parts of the input directly correspond to the memory or registers at runtime”. Similarly, GREYONE infers taint by monitoring the value of variables as input bytes are mutated.

Alternatives to code coverage metrics are also being explored. Coppik et al. instrument the target’s memory accesses, storing this information in the fuzzer’s coverage map. JON 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). Finally, INVSOC 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).
Despite the body of work related to fuzzer coverage metrics, pure data flow coverage remains an underexplored metric. This is likely due to the perceived runtime cost of measuring data flow [32, 34]. Nevertheless, we hypothesize that lightweight data flow tracking is possible. To this end, we introduce DATAFLOW, the first data-flow-driven greybox fuzzer with a tunable sensitivity range.

III. MOTIVATING DATA-FLOW COVERAGE

```c
1 unsigned int max; // Set by the user
2 unsigned int i = 0, j = 0;
3 char *prime = (char *) 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: Motivation for data-flow coverage. This example code implements the Sieve of Erathosthenes for finding all prime numbers up to max value.

A fuzzer’s coverage metric should accurately capture/approximate program behavior with minimal runtime overheads. Here we discuss why control-flow-based metrics are not enough 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 runtime 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 determine which input mutations lead to new program behaviors). However, the process for uncovering new behaviors can be highly inefficient, because a fuzzer driven by code coverage alone cannot identify which mutated input bytes led to new program behavior. Differences in data access and manipulation within a single code path are lost.

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 5, 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 may not be captured by the taint policy. For example, compiler-based DTA—e.g., LLVM’s DataFlowSanitizer (DFSan) [35]—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 a significant amount of 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, unsupported instructions) [20].

DTA is also expensive. She et al. [18] found that 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 top of DFSan) exhibited a runtime overhead of 33.31× over the same uninstrumented code from the SPEC CPU2006 benchmark suite. 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) [18].

Given the disadvantages of DTA (accuracy and cost), we propose an alternate 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.

IV. DESIGN AND IMPLEMENTATION

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, focusing on uncovering memory-safety vulnerabilities.

A. Coverage Sensitivity

Based on §II-B, we define data-flow coverage as follows:

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

This definition allows us to explore data-flow-based coverage metrics with different sensitivities [32, 36]. We adhere to 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 [32]. Different sensitivities allow us to balance efficacy and performance: more sensitive metrics incur a higher performance penalty. For example, edge coverage can be made more sensitive by incorporating context-sensitivity. However, this requires additional instrumentation, increasing runtime overhead [36].

Like traditional data-flow analysis (§II-B), our data-flow coverage metrics require the identification of variable def and
would traditionally kill data flow variable use sites. Following Horgan and London [26], we define a definition. 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 [17, 38]. We believe reducing precision by not killing definitions is a suitable tradeoff to maintain scalability.

Once def and use sites are identified, DATAFLOW instruments these sites (using compiler-based instrumentation, discussed in IV-B) so that defluse chains can be tracked at runtime. However, exactly which defluse sites are instrumented (and hence which are tracked) depends on the required sensitivity. Inspired by Wang et al. [32], 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 the threats to validity with this approach in §IV-C).

1) Def Site Sensitivity: Complete data-flow coverage requires all variable def sites to be identified and instrumented. Unfortunately, the overhead to achieve this level of sensitivity is prohibitively expensive [39]. 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, to reduce 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 [40, 41]. Type-based partitioning requires some upfront knowledge of the target to ensure meaningful variables are tracked at runtime. For example, important program behaviors (and hence bugs) may be missed 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.

2) Use Site Sensitivity: Fig. 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. This is akin to traditional data-flow testing, which focuses on the values that variables take at runtime [14, 17], and is similar to GREYONE, which monitors (a subset of) program variables and their values to infer taint [19]. Depending on the def site sensitivity, this approach will quickly saturate the fuzzer’s coverage map (due to the path collision problem [9]); a middle ground between this overly sensitive approach and simple accesses is required.

This middle ground is achieved 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 Line 9 in Fig. 2), as defluse chains are now differentiated by the offset at which an array/struct is accessed.

Information at different granularities is recorded at use sites. When recording more precise information, care must be taken to ensure the coverage map does not saturate, clogging the fuzzing queue.

