Fuzzing with Data Dependency Information

Alessandro Mantovani
EURECOM
mantovan@eurecom.fr

Andrea Fioraldi
EURECOM
fioraldi@eurecom.fr

Davide Balzarotti
EURECOM
balzarot@eurecom.fr

***Abstract*—Recent advances in fuzz testing have introduced several forms of feedback mechanisms, motivated by the fact that for a large range of programs and libraries, edge-coverage alone is insufficient to reveal complicated bugs. Inspired by this line of research, we examined existing program representations looking for a match between expressiveness of the structure and adaptability to the context of fuzz testing. In particular, we believe that data dependency graphs (DDGs) represent a good candidate for this task, as the set of information embedded by this data structure is potentially useful to find vulnerable constructs by stressing combinations of def-use pairs that would be difficult for a traditional fuzzer to trigger. Since some portions of the dependency graph overlap with the control flow of the program, it is possible to reduce the additional instrumentation to cover only "interesting" data-flow dependencies, those that help the fuzzer to visit the code in a distinct way compared to standard methodologies.**

To test these observations, in this paper we propose `DDFuzz`, a new approach that rewards the fuzzer not only with code coverage information, but also when new edges in the data dependency graph are hit. Our results show that the adoption of data dependency instrumentation in coverage-guided fuzzing is a promising solution that can help to discover bugs that would otherwise remain unexplored by standard coverage approaches. This is demonstrated by the 72 different vulnerabilities that our data-dependency driven approach can identify when executed on 38 target programs from three different datasets.

1. Introduction

In a society that makes software applications the central core of many every-day activities is essential to make such software as secure as possible before it is released to the public. This has led to a large amount of research focused on the development of increasingly sophisticated techniques to discover vulnerabilities, such as static software testing [36], [60], [77], symbolic execution [61], [62], [71] and dynamic analysis [73].

In the context of dynamic analysis, researchers have proposed many approaches to measure the coverage that a certain input produces in the software under testing. One of the possible metrics is *path coverage*, which refers to all independent paths present in a program. For example, in software testing, the community has focused on path coverage for tests generation [64], [70] with the goal of automatically producing inputs that can reach nested code locations. The main limitation of path coverage is that any non trivial application can contain a huge, and potentially infinite, number of paths [48], [67] thus making this approach a poor choice for a large set of programs. Because of this, other dynamic testing techniques, like *fuzz testing*, mostly rely on simpler forms of coverage, such as the popular *edge coverage*.

Fuzz testing or *fuzzing* is a dynamic analysis technique that consists of feeding a program with different input variations, with the goal of revealing a flaw in its underlying code. Originally proposed in the early '90 [52], fuzz testing has evolved enormously over the past decade thanks to numerous efforts that have been made to improve its individual components. For instance, researchers have tried to enhance fuzzers by designing better seed schedulers [21], [66] or by introducing novel instrumentation techniques [10], [31], [40], [49]. Fuzz testing is probably nowadays the most effective approach employed for automatic vulnerability discovery and for this reason public research on this topic has advanced extremely fast as confirmed by the fact that in 2020 alone more that 120 papers were published on this topic [51].

As a necessary condition for a fuzzer to discover a vulnerability is the ability to generate an input that reaches the location of the flaw in the target program, code-coverage has become one of the most common metrics to gauge the effectiveness of a fuzzer and the success of a fuzzing campaign. In this context, *edge coverage* became the de-facto standard to measure code coverage. According to this paradigm, all basic blocks in the control flow graph (CFG) of the binary are instrumented to reward the fuzzer with a feedback when new edges connecting two basic blocks are discovered. In this way, the fuzzing engine can keep track of the new discoveries in terms of program points and mutate the generated inputs so that they evolve towards the exploration of nested portions of code. This gave origin to what is commonly referred as *coverage-guided* fuzzing.

A common goal among researchers is to develop new techniques to increase edge coverage, which has led current state-of-the-art fuzzers to be very effective at visiting difficult-to-trigger code locations. For instance, Redqueen [11] can generate inputs that satisfy complicated conditions, like the ones that involve a comparison with some magic bytes. However, high code coverage alone does not always translate in a better ability to discover bugs, and therefore the fuzzing community had also explored other approaches based on alternative coverage metrics. In this direction, Parmesan [55] relies on the code locations instrumented with `AddressSanitizer` [69] and `UndefinedBehaviorSanitizer` [1] to build a

directed fuzzer that tries to uncover and stress such locations. Ankou [49] adopted instead a different fitness function to guide the fuzzer by considering different combinations of the branches during the execution of the target. These approaches have shown promising results at detecting new bugs, thus suggesting that looking for alternative coverage approximations could help fuzzers to find different vulnerabilities compared to those normally found by employing edge-coverage.

Historically fuzz testing has taken inspiration from program analysis techniques to implement novel solutions that, in the past, have allowed to increase the fuzzers' performances. In 2019, Chowdhury et al. [23] rely on static analysis to make the target easier to be fuzzed, by simplifying the loop exit conditions and determining the ranges of valid input that can reach some error locations. More recently, the study presented by Fioraldi et al. [28], relies on a sophisticated program analysis technique to retrieve likely invariants from the execution of a program, with the idea of adopting the violations of such invariants as a feedback for their prototype fuzzer. Another example of the combination of fuzzing and program analysis approaches is the use of taint analysis to support fuzzing. This idea was adopted, for instance, by Rawat et al. in 2017 [65] and Chen et al. in 2018 [20] to infer properties of the application that were used to generate more suitable input values, thus increasing the amount of visited code.

A common representation used in program analysis is the *Data Dependency Graph* (DDG), which could be used as a possible approximations of code coverage. DDG are very often adopted in other fields, such as compilers [38], [56], mostly used for code transformations or optimisations, and static software testing [77], [78] where interestingly it is used to determine the existence of vulnerable patterns in the source code. The fact that the DDG is already used for vulnerability discovery purposes suggests that it can be a good feedback candidate to drive a fuzzer to discover vulnerable paths. Our intuition behind this, is that the set of information contained in the DDG can provide the fuzzer with an additional feedback that the CFG alone cannot capture.

To verify our hypothesis we implemented a custom instrumentation approach, built on top of AFL++ [30] and LLVM [45], where the fuzzer is rewarded, on top of the traditional CFG instrumentation, every time a new significant DDG edge is explored. We encountered several challenges to make our instrumentation lightweight, incremental with respect to the edge coverage, and effective in terms of discovered bugs. We tested our prototype implementation, `DDFuzz`, over three different datasets including a custom one of 10 real world applications, the popular benchmarking service Fuzzbench [51] and a third suite of programs previously used in the state-of-the-art [13].

The findings show that adopting the DDG coverage to guide fuzzers can lead to the discovery of additional (and different) bugs compared with traditional approaches. For instance, our data-dependency empowered fuzzer revealed 26 bugs that the traditional AFL++ strategy missed in our custom evaluation, by introducing a modest overhead of 10% compared to a normal edge-coverage instrumented binary. Moreover, our approach results to be incremental also to other state-of-the-art approaches such as Context Sensitivity and Ngrams. The second evaluation on Fuzzbench [51] that contains a larger variety of programs and bugs emphasizes these aspects as our approach performs better than AFL++ in 5 cases while for 3 further applications it is still able to find different bugs. As a confirmation of our methodology, `DDFuzz` discovers 12 more bugs compared to AFL++ in the third and final evaluation.

In short, this paper provides the following contributions:

- We propose a novel instrumentation method, `DDFuzz`, and we show its benefits as well as its limitations on a total of 38 target applications.
- We establish a reliable criteria based on the codebase structure which allows to predict when the use of our approach should be adopted for a fuzzing campaign.
- We test `DDFuzz` against several other state-of-the-art instrumentation approaches as well as a large range of targets demonstrating how, for each case, our custom instrumentation differs in terms of detected vulnerabilities.

The code of our prototype is available at https://github.com/elManto/DDFuzz.

2. Background

2.1. Data Dependency Graphs

Data Dependency Graphs (DDGs) were first introduced by Ferrante et al. [27] in 1987 as a program representation to capture the data-flow relationship among each instruction in the program. More formally, the LLVM documentation defines a data dependence graph as a structure that "*represents data dependencies between individual instructions. Each node in such a graph represents a single instruction and is referred to as an 'atomic' node [...]*" [6] while edges are defined as "*def-use dependencies between the atomic nodes*".

The introduction of DDGs paved the way to new program analysis techniques, such as program slicing [19], [76] (i.e., the set of statements that affect the value of a certain variable) and reaching definitions [8] (i.e., the set of definitions that could hit a certain point in the code). Over the years, researchers have proposed DDG-based techniques to build new approaches in compiler optimizations, as reported by the seminal work in this field by Kuck et al. [44]. For instance, Heffernan et al. [38] used the data dependencies that exist inside a program to improve instruction reordering and increase the CPU pipeline performances. Other works [39], [46] proposed advanced scheduling approaches based on the adoption and transformation of the DDG to measure the dependencies among instructions and evaluate when to perform a reordering operation. Another common application of DDGs is dead code elimination, which aims at identifying which assignments in the code can be removed after checking that no subsequent operations depend on them [14], [18], [43].

In software security, DDGs are often used to verify if potentially unsafe or poorly sanitized data (the *source*) can

propagate information inside the program until it reaches a certain statement that can trigger a vulnerability (the *sink*). This led to a set of applications of the DDG, especially in static software testing. For instance, in 1994 Kinloch et al. [41] suggested that the combination of the DDG with the Control Flow Graph (CFG) could help programmers at detecting bugs. More recently, Yamaguchi et al. [77] proposed a program representation, known as the Code Property Graph, that combines the CFG, the DDG, and the Abstract Syntax Tree (AST) of a program. The authors then designed specific queries over this data structure to detect vulnerable patterns in the code. The popularity of DDGs for vulnerability discovery is confirmed by the comparison performed by Zhioua et al. [81] among static software testing tools. The authors found that 3 out of the 4 investigated frameworks implemented a data dependency analysis component when scanning C source code. However, data dependencies are not just used to analyze source code but play a very important role also in other scenarios. For instance, Cheng et al [22] use them to perform taint analysis on IOT firmware images with the goal of finding vulnerable flows, and several model checking techniques rely on them to detect unsafe program points [50], [75].

