Arvin: Greybox Fuzzing Using Approximate Dynamic CFG Analysis

Sirus Shahini
sirus.shahini@utah.edu
University of Utah

Mu Zhang
muzhang@cs.utah.edu
University of Utah

Mathias Payer
mathias.payer@nebelwelt.net
EPFL

Robert Ricci
ricci@cs.utah.edu
University of Utah

ABSTRACT

Fuzzing has emerged as the most broadly used testing technique to discover bugs. Effective fuzzers rely on coverage to prioritize inputs that exercise new program areas. Edge-based code coverage of the Program Under Test (PUT) is the most commonly used coverage today. It is cheap to collect—a simple counter per basic block edge suffices. Unfortunately, edge coverage lacks context information: it exclusively records how many times each edge was executed but lacks the information necessary to trace actual paths of execution.

Our new fuzzer Arvin gathers probabilistic full traces of PUT executions to construct Dynamic Control Flow Graphs (DCFGs). These DCFGs observe a richer set of program behaviors, such as the "depth" of execution, different paths to reach the same basic block, and targeting specific functions and paths. Prioritizing the most promising inputs based on these behaviors improves fuzzing effectiveness by increasing the diversity of explored basic blocks.

Designing a DCFG-aware fuzzer raises a key challenge: collecting the required information needs complex instrumentation which results in performance overheads. Our prototype approximates DCFG and enables lightweight, asynchronous coordination between fuzzing processes, making DCFG-based fuzzing practical.

By approximating DCFGs, Arvin is fast, resulting in at least an eight-fold increase in fuzzing speed. Because it effectively prioritizes inputs using methods like depth comparison and directed exclusion, which are unavailable to other fuzzers, it finds bugs missed by others. We compare its ability to find bugs using various Linux programs and discover 50 bugs, 23 of which are uniquely found by Arvin.

CCS CONCEPTS

• **Security and privacy → Vulnerability scanners**; **Software security engineering**.

KEYWORDS

fuzzer, vulnerability, Control Flow Graph, Input Prioritization

ACM Reference Format:
Sirus Shahini, Mu Zhang, Mathias Payer, and Robert Ricci. 2023. Arvin: Greybox Fuzzing Using Approximate Dynamic CFG Analysis. In *ACM ASIA Conference on Computer and Communications Security (ASIA CCS '23), July 10–14, 2023, Melbourne, VIC, Australia.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3579856.3582813

1 INTRODUCTION

Fuzz testing is a key technique to improve the resilience of software against bugs [2, 3, 7–9, 15, 19, 25, 27, 28]. In the last few years, fuzzers have discovered thousands of critical security bugs [3, 16, 19]. Coverage-guided or *greybox* fuzzing [3, 8, 12, 14, 15, 18, 20, 23, 25] uses a light-weight instrumentation to observe which basic blocks of the Program Under Test (PUT) are executed. While the feedback that the fuzzer gathers is used to make better decisions about the mutation process and generating more quality inputs, all fuzzers still spend a large amount of their time with millions of inputs that will never yield any new observations.

Naturally, "better candidates" for each mutation increase the likelihood of generating better "mutated" inputs. The central challenge in fuzzing is the huge number of inputs to try: only a minuscule fraction of possible inputs will trigger bugs. Most fuzzing techniques do not discuss how to distill this small portion of "better" inputs from numerous candidates but simply aim to increase the likelihood of finding inputs that belong to this tiny portion of the sample space. Prior work has managed to prioritize the mutation of specific bytes in an input file, which have control or data dependencies on certain target code such as compare instructions [6, 23, 27], error handlers [23], sanitizers [22] or array accesses [11]. Nonetheless, prior to mutating an input from the input queue, we must identify a "better" input among many candidates generated from previous mutations. Intuitively, these prior inputs are not equally important—mutating an input that has greater potential for improving coverage or finding crashes makes a fuzzer more efficient and thus needs to be prioritized.

An overall observation of the change of edge coverage is not sufficient to differentiate the path-finding potential of inputs. Fuzzers create a lot of inputs that cause the program to exit early, such as exiting due to errors with malformed input. Various techniques like taint-analysis, hardware-assisted analysis, in-memory patching, and hooking of instructions have been used to improve mutation strategy, and seed generation [6, 10, 23]. Another approach is to create better inputs that observe behavior deeper in the call graph, exercising more application logic without spending too many iterations creating inputs. Seed prioritization, which is of paramount importance in fuzzing, remains under-explored in the literature.

While previous work has improved *which inputs are mutated* and managed to mutate specific inputs selectively, *prioritization of mutated inputs remains ad hoc.* Fuzzers simply unconditionally store the very first input that reaches a new program area or increases its hit count. Our core contribution is to distinguish *how a target area is reached*, allowing the fuzzer to select inputs based on reaching paths. As we demonstrate, *CFG-awareness* enables effective prioritization of mutated inputs. Control Follow Graphs (CFGs), which indicate how basic blocks (or other execution units like functions) are related to one another, provide great contextual information regarding the PUT. This information, if used correctly, significantly improves runtime decisions to mutate and prioritize inputs. CFG analysis can be performed statically or dynamically. Dynamic CFGs (DCFGs) are generated at runtime through monitoring what basic blocks of the PUT are executed and how the transitions happen between those basic blocks. A DCFG provides a fine-grained and more accurate image of what happens when the PUT is fed with a new mutated seed. DCFGs capture the precise execution context of the discovered basic blocks, like their depths and relative distances in the control flow. Our observation is that different inputs that cover the *same basic blocks* can have *different* potential with respect to being mutated to reach newer or deeper regions due to different contextual information in their DCFGs. As a result, we propose to leverage DCFG information to guide the selection of the "best" inputs which directly improves the fuzzer's ability to grow code coverage and find bugs. In short, *context awareness matters when determining the most-viable inputs.*

Control flow graphs are complex structures, and this complexity can create serious challenges if the fuzzer needs to construct and analyze dynamic CFGs at runtime. The added complexity will result in a significant slow-down of the entire fuzzing process. It is important to find a balance between the benefits that a complex but valuable structure like a DCFG provides to the fuzzer and the extra cost that is required to manage this complexity. We observed that by introducing approximation to the graph structure, we can leverage the benefits of using a DCFG without sacrificing execution speed. We design *directed approximation* of DCFGs through observing the frequency of running basic blocks to selectively focus on *parts* of a DCFG: by tracking at runtime which basic blocks of the PUT have less potential to increase coverage and trigger crashes, we can dynamically adjust the instrumentation and graph analysis to spend more time on important parts.

We present Arvin, a context-aware greybox fuzzer that uses dynamic CFG analysis and directed approximation to find high-quality inputs. Arvin uses binary rewriting to inject instrumentation that on-the-fly builds DCFGs. To offset the cost of DCFG construction, Arvin dynamically identifies less important parts of the PUT based on their execution frequency to exclude them from instrumentation and steer the iterations towards the parts of the program most likely to lead to growth in coverage. At each iteration, using a novel technique called Decremental CFG Growth (DCG), only a subset of the basic blocks will be instrumented, and the fuzzer constantly rewrites the PUT in memory to change the instrumentation. Arvin can selectively collect a rich set of CFG information for input prioritization. We have designed a flexible prioritization policy that can be tailored to the characteristics of the individual targets. The collective result of DCFG-based input prioritization and graph approximation effectively increases the chance of finding better inputs within far fewer fuzzing iterations.