3) Composing Sensitivity Lattices: Different defluse sensitivities can be composed to track data flow at different granularities. We reuse the code in Fig. 2 to illustrate how we achieve 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 ⇝ Line 7 and Line 3 ⇝ Line 9. This is essentially equivalent to basic block coverage (per §II-A): 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 1 and 3). This provides a more complete view of how prime is used with negligible overhead (our implementation incurs a 3% overhead over the simple data flow coverage for the code in Fig. 2). In some respects this is similar to MemFUZZ’s approach, which incorporates memory accesses into code coverage [7]. This results in \(2 \times (\max - 2)\) 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 [14]. This is also similar to GREYONE’s “taint inference”, which looks at the value of variables used in path constraints [19].

Again, this level of sensitivity results in \(2 \times (\max - 2)\) def/use chains. Here, the values prime can take are fully deterministic. However, in general these values may depend on user input, and therefore will quickly saturate the fuzzer’s coverage map.

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

B. Implementation

...Figure 4 depicts DATAFLOW’s high-level architecture, including: (i) compiler instrumentation for capturing def/use sites at the desired sensitivity (§IV-B1); and (ii) runtime libraries for tracking data flows between instrumented def/use sites and feeding this information to the fuzzing engine (§IV-B2).

Our architecture is agnostic to the underlying fuzzer. Thus, the instrumented target produced by the compiler and linked with our runtime libraries 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.

1) Compiler Instrumentation: DATAFLOW’s compiler-based instrumentation is realized through a set of LLVM (v12) passes (2,270 LOC). These passes identify and instrument def and use sites (at the IR level) so flows between these sites—i.e., def/use chains—can be tracked at runtime.

Def/use site identification: Variable def and use sites must first be identified so data flows between these sites can be tracked. Per §IV-A, the selection of def sites to instrument impacts coverage sensitivity: more instrumented def sites leads to more complete data flow coverage. We implement a number of def site instrumentation schemes based on the type-based partitioning described in §IV-A1. Restricting def sites to arrays (allowing us to focus on memory-safety bugs, which remain one of the most common bug classes [42]) limits use sites to memory access instructions. We apply existing LLVM transformations, allowing us to focus on load and store instructions (both of which are trivial to identify and hence instrument). Which of these instructions are instrumented depends on the use sensitivity required (configurable at compile time).

Def/use site instrumentation: Previously-identified def and use sites are instrumented so the fuzzer can track def/use chains. Dynamically-allocated array def sites are instrumented by replacing the memory allocation function (e.g., malloc) with a tagged version (e.g., __tagged_malloc) accepting an additional argument: a random 16-bit integer identifying (i.e., tagging) the def site. This approach is analogous to AFL’s static assignment of basic block identifiers (which are also random 16-bit integers) to track edge coverage. For static arrays (i.e., stack, global), we adopt an approach similar to CCured’s and heapify these variables [43]. While heapification incurs runtime overheads unacceptable in production environments, we find these overheads acceptable for fuzzing.

We reuse the code from Fig. 2 to demonstrate DATAFLOW’s tagging operation. The allocation of prime (Line 3 in Fig. 2) is rewritten and tagged with the identifier 0x123 (Line 6 in Fig. 5). Use sites (i.e., memory accesses) are similarly instrumented. Figure 5 is an example of this instrumentation: both writes (Line 13) and reads (Line 10) to/from prime are instrumented with a call to __mem_access (discussed in §IV-B2). The offset at which prime is accessed is also statically determined (or set to zero for less-sensitive coverage metrics). We reuse a number of techniques from LLVM’s AddressSanitizer (ASan) [44] to limit the number of instrumentation sites, thereby reducing overhead without sacrificing precision.

This combination of heapification, allocation site tagging, and memory access instrumentation enables tracking the run-
extern void __tagged_malloc(tag_t tag, size_t s);
extern void __mem_access(void *ptr, int offset);

size_t max; // Set by the user
unsigned int size_t max; // Set by the user

for (i = 2; i < max; ++i) {
    __mem_access(prime, i);
    __mem_access(prime, i * j);
}

Fig. 5: Instrumented Sieve of Eratosthenes.

time uses of variables. We achieve this via our memory allocator, fuzzalloc.

2) Runtime Libraries: We reduce the runtime tracking of data-flow to a metadata management problem (def site tags are the metadata that must be efficiently retrieved at use sites). We adopt a form of low-fat pointer [45–47] to implicitly store the 16-bit def site tag within the pointer itself. This approach provides a number of advantages—particularly over (mid-)fat and tagged pointers [43, 48–50]—including compatibility with uninstrumented/legacy code and cheap metadata access.