Possible applications of the Data Dependency Graph are not limited to unsafe languages such as C/C++. In 2009, Hammer et al. [34] proposed an approach based on path conditions in dependency graphs that can be used to reveal security-sensitive flows inside Java code. Moreover, in 2015 Qian et al. [63] developed a static analysis framework to detect vulnerabilities in Android applications by traversing the DDG, similarly to what already done in [77] for C source code.

2.2. Coverage Guided Fuzzing

Depending on the instrumentation applied to the target binary, we can split fuzzing approaches into three major categories. *Blackbox* fuzzing tries to trigger crashes and error conditions by randomly mutating input without receiving any information from the target program. At the opposite end of the spectrum, *whitebox* fuzzing strongly relies on code analysis and instrumentation to generate inputs that can lead to vulnerable program points. For instance, approaches based on symbolic execution [17], [61] belong to this category.

Greybox approaches are somewhere in between the two aforementioned solutions. They try to achieve the same performances of *blackbox* fuzzing by adding lightweight instrumentation to the target application to collect some form of *feedback* during the execution. The *feedback* information can then drive the fuzzer to reach more code and trigger multiple interesting states in the target application.

Different implementations of *greybox* fuzzing adopt distinct ways to measure when a given input can trigger something new in the program. For instance, traditional approaches measure either the basic block coverage, like in [54], or the edge coverage, like in [47]. AFL [80], probably the most popular greybox fuzzer, extended edge coverage to capture information about how many times a program point is executed. In this case, the fuzzer keeps track of the visited edges inside a bitmap, where each entry of the bitmap represents how many times a certain edge has been hit, and AFL considers a new testcase as interesting if at least one of the entries has a new value that falls in a previously unseen bucket. To encode the edge as an index of the bitmap, the approach that AFL adopts consists of computing the `XOR` between the current and the previous location.

On top of this approach, many edge coverage variations have been proposed [15], [31], [40] that try to improve the basic strategy to augment the discovered program points. For instance, AFL++ [30], a modern fork of AFL, allows the analyst to adopt different coverage instrumentation strategies, such as Context Sensitivity [24] and N-grams (which takes track of a list of consecutive edges). In 2019, Wang et al. [74] have performed a study to compare many state-of-the-art coverage metrics to assess how they differ from the points of view of both code coverage and detected bugs. The authors found that none of the metrics outperforms all the others, and each coverage has pros and cons depending on the target application.

A different line of research has focused instead on finding new techniques that rely on alternative forms of feedback. For instance, Aschermann et al. [10] proposed a set of manual annotations to give the fuzzer a feedback also on the state of the program. Other promising methodologies [55], [57] use a different approximation of code coverage to reward the fuzzer. Manes et al. [49] instead implement what they define as *distance-based* fuzzing. Their goal is to build an informative fitness function that is used as the base for the feedback which is sent to the fuzzer. Finally, at the recent Usenix 2021 symposium, Fioraldi et al. [28] illustrate how violations of likely invariants can also be used as feedback to better fuzz the state of the program.

3. Methodology and Implementation

We now illustrate the methodology and the implementation challenges that we encountered while developing our solution. The technique presented in this paper is implemented as an LLVM [45] pass. While the code is compatible with different versions of the LLVM APIs, for our experiments we used LLVM 13. We chose LLVM for two main reasons: first, its intermediate representation is emitted in SSA [68] (Single Static Assignment) form, which means that each variable is assigned only once, and all variables must be defined before their first use. As we will see in the rest of the section, this simplifies the recovery of dependencies between LLVM IR variables with respect to other code representations where multiple definitions are allowed. Second, the LLVM toolchain is already well integrated into popular fuzzing projects, thus making our solution easy to plug into existing fuzzers' implementations, such as AFL++ [30]. This gives us the possibility to deploy our solution in an effective way, as well as to compare it with other instrumentation approaches that are already shipped with the AFL++ package. The following three subsections describe how our pass works by dividing the process into three main parts: DDG construction, dependency filtering, and target instrumentation.

3.1. DDG construction

The first decision we had make, when we started to develop our static analysis part, was the choice of the proper LLVM IR variables to construct the data dependency graph. The first intuitive approach was to recover the dependencies of *each* variable present in the LLVM bitcode by relying on the def-use edges to represent a dependency. However, we immediately noticed that such a technique does not fit well with our context. Indeed, because of the SSA form of the bitcode, this would produce too many dependencies to account for, which in turn would result in poor feedback for the fuzzer and in a large overhead in the execution of the target binary.

The LLVM framework applies some optimizations to construct its internal *Dependence Graph* [5], such as considering strongly connected components as a single node (the so-called *P-Node*). From now on, we will refer to this graph as $\texttt{DDG}_{\texttt{raw}}$, to indicate that we obtained it by the default LLVM implementation without applying any further transformations/optimizations. Although $\texttt{DDG}_{\texttt{raw}}$ is already an improvement with respect to the base strategy, it was still insufficient to overcome the performance issue. Nevertheless, before completely abandoning this road, we performed a benchmark where we evaluated this approach against the one that we ended up selecting. This was needed to ensure not to discard valid possibilities for our final prototype tool. Results of this preliminary experiment are reported along with the other evaluations in Section 4.3.

Our intuition then was to consider only a subset of LLVM variables, depending on how they are defined and used in the bitcode, and to recover the data dependencies that involve only this subset. This implies that we had to choose which instructions to consider as definitions and which one to retain as uses. The first observation was that, at the binary level, the actual data flow happens only when the memory is read or written. At the IR level, this led us to adopt the *Load* and *Store* instructions as a possible source and sink of our data flows. In addition to this, we added the *Call* instructions, and we considered them as uses of the variables, i.e., we track the dependency that reaches the function call parameters. Finally, we selected the *Alloca* instructions as a potential source of the def-use edge. Even though the compiler optimization passes would remove the majority of the *Alloca* defined variables, it can still be useful to track the dependencies that come from these variables when they are maintained in the emitted bitcode.

Algorithm 1 shows our solution. The two main data structures that we use are the DDG itself (a map of sets) and what we call the *Data Flow Tracker* (DFT), initialized at the beginning of the algorithm. The DFT is as a map of sets, where the key is a LLVM *Value* and for each key we get a set of LLVM *Values* the key depends on. Our approach iterates over all the instructions present in each basic block (line 7) and, when we meet a *defining instruction* (i.e., *Load* and *Alloca*), we add an entry in the DFT as shown in the first `if` block (lines 8-10). For general purpose instructions, i.e., those instructions that are neither a source nor a sink, we first extract the operands and subsequently the return value of the instruction (12-14). The primitive `InsertDataFlow` is then

```
1  function BuildDDG(module)
2      blocks ← GetBasicBlocks(module)
3      DDG_b ← {}, ∀ b ∈ blocks
4      DFT ← {}
5      for b ∈ blocks do
6          instructions ← GetInstructions(b)
7          for i ∈ instructions do
8              if IsDefinition(i) then
9                  val ← GetValue(i)
10                 DFT_val ← {}
11             if IsGeneric(i) then
12                 val ← GetValue(i)
13                 ops ← GetOperands(i)
14                 InsertDataFlow(DFT, val, ops)
15             for u ∈ GetUses(i) do
16                 def ← QueryDataFlow(DFT, u)
17                 src ← GetParentBlock(def)
18                 DDG_b ← AddToSet(src)
```

Algorithm 1: Data Dependency Graph builder

responsible to track the fact that the return value is actually depending on the operands variables (line 15). To achieve this, it stores the Value `val` in the corresponding DFT set, whose key has a dependency with the operands `ops` that are involved in the instruction. The inner `for` loop iterates over all the uses in the instruction (line 17). For each of them, we extract the defining instructions by means of the DFT (with the primitive `QueryDataFlow`) and we add a new edge to the DDG (lines 18-20). `QueryDataFlow` iterates over the keys of DFT, looking for a match between `u` (the use) and one of the elements in the corresponding set, thus performing a recovery of all definitions that `u` depends upon. The final output is a data dependency graph, where an edge connects respectively a definition of a variable D and a use of a variable that depends on D. From now on, we will refer to such a graph as $\texttt{DDG}_{\texttt{full}}$.

3.2. Filtering

Given the goal of producing a lightweight instrumentation that has a limited impact on the performance of the compiled binary, we introduce a set of optimizations and filters to reduce the number of locations to instrument. This filtering phase helps us to discard dependencies that would not add any additional feedback to the fuzzer, as the associated transition is already captured by edge coverage. First of all, since our reference granularity is the basic block, any dependency within the same block of code is not significant. Similarly, multiple dependencies that connect the same two basic blocks are merged into a single one.

It is important to remember that the purpose of a DDG coverage instrumentation is not to help the fuzzer to reach complicated nested regions of code, but rather to revisit a determined program point by examining the different dependencies of the variables involved in such a path. In other words, not by visiting more code, but by triggering additional paths in already-explored code. Our hypothesis is that by exploring these additional dependencies we can uncover new flaws which would normally go undetected. Because of that, we designed our instrumentation to coexist with the classic edge coverage mechanism. This allows our fuzzing engine to receive two different feedbacks, the former useful to test different dependencies and the latter to further explore the application code. However, this also means that we had to reason about potential intersections of DDG and CFG coverage, which would

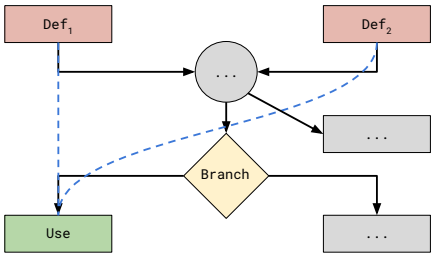

Figure 1. The configuration of the Definition and Uses that we want to isolate

lead to duplicate feedback. Thus, we now clarify which data dependencies we track and which ones we discard with our pass.

Essentially, we implemented two main rules to filter out redundant data dependencies. The first one is to check if a dependency is among two connected basic blocks, i.e., in which one is the successor of the other in the CFG. In this case, the dependency would not add any additional information that is not captured already by the edge coverage, and therefore would just increase the overhead without providing any useful feedback to the fuzzer. Therefore, in this condition we discard the data flow.