In summary, we make the following contributions:

- Design and implementation of a new coverage-guided fuzzer that automatically identifies and prioritizes high-quality inputs during fuzzing.
- New data structures for basic block instrumentation, with enough flexibility to add or change Arvin's default low-level prioritization factors based on the nature of the programs and their inputs.
- A new approximative technique to speed-up CFG analysis while maintaining sufficient accuracy in each graph in each iteration. Our design allows the use of a wide class of different CFG properties for input prioritization.
- A *co-operative* parallel fuzzing scheme that uses a heavily-instrumented version of the PUT to guide the runs of lightweight instances.
- A greybox fuzzer that works independently from any external library and does not require access to the PUT's source code. Arvin works directly on the target binary.

2 DESIGN

To effectively prioritize generated inputs, we leverage dynamic control flow information from each iteration and at the same time alleviate the high cost of graph analysis using approximation. The architecture of Arvin has two main components. The fuzzing engine and the DCFG library. The fuzzing engine generates inputs, orchestrates the execution of the instrumented `PUT`, and analyzes the resulting CFGs. The DCFG library is the set of two (`libarv` and `libarvp`) native-code libraries that manage graph construction, instrumentation and synchronization. `libarv` contains the full instrumentation logic, and the executions of the PUT that it uses to produce DCFGs. Dynamic graphs are created in the DCFG library and then are fed back to the fuzzing engine, which extracts runtime information from the DCFGs at different granularities. This information is used to analyze the execution trace and is compared with other DCFGs from previous iterations. This provides us with the fundamental means using which we find an accurate and effective order for the inputs. Arvin uses graph metrics, like size, depth, the number of times different basic blocks are executed and the presence of sensitive functions to prioritize inputs. `libarvp` is a lightweight load-balancing library used in co-operative parallel fuzzing, and does not generate a CFG on its own. It helps the fuzzing engine to distribute the inputs on fuzzing instances after inputs have been prioritized through DCFG analysis.

A graph in Arvin for any iteration shows either the precise (in precision mode) or an approximate (DCG mode) control flow information of the PUT being executed using the tried test input. The metadata of each node describes the type of the exercised basic block and the number of times it has been executed (or hit).

2.1 Building the CFG

Our instrumentation traces the PUT to build a CFG that precisely reflects what has happened during the execution and then approximate the new graphs for next executions using the collective information obtained from previous iterations. The correct structure of the CFG to show the accurate transitions between basic blocks is the basis for our analysis to specify the priority of each input.

We divide basic blocks into two general categories of *independent* and *nested* basic blocks. This classification is used to improve the accuracy of the recorded transitions in the graph. Normally, a transition from a basic block to another can be either done using a conditional jump (*branch*) instruction or by a *call* instruction. In a greybox fuzzer, the coverage instrumentation is usually at the beginning of each basic block. The instrumentation informs the fuzzer that the basic block is going to execute. This simply means that the fuzzer can see a basic block when an explicit *branch* or *call* instruction is executed. However, fuzzers (including AFL) generally do not consider a *call* instruction as the prologue of a new basic block because what comes after that, deterministically is executed after the called function returns (the execution flow does not depend on the evaluation result of a branch instruction). As a result, the fuzzer will recognize the start of the execution of all traversed basic blocks, however, the correct representation of many transitions will be missed. Although this strategy is not problematic in a fuzzer like AFL for *marking* a basic block or an edge in its basic block/edge map, it poses a serious challenge to build a dynamic CFG as Arvin needs. Arvin needs to know at any given moment during the execution of the PUT, which basic block the processor is executing regardless of whether there have been some jumps back and forth between the current basic block and the neighboring basic blocks. We need to break normal basic blocks whenever we come across a *call* instruction and begin a new basic block right after that. We define a *nested* basic block as any basic block that starts with a *call* instruction. Any other basic block which is not nested is an *independent* basic block.

The graph nodes are mapped to the executed independent basic blocks during the iteration. We do not create new nodes in the DCFG for nested blocks when they are executed. The purpose of having nested blocks is solely to maintain the precision of the dynamic graph as it is being made.

Figure 1 demonstrates how we address the challenge of creating a dynamic CFG during execution. In this example, we aim to record the correct transitions between three basic blocks A, B and C in a simple control flow graph. The control flow starts from basic block A. Supposing A has not been visited before in this iteration, the fuzzer makes a new node for A. We describe the three possible cases as different scenarios to demonstrate how nested blocks help the fuzzer ensure the needed precision in tracking basic blocks:

(1) A branch instruction at the end of basic block A takes control to basic block B. The fuzzer adds B as a new child of A. B finishes execution and branches back to A. Since a branch always lands on the beginning of the destination basic block, the fuzzer sees that A is about to run but A already exists in the graph and hence it is not recorded as a child node of B. A then branches to C and a similar procedure to the first branch is performed.

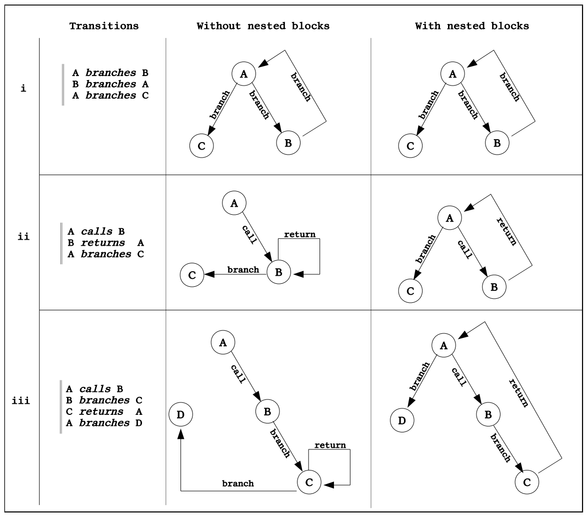


Figure 1: Example iterations *i* through *iii* with and without nested blocks.

(2) B is the first basic block of a function. We call such a basic block, a function basic block which is also an independent basic block. A *calls* B and B is added to the graph. B finishes execution and gives control back to A using a *return* instruction. Technically what happens is that the processor jumps back to the middle of basic block A while the last known visited basic block for the fuzzer, is B. What happens now is, if A jumps to C using a branch instruction, the new transition will be seen as $B \longrightarrow C$ instead of $A \longrightarrow C$. We solve this issue by creating a nested basic block inside A right after the *call* instruction which will help the DCFG library correctly track the transitions after call instructions. The metadata value of a basic block is defined to have various information bits, one of which is the *BLOCK_TYPE*. A nested basic block has the same identifier as its encompassing independent basic block with a different *BLOCK_TYPE* value in its metadata. Considering this new basic block, after B executes the *return* instruction, `libarv` sees the new transition $B \longrightarrow A$, and when A finishes its execution by branching to C, the transition is correctly recorded in the graph as $A \longrightarrow C$.

(3) B is a function basic block. A calls B and the $A \longrightarrow B$ transition is seen by `libarv` and recorded in the graph. B branches to another basic block C of the same function which B has started. The transition $B \longrightarrow C$ is executed and recorded. C executes a *return* instruction and the processor jumps to the last address stored on the stack, which is the address of the next instruction after the initial *call* instruction to B. Again, what happens is that without the nested block after the call instruction, the last transition would be invisible to the fuzzer and if A branches to basic block D at the end of its execution, the incorrect transition $C \longrightarrow D$ would be recorded. However, with nested basic blocks, the library sees the transition $A \longrightarrow D$ after the previous transition of C to the nested block in A and the correct graph will be built.

After each iteration, the fuzzing engine traverses the graph stored in shared memory and makes a copy of the relevant information into its private memory so that it can be compared with the results of other iterations. This process determines the priority of all inputs in the queue. Properties like depth, number of new unvisited basic blocks, presence of important or dangerous functions in the CFG and the proximity of new basic blocks to the important blocks are examples of prioritization factors. A new graph will then be made for the next iteration using the same process.