The design of our low-fat pointer system is similar to Duck and Yap [45–47]: we implement a custom memory allocator, fuzzalloc, that exploits the large virtual address space provided by the x86_64 architecture (which we assume for DATAFLOW, because low-fat pointers are only practical on architectures with sufficient pointer bit-width). The fuzzalloc API consists of tagged versions of malloc, calloc, and realloc. These tagged functions (inserted by the compiler at def sites, per [LV-BT]) provide a mechanism for mapping heap-allocated data to def sites tags.

This mapping is achieved by allocating separate “memory spaces” for each def site such that the tag is stored in the upper 16-bits of the memory space’s address. Consequently, our low-fat pointer can be encoded in the following type:

union {
    void *ptr;
    struct {
        uintptr_t def_site:16; // MSB
        uintptr_t unused:48;
    };
} p;

Fuzzalloc leverages ptrmalloc’s (v3) mspace feature for partitioning the heap into independent “memory spaces” [51] (237 LOC). Each allocation site is assigned its own mspace, allowing us to directly map def site tags to msapces. Mspaces are mapped into memory (via a combination of address-space shrinking techniques [48] and mmap) such that the upper 16-bits of the mspace’s address space contains the def site tag. This process is illustrated in Fig. 6 (at 1 and 2).

Fig. 6 shows how a def site tag is retrieved from a low-fat pointer allocated by fuzzalloc (at 3). X86_64 restricts addresses to the lower 48-bits of a pointer, so the tag can be retrieved by right-shifting the pointer by 32-bits (in __mem_access).

Unlike def sites, which are identified by a compile-time tag, we use the program counter to identify use sites (at 4). Retrieving the program counter at the use site is an inexpensive operation: on x86_64 it is accessible via the lea instruction.

3) Fuzzer Integration: Fuzzalloc constructs a def/use chain by hashing together the def and use sites (at 3). This hash is used as a lookup into the fuzzer’s coverage map to guide the fuzzer towards 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 [29].

In particular, we use coarse data-flow coverage metrics—def/use chain hit counts stored in a compact bitmap—to achieve efficient fuzzing. While it is well known such techniques result in path collisions [9], we are willing to tolerate such imprecision to limit overhead costs. Coarse coverage metrics also lower implementation costs, as they enable the reuse of existing fuzzing engines (in our case, AFL++) [8].

The following hash function maintains coverage:

\[(3 \times (\text{def} - \text{DEFAULT_TAG} \oplus \text{use}) - \text{use})\]

This hash function is designed such that uninstrumented def sites (e.g., allocations made in linked libraries) all resolve to the same bitmap index. All uninstrumented allocations are implicitly tagged with the DEFAULT_TAG def site identifier. This results in the hash calculation \((3 \times 0 \oplus \text{use}) - \text{use}\), which simplifies to zero.

Finally, we modify ASan in order to detect a greater range of memory safety bugs. This ensures dynamic memory allocation requests are always routed through fuzzalloc.

C. Threats to Validity

1) Def Site Selection: Our def site selection approach ([LV-A1]) is incomplete: important data flows may be missed if the appropriate def sites are not instrumented. For example, our focus on array def sites means we may miss other relevant data flows. We are willing to accept this trade-off, given (a) our focus on memory safety vulnerabilities, and (b) the prohibitive runtime overheads when tracking all def sites.

2) Array Def Site Identification: Identifying array def sites is complicated by the fact that 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 our prototype requires the user to manually find these wrappers, existing tools [52] could assist in this process. We statically track the use of memory allocation routines (including wrappers) and detect when they
3) C++ Dynamic Memory Allocation: C++ new calls are rewritten as malloc calls to simplify our instrumentation. However, this prevents us from handling any std::bad_alloc exceptions. This means any failed allocations will cause a program crash, irrespective of any exception handlers in place. These false negatives are filtered out by replaying the inputs through the original binary.

4) Coverage Imprecison: Storing coarse coverage information in a compact bitmap is inherently inaccurate and incomplete [9]. 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][7][10][12][19][31][34].

V. EVALUATION

We perform a preliminary evaluation of DATAFLOW, comparing it against state-of-the-art greybox and DTA-based fuzzers (\texttt{V-B}). We describe future evaluation in \texttt{V-C}.