The second rule is an extension of the previous one and covers other scenarios where a data-dependency is already captured by edge coverage. In particular, we identified two additional types of data-flow. In the first one, the use U of a variable depends on a **single** definition D that can be located many basic blocks before U. In this scenario, we noticed that it is not worth maintaining the edge connecting the two basic blocks in our data structure because by definition if the program reaches the code block of U, it must have passed through the definition D already (since it was the only definition). Traditional edge coverage instrumentation already rewards the fuzzer when following this path, and therefore, for these situations, we do not track the data flow and discard the edge from our DDG.

Differently, we can have a configuration in which the use U of a variable depends on **more** than one definition, for instance the use of a ϕ-node variable [1]. As an example, we can consider Def_1 and Def_2, as represented in Figure 1. In the graph, the black arrows represent CFG edges while the blue arrows represent DDG edges. In this case, a fuzzer can reach 100% edge coverage while still triggering only one of the two def-use pairs. In fact, because of the mechanism used to log the edge coverage (i.e., the `XOR` of the current and previous BBs), the execution can reach the use always from the same edge, and therefore the fuzzer would not consider the two paths as two separate discoveries. As a result, this type of dependency is the only type of flow that we keep, as it is fundamental to reward the fuzzer in a different way compared to standard approaches (edge-coverage).

In the rest of the paper, we will refer to the DDG obtained after the filtering phase as $\texttt{DDG}_{\texttt{filtered}}$. To summarize, $\texttt{DDG}_{\texttt{filtered}}$ is a graph that contains only flows represented with a def-use relationship and that have at least two definitions for the same used variable.

1. A ϕ-node merges many versions of a variable into a new one in relationship with the incoming control flow. https://gcc.gnu.org/onlinedocs/gccint/SSA.html

```
1  // X, Y, Z are random IDs set at compile time
2  void function() {
3      u16 on_block_X = 0;
4      u16 on_block_Y = 0;
5
6      int val;
7
8      // this if-statement leads to two distinct definitions of the
          variable 'val' that is then used in the last if-statement,
          producing a dependency that we want to instrument
9      if (...) {
10         // block X
11         on_block_X = X;
12         int a = load_a();
13         val = a;
14     } else {
15         // block Y
16         on_block_Y = Y;
17         int b = load_b();
18         val = b;
19     }
20
21     // without these branches here the DDG edges would be included in
          the CFG and so pruned by our filtering pass
22     if (...) {
23         ...
24     } else {
25         ...
26     }
27
28     if (val == 0) {
29         // block Z
30         u16 idx = on_block_X ^
31                   on_block_Y ^
32                   Z;
33         __afl_area_ptr[idx]++;
34         use(val);
35     } else {
36         ...
37     }
38 }
```

Listing 1. A simple example of how our instrumentation looks like.

3.3. Instrumentation

As we explained in Section 2.2, typically AFL-based fuzzers log an edge visit by computing the `XOR` between the `ID`s of the current and previous locations and use this value as an index to access a bitmap that stores the number of times a certain edge has been hit. For our purposes, this is still necessary, because the DDG does not add any information to make the fuzzer explore deeper code but only to improve *how* to fuzz some specific code locations. Therefore, as a first instrumentation layer, we keep the traditional approach that is used in off-the-shelf fuzzers to log new edges inside a bitmap.

However, in the context of the DDG this solution does not work, since the two locations that we want to use as input for the `XOR` are not consecutive and the execution could go through many intermediate BBs before reaching the one that contains the data-dependent use. To solve this issue, we add an additional marker variable, originally set to 0, for each basic block that contains a definition we want to keep track of. Our instrumentation then changes the marker value to the ID of the block when it reaches the block itself. Finally, we instrument each basic block containing a use by generating a new bitmap index obtained by `XOR`ing the corresponding marker variables with the ID of the destination block. Because $N \oplus 0 = N$, the result will account only for the definition that was actually executed, thus resulting in different values when the same basic block is reached by following different data-flow paths.

For instance, we can consider the code in Listing 1, in which we highlighted the instrumentation inserted at

the IR level. At the beginning of the function, we insert a marker variable for each definition block that we want to track, `on_block_X` and `on_block_Y` in our example. In the basic blocks containing the definitions, we set the corresponding marker to the ID of the block (`X` and `Y` respectively). Then, the basic block that contains the use is instrumented to compute the hash of the markers linked to `val` with the ID of the current block (`Z`). This hash is used as an index in the AFL bitmap.

4. Evaluation

To assess the validity of our approach, we performed a number of experiments to measure and compare the effects of our instrumentation in terms of bug detection, performance overhead, achieved code coverage, and the possibility to cause queue explosion. We believe that all these points are equally important to evaluate when proposing a new fuzzing approach, as they play a role in the overall effectiveness of a solution. Based on the results of our experiments, we try to understand when the adoption of our approach is useful for a fuzzing campaign and when, instead, it does not add any clear advantage with respect to the traditional edge coverage solution.

For our evaluation, we used 3 datasets. The first includes old versions of 10 real-world open-source projects known to contain bugs. For the first five applications, we selected software that we expected to contain a large number of data dependencies. For instance, we decided to include the `pcre2` library because of its frequent use of lookup tables, a C compiler frontend (`c2m`) and backend (`qbe`), a popular parser generator (`bison`), and `faust`, a compiler for a functional programming language used mostly for real-time signal processing. For the remaining five applications, we selected instead parser-related projects in which we expect data dependencies to be less relevant. Three of them (namely `readelf`, `objdump` and `libmagic`) operate on the ELF file format while `tiff2pdf` performs image parsing and `openssl` is used to perform cryptographic operations.

The goal of this initial distinction, based purely on our expectation of the amount and impact of data dependencies, is to select some applications where our approach can provide an advantage and others where probably it is less useful. This can help us to better assess in which scenario an analyst should deploy a fuzzing campaign with our approach and when instead the traditional edge-coverage can provide better results.

Table 1 reports the list of applications in our first dataset, along with the hash commit that we tested and the command line that we used for the fuzzer. The fourth column reports the lines of code and clearly shows how our dataset contains software of different sizes, ranging from 10 KLOCs (`qbe`) to 234 KLOCs (`openssl`). In three cases, namely `pcre2`, `file` and `openssl`, we had to build an harness that invokes the core functionalities of the libraries. Furthermore, for each fuzzing campaign, we enabled the ASAN sanitizer, and, when it was possible, we also added the UBSAN sanitizer (some of the applications failed at building when UBSAN was enabled). We did not modify the compiler optimization level adopted by AFL++ (by default set to `O3`), to make the code execution more performant.

In addition to this first set of experiments, we also wanted to extend our evaluation to other datasets, to confirm our findings and prove that our approach can work for different codebases. However, many popular benchmark datasets which are used in other studies for fuzzing experiments are not suitable for our purpose. This is due to the fact that such benchmarks were designed to evaluate the amount of code covered by different fuzzers, often by inserting artificial bugs. Although code reachability is a fundamental aspect of a fuzzing approach, our solution does not add any improvement in terms of code coverage (at least not directly). Rather, it tries to use data dependencies to augment the amount of discovered bugs – which does not necessarily imply exploring new program points. As a consequence, we believe that datasets such as LAVA-M [25] are not ideal candidates for a proper evaluation. On the other hand, both MAGMA [37] and Fuzzbench [51] satisfy our requirements as they focus on a number of bugs and on a large variety of applications. Among the two, we selected Fuzzbench [51] because it contains a larger number of programs. In total, this second dataset includes 22 different programs (we had to disable three applications because they did not compile under clang-based instrumentations).

Finally, we added a third dataset to our evaluation, this time taken from a recent paper by Blazytko et al. [13]. This last benchmark is composed of targets not selected by us, thus without any a-priori knowledge about the structure of the programs, but that may still contain targets with relevant data-dependencies as the focus of the original paper was to deal with highly-structured inputs and complex parsing code. This final dataset included 6 applications.

4.1. Experiment Setup

We performed the experiments with the first and the third dataset on a `x86_64` server containing an Intel Xeon Platinum 8260 CPU with a clock frequency of 2.40GHz. As already explained, we adopted AFL++ (version 3.14) as a reference fuzzer, both for our implementation and evaluation. One of the advantages that come with using this project is that it already implements many feedback approaches, such as Edge Coverage, Context Sensitivity, and NGram coverage (from now on respectively `Edge`, `Ctx` and `Ngram`). Thus, it provides the ideal comparison for our evaluation purposes. In the follow-up of the paper, we will use interchangeably `Edge` and AFL++ to indicate the use of standard edge coverage mechanism whereas we will specify `Ctx` and `Ngram` to identify the adoption of alternative feedback techniques.

Each experiment was made of 5 trials of 24 hours to limit the randomness of the fuzzer. The initial seeds were mostly taken from the `test` directories that were already present in the repositories of the projects to ensure that they represented valid input files for that application. The initial queue size is reported in Table 1 for each target program in our custom evaluation set.

For the experiments on the Fuzzbench targets, we adopted the default configurations described on the public website page [7]. In this case, each test consists of 20 trials of 23 hours for each fuzzer.