The prioritization weight of the graph properties is decided upon based on the active priority model in Arvin. A priority model describes in what order different graph metrics like depth and size should be evaluated to score inputs. New priority models can be defined for Arvin based on the characteristics of each individual PUT. In each priority model, the graph properties that should be evaluated for input prioritization and the order in which those properties should be evaluated are specified. We defined two default priority models, TNF ("Tree Nodes First", which prioritizes finding large graphs) and TDF ("Tree Depth First", which prioritizes deep graphs) for Arvin. TNF has the following metrics evaluated for each generated input, in the order written below:

- Presence of user-defined marked functions in the input graph.
- Size of the graph.
- Depth of the graph.
- Hit count (execution frequency) of basic blocks in the graph.

TDF evaluates the same properties in the order below:

- Presence of user-defined marked functions in the input graph.
- Depth of the graph.
- Size of the graph.
- Hit count (execution frequency) of basic blocks in the graph.

We also defined a third model, TNS ("Tree No Sort"), that simulates AFL's input selection and ignores all graph properties.

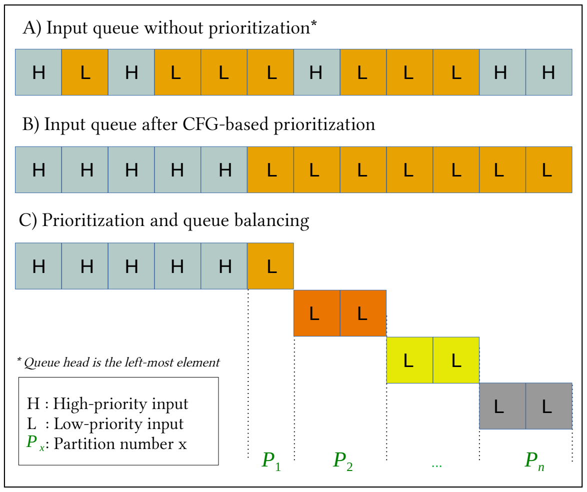

A) Input queue without prioritization*
H L H L L L H L L L H H
B) Input queue after CFG-based prioritization
H H H H H L L L L L L L
C) Prioritization and queue balancing
H H H H H L
L L
L L
L L
* Queue head is the left-most element
H : High-priority input
L : Low-priority input
P_x: Partition number x
P_1
P_2
...
P_n

Figure 2: High-level view of an input queue without Arvin prioritization (A) and with Arvin prioritization (B and C) and the effect of partitioning similar low-priority inputs.

2.2 Balancing the Priorities

Prioritization allows the fuzzer to focus on more important inputs which consequently reduces the total amount of time spent on similar inputs. It is possible, however, that a small portion of lower-priority inputs is worth earlier evaluation. For example, a new mutation might generate an input that increases the frequency of execution of a specific basic block (e.g., to access a linear memory range) close to a fault but barely not triggering the fault. It is possible that a lower-priority input is quite close to crashing the PUT.

Mutated inputs that score high enough for further mutation are stored in a multi-level queue structure; a coarse-grained division to a high priority (HP) and a low priority (LP) segment, and a fine-grained prioritization within each segment. The LP segment will be used to hold similar inputs to those already in the HP segment.

We define two inputs to be similar if they have extremely close or equal prioritization metrics in their CFGs, irrespective of the contents of the two inputs. This means that two inputs with entirely different contents might be considered similar because of triggering similar behavior in the PUT.

We dynamically and randomly increase the priority of a small number of LP inputs, on the chance that, while they themselves may not score well, mutations of them might lead to higher-quality inputs. However, there is a massive number of similar inputs that we want to identify and deprioritize. Since we spend a high analysis cost for evaluating each input, we have to carefully specify what inputs are worth the added cost, which is resulted from graph traversal. In contrast to a fast greybox fuzzer which stores an input if the input shows any form of change in the coverage map—irrespective of how contextually similar it might be to the previously stored inputs—we cannot afford to spend all the fuzzing time on millions of inputs that are similar to each other. We need to balance exploration vs. exploitation carefully and effectively in Arvin.

We figured there would be a notable possibility of starvation in the LP segment due to the high volume of similar inputs in the sample space. When the fuzzer chooses an input from this segment, the generator may make additional similar inputs with close scores to their parent, preventing such inputs from being added to the HP segment, and they will be assigned to an index in close proximity to their parent, being also in the LP segment. When this cycle repeats, other inputs in the LP segment will get starved. Further, we want to limit the evaluation of similar inputs to increase coverage faster.

To resolve these two issues (starvation and the presence of a huge number of similar inputs), we expanded our fine-grained prioritization by "partitioning" each class of similar inputs in the LP segment (Figure 2) and defining a limited capacity for each partition. For example, there is a maximum capacity for the LP inputs with the depth of 200. If the capacity of a partition has been fully consumed, the partition gets frozen and the fuzzer avoids placing further similar inputs next to their parent in that partition, effectively eliminating the chance of starvation while keeping the priority of the inputs in the frozen partition unchanged. After each iteration, we post-process the queue and the assigned capacities. A partition will get unfrozen if it has remained frozen for enough cycles. This strategy improves performance, avoids starvation and stays fair to the low-priority inputs. In summary, through deprioritization of lower-quality inputs and enforcing fine-grained limitations on the queue partitions, Arvin refrains from spending too much time on similar inputs, which is a major issue in other fuzzers.

2.3 Decremental CFG Growth

In any PUT, many basic blocks (along with the instrumentation code) are executed in each iteration. Multiple executions of the same basic block do not increase coverage, and do not necessarily lead to finding bugs. However, they *do* strongly influence the runtime of the PUT, due to the repeated calls to the instrumentation code. In Arvin, this time will be spent on the transition of the control to the fuzzer and traversing the graph structures in the DCFG library. Our strategy for speeding up fuzzing is to avoid calling the instrumentation code for basic blocks that execute repeatedly without providing us new CFG information, such as loop bodies. Appendix C presents a code example.

We introduce *directed approximation* to reduce the amount of instrumentation required, excluding instrumentation on less-promising basic blocks (and consequently less-promising paths) to spend more time on more-promising paths. We do so by removing instrumentation from expensive basic blocks: a basic block is *expensive* if the DCFG library observes that it was executed more frequently than a defined threshold called the *DCG Hit Threshold* (*DHT*).

Note that we have two types of runtime analysis. The first happens in the DCFG library to record and manage the graph while the PUT is running during an iteration. The second happens in the fuzzer outside the library after each iteration, to compute the properties of the graph and prioritize inputs. DCG aims to lower the performance cost of the former while refraining from diminishing the information needed by the latter to prioritize inputs.

2.3.1 Identifying Expensive Basic Blocks. *Identification* is the process of finding the basic blocks that have been executed more than the defined DHT in an iteration. Doing so is straightforward, as Arvin's CFG structure includes a hit counter on each node that is incremented every time the associated given basic block is executed. After an iteration finishes, we traverse the CFG, comparing the hit count for each node with DHT to mark expensive blocks.

Note that we cannot use static analysis to find expensive nodes for three reasons. First, counting the exact number of times each basic block executes requires dynamic analysis: this number changes based on the dynamic execution context. Second, while finding loops using static analysis is possible, we do not necessarily mark all loops as expensive. Third, we need to achieve *identification* and *exploration* for all traced basic blocks: we do need to instrument all reached basic blocks, even the expensive ones, in at least one iteration to include them in our dynamic CFG. Additionally, we still need an input that can reach a given expensive block to evaluate it more exhaustively in the next cycles (explained in Section 2.4) and to find neighboring blocks that are reached through it.