A. Methodology

Fuzzer Selection: Our evaluation aims to compare the performance of fuzzers using (i) pure control-flow-based coverage; (ii) pure data-flow-based coverage; and (iii) exact and approximate DTA, combining control-flow coverage with data-flow tracking. We select AFL++ as the pure control-flow-driven fuzzer because it is the current state-of-the-art coverage-guided greybox fuzzer. We configure AFL++ with: (i) link-time optimization instrumentation, eliminating hash collisions; (ii) the forkserver disabled, because it is currently unsupported by fuzzalloc; and (iii) with and without “CmpLog” instrumentation. Cmplog—inspired by REDQUEEN’s input-to-state correspondence—approximates DTA by capturing comparison operands. We select Angora as the exact-DTA-based fuzzer. We configure DATAFLOW by (a) restricting def instrumentation to arrays (static and dynamic), and (b) using two use site sensitivities: simple access and access with offset. We refer to these sensitivities as “A” and “A+O”, respectively.

Benchmark Selection: We evaluate our fuzzers on a subset of the Magma benchmark [53] (for bug finding) and the jq JSON processor [54] (for coverage). We select the subset of Magma targets all fuzzers successfully build and run\footnote{Angora failed to run sndfile_fuzzer, \texttt{php} failed to build with CmpLog instrumentation, and DATAFLOW failed to build openssl.}. We select jq because its yacc-based LR parser exemplifies the decoupling of control structure from semantics [13].

Experimental Setup: All experiments were conducted on an Ubuntu 20.04 AWS EC2 instance with a 48-core Intel® Xeon® Platinum 8000 3.6 GHz CPU and 192 GiB of RAM. Each fuzz run was conducted for 24 h and repeated five times. Magma targets were bootstrapped with the provided seeds, while jq was bootstrapped with an afl-cmin-minimized corpus of JSON files sourced from Herrera et al. [24]. 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.

B. Preliminary Results

Following prior work [24][53][55], we use survival analysis to summarize our bug-finding results. Table\footnote{Angora failed to run sndfile_fuzzer, \texttt{php} failed to build with CmpLog instrumentation, and DATAFLOW failed to build openssl.} uses the restricted mean survival time (RMST), measuring the average time for a bug to “survive” (i.e., remain undiscovered) up to a specified time point (here, 24 h, the length of a fuzz run).

The number of bugs triggered across all fuzzers is lower (and their RMSTs higher) compared to previous Magma evaluations [24][53]. We attribute this to the disabled forkserver, which impacts fuzzer throughput. Even with this performance regression, DATAFLOW triggered two bugs (LUA002 and LUA003) not previously triggered by any other fuzzer in prior evaluations. DATAFLOW also found XML001, which remained untriggered by AFL++ and Angora in this evaluation.

Like FuzzBENCH [50], we compare coverage by replaying the fuzzing corpus through Clang’s source-based coverage instrumentation (Fig.\footnote{Angora failed to run sndfile_fuzzer, \texttt{php} failed to build with CmpLog instrumentation, and DATAFLOW failed to build openssl.}). We also replay the same corpora through the DATAFLOW-instrumented jq to gain a sense of...
the `def/use` chains covered by each fuzzer (Figs. 7b and 7c).  

Figure 7a shows Angora and AFL++ (with and without `CmpLog`) cover ∼3% more code than DATAFLOW. DATAFLOW remains competitive, despite the other fuzzers using a control-flow-based coverage metric. Notably, the use of data-flow analyses (exact and approximate DTA) offers no statistically-significant improvement over using edge coverage alone. Figures 7b and 7c show the value of a tunable sensitivity range. When using memory accesses alone (the least sensitive metric), `def/use` coverage is subsumed by edge coverage. However, when access offsets are considered, DATAFLOW offers a ∼5% improvement in `def/use` coverage over AFL++, and ∼35% over Angora. This is likely due to `g`’s table-driven parser, supporting the intuition of Xin et al. [13].