Moreover, as the authors of [42] suggest, we computed the p-value resulting from the Mann-Whitney U test, a

TABLE 1. DATASET OF THE APPLICATIONS USED FOR OUR STUDY, ALONG WITH THEIR VERSION, LINES OF CODE, COMMAND LINE AND COMPILATION INFORMATION

Application	Package	Hash commit	KLOC	Command line	Sanitizers	Initial Queue
`bison`	bison	0ac1584	100	@@	ASAN	3
`pcre2`(*)	pcre2	65457aa	68		ASAN, UBSAN	3
`c2m`	mir	7670d7e	200	@@	ASAN, UBSAN	65
`qbe`	qbe	e0b94a3	10	@@	ASAN	52
`faust`	faust	f9aac26	115	@@	ASAN, UBSAN	27
`readelf`	binutils	68b975a	120	-s @@	ASAN	3
`objdump`	binutils	68b975a	120	-d @@	ASAN	3
`libmagic`(*)	file	d1ff3af	14		ASAN, UBSAN	3
`tiff2pdf`	libtiff	7d3b9da	63	@@	ASAN, UBSAN	10
`openssl`(*)	openssl	0d87763	234		ASAN	9

(*) indicates that an harness was used

TABLE 2. MEDIAN NUMBER OF UNIQUE BUGS DETECTED IN 5 TRIALS OVER 24H OF OUR EVALUATION AGAINST EDGE-COVERAGE BASED FUZZING ALONG WITH THE INTERSECTION OF BUGS DETECTED IN THE MEDIAN CASE. RESULTS INCLUDE ALSO THE DD RATIO FOR EACH OF THE TARGETS, THE SLOWDOWN INTRODUCED DUE TO OUR DDG INSTRUMENTATION AND THE P-VALUE OBTAINED WITH MANN-WHITNEY

Target	DD ratio	DDFuzz	Edge	DDFuzz ∩ Edge	Slowdown	P-value
`bison`	15%	5	3	3	2%	0.05
`pcre2`	14%	30	28	13	6%	0.46
`c2m`	23%	26	26	23	21%	0.55
`qbe`	15%	5	3	2	2%	0.04
`faust`	20%	4	2	2	18%	0.03
Total	-	70	62	43	-	-
Geomean	17.06%	9.51	6.66	5.14	6%	-
`readelf`	7%	4	4	4	2%	0.45
`objdump`	7%	5	5	4	1%	0.56
`file`	5%	1	1	1	1%	0.38
`tiff2pdf`	8%	9	13	9	4%	0.002
`openssl`	5%	2	2	2	5%	0.45
Total	-	21	25	20	-	-
Geomean	6.28%	3.49	3.24	3.10	2%	-

nonparametric test used to compare differences between two independent randomly selected sample values that originate from two distinct populations. For our purposes, the test checks that our prototype fuzzer is statistically different from the competitor one, i.e., the default AFL++, in terms of bugs detected during the trials.

4.2. Comparison against edge coverage

For our initial evaluation, we want to compare our data dependency coverage instrumentation against the traditional edge coverage approach. The first question that we want to answer is whether our approach is helpful in terms of discovered bugs. We also report a simple metric to capture the impact of our pass by counting the amount of data dependencies in the target application. This can tell the analyst if it makes sense to enable our instrumentation for a new fuzzing campaign. This metric, which we call *DD ratio*, is the ratio between the basic blocks that we instrumented with the dependencies information (as described in Section 3) over the total number of basic blocks in the program.

We computed such a value for the ten applications in our dataset. Results are reported in the first column of Table 2. We can observe that the value of *DD ratio* changes quite sharply from application to application, with a minimum of 5% for `openssl` to a maximum of 23% for `c2m`. The other interesting and more important aspect is that the two sets of applications are clearly separated by their `DD ratio` value, confirming our intuition that the first five were more data-flow intensive. For instance, if we adopt a threshold of 10%, that derives from the geometrical mean of all ratios for each target, we can easily classify our codebases into *strongly* (above the threshold) and *weakly* (below the threshold) data-dependent applications.

We then ran our fuzzer evaluation against the traditional edge coverage instrumentation. Results are reported in the remaining columns of Table 2, where `DDFuzz` and `Edge` indicate respectively the median of the unique bugs found by the two approaches while the column with label `DDFuzz` ∩ `Edge` represents the findings that are common among the two (i.e., the intersection). Such numbers are the result of a careful triage phase that was conducted both in an automated fashion along with a rigorous manual check. As a first step, we de-duplicated bugs by grouping them according to their call-stack hash collected from the stack trace that we obtained by the sanitizers. However, this simple approach is generally error-prone (as shown in [42]) and thus we manually inspected all remaining test cases to avoid any possible duplicates.

The first way to interpret the results is by comparing the number of discovered bugs by the two approaches. Overall, `DDFuzz` found 8 vulnerable flaws more than `Edge` in the 5 strongly data-dependent applications. With

the exception of c2m, where the median is the same, for the other four cases, the fuzzer equipped with our instrumentation discovered two additional bugs each. Viceversa, for the remaining five targets that do not exhibit a high dependency in their code base, the number of discovered bugs remained mostly the same. The only exception is tiff2pdf, where DDFuzz found 4 bugs less than Edge – probably due to the fact that triggering the bugs was mostly a code reachability problem.

While these results are promising for strongly data-dependent programs, the intersections between the discovered bugs can tell us even more. In fact, not only our approach allows to detect more bugs on average, but it can find some that are never discovered by Edge in any of the trials. For instance, for c2m, even if the number of unique bugs is the same, there are three vulnerable points that are only detected by our approach. This is even more evident if we consider the intersections for pcre2, where DDFuzz finds a total of 17 unique bugs which are not revealed by the vanilla AFL++. For the remaining cases instead (bison, qbe and faust) the unique bugs identified by DDFuzz are mostly a super set of the ones individuated by Edge. This trend holds until a sufficiently high data dependency exists in the analyzed code. Indeed, if we consider the last five rows, where the DD ratio is always under 8%, the intersections converge towards the total number of bugs – i.e., the additional instrumentation of DDFuzz had a negligible impact on the results.

We also measured the overhead introduced by our additional instrumentation, compared again with that introduced by Edge alone. The results, reported in Table 2, show that the overhead introduced by our approach is overall modest. The slowdown factor was computed as the difference between 1 and the ratio among the DDFuzz average executions per second over the Edge executions per second (and then transformed in percentage). For the five strongly data-dependent applications, the average slowdown is 10% (6% if we consider the geometrical mean), despite the fact that we observed a non-negligible variation of this value. Such a variation is only in part justified by the DD ratio, because other factors influence it, such as the execution path, the control flow executed and the fuzzer execution mode (i.e., if we used an harness or we make use of the fork system call to spawn the target process). For the remaining five applications instead, the executions per second are almost the same between the two fuzzing approaches, demonstrating again that, in the case of a weakly data-dependent code structure, our methodology converges towards the default AFL++. The last column of Table 2 concludes our measurements about this experiment with the p-value resulting from the Mann-Whitney U test computed over the bugs found for each trial. The p-value is significant ($<= 0.05$) in 3 out of the 5 cases for the strongly-dependent programs while only one weakly-dependent application reports a p-value statistically significant. This again suggests that for high values of DD ratio, DDFuzz behaves differently from Edge.

4.3. Effects of our instrumentation filters

As presented in Section 3, before performing the actual instrumentation of the DDG, we obtain three different versions of the data dependency graph. The first is $\texttt{DDG}_{\texttt{raw}}$ which is the result of the default LLVM dependence graph implementation. The second instead is $\texttt{DDG}_{\texttt{full}}$ which is the output of Algorithm 1 and serves as base graph to produce our final version, $\texttt{DDG}_{\texttt{filtered}}$. As already explained, we ended up instrumenting this last version of the DDG in our final implementation of DDFuzz, which is used across the whole paper. However, in the current section, we want to measure the effects of our optimizations compared to the other two flavors of the DDG ($\texttt{DDG}_{\texttt{raw}}$ and $\texttt{DDG}_{\texttt{full}}$).

TABLE 3. COMPARISON OF THE EFFECTS OF OUR FILTERING STRATEGIES IN TERMS OF MEDIAN NUMBER OF BUGS OVER 5 TRIALS OF 24H EACH

Target	DDFuzz	$\texttt{DDFuzz}_{\texttt{full}}$	Intersect.	Decrease
bison	5	4	4	67%
pcre2	30	29	24	41%
c2m	26	16	13	44%
qbe	5	4	4	49%
faust	5	1	1	52%
Total	71	54	46	-
Geomean	9.95	5.94	5.49	49.85%

As a first step, we compare the performances that result from the instrumentation of $\texttt{DDG}_{\texttt{raw}}$ (the LLVM default implementation) against the final version of the DDG ($\texttt{DDG}_{\texttt{filtered}}$). For this, we selected an applications in our dataset (qbe) for which our approach had an average performance and instrumented it by using either the default LLVM data dependency graph (i.e., $\texttt{DDG}_{\texttt{raw}}$) or our optimized version. We then launched two fuzzing campaigns according to the experiment setup described in the previous subsection.

We immediately noticed that the instrumentation based on $\texttt{DDG}_{\texttt{raw}}$ was introducing a major overhead since we had to increase the timeout of AFL++ due to the fact that the instrumented program was not terminating within the default time interval. After 24 hours, the average executions per second was 50% higher with our optimization (362 vs. 240), and this resulted in a median of five discovered bugs for DDFuzz versus two (always a subset of the five) when using $\texttt{DDG}_{\texttt{raw}}$.

For the second measurement, we wanted to quantify the effects of our pruning strategies. As described in Section 3.2, we filtered $\texttt{DDG}_{\texttt{full}}$ to exclude the edges which are already covered with Edge, producing as a final output what we call $\texttt{DDG}_{\texttt{filtered}}$. To assess to which extent these filtering strategies impacted the performances of our final implementation, we performed another experiment with the same setup described in Section 4.1, comparing the filtered with the unfiltered graphs. In this case we selected only the five applications that exhibit a sufficient amount of data dependency as we are interested at measuring the effects of our filter strategies, and it would not be meaningful to consider applications where our approach instruments only a small amount of locations. Results of this evaluation are listed in Table 3 where we refer to DDFuzz as our prototype implementation after the application of the filters and to $\texttt{DDFuzz}_{\texttt{full}}$ as the implementation obtained from our pass without the filtering steps (i.e., the instrumentation of $\texttt{DDG}_{\texttt{full}}$). Moreover, we computed the intersections of bugs between $\texttt{DDFuzz}_{\texttt{full}}$

TABLE 4. A COMPARISON OF DATA DEPENDENCY COVERAGE AGAINST NGRAM2, NGRAM4 AND CTX INSTRUMENTATION APPROACHES IN TERMS OF MEDIAN NUMBER OF BUGS OVER 5 TRIALS OF 24H EACH

Target	DDFuzz	Ngram2	Ngram4	Ctx	DDFuzz ∩ Ngram2	DDFuzz ∩ Ngram4	DDFuzz ∩ Ctx
bison	5	5	4	3	3	3	3
pcre2	30	29	23	33	15	14	17
c2m	26	27	27	17	23	22	12
qbe	5	3	4	5	3	4	4
faust	5	2	2	1	2	2	1
Total	71	66	60	59	46	45	37
Geomean	9.94	7.48	7.23	6.09	5.73	5.93	4.76

and DDFuzz (third column) and the percentage decrease of the instrumented locations (fourth column).