2.3.2 Excluding Basic Blocks From Instrumentation. *Exclusion* is the process of modifying the PUT image in memory to remove the instrumentation from basic blocks that have been identified as expensive. Thus, they do not transition to the DCFG library and effectively do not pass through the runtime CFG traversal code inside the library. Note that the excluded blocks are still *executed*, they are simply not *instrumented*. This technique does not remove them

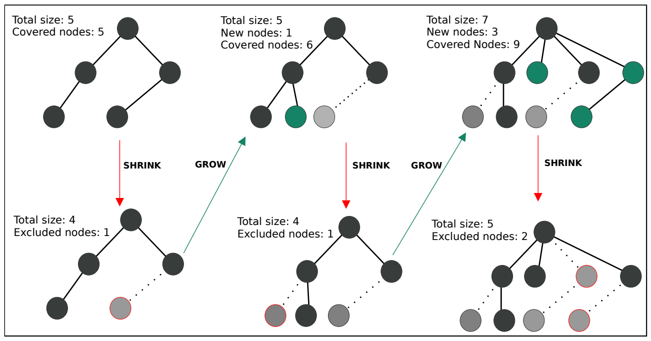


Figure 3: An example of DCG balancing tracing cost vs coverage growth.

entirely from the CFG: excluded basic blocks are still kept in an *information base* stored in the fuzzer and used in the graph analysis to prioritize inputs. They simply do not appear in *the runtime CFGs of the next iterations*. Effectively, DCG *deprioritizes* less important basic blocks *in the DCFG library* which is affected the most by the size of the CFG. The fuzzer, *outside* the library does have access to the information of the excluded nodes and uses that information for CFG analysis and prioritization. This has two important consequences. First, the information of the excluded nodes is preserved in the fuzzer and used to prioritize inputs and tailor mutation to increase the likelihood of finding bugs in loops and finding code parts that access sequential memory. Secondly, In the case of executing the excluded basic blocks in the next iterations, we reduce the load of graph analysis by not tracing excluded nodes. Effectively we increase the chance of finding *new basic blocks* instead of repetitively adding and traversing the known more frequent basic blocks.

Note that if an input causes the PUT to transition from the excluded loop to a new basic block, the new basic block is correctly recorded in the graph, and this information is conveyed to the fuzzer for prioritization; exclusion of a basic block by the DCG will not cause the fuzzer to miss the new basic blocks that are reached through the excluded basic block.

2.3.3 The Shrink-Grow Cycle in DCG. DCG has two operational stages, *shrink* and *grow*. In each *shrink* stage, after the CFG has been analyzed, the fuzzer excludes expensive nodes, reducing the tracing cost. In the *grow* stage, when new nodes are added to the graph, the analysis cost does not proportionally increase: this stage takes advantage of the time saved by the *shrink* stage. The combination of the two stages produces an *approximated* graph rather than a precise graph. However, the key point is that, instead of introducing randomness to approximate and decrease the size of the graph, which might lead to the exclusion of important nodes, the approximation in Arvin is *directed* to focus on the exclusion of less potential and slow basic blocks.

When we have gathered the metadata for the expensive nodes, having them profiled again in the next graphs is not necessary. While we do need precision for graph analysis, we only require detailed tracing for the newly explored areas. Precise tracing information is achieved *collectively* and *separately* in different areas of

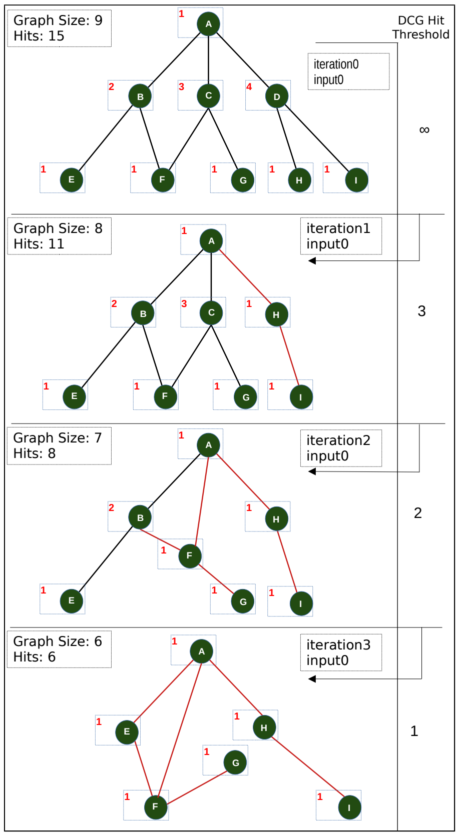


Figure 4: An example of creating multiple dynamic CFGs using DCG technique with the same input

the PUT and at the same time the unnecessary high cost of precision for unwanted and older areas is avoided. An example of these stages operating on a simple CFG is shown in Figure 3. It shows how the shrink-grow strategy works in Arvin to increase coverage while preventing the tracing cost from increasing proportionally to the added coverage. The fuzzer memorizes the information about all the visited basic blocks (including the excluded ones) while DCG keeps the fuzzing speed close to a constant number of iterations per second for the majority of iterations and increases the coverage at the same time. This helps Arvin reach a stable and reasonable speed even for very large programs.

2.3.4 Example of DCG in Action. Figure 4 shows an example of this approximation. At the top of the figure, we have the precise graph showing a simple execution of a program. This is the graph that is generated in the first execution irrespective of the DHT value. However, DHT specifies how the approximation will be carried out for these basic blocks in the next iterations. Each node and the number of times the corresponding basic block has been executed is written in the graph. For example, basic blocks C and D have been executed 3 and 4 times respectively. In the first execution, we have 15 trace calls in the library as the set of 9 basic blocks are in total executed 15 times. D transitions to H through a `call` instruction. A nested block in D is used to register H and I as the children of D. One level lower in the figure, the result of the second execution with the same input when DHT is set to 3 has been shown; node D is *excluded* from the graph. Note that we still have the information of node D in the first graph but we do not want this node to be evaluated again in the second graph. Since normally D and its nested blocks have equal hit counts, D will be excluded along with its nested basic blocks. Figure 5 shows how basic blocks D, H, I and their transitions are identified before and after the exclusion stage that targets basic block D. At the top of the figure, it can be seen that A *branches* to D which has a *call* to H. This means that we have a nested block after the `call` instruction - N_{D_1} - which will be executed after H, *returns* to D. Next, D *branches* to I. The result of these transitions is registering H and I as the children of D. However, after the exclusion of D and N_{D_1}, in the second iteration, the transitions to and from D and N_{D_1} are missed which is the expected behavior. Instead the fuzzer observes the sequential execution of H right after A and I after H. This causes I to be recorded as the child of H in the dynamic graph built for the second iteration, while H and I are originally the children of the excluded node D. This is an example of precision loss. Similarly, lower graphs show the results of the third and fourth iterations when we decrease DHT to 2 and 1 respectively.