C. Future Evaluation

We intend to test the following hypothesis:

\[ \text{Data-flow-driven fuzzing offers superior performance on targets where control-flow is decoupled from semantics.} \]

Testing this hypothesis requires a more thorough evaluation of DATAFLOW on a wider range of targets (e.g., those from Google’s FuzzBench [56]) and a comparison against more data-flow-driven fuzzers (e.g., SIVO [55]) and a wider range of `def/use` sensitivities. We intend to use the same methodology and statistical analyses described in Sections V-A and V-B (i.e., survival analysis to summarize bug-finding results and corpus replay to compare control- and data-flow-based coverage). Specifically, we will perform the following experiments:

Understanding overheads: Does improving DATAFLOW’s performance change our results? We will answer this question by (a) adding support for AFL++’s forserver (which we have completed), and (b) investigating where heapify operations can be removed (e.g., on local variables that do not escape to other execution threads or functions). Similarly, we aim to better understand the impact of heapification by also heapifying `def` sites in AFL++.

Characterizing programs: Can we determine a priori if a given target is amenable to data-flow-driven fuzzing? To answer this question, we propose developing a static analysis—based on techniques proposed by Chaim et al. [33]—for determining whether a `def/use` chain is subsumed by a control-flow measure (e.g., node, edge coverage). Fuzzing with DATAFLOW may be redundant if the majority of `def/use` chains are subsumed by control-flow measures.

Quantifying data-flow coverage: Control-flow coverage can be quantified by reasoning over the target’s CFG. This is commonly achieved by replaying the fuzzer’s queue through an independent, collision-free coverage metric (e.g., Clang’s source-based coverage [56]). However, the equivalent process for quantifying data-flow coverage does not exist. We propose computing an upper-bound of the target’s `def/use` chains using an LLVM-based static analysis (e.g., based on SVF [59]). This allows us to quantify the percentage of `def/use` chains executed during fuzzing, much like the percentage of lines-of-code is quantified for control-flow coverage.

VI. 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 expected data-flow-driven fuzzing to offer drastic benefits over traditional control-flow-driven greybox fuzzing. Instead, our preliminary evaluation shows data-flow-based coverage metrics offer little benefit over traditional control-flow-based coverage metrics on most targets. However, data-flow-driven fuzzing does show promise on programs where control flow is decoupled from semantics.

Notably, we also found other data-flow analyses used by fuzzers (exact and approximate DTA) provided little benefit over pure control-flow-based coverage metrics in most cases. Further investigation is required to shed light on why particular targets and bugs do benefit from data-flow analyses.

We intend to perform further evaluation and analysis to understand the advantages/disadvantages of control- vs. data-flow-based coverage metrics.

REFERENCES

TABLE I: Magma bugs, presented as the RMST (in hours) with 95% confidence interval (CI). Bugs never found by a particular fuzzer have an RMST of $\top$ (to distinguish bugs with a 24h RMST). We only report the RMST for bugs triggered; bugs not triggered by any fuzzer are omitted. Lower RMSTs are better.

<table>
<thead>
<tr>
<th>Target</th>
<th>Driver</th>
<th>Bug</th>
<th>AFL++</th>
<th>AFL++ (cmplog)</th>
<th>Angora</th>
<th>DATAFLOW (A)</th>
<th>DATAFLOW (A+O)</th>
</tr>
</thead>
<tbody>
<tr>
<td>libpng</td>
<td>libpng_read_fuzzer</td>
<td>PNG003</td>
<td>0.13 ± 0.05</td>
<td>0.07 ± 0.05</td>
<td>0.01 ± 0.01</td>
<td>$\top$</td>
<td>$\top$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PNG006</td>
<td>$\top$</td>
<td>1.32 ± 0.56</td>