We can immediately notice that bugs discovered by DDFuzz are mostly a superset of those which are detected with $\text{DDG}_{\texttt{full}}$. This is because the feedback produced by the more dense instrumentation is less informative in terms of the dependencies that were reached. However, the approach can still find some interesting program points as demonstrated by the intersections with DDFuzz. In facts, for pcre2 and c2m, 8 distinct bugs are detected by the more naive approach and missed by DDFuzz in the median case.

Overall, the instrumented locations decreased by 52% in average (49% geometrical mean) with respect to the version without the filters enabled. This resulted in better performances and indeed we observed an average slowdown of roughly 20% compared to Edge (while, as already shown, DDFuzz slowdown was 10%). Therefore, we can conclude that our filtering strategies represent an improvement of the standard instrumentation of $\text{DDG}_{\texttt{full}}$ and justify our design choices.

4.4. Comparison against different instrumentation strategies

For the second evaluation of our approach on the custom dataset, we want to compare DDFuzz against different instrumentation approaches. As we have shown in Section 3, one of the major points of our instrumentation technique is that the edges in the DDG can connect basic blocks which are not necessarily consecutive, i.e., when other intermediate basic blocks exist between the source and the sink. Thus, our first hypothesis was to examine instrumentation passes such as the ones that rely on Ngram coverage (Ngram). The simple idea behind these is that when computing the XOR of the edges to find the bitmap index where to log the new discovered program point, Ngram approaches take into account multiple edges. For instance, by adopting Ngram2, the index of the bitmap involves the XOR of the previous 2 locations, while Edge would consider only the previous location. In other words, Edge is equivalent to a Ngram1 instrumentation. For our tests, we chose to compile our target binaries with Ngram2 and Ngram4 because, according to Wang et al. [74], these are the two approaches that provide the best results in terms of number of discovered bugs. We also compared against Context Sensitivity [74] (Ctx), a recent approach that takes into account the callstack in addition to the reached basic blocks. According to its authors, together with the two aforementioned Ngram instrumentations context sensitivity was the solution that provided better results.

The results of our second evaluation are showed in Table 4, limited again to only the five strongly data-dependent binaries. As in the previous experiment, we report the median of unique bugs, de-duplicated as explained in 4.2. The first four columns contain the median detected bugs depending on each of the four instrumentations that we wanted to include, while the remaining three columns instead show the intersections with DDFuzz. There is no solution that outperforms all the others for all programs, but DDFuzz results are the best for three out of five applications and the best overall – with 71 discovered bugs vs. 66 of NGram2 (the second-best performer).

Again, by looking at the intersection, we can see that the bugs detected thanks to the use of our data dependency instrumentation are very different for each of the tested applications. In each one, DDFuzz finds at least one different bug that the other approaches could not find. It total, it was able to detect 25 bugs that were never triggered by Ngram2, 26 more than Ngram4, and 34 not captured by the Ctx instrumentation. This shows once more that our extension could provide a very clear benefit for applications that have a rich set of data dependencies and that even when DDFuzz is not the approach that finds more bugs overall, it always leads to detect some different ones. Moreover, it is important to underline that we believe that all these approaches can (and should) be combined during a fuzzing campaign.

4.5. Queue Explosion

If the number of vulnerable points detected is an important metric for a fuzzer effectiveness, the number of inputs generated influences its usability and could result in poor feedback for the fuzzer.

The authors of [74] found that an increase factor up to ∼8x is still manageable by the fuzzer and would allow to provide relevant feedback without incurring in a queue explosion problem. These results were derived from experiments with different coverage instrumentations, such as Ngram and Ctx. On the other hand, the authors observed growth factors of 21x and 14x when using two types of memory feedbacks and concluded that such values were potentially leading to the explosion in the queue size.

For this, we compute the average size of the queue for our five strongly data-dependent applications, both in the baseline case with only edge coverage instrumentation and with our data dependency instrumentation. Table 6 shows the results along with the ratio obtained by dividing

TABLE 5. A COMPARISON OF THE MEDIAN VALUES OF LINE AND FUNCTION COVERAGE AMONG THE PROGRAMS ACCORDING TO EACH DIFFERENT INSTRUMENTATION OVER 5 TRIALS OF 24H EACH

Target	DDFuzz		Edge		Ngram2		Ngram4		Ctx	
	Lines	Functions	Lines	Functions	Lines	Functions	Lines	Functions	Lines	Functions
bison	46.1%	49.7%	43.9%	47.6%	47.5%	50.3%	47.5%	50.3%	47.5%	50.3%
pcre2	53.2%	32.2%	53.8%	32.4%	54.1%	32.4%	54.2%	32.4%	54.2%	32.4%
c2m	48.8%	55.2%	48.8%	55.2%	48.9%	55.2%	48.9%	55.2%	49.2%	55.8%
qbe	76.9%	85.2%	76.6%	85.2%	77.0%	85.0%	77.1%	85.0%	77.0%	85.0%
faust	26.3%	28.5%	26.7%	28.4%	26.7%	28.5%	26.8%	28.5%	26.8%	28.5%

TABLE 6. COMPARISON AMONG THE AVERAGE QUEUE SIZES OVER 5 TRIALS OF 24H EACH WHEN DATA DEPENDENCY COVERAGE AND EDGE COVERAGE ARE APPLIED, ALONG WITH THEIR RATIO

Target	DDFuzz	Edge	Ratio
bison	5,173	2,986	x1.7
pcre2	119,180	16,282	x7.3
c2m	14,323	13,269	x1.1
qbe	2,748	1,814	x1.5
faust	2,934	2,633	x1.1

the DDFuzz queue size by the Edge size. The numbers show that the overall increase is quite moderate. With the exception of pcre2, where the increase accounts for a factor of 7.3x, all other factors are below 2x, thus showing that our technique results in additional feedback but not large enough to cause problems in the input queue.

4.6. Code Coverage

As a last experiment on our custom benchmark, we decided to compare the values of the code coverage that we obtained for our strongly data-dependent applications. For this test, we used afl-cov [4] which represents the default solution to measure this metric and is compatible with AFL++ based fuzzers. The tool was launched against the different instrumentation types that we tested and produced two values for each one: line coverage and function coverage. Results are reported in Table 5 and show the median values that we observed in our experiments.

If we look at the first two columns that correspond to DDFuzz and Edge, we observe that there is not one of the two which dominates the other. In two cases (bison and qbe) DDFuzz performs better whereas for c2m we registered the same line coverage. For the remaining two projects instead, Edge reaches a major number of program points.

Other forms of instrumentation led instead to better code coverage. However, in the worst case (bison), the line coverage difference with the best approaches is 1.4% while for the other projects is always less than 1%. This result is in line with our expectations since our approach is not designed to increase coverage, but only to increase paths on already-covered areas of code.

4.7. FuzzBench

We now look at the results we obtained from the experiments we performed on Fuzzbench [51]. The main goal was to extend our approach to a broader range of applications and bugs to verify whether our findings could be generalized beyond our test programs.

In these experiments we limited our comparison to DDFuzz and Edge, because the reports generated by Fuzzbench do not include the information about the intersections of discovered found but only an aggregated value representing the sum of the unique bugs for all fuzzers. Therefore, if we had included other instrumentation approaches (as, for instance, Ctx and Ngram) we would have not been able to distinguish among the bugs found by each technique. The experiment consisted of 20 trials of 23 hours over a total of 22 real-world projects. We report the results in Table 7, where, for each target, we list the bugs revealed by the two instrumentation approaches, the sum of the two, the intersection as well as the DD ratio, that we introduced in Section 4.2. To compute the DD ratio, we built the 22 projects on our local machine, according to the docker specifications reported by Fuzzbench to mimic the same environment setup. This allowed us to run our instrumentation pass and log the instrumented locations and the total number of basic blocks necessary to compute the ratio. The targets in Table 7 are sorted by the DD ratio with the weakly data-dependent applications in the first half and the strongly data-dependent projects in the second.

If we look at the total unique bugs detected by the standard AFL++, we notice that they are more than the amount of vulnerabilities triggered by DDFuzz (respectively 187 and 183). On the other hand, the geometrical mean, which indicates a central tendency by flattening the outlier values, tells us that DDFuzz finds in the mean case 8.20 bugs while Edge stops at 7.98 vulnerabilities.

By looking at individual applications, we can notice that Edge performs better in 4 cases while DD-Cov in 5. For the remaining 13 benchmarks, in 10 cases none of the fuzzers reported any interesting finding (both fuzzers properly ran but did not trigger any vulnerable location) whereas for 3 cases the bugs discovered are the same (libgit2_objects_fuzzer, stb_stbi_read_fuzzer, php_-php-fuzz-parser-2020-07-25), despite the fact that for php_php-fuzz-parser-2020-07-25 the two fuzzers find 2 different bugs. Moreover, it is interesting that in two cases where Edge finds more vulnerabilities, our data dependency instrumentation can still trigger different buggy locations that were not detected by the edge coverage. More specifically, for arrow_parquet-arrow_fuzz the intersection of bugs is 74 which means that DDFuzz finds 12 different bugs, and the same happens for one bug discovered in poppler_pdf_fuzzer. Overall the outcomes of this experiment confirms that