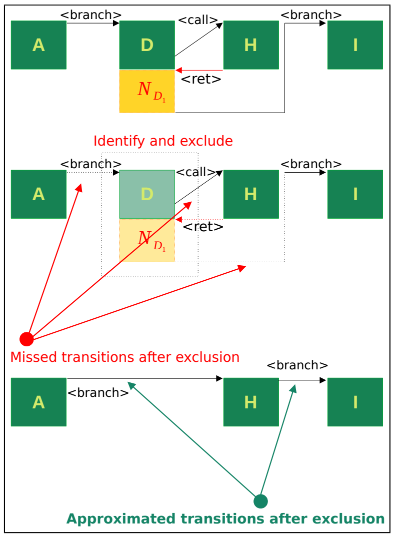

<branch>
<call>
<branch>
A
D
H
I
N_{D_1}
<ret>
Identify and exclude
Missed transitions after exclusion
Approximated transitions after exclusion

Figure 5: Effect of exclusion after the first iteration on identification of nodes and transitions in the second iteration

2.3.5 Additional Challenges. With an approximate CFG, fuzzing multi-threaded targets is possible as the major problem with multi-threaded targets while using one shared memory instance is the loss of precision in the CFG. The disadvantage of reducing precision (which is comparing approximate graphs instead of precise graphs) will be outweighed by the benefit of a higher speed and more flexibility in managing the graphs. Also, since approximation happens in a directed manner and the fact that DCG naturally targets dependent (nested) basic blocks more frequently, the precision loss mostly happens in less important parts of the graph. Note that loss of precision also happens even without DCG in effect, when the PUT jumps to an external uninstrumented library. In this regard, we can safely assume that the nodes that are excluded by DCG in

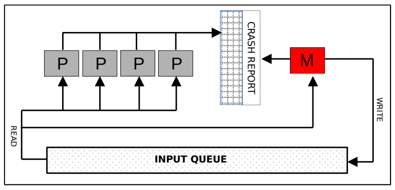


Figure 6: Distribution of seeds in the input queue between the master process (M) and the parallel processes (P).

Arvin, belong to an external entity that we are not interested in tracing. There is an important difference though; a node is only excluded from the graph if and only if we have previously traced it in a DCFG and identified it as an expensive node. This is opposite to a pure external transition into which the fuzzer does not have any insight to analyze what has happened during the transition.

DCG improves speed by excluding the shared slow nodes between different inputs. DCG always increases relative speed. This means that the result of applying DCG to the dynamic graph of any iteration always decreases the evaluation time for the same input. And given that there are usually many shared nodes between the mutations of an input and the input itself, the overall effect of applying DCG, is uniform speed improvement for the mutations of any given input. Finally, note that the implementation of DCG is only possible due to the use of dynamic runtime instrumentation as opposed to static file-based instrumentation.

2.4 Parallel Fuzzing and Adaptive Mutation

Arvin uses a complex memory layout to record and manage DCFGs. This complicates parallelism in the fuzzer due to various concurrency issues. We addressed this problem by creating a separate library for parallel fuzzing and defining a different fuzzing mode for parallel instances. The fuzzing strategy is different in the *master* mode from the *parallel mode* with each mode having its own native library. The master process tackles the DCFG construction and analysis, stores and queues the inputs, and then has the parallel processes pick up the generated inputs and perform non-deterministic mutations. The parallel mode uses a separate fast fuzzing model that does not incur the cost of CFG tracing and analysis. Figure 6 shows how the master and parallel processes cooperate with each other to cover the input queue.

Arvin tunes mutation dynamically. Better coverage does not necessarily mean finding more bugs [29]. Executing a bogus basic block is not enough by itself to crash the PUT: the right input must be fed into the target to create the necessary context for the basic block to trigger the bug. For example, in `div %rcx` instruction, the CPU will fail to complete the operation only if `rcx` register contains zero. This means that the more possibilities we try for each given set of values that can be consumed in a basic block, the more likely it is to crash the PUT. However, trying more possibilities means investing more time on a given input file.

Arvin uses two mutation stages which is a common fuzzing strategy; the mutation functions (like bit-flipping or arithmetic operations) are also similar to other fuzzers. The first stage deterministically executes different mutation functions on the input (IT stage). The second stage (NI stage) executes mutation functions

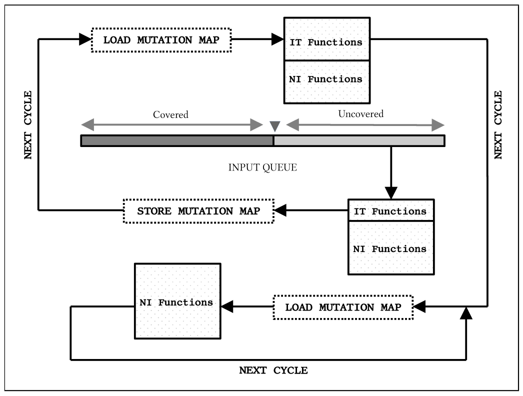


Figure 7: A high-level view of automatic tuning of mutation functions. The area of NI or IT functions is proportional to the number of times they're used for mutation of the input.

probabilistically. We figured careful balancing of these two stages in Arvin is important to find more quality inputs. NI stage (analogous to *havoc* in some other fuzzers), normally yields a better coverage growth (also discussed by Wu et al. [31]). However, after completing one cycle—in any fuzzer—the coverage growth gets much slower and leaves enough room for spending more time on each input individually in the queue. This will also cover the *excluded* basic blocks by DCG; while we reduce the graph analysis cost in the DCFG library, later on, we spend more mutation time on the expensive basic blocks, including loops and other code parts that might relate to memory corruption bugs.

Arvin decides how the mutation functions should be used to create a balance between the two mutation stages to increase the likelihood of finding crashes. This works based on how the PUT responds to mutations in either IT or NI stages. Arvin automatically changes mutation patterns after it gathers enough basic blocks and the coverage reaches a relative saturation point which causes the fuzzer to execute more aggressive deterministic mutations (example in Appendix A). This combines the capability of increasing the coverage using the information the DCFGs expose and increasing the chance of finding more crashes. In Figure 7 we show a high-level view of how our dynamic mutation tuning works.

Initially, an input in the input queue is uncovered. It is considered covered when it passes a full cycle of mutation. A cycle of mutation for an input is complete when the fuzzer advances to the next input in the queue. Generally, the queue, for the most part, consists of mutated inputs. When the queue is completely covered and the fuzzer jumps back to the first element of the queue, it is said one cycle of fuzzing has elapsed. Initially, Arvin tries to make a mutation map for the current input under evaluation that shows which parts of the input respond better to mutation in terms of reflecting a dynamic behavior in the PUT. In the first cycle, we do not wait in the deterministic stage for a long time. This is shown in Figure 7 as the smaller area of *IT Functions* versus the bigger area of *NI Functions* in the corresponding rectangles. When Arvin gets back to the same input in the next cycle, Mutation Map is loaded, and deterministic checks are executed exhaustively based on the map. In the third cycle, we only run NI stage for the input as we have already exhausted all deterministic checks in the previous cycle. This also happens in the next cycles for the same input. This whole process is performed on a *per-input* basis; any individual input goes through the same stages as described.

3 IMPLEMENTATION

Given the novel instrumentation strategy and using a native DCFG library in the design, we decided to build Arvin as a new greybox fuzzer. We implemented Arvin as a standalone fuzzer from the ground up in 15,000 lines of C. Arvin performs the instrumentation at runtime over the loaded memory image of the executable PUT. Transitions between the PUT and the fuzzer are managed through "ptrace" mechanism of the Linux kernel. The DCFG library traces the PUT, builds the DCFGs, and relays the information back to the fuzzing engine. Both the master and parallel libraries are native-code shared objects injected into to the PUT's address space depending on the fuzzing mode to monitor the control flow and manage the PUT's execution. DCG finds the candidate graph nodes for exclusion and the fuzzing library skips tracing of the corresponding basic blocks in the next iterations. We defined two main priority models in the fuzzing engine, TDF and TNF. These models prioritize inputs based on the graph depth, graph size and coverage, the presence of marked basic blocks (in targeted fuzzing) and the frequency of the execution of the basic blocks. TDF however, assigns a higher score to a *deeper* input while TNF prioritizes coverage over depth. For the purpose of comparing coverage depth of our fuzzer to AFL, we defined an AFL-simulated mode, TNS, in Arvin. By default. Arvin takes input from STDIO and files but also supports network sockets.