<td>0.05 ± 0.02</td>
<td>$\top$</td>
<td>$\top$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TIF002</td>
<td>$\top$</td>
<td>$\top$</td>
<td>22.82 ± 2.66</td>
<td>$\top$</td>
<td>$\top$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TIF007</td>
<td>0.62 ± 0.29</td>
<td>0.51 ± 0.31</td>
<td>0.70 ± 0.73</td>
<td>$\top$</td>
<td>$\top$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TIF012</td>
<td>2.89 ± 1.48</td>
<td>10.54 ± 4.18</td>
<td>16.42 ± 7.63</td>
<td>$\top$</td>
<td>$\top$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TIF014</td>
<td>12.50 ± 3.42</td>
<td>20.85 ± 3.30</td>
<td>$\top$</td>
<td>$\top$</td>
<td></td>
</tr>
<tr>
<td>libtiff</td>
<td>tiff_read_rgba_fuzzer</td>
<td>TIF002</td>
<td>$\top$</td>
<td>$\top$</td>
<td>19.46 ± 10.28</td>
<td>$\top$</td>
<td>$\top$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TIF005</td>
<td>$\top$</td>
<td>$\top$</td>
<td>14.37 ± 9.87</td>
<td>$\top$</td>
<td>$\top$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TIF006</td>
<td>$\top$</td>
<td>$\top$</td>
<td>9.51 ± 8.87</td>
<td>15.99 ± 6.61</td>
<td>18.82 ± 6.62</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TIF007</td>
<td>0.20 ± 0.06</td>
<td>0.57 ± 0.38</td>
<td>0.46 ± 0.22</td>
<td>0.12 ± 0.03</td>
<td>0.21 ± 0.12</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TIF009</td>
<td>$\top$</td>
<td>$\top$</td>
<td>19.58 ± 6.37</td>
<td>$\top$</td>
<td>$\top$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TIF012</td>
<td>4.15 ± 1.43</td>
<td>13.97 ± 6.23</td>
<td>7.90 ± 6.77</td>
<td>1.38 ± 1.09</td>
<td>$\top$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TIF014</td>
<td>12.33 ± 3.97</td>
<td>17.99 ± 5.95</td>
<td>23.43 ± 1.28</td>
<td>11.95 ± 8.92</td>
<td>7.74 ± 6.97</td>
</tr>
<tr>
<td>libxml2</td>
<td>xml_read_memory_fuzzer</td>
<td>XML001</td>
<td>$\top$</td>
<td>$\top$</td>
<td>5.98 ± 1.67</td>
<td>$\top$</td>
<td>$\top$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>XML003</td>
<td>8.27 ± 6.14</td>
<td>5.37 ± 2.98</td>
<td>0.01 ± 0.01</td>
<td>$\top$</td>
<td>$\top$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>XML009</td>
<td>5.75 ± 2.22</td>
<td>11.76 ± 5.58</td>
<td>$\top$</td>
<td>$\top$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>XML017</td>
<td>0.21 ± 0.19</td>
<td>0.07 ± 0.06</td>
<td>0.01 ± 0.01</td>
<td>$\top$</td>
<td>$\top$</td>
</tr>
<tr>
<td>libxml2</td>
<td>xmlint</td>
<td>XML001</td>
<td>$\top$</td>
<td>$\top$</td>
<td>0.03 ± 0.04</td>
<td>0.03 ± 0.03</td>
<td>$\top$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>XML009</td>
<td>5.75 ± 2.97</td>
<td>6.14 ± 3.08</td>
<td>6.68 ± 0.44</td>
<td>7.28 ± 4.73</td>
<td>6.78 ± 2.04</td>
</tr>
<tr>
<td></td>
<td></td>
<td>XML017</td>
<td>0.05 ± 0.04</td>
<td>0.17 ± 0.15</td>
<td>0.01 ± 0.01</td>
<td>0.03 ± 0.03</td>
<td>0.03 ± 0.03</td>
</tr>
<tr>
<td>lua</td>
<td>lua</td>
<td>LUA002</td>
<td>$\top$</td>
<td>$\top$</td>
<td>20.56 ± 7.79</td>
<td>$\top$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>LUA003</td>
<td>$\top$</td>
<td>$\top$</td>
<td>22.35 ± 3.74</td>
<td>$\top$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>LUA004</td>
<td>21.32 ± 6.07</td>
<td>18.87 ± 6.89</td>
<td>0.08 ± 0.02</td>
<td>$\top$</td>
<td></td>
</tr>
<tr>
<td>sqlite3</td>
<td>sqlite3_fuzz</td>
<td>SQL002</td>
<td>8.97 ± 6.01</td>
<td>17.93 ± 8.14</td>
<td>$\top$</td>
<td>$\top$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQL014</td>
<td>22.08 ± 4.34</td>
<td>22.34 ± 3.67</td>
<td>$\top$</td>
<td>$\top$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQL018</td>
<td>23.90 ± 2.22</td>
<td>$\top$</td>
<td>$\top$</td>
<td>$\top$</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 7: jfq coverage for AFL++, AFL++ (cmplog), Angora, DATAFLOW (A), and DATAFLOW (A+O). Each plot shows the mean coverage (over five repeated runs) and 95% bootstrap CI. The x-axis shows wall clock time (in seconds) on a log scale.