TABLE 7. TOTAL UNIQUE BUGS FOUND ACCROSS ALL 20 TRIALS OF 23H BY DDFUZZ AND EDGE ON FUZZBENCH

Benchmark	Total bugs	Edge	DDFuzz	DDFuzz ∩ Edge	DD ratio
arrow_parquet-arrow-fuzz	105	93	86	74	1%
proj4_standard_fuzzer	0	0	0	0	2%
muparser_set_eval_fuzzer	0	0	0	0	3%
openh264_decoder_fuzzer	10	10	8	8	3%
aspell_aspell_fuzzer	0	0	0	0	4%
systemd_fuzz-varlink	0	0	0	0	4%
tpm2_tpm2_execute_command_fuzzer	0	0	0	0	4%
file_magic_fuzzer	0	0	0	0	7%
libgit2_objects_fuzzer	2	2	2	2	5%
grok_grk_decompress_fuzzer	4	4	2	2	7%
stb_stbi_read_fuzzer	11	11	11	11	8%
njs_njs_process_script_fuzzer	0	0	0	0	11%
php_php-fuzz-execute	21	13	16	8	11%
libxml2_libxml2_xml_reader_for_file_fuzzer	13	11	12	10	12%
libhtp_fuzz_htp	7	6	7	6	13%
matio_matio_fuzzer	22	20	21	19	14%
php_php-fuzz-parser-2020-07-25	13	12	12	10	14%
poppler_pdf_fuzzer	5	4	3	2	14%
libarchive_libarchive_fuzzer	0	0	0	0	15%
zstd_stream_decompress	0	0	0	0	15%
usrsctp_fuzzer_connect	0	0	0	0	17%
libhevc_hevc_dec_fuzzer	3	1	3	1	19%
Total	214	187	183	150	-
Geomean	9.72	7.98	8.20	6.38	7%

TABLE 8. MEDIAN RELATIVE CODE-COVERAGE ACCROSS ALL 20 TRIALS OF 23H BY DDFUZZ AND EDGE ON FUZZBENCH

Benchmark	Edge	DDFuzz
arrow_parquet-arrow-fuzz	96.43	95.02
proj4_standard_fuzzer	100.00	100.00
muparser_set_eval_fuzzer	97.91	97.91
openh264_decoder_fuzzer	99.38	98.91
aspell_aspell_fuzzer	99.91	99.86
systemd_fuzz-varlink	100.00	100.00
tpm2_tpm2_execute_command_fuzzer	96.16	88.62
file_magic_fuzzer	81.65	78.34
libgit2_objects_fuzzer	99.75	99.63
grok_grk_decompress_fuzzer	94.58	91.41
stb_stbi_read_fuzzer	94.81	93.01
njs_njs_process_script_fuzzer	96.22	93.66
php_php-fuzz-execute	96.15	92.77
libxml2_libxml2_xml_reader_for_file_fuzzer	94.73	92.85
libhtp_fuzz_htp	99.94	99.85
matio_matio_fuzzer	99.33	99.29
php_php-fuzz-parser-2020-07-25	99.12	98.08
poppler_pdf_fuzzer	97.95	97.96
libarchive_libarchive_fuzzer	96.51	81.32
zstd_stream_decompress	98.12	97.84
usrsctp_fuzzer_connect	99.58	99.65
libhevc_hevc_dec_fuzzer	96.14	93.28

the feedback produced by our instrumentation is different from the standard edge coverage and results in different program points reached during the fuzzing session. This inherently does not always imply more bugs but can result in some different ones.

For the 5 projects where DDFuzz works better, the DD ratio indicates a high value of data dependency (above the 10% threshold we defined in Section 4.2) whereas for 3 out of the 4 projects where Edge wins, the DD ratio suggests that the application has a lower amount of data dependencies. The only exception is poppler_pdf_fuzzer, where Edge detected 4 bugs vs 3 of DDFuzz, despite a data-dependency ratio of 14%.

Overall, this confirms once more that the DD ratio can be used as a criteria to predict the type of instrumentation that would provide better results. To conclude our overview of the bugs, we extracted the statistical significance data included in the Fuzzbench report. Interestingly, both for *DDFuzz* and Edge the bug coverage is statistically significant (i.e., p-value $\leq \sim 0.05$) in 7 cases out of the 12 projects where they can detect at least one vulnerability.

As a parallel consideration, we computed the coverage that our prototype fuzzer reached in the Fuzzbench targets. Table 8 shows for AFL++ and DDFuzz alike, the median relative code coverage that we obtained across the 20 trials for each target. According to the Fuzzbench documentation, the relative code coverage for a trial is computed as the ratio between the single-trial coverage over the maximum coverage obtained during the experiment. It is quite evident that for the majority of the programs (17 out of 22), Edge results in a better code coverage while DDFuzz can do better only in 2 cases. However, it is interesting to observe that for all targets where DDFuzz finds more bugs, it also reaches a lower code coverage. This confirms once more our intuition that fuzzing the data dependency edges does not help to discover new program points but suggests how to ”stress“ the already discovered code locations in a different way.

TABLE 9. OUTCOME OF THE THIRD EVALUATION OVER A DATASET OF 6 PROGRAMS. THE RESULTS INCLUDE THE DD RATIO, THE MEDIAN NUMBER OF BUGS FOUND WITH DDFUZZ AND EDGE COVERAGE, THEIR MEDIAN INTERSECTION, THE SLOWDOWN INTRODUCED WITH OUT INSTRUMENTATION AND THE P-VALUE THAT WE OBTAINED WITH THE MANN-WHITNEY TEST.

Target	DD ratio	DDFuzz	Edge	DDFuzz ∩ Edge	Slowdown	P-value
nasm	3%	4	4	4	1%	0.12
sqlite	7%	1	2	1	15%	0.001
lua	11%	9	2	2	20%	0.005
boolector	14%	3	1	1	6%	0.13
mruby	17%	3	3	3	35%	0.45
tcc	12%	11	8	8	7%	0.05
Total	-	31	20	19	-	-
Geomean	9.3%	3.9	2.6	2.4	8.7%	-

4.8. Third Dataset

As a final proof of our approach efficacy, we opted to select another dataset, made of real-world programs, and adopt it to compare against Edge. We reviewed recent state-of-the-art papers looking for other fuzzing solutions dedicated to specific classes of target applications (since *DDFuzz* works best with highly data-dependent binaries). At the end we settled for the dataset of Blazytko et al. [13], who evaluated their structure-aware fuzzer on a set of 8 programs, that we report in Table 9 (note that we excluded PHP and libxml2 as they are already included in the experiment we ran on the Fuzzbench dataset, and a second evaluation of these targets would be unfair, as they both resulted in more bugs for *DDFuzz*, see Tab. 7). For the experiment setup, we used the same configuration described in Section 4.1.

DDFuzz outperformed Edge in 3 targets out of the 4 with a DD ratio above the threshold (10%), with 11 more bugs overall. On the two targets with a low DD ratio, our technique performs like the baseline on nasm while on sqlite it underperforms Edge missing one bug. The Mann-Whitney U test results in a significative p-value in 3 of the 6 considered cases. With the exception of boolector, where the test produces a p-value major than 0.05, the two remaining applications (nasm and mruby) that result in a high p-value justify this due to the similar number of bugs during the 5 trials. This evaluation further confirms that the DD ratio is a good predictor of the efficacy of DDFuzz and thus the insight that our technique should be applied to a specific class of data-dependent programs.

4.9. Classes of Bugs

Another interesting measurement that we carried out on top of our experiments is to study the classes of bugs that DDFuzz revealed during the several trials. For each of the detected bugs in the median case, we analysed the reports generated with ASAN for the two datasets that we tested on our servers. Table 10 reports the results in the column **Total** while the last column shows the bugs that only DDFuzz could find.

The table shows that DDFuzz can detect several types of bugs, with a prevalence for Heap Buffer Overflows (38) and Undefined Behaviors that generate the signal ILL (36). However, many of these bugs are the same that also the vanilla version of AFL++ can spot during our tests.

TABLE 10. CLASSES OF BUGS DETECTED WITH DDFUZZ IN THE MEDIAN CASE

Bug Class	Total	DDFuzz Only
Heap BOF	38	11
Global BOF	5	3
Stack BOF	1	0
Heap UAF	4	2
Stack Overflow	19	11
Invalid Ptr Deref	11	3
Invalid Allocation Size	1	0
ABRT	2	1
ILL	36	5

Thus, we isolated the ones that are different among the two fuzzers, and found that the majority accounts for Stack Overflows (11 instances) and Heap Buffer Overflows (11) while all the other cases are almost equally distributed between the other classes of bugs.

5. A Bug Case Study

In this section, we present a case study about a sample bug that we found while triaging the crashes that we obtained during our experiments. Our goal is to show an example of how DDFuzz succeeds in real life and differs from previous approaches at inspecting the program state.

The application that we consider is tcc, a project that implements a fast and small C compiler (roughly 50K lines of code). During the triage phase, AddressSanitizer reports the presence of a Global Buffer Overflow. Interestingly, no edge-coverage trial reports the same bug and therefore we opted to re-implement the approach described in the paper by Wang et al. [74] to investigate the crash and understand the reason why only one of the two fuzzers was able to detect it.

The first step consists of reconstructing the testcase tree that originated the crash. This is possible because AFL++ stores the newly generated testcases in the queue by naming them with additional info such as the time, the mutation strategy and the previous testcase whose the mutation originated the current one. Moreover, when the fuzzer applies a splicing strategy, the two parents' names are preserved, thus allowing to recover all original testcases also in this second scenario. After recovering the testcase tree, the next step is to show if each mutation generates novelty according to the fuzzer bitmap. Indeed, in case a certain testcase in the tree does not generate novelty for the edge-coverage fuzzer, it means that the

fuzzer would have skipped the testcase, loosing one step towards the crashing input. Note that for the bug we show, we found only one input that generated the corresponding crash.

After running our set of scripts that implements the previously described technique, we find that the Global Buffer Overflow is the result of 168 mutations deriving from 2 initial seeds and divided into 133 havoc and 35 splicing. More importantly, for 3 havoc mutations, the resulting testcase does not generate any novelty according to edge-coverage, while `DDFuzz` classifies it as interesting. The first of these intermediate testcases appears already after the first 10 mutations of the tree that lead to the bug. This demonstrates how our Data Dependency instrumentation can affect the findings of the fuzzer already after few initial mutations. The other two mutations that were retained because of the DDG instrumentation come later in the tree, respectively at the 18th and 105th mutation round. This case is a good example of how the augmented sensitivity caused by our data-dependency feedback can help the fuzzer to retain input that can later help to discover new bugs.