Before starting the fuzzing engine, Arvin performs a pre-processing pass over the PUT using angr framework [26]. To avoid the limitations and costs of simulation/emulation, we instrument and execute the PUT natively. Although we have written the current prototype of Arvin for x86-64 architecture, with moderate changes only to the instrumentation code, it can run on other architectures as well. Arvin has been implemented to run on Linux.

4 EVALUATION

Our test environment was a CloudLab [13] node with an Intel Xeon Gold 6142 processor running Ubuntu 20 with Linux kernel 5.4. For our fuzzing sessions we mainly used binutils package and unzip because of their ubiquity and the fact that they are secure and highly tested programs which make finding bugs in them relatively quite challenging. We also fuzzed several programs from the OSS-Fuzz project [25]. In this section, we present our evaluation results for coverage growth, fuzzing speed, the effectiveness of our prioritization policies, finding bugs, and parallel fuzzing. The default priority model in all Arvin sessions has been TNF[1]. To understand the effects of our DCG technique, we include *Arvin_pr* (Arvin in precision mode) in several tests. In this mode, DCG is disabled, and native static instrumentation is used instead of runtime instrumentation. In this mode, we do not need the modification of the PUT memory image, using "ptrace" is not necessary and we write the instrumentation payload at compile time. For Arvin_pr, the fuzzing library ("libarv") is still loaded into the PUT's address space, but

[1]For the majority of PUTs, TNF and TDF yield similar results.

Table 1: Comparison of Arvin and AFL++ time and coverage.

Target	Coverage Delta	Arvin Time	AFL++ Time	**Coverage Ratio**
objdump	+3%	34%	99%	**7**
size	+12%	57%	98%	**7**
addr2line	+3%	60%	95%	**8**
readelf	+6%	39%	96%	**9**
nm	+3%	87%	97%	**7**
as	+1%	32%	98%	**3**
unzip	+23%	84%	98%	**3**
Average	+7%	56%	97%	**6**

Table 2: Average number of total iterations in the four 120-hour experiments.

	afl++	**arvin_pr**	**arvin**
strip	644M	48M	74M
objdump	503M	32M	69M
readelf	683M	87M	78M
libpng	937M	11M	15M
Average	691M	44M	59M

the instrumentation aims at maximizing the CFG precision and enhancing the accuracy of the benchmarks.

4.1 Coverage Growth

In this test, we chose AFL++ (given its impressive coverage) and compared it with Arvin when both are given the same amount of time to run in the same fuzzing configurations. Arvin fuzzes in binary mode, while AFL++ is used in open-source mode to achieve its maximum speed and coverage.[2] The results are shown in Table 1. Each program was fuzzed 3 times for 48 hours and the numbers reported are the averages. In Table 1, *Coverage Delta* is the difference in the basic block coverage reached by each fuzzer. A positive number shows a better coverage in Arvin; for all PUTs, Arvin had better coverage, ranging from 1% to 23%. *Arvin Time* and *AFL++ Time* show the portion of the fuzzing time that each fuzzer used to reach the maximum *common* coverage. For example, if one fuzzer finds 2,000 basic blocks while the other finds 1,800 in the same PUT, these columns show the time it took each fuzzer to find 1,800 basic blocks. The lower this number the better, as it shows the fuzzer can increase coverage faster.

Coverage Ratio looks at how efficient each fuzzer is in terms of *iterations*. We define the coverage score (s) as the total number of covered basic blocks (b) divided by the total number of iterations (i) executed in a given fuzzing session ($s = \frac{b}{i}$). A higher coverage score shows the fuzzer can increase coverage *within fewer number of iterations*. In all cases, Arvin's coverage score is much higher than AFL++'s, so we report the Coverage Ratio, which is simply Arvin's coverage score divided by AFL++'s. On average, Arvin needs only 0.15 (or $\frac{1}{6}$) times the number of iterations of AFL++ to reach the same basic block coverage in 41% shorter fuzzing time; Arvin is

[2]AFL++ does have a binary-only qemu mode, but it is much slower; using source mode gives AFL++ the biggest possible advantage.

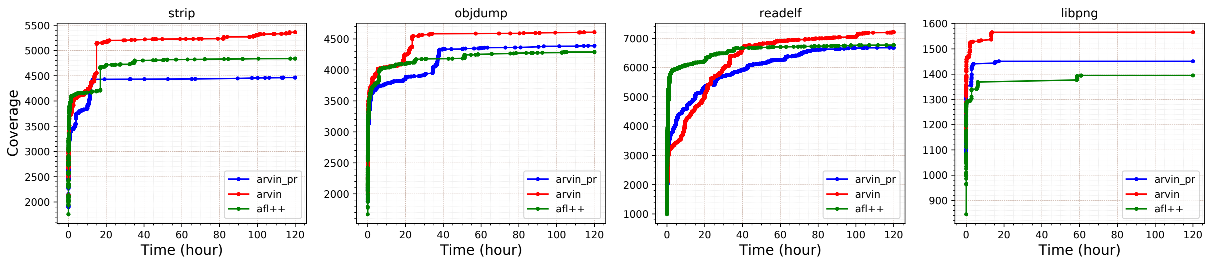


Figure 8: Coverage growth of Arvin and AFL++ over the course of 120 hours for four different programs.

more efficient than AFL++ in terms of both total fuzzing time and total number of iterations.

To evaluate the performance of Arvin compared to AFL++ over long time periods, we also ran four 120-hour sessions. Graphs of coverage growth (in terms of basic blocks) during these sessions can be seen in Figure 8. In all cases, Arvin finishes the session with higher coverage than AFL++, though in the `readelf` case, AFL++ does temporarily outpace Arvin at the beginning of the session. Table 2 shows the total number of iterations during these sessions. While Arvin, on average, has more iterations than Arvin_pr (about 15M more), it has significantly fewer iterations compared to AFL++ (about 632M less) while reaching better coverage than both. Arvin's superior coverage is, in most part, explained by the effect of directed approximation. By excluding less important nodes, we save processing time to find new nodes.

4.2 Fuzzing Depth

Our next test looks at Arvin's ability to achieve coverage that goes deep (i.e., further from the root of the DCFG at the program's entry). We fuzzed *readelf*, *addr2line*, and the *GNU assembler* five times with two different settings (30 total fuzzing sessions). As a baseline, we use *TNS* mode which approximates AFL's input-selection strategy, ignoring depth[3]. We compare this to Arvin's *TDF* mode, which *does* prioritize depth when selecting inputs to mutate.

The result of this comparison is shown in Figure 9. Since our goal is to measure how much Arvin increases the depth of generated inputs, we use the depth of the initial input as the starting point: for each PUT, we observed the depth of the PUT for the initial seed and subtracted it from the depth of all generated inputs in both fuzzing modes. This result demonstrates the effectiveness of our proposed technique to reach deeper parts of the target. In depth-sensitive TDF mode, Arvin reaches on average 83% greater depth than the depth-oblivious TNS mode.

4.3 Targeted Fuzzing

Arvin is also able to focus on specific functions or code parts such as known unsafe functions. The user can have the fuzzer *mark* these functions so that if an input reaches them, the library records this in the graph to signal the fuzzer to increase the priority of the input. This feature is called "targeted fuzzing". To test it, we fuzzed the *file* program with different inputs and marked one specific function[4]

[3]We cannot do this comparison with AFL itself, as AFL does not construct a DCFG and therefore does not collect enough information to compute depth.
[4]The marked function was `json_isxdigit()` defined in `src/is_json.c`

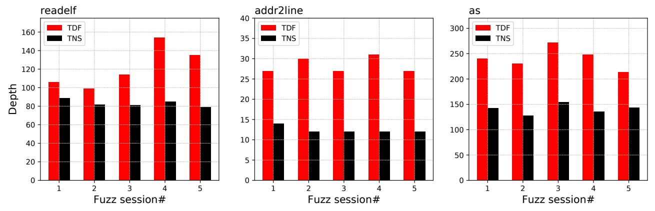


Figure 9: Depth reached by depth-sensitive TDF mode and depth-oblivious TNS mode.