6. Discussion

Overall `DDFuzz` experimented on three different datasets for a total of 38 different target applications. Our numbers and experiments show that embedding data flow information in coverage-guided fuzzing is a useful practice. The feedback produced by such an instrumentation can reward the fuzzer in a different way compared to current state-of-the-art techniques and lead to reaching different program points that would remain unexplored with coverage-guided approaches proposed so far. This is evident when we compare the intersections of vulnerabilities triggered by our own approach against the other instrumentations that we tested. In our custom dataset, it helped to reveal 27 new bugs compared to `Edge`. Moreover, in Fuzzbench, it triggered 33 different buggy locations and in our third real-world evaluation set it was able to find 12 different vulnerabilities. However, `DDFuzz` functioning is strictly related to the internals of the tested codebase. Indeed, as we have seen across our evaluation it is able to spot interesting and different bugs only in those cases where the tested application exhibits an high data-dependent structure of the code. Therefore we tried to develop a reliable heuristic (that we referred as `DD ratio`) that helps at recognizing these cases before running the actual fuzzing session. Moreover, we tested the impact of our technique from the point of view of the growth in the fuzzer queue, demonstrating that in the average case the increase is minor than x2, while in the worst case, it is smaller than many coverage guided approaches. Finally, the code coverage reached by `DDFuzz` is affected negatively only in a minimal part (-1.4% in the worst case against `Ctx`) while in some cases, it can allow to trigger different program points. All these aspects come at the expense of a moderate slowdown of 10-14% in the average case (6-8% if we use the geometric mean) compared to traditional `Edge`.

In total our dataset includes 38 programs, therefore respecting the fuzzing guidelines indicated by the authors of [42]. While we cannot make sure to cover all possible scenarios we thing that the targets we selected for our experiments are quite representative from different points of view. For instance, they provide a good variety of weakly and highly data-dependent applications as well as they show different trends in terms of performances and discovered bugs. In particular, we found that the highest amount of data-dependency present in an app corresponds, in our set, to 23% (`c2m`) that resulted in a slowdown of 21%. This hints that in a worst-case scenario with higher values of the DD ratio, our instrumentation could penalize the fuzzing session by injecting too much instrumentation. However, during our research of the targets, especially w.r.t. the first custom dataset, we did not meet any application producing a DD ratio so high to hinder the fuzzing process.

That being said, our approach is first of all a sub-approximation of the path coverage that allows the fuzzer to reach new program points. As we described in the Introduction, path coverage is not a good solution for many applications where it results in state explosion. On the other hand, the fact itself that `DDFuzz` is an under-approximation in part justifies why our approach can only work in a subset of cases. However, we believe that relying on such approximations to produce alternative feedbacks could, and should, represent a possible road rather than just focusing on edge coverage based instrumentations and we hope, with our work, to put more emphasis on this aspect for future research.

7. Limitations and Future Work

Although we believe that our implementation well describes the potential of the data flow information as feedback for fuzzing, there are some points that we did not investigate, and that could additionally extend and improve our approach for future uses.

Firstly, our approach is useful particularly for what we called strongly data-dependent applications. This is at the same time a limitation and a feature, because it restricts the range of programs for which our fuzzing approach should be deployed.

Another point of improvement is that our current implementation does not avoid edge collisions. This could result in some paths that are ignored as the result of the `XOR` computation returns the same value for two different sets of edges. Note that this is similar to what happens with the Ngram instrumentation that solves the collisions problem only in part, introducing a more sensitive feedback that results in more collisions in the bitmap. Since the scope of our study was to show the efficacy of the DDG instrumentation, we did not implement a mechanism to avoid this issue. However, we plan to address this point in future work on this topic. For instance, a possible solution could consist of two bitmaps, one collision-free for edge coverage that implements the AFL++ approach transforming the program by breaking the critical edges in the CFG [2] and a second smaller one with collisions for DDG coverage.

For the same reason, we did not try to adapt our approach to binary-only fuzzing. Although this is technically possible, it would require a different approach to recover the DDG of the binary, and instrumentation should be injected either by binary rewriting [26] or

by emulation [12]. In any case, we believe this could represent a promising future research direction, and we hope it will be considered by researchers on this topic.

Finally, our implementation is based on AFL++ and we did not adapt it to other fuzzing engines. Given the large number of fuzzers, it is possible that our approach could work in a different way depending on the underlying implementation of the engine.

8. Related Work

We discuss now some problems and the respective proposed solutions that are orthogonal to DDFuzz.

8.1. Increasing code coverage

One of the main objectives for many proposed optimizations to coverage-guided fuzzing is to reach a higher amount of covered program points in the same time window compared to previous solutions. This metric comes from the observation that a fuzzer cannot uncover a fault in an unxeplored portion of the code.

During the last years, the community identified some artifacts in the code that prevent a fuzzer to explore the code behind these so-called "roadblocks". The main types of roadblocks are:

Multi-byte comparisons The probability that a generic byte-level mutator guess from scratch the value needed to bypass a certain multi-byte comparison is near to 0. Several approaches try to overcome this problem, like [3] that splits them into several single-byte comparisons at the compiler level, [65] and [20] that identify portions of the input related to the values in the comparisons using taint analysis, or [11] that heuristically replaces correspondences between input and comparisons.

Checksum checks Checksums checks are a particularly hard version of comparison. While they can be solved by providing valid inputs, the problem is that any generic mutation invalidates it generating an invalid input, making the fuzzer unable to explore the code behind these checks. The most common way to handle these checks is to patch them out and restore them when the fuzzer finds a crash, manually or automatically, as proposed in [11], [29], [58].

Hashtable lookups The third important code pattern identified as a problem for coverage-guided fuzzers is the hashtable lookups as the information needed to get the right item is not explicit in code coverage and eventual comparisons are between encoded versions of portions of the input. Current automatic solutions, like [11], [30], are quite naive because they can solve the roadblock if the lookup is in a specific helper function looking at its pointer arguments. While using a helper function is a common coding pattern for hashtable lookups, this is not an exhaustive solution and the state-of-the-art [10] requires manual annotations.

In addition, a popular technique to handle roadblocks is hybrid fuzzing, in which a concolic executor is used to aid the fuzzer [16], [61], [79]. For the checksum checks, the concolic engine can be used to repair the checksums on crashing testcases. For the other two kinds of roadblocks, in theory, it should be possible to solve them with this technique but concolic engines struggle to solve hashmap-like lookups due to the complexity of handling symbolic pointers.

8.2. Meaningful inputs generation

A problem that fuzzers face is the inability to stress code paths behind the parsing stage. While generic mutators are very good in stressing parsers with invalid inputs, in order to fuzz deep in the program we need mutators able to go beyond the parser.

A widely adopted solution is to guide the input generation with a model of the input format, that can be a grammar [9], [72] or a block-based model [59], using an internal representation, like the AST, that is easy to mutate while preserving validity.

An important field is the research into the automation of this process, as writing an input model is still a human task. So in the past year, some solutions were proposed to infer how the input bytes are handled by the program with the goal of mutating them accordingly [13], [29].

Another important property of inputs that some domain-based fuzzers may want to preserve is semantic validity. For instance, when fuzzing a compiler, a testcase that uses undefined variables is syntactically valid, but the compilation will fail. Fuzzers such as [32], [35] implement semantic-preserved mutations to fuzz JavaScript interpreters behind the compilation stage.

8.3. Hunting non-crashing faults

While the algorithmic improvements to fuzz testing play an important part in the game, the discrimination between testcases that trigger or not a fault is another important room for fuzzers' enhancement.

It is easy to spot bugs that cause a crash in the application just by observing the exit status for instance on UNIX systems, but many bugs are silent and do not corrupt the application state enough to cause a crash [53].

An example of this kind of bugs is many logic bugs related to integer arithmetic. While in some application the invalid state may be propagated further in the code and cause a crash, in many others this kind of faults remain silent.

In order to catch these bugs, the targets, in addition to the fuzzer instrumentation, are transformed to include tripwires and assertions to check the validity of the program's states. Many fuzzers offer the support to the LLVM sanitizers, ASan [69] for many memory corruption bugs and UBSan [1] for generic undefined behavior in C/C++ for instance. Other sanitizers to catch specific bug classes exist too, like [33].

9. Concluding Remarks

In this paper we discussed how the information extracted from the Data Dependency Graph can be integrated with the classic edge coverage to provide a better feedback for fuzzing. Our experiments showed that, for applications that have a rich set of data dependencies, this approach leads to the discovery of more and diverse bugs. Moreover, we hope that our technique and prototype based on AFL++ and LLVM will be adopted by users to fuzz programs alongside the existing coverage metrics.

Acknowledgments

We would like to thank the anonymous reviewers for their constructive feedback and Slasti Mormanti for the useful tips. This project has been supported by the Defense Advanced Research Projects Agency (DARPA) under agreement number FA875019C0003.

References

[1] Clang User's Manual. Undefined behavior sanitizer. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html. [Online; accessed 17 Sep. 2021].

[2] Critical Edges Elimination Pass. https://llvm.org/doxygen/BreakCriticalEdges_8cpp_source.html. [Online; accessed 08 Feb. 2022].

[3] Circumventing Fuzzing Roadblocks with Compiler Transformations. https://lafintel.wordpress.com/2016/08/15/circumventing-fuzzing-roadblocks-with-compiler-transformations/, 2016. [Online; accessed 10 Sep. 2021].

[4] afl-cov. https://github.com/mrash/afl-cov, Accessed February 28, 2022.

[5] Datadependencegraph class reference in the llvm framework. https://llvm.org/doxygen/classllvm_1_1DataDependenceGraph.html, Accessed February 28, 2022.

[6] Definition of ddg in the llvm framework. https://llvm.org/docs/DependenceGraphs/index.html, Accessed February 28, 2022.

[7] Fuzzbench configuration. https://google.github.io/fuzzbench/, Accessed February 28, 2022.

[8] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. *Compilers: principles, techniques and tools*. 2020.

[9] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. NAUTILUS: Fishing for deep bugs with grammars. In *NDSS*, 2019.

[10] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. Ijon: Exploring deep state spaces via fuzzing. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1597–1612. IEEE, 2020.

[11] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. Redqueen: Fuzzing with input-to-state correspondence. In *NDSS*, volume 19, pages 1–15, 2019.

[12] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX annual technical conference, FREENIX Track*, volume 41, page 46. Califor-nia, USA, 2005.

[13] Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. GRIMOIRE: Synthesizing structure while fuzzing. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1985–2002. USENIX Association, August 2019.