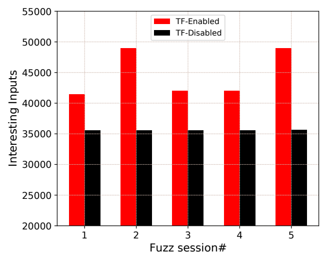


Figure 10: The number of generated inputs that reach a marked function with and without targeted fuzzing.

as an important function in the PUT. Only one of the initial seeds reached the marked function. We ran Arvin 10 times to fuzz *file*: 5 times each with the targeted fuzzing feature disabled and enabled. Figure 10 shows the number of the total inputs that reached the marked function in each fuzzing session. In all of the sessions that targeted fuzzing was enabled, Arvin could successfully generate more inputs that ended up calling the marked function in the fuzzed program, effectively increasing the chance of a more aggressive evaluation of the chosen important function.

4.4 DCG and Fuzzing Speed

DCG approximates graphs, diminishing their size, and at the same time, it reduces the number of transitions between the fuzzer and the PUT. This yields a significant speed improvement. It decreases graph analysis time and moderates the cost of using "ptrace", which is known to be slow in Linux kernel [1].

4.4.1 DCG Hit Threshold (DHT). DHT controls the exclusion of nodes from the graph. To show its effect on fuzzing speed, and to select a value to use in practice, we evaluated ten different DHT levels on five different programs. These levels from L_1 to L_{10} map

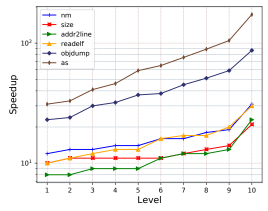


Figure 11: Log-scale plot of the effect of DCG Hit Threshold on fuzzing speed of 6 different programs.

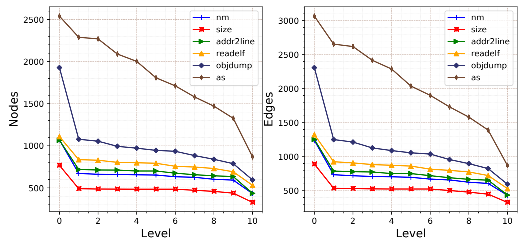


Figure 12: The effect of DCG Hit Threshold on graph size in terms of number of nodes (left) and edges (right).

to numbers 1 to 10 in the reverse order. ($L_1 = 10, L_2 = 9 \ldots L_{10} = 1$). This means a higher level ends up excluding more nodes. For each program, we ran Arvin 10 times, each time with one of the defined levels for its DHT (50 total runs). In each run, we calculated the speed improvement based on the first few iterations. The reason is that the effect of DHT is normally best seen in the first iterations of mutating any seed where the majority of repetitive basic blocks are identified by the DCFG library. The speed improvement in this experiment at each level is the reverse of dividing the execution time of one iteration before and one iteration after the DCG completes the identification and exclusion phase of the tried seed. This number is usually the upper bound for relative speed improvement for that individual seed at the evaluated DHT level, throughout the whole fuzzing session for any two iterations affected by DCG. The result of this test is shown in Figure 11. The worst case is for `addr2line` with an speedup of 8 for L_1 and the best case is a speedup of 173 for `as` at L_{10}. This means that at the worst-case scenario in these experiments, DCG improved speed 8 fold. While the speed in the worst case is increased 8-fold, the graph size reduction in most tests is between 60% to 70%. This shows that DCG correctly targets expensive nodes for exclusion; on average, for each 1% reduction in the graph size, DCG results in a 45% speedup. The effect of DHT levels on the size of an average graph has been shown in two different plots in Figure 12.

For most programs, the low and middle levels (L_1 and L_5) have a relatively high number of discovered paths. This is exactly because the way DCG is supposed to work; we reduce the number of evaluations of different execution traces (paths in this context) by excluding specific basic blocks from the analysis. We have added the detailed graphs of DCG effects in Appendix B. For the rest of the tests in our evaluation, we have used L_{10}.

Also, our partitioning strategy (explained in Section 2.2), reduces the total time needed to finish a cycle on average by 40% while in some cases, even one cycle is not finished after the session ends if we disable partitioning.

As a conclusion, Arvin generally performs better in longer fuzzing sessions and for bigger programs.

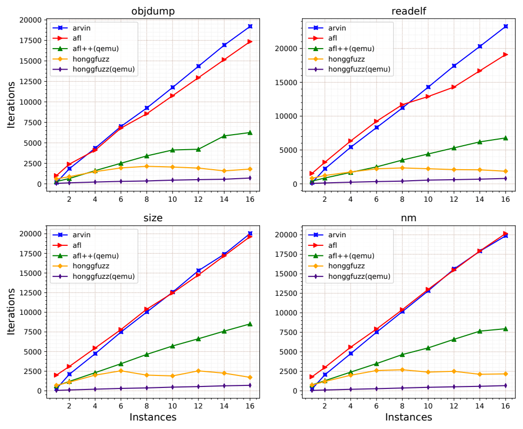


Figure 13: Fuzzing speed versus the number of fuzzing instances for 4 different programs and 4 different fuzzers run in total 5 different sessions.

4.5 Parallel Fuzzing Mode

The fast parallel mode in Arvin effectively makes up for the performance cost of using `ptrace`. In this subsection, we present the result of comparing fuzzing speed in terms of the number of iterations per second between Arvin and other fuzzers as shown in Figure 13. For this experiment, we chose 4 programs and fuzzed them with three fuzzers. honggfuzz and AFL have been used to fuzz in open-source mode and AFL++-qemu and honggfuzz-qemu for binary-only mode. The number of instances is the total number of fuzzing units for each specific fuzzer, including the Master instance in Arvin. For example, in Arvin, an instance number of 8 means one master instance plus 7 parallel instances (this is why for the single-instance case, Arvin is slow). The result is, except `nm`, Arvin is faster than other tested fuzzers. And for `nm` the speed difference between Arvin and the fastest fuzzer (AFL) is negligible.

As is shown in the figure, while different tests show a monotonic speed growth by increasing the number of instances, honggfuzz results in some inconsistencies. AFL is almost as fast as Arvin, however, the speed of AFL is for *open-source* mode. In binary-only mode, AFL++-qemu is significantly slower. Honggfuzz-qemu also has an almost linear speed growth by increasing the number of its parallel threads, however, it is slower than other tested fuzzers in both modes. These tests show Arvin has, in general, better performance in parallel mode than the other three popular evaluated fuzzers.

4.6 Discovered Bugs

Table 3 summarizes the previously-unreported bugs found by Arvin and seven other fuzzers. All the bugs found by other fuzzers, were also found by Arvin. We used the latest version of each program in this table and fuzzed them for 12 hours separately with each reported fuzzer. In each session, for the fuzzer under test, we ran

Table 3: Bugs found using different tested fuzzers. honggfuzz (hf) and AFL++ (APP) abbreviated to save space.