[14] Rastislav Bodik and Rajiv Gupta. Partial dead code elimination using slicing transformations. In *Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 159–170, 1997.

[15] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, 45(5):489–506, 2017.

[16] Luca Borzacchiello, Emilio Coppa, and Camil Demetrescu. Fuzzing Symbolic Expressions. In *Proceedings of the 43rd International Conference on Software Engineering*, ICSE '21, 2021.

[17] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[18] Qiong Cai, Lin Gao, and Jingling Xue. Region-based partial dead code elimination on predicated code. In *International Conference on Compiler Construction*, pages 150–166. Springer, 2004.

[19] Jian-Liang Chen, Feng-Jian Wang, and Yung-Lin Chen. An object-oriented dependency graph for program slicing. In *Proceedings. Technology of Object-Oriented Languages. TOOLS 24 (Cat. No. 97TB100240)*, pages 121–130. IEEE, 1997.

[20] P. Chen and H. Chen. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 711–725, 2018.

[21] Yaohui Chen, Mansour Ahmadi, Boyu Wang, Long Lu, et al. MEUZZ: Smart seed scheduling for hybrid fuzzing. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, pages 77–92, 2020.

[22] Kai Cheng, Qiang Li, Lei Wang, Qian Chen, Yaowen Zheng, Limin Sun, and Zhenkai Liang. Dtaint: detecting the taint-style vulnerability in embedded device firmware. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 430–441. IEEE, 2018.

[23] Animesh Basak Chowdhury, Raveendra Kumar Medicherla, and R Venkatesh. Verifuzz: Program aware fuzzing. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 244–249. Springer, Cham, 2019.

[24] Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. Mining hot calling contexts in small space. *Software: Practice and Experience*, 46(8):1131–1152, 2016.

[25] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. Lava: Large-scale automated vulnerability addition. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 110–121. IEEE, 2016.

[26] Gregory J Duck, Xiang Gao, and Abhik Roychoudhury. Binary rewriting without control flow recovery. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 151–163, 2020.

[27] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.

[28] Andrea Fioraldi, Daniele Cono D'Elia, and Davide Balzarotti. The use of likely invariants as feedback for fuzzers. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2829–2846. USENIX Association, August 2021.

[29] Andrea Fioraldi, Daniele Cono D'Elia, and Emilio Coppa. WEIZZ: Automatic grey-box fuzzing for structured binary formats. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2020. Association for Computing Machinery, 2020.

[30] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. Afl++: Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, 2020.

[31] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 679–696. IEEE, 2018.

[32] Samuel Groß. Fuzzil: Coverage guided fuzzing for javascript engines. 2018.

[33] Istvan Haller, Yuseok Jeon, Hui Peng, Mathias Payer, Cristiano Giuffrida, Herbert Bos, and Erik van der Kouwe. Typesan: Practical type confusion detection. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 517–528, New York, NY, USA, 2016. Association for Computing Machinery.

[34] Christian Hammer. *Information flow control for Java: a comprehensive approach based on path conditions in dependence graphs*. 2009.

[35] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. CodeAlchemist: Semantics-aware code generation to find vulnerabilities in javascript engines. 2019.

[36] Dominik Harmim, Vladimır Marcin, and Ondrej Pavela. Scalable static analysis using facebook infer. *Excel@ FIT'19*.

[37] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. Magma: A ground-truth fuzzing benchmark. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 4(3):1–29, 2020.

[38] Mark Heffernan and Kent Wilken. Data-dependency graph transformations for instruction scheduling. *Journal of Scheduling*, 8(5):427–451, 2005.

[39] Mark Heffernan, Kent Wilken, and Ghassan Shobaki. Data-dependency graph transformations for superblock scheduling. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pages 77–88. IEEE, 2006.

[40] Chin-Chia Hsu, Che-Yu Wu, Hsu-Chun Hsiao, and Shih-Kun Huang. Instrim: Lightweight instrumentation for coverage-guided fuzzing. In *Symposium on Network and Distributed System Security (NDSS), Workshop on Binary Analysis Research*, 2018.

[41] David A Kinloch and Malcolm Munro. Understanding c programs using the combined c graph representation. In *ICSM*, pages 172–180, 1994.

[42] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138, 2018.

[43] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Partial dead code elimination. *ACM SIGPLAN Notices*, 29(6):147–158, 1994.

[44] David J Kuck, Robert H Kuhn, David A Padua, Bruce Leasure, and Michael Wolfe. Dependence graphs and compiler optimizations. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 207–218, 1981.

[45] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[46] Chingren Lee, Jenq Kuen Lee, and TingTing Hwang. Compiler optimization on instruction scheduling for low power. In *Proceedings 13th International Symposium on System Synthesis*, pages 55–60. IEEE, 2000.

[47] LLVM Project. libFuzzer – a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html, September 2018. [Online; accessed 17 Sep. 2021].

[48] Shan Lu, Pin Zhou, Wei Liu, Yuanyuan Zhou, and Josep Torrellas. Pathexpander: Architectural support for increasing the path coverage of dynamic bug detection. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pages 38–52. IEEE, 2006.

[49] Valentin JM Manès, Soomin Kim, and Sang Kil Cha. Ankou: Guiding grey-box fuzzing towards combinatorial difference. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 1024–1036, 2020.

[50] Masahiro Matsubara, Kohei Sakurai, Fumio Narisawa, Masushi Enshoiwa, Yoshio Yamane, and Hisamitsu Yamanaka. Model checking with program slicing based on variable dependence graphs. *arXiv preprint arXiv:1301.0041*, 2013.

[51] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. Fuzzbench: an open fuzzer benchmarking platform and service. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1393–1403, 2021.

[52] Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44, December 1990.

[53] Marius Muench, Jan Stijohann, Frank Kargl, Aurélien Francillon, and Davide Balzarotti. What you corrupt is not what you crash: Challenges in fuzzing embedded devices. In *NDSS 2018, Network and Distributed Systems Security Symposium*, 2018.

[54] Stefan Nagy and Matthew Hicks. Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 787–802, 2019.

[55] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Parmesan: Sanitizer-guided greybox fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2289–2306, 2020.

[56] Karl J Ottenstein and Linda M Ottenstein. The program dependence graph in a software development environment. *ACM Sigplan Notices*, 19(5):177–184, 1984.

[57] Rohan Padhye, Caroline Lemieux, Koushik Sen, Laurent Simon, and Hayawardh Vijayakumar. FuzzFactory: Domain-specific fuzzing with waypoints. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019.

[58] H. Peng, Y. Shoshitaishvili, and M. Payer. T-fuzz: Fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 697–710, May 2018.

[59] V. Pham, M. Boehme, A. E. Santosa, A. R. Caciulescu, and A. Roychoudhury. Smart greybox fuzzing. *IEEE Transactions on Software Engineering*, 2019.

[60] Marco Pistoia, Satish Chandra, Stephen J Fink, and Eran Yahav. A survey of static analysis methods for identifying security vulnerabilities in software systems. *IBM Systems Journal*, 46(2):265–288, 2007.

[61] Sebastian Poeplau and Aurélien Francillon. Symbolic execution with symcc: Don't interpret, compile! In *29th USENIX Security Symposium (USENIX Security 20)*, pages 181–198, 2020.

[62] Sebastian Poeplau and Aurélien Francillon. Symqemu: Compilation-based symbolic execution for binaries. In *Proceedings of the 2021 Network and Distributed System Security Symposium*, 2021.

[63] Chenxiong Qian, Xiapu Luo, Yu Le, and Guofei Gu. Vulhunter: toward discovering vulnerabilities in android applications. *IEEE Micro*, 35(1):44–53, 2015.

[64] Dudekula Mohammad Rafi, Katam Reddy Kiran Moses, Kai Petersen, and Mika V Mäntylä. Benefits and limitations of automated software testing: Systematic literature review and practitioner survey. In *2012 7th International Workshop on Automation of Software Test (AST)*, pages 36–42. IEEE, 2012.

[65] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *NDSS*, volume 17, pages 1–14, 2017.

[66] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. Optimizing seed selection for fuzzing. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 861–875, 2014.

[67] Marc Roper. Software Testing, 1994.

[68] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 12–27. Association for Computing Machinery, 1988.

[69] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *2012 USENIX Annual Technical Conference (USENIXATC 12)*, pages 309–318, 2012.

[70] Chayanika Sharma, Sangeeta Sabharwal, and Ritu Sibal. A survey on software testing techniques using genetic algorithm. *arXiv preprint arXiv:1411.1154*, 2014.

[71] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 138–157. IEEE, 2016.

[72] Prashast Srivastava and Mathias Payer. Gramatron: Effective grammar-aware fuzzing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2021, New York, NY, USA, 2021. Association for Computing Machinery.

[73] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[74] Jinghan Wang, Yue Duan, Wei Song, Heng Yin, and Chengyu Song. Be sensitive and collaborative: Analyzing impact of coverage metrics in greybox fuzzing. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, pages 1–15, 2019.

[75] Lei Wang, Qiang Zhang, and PengChao Zhao. Automated detection of code vulnerabilities based on program analysis and model checking. In *2008 Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 165–173. IEEE, 2008.

[76] Mark Weiser. Program slicing. *IEEE Transactions on software engineering*, (4):352–357, 1984.

[77] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*, pages 590–604. IEEE, 2014.

[78] Fabian Yamaguchi, Alwin Maier, Hugo Gascon, and Konrad Rieck. Automatic inference of search patterns for taint-style vulnerabilities. In *2015 IEEE Symposium on Security and Privacy*, pages 797–812. IEEE, 2015.

[79] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. Qsym: A practical concolic execution engine tailored for hybrid fuzzing. In *Proceedings of the 27th USENIX Conference on Security Symposium*, SEC'18, pages 745–761. USENIX Association, 2018.

[80] Michał Zalewski. American Fuzzy Lop - Whitepaper. https://lcamtuf.coredump.cx/afl/technical_details.txt, 2016. [Online; accessed 17 Sep. 2021].

[81] Zeineb Zhioua, Stuart Short, and Yves Roudier. Static code analysis for software security verification: Problems and approaches. In *2014 IEEE 38th International Computer Software and Applications Conference Workshops*, pages 102–109. IEEE, 2014.