Program	Arvin	AFL	hf	hf-qemu	APP	APP-qemu	TortoiseFuzz	Angora*	VUzzer	ParmeSan
GNU assembler (v 2.35, 2.36, 2.37)	13	0	4	1	5	1	3	0	0	0
unzip (v 6.00)	4	2	2	0	3	1	0	0	0	0
gif2png (v 2.5.8-1)	2	2	2	2	2	2	2	2	0	2
readelf (v 2.35)	1	0	0	0	0	0	0	0	0	0
bison (v 3.5, 3.7)	7	3	4	2	4	4	3	1	0	1
fig2dev (v 3.2.8a)	3	1	2	1	2	2	2	1	0	0
flvmeta (v 1.2.2)	1	0	1	0	1	0	0	0	0	0
nasm (v 2.15.05)	1	0	0	0	0	0	0	0	0	0
ibmtpm (last commit May 15 2021)	1	0	0	0	0	0	0	0	0	0
libraw (v 0.21.0)	2	0	0	0	1	0	0	0	0	0
pnmtopng (v 1.17.dfsg-4)	2	1	1	1	2	1	1	0	0	0
pnmtojpeg (v 1.17.dfsg-4)	2	0	2	2	1	1	1	0	0	0
pnmtofiasco (v 1.17.dfsg-4)	2	2	2	2	2	2	2	2	0	2
pstopnm (v 1.17.dfsg-4)	3	0	2	1	1	1	1	0	0	0
psnup (v 1.17.dfsg-4)	2	0	1	1	2	1	1	0	0	0
xpdf (v 4.03)	1	0	0	0	0	0	0	0	0	0
xvid (last version from http://svn.xvid.org/trunk)	3	0	0	0	1	0	0	0	0	0
Total	50	11	23	13	27	16	16	6	0	5

* Angora failed to fuzz unzip with this error message: "There is none constraint in the seeds". Angora also failed to compile binutils-gnu 2.36 in taint mode.

four parallel instances. We used a small set of simple random seeds with an average size of 4KB. Arvin, AFL++-qemu and honggfuzz-qemu fuzzed in binary mode, and the other fuzzers instrumented the target's source code. We repeated the tests five times. Since VUzzer and ParmeSan were extremely slow (on average more than 230x and 50x slower than Arvin respectively), we reran the same fuzzing sessions for both these fuzzers and let them run for 48 hours (4 times longer than Arvin) with exactly the same results.

Arvin found thousands of crashes. Obviously, each crash does not belong to a distinct new bug. Arvin performs a fast hash calculation over every CFG to report whether a similar crash has happened more than once. However it is possible that different inputs that trigger the same bug have different execution paths and hence different CFGs. To find the number of unique bugs we categorized the crash reports based on the fuzzed program and the proximity of reporting times, then chose a couple of sample inputs from each category. We identified a lower bound of 50 distinct-zero day bugs discovered by Arvin[5]. As reported in Table 3, this is 23 bugs more than the next best fuzzer, AFL++, and 27 more bugs than honggfuzz.

4.6.1 *Discussion of the effects of design decisions on finding bugs.*

So far, we demonstrated how our prioritization policies increase code coverage which generally translates into finding more bugs. To directly study how Arvin's prioritization affects finding bugs, we completely disabled prioritization in Arvin. Then we repeated the bug-finding sessions three times in this mode and recorded the average number of bugs that were found. Arvin only detected between 20% to 33% of the same bugs reported in Table 3, depending on the size of the PUT and, consequently, the number of generated inputs in the queue. We observed the poorest result for GNU assembler. The effect of prioritization is more significant on larger PUTs. On average, only 22.5% of all the bugs were discovered.

[5]We have reported the bugs to the software maintainers.

To evaluate the direct effect of DCG and targeted fuzzing, we repeated the bug-finding sessions three times for each test. First, we executed the sessions in arvin_pr mode which does not use DCG. In this mode we were able to find, on average 44% of the bugs. For the second test, we used three of the bugs that we had previously found in GNU assembler and unzip[6] For the reported bugs, we marked the responsible functions in which the bug existed. For GNU assembler, the functions were `tc_gen_reloc()` and `elf_copy_symbol_attributes()` defined in `gas/config/tc-i386.c` and `gas/config/obj-elf.c` respectively. For unzip, the marked function was `BZ2_decompress()` defined in `bzip2/bzlib.c`. On average, the total number of correct inputs to trigger any of the three bugs increased by 20.6% and the time needed to find the first input to trigger the related bug decreased by 13.8%.

5 RELATED WORK

The major contribution of prior work in greybox fuzzing has been to improve mutation or prioritization and, in most cases, a combination of both. Prioritization happens at the *individual-input* and *input-set* levels. The former discusses better ways of finding *promising parts* of inputs while the latter focuses on selecting *better inputs* from the set of *mutated inputs*. At either level, the information the fuzzer gains from the PUT—statically or dynamically—is an important factor that determines to what extent a greybox fuzzer can successfully perform prioritization. The more information the fuzzer has about the internals, structure, and run-time context of the PUT, the more metrics it has to prioritize inputs.

Correlating different parts of inputs to PUT's functionality or CPU registers using techniques like memory taint analysis have been used by various state-of-the-art fuzzers. REDQUEEN [6] for

[6]Note that targeted fuzzing was not originally used to discover the reported zero-day bugs.

example, uses the QEMU emulator to identify the compare instructions and tries to match input bytes with the operands of the identified instructions to satisfy branch conditions to increase coverage. Steelix [18] uses IDA and Dyninst framework and VUzzer [23] uses Intel Pin framework [4] to reach a similar goal, finding the correlation between different parts of the input and operands of compare instructions to resolve magic bytes. These strategies are quite different from, but complementary to, the one used by Arvin: Arvin instead focuses on selecting and mutating *entire inputs* rather than individual bytes within them.

Full speed fuzzing [21] considers removing instrumentation from basic blocks to increase speed. Although it uses a relatively similar idea to how we identify expensive basic blocks, their goal is entirely different from what we do in Arvin. Arvin aims to perform DCFG analysis to gauge the contextual value of the inputs and then use approximation to simplify the graphs. The way they modify instrumentation and consequently trace the PUT is also completely different from Arvin. In Full speed fuzzing they have two separate binaries of the PUT, the oracle and the tracer. When a new basic block is found in the oracle, the oracle is terminated, and the tracer is executed to find the new basic block(s). This cycle is repeated to gradually reduce the number of instrumented basic blocks in the oracle. In Arvin, we do not stop the PUT upon finding a new basic block because we need to acquire the run-time context information from the PUT's control flow. The fuzzer keeps monitoring the complete execution of the PUT to construct its DCFG, which is then used by the fuzzing engine for input prioritization.

ParmeSan [22] makes dynamic CFG by heavily using external data-flow analysis. In contrast to Arvin, ParmeSan builds an initial graph and gradually adds edges to the structure upon visiting them during fuzzing. The CFG is then used to calculate distances between different parts or *targets* of the program under fuzz. The way ParmeSan uses and builds the dynamic CFG is fundamentally different from Arvin; ParmeSan uses data-flow analysis to make a CFG that contains all possible traces using different inputs observed during fuzzing. Arvin however, natively constructs a new CFG for each iteration that demonstrates the exact execution trace for that individual execution to be directly compared to other inputs.

SAVIOR [11] prioritizes inputs based on static analysis of the PUT and predicting potentially vulnerable locations in code then labeling them to be identified during fuzzing. TortoiseFuzz [29] prioritizes inputs based on the potential execution of some known security-sensitive functions and code structures (e.g loops) in the PUT. Although these techniques are fundamentally different from our approach, they are two other examples that show the importance of input prioritization in fuzzing. Input selection in general has been discussed in multiple other works [17, 24, 30].

To the best of our knowledge, Arvin is the first greybox fuzzer to natively build an approximate DCFG in each iteration for effective input prioritization.

6 CONCLUSION

Input prioritization is essential to improve fuzzing effectiveness but challenging to achieve. Fuzzers usually fail to accurately estimate the quality of a mutated input due to a lack of sufficient context information. We showed how to leverage DCFG run-time information for effective input prioritization and at the same time use approximation to reduce the cost of the analysis without incurring significant negative consequences of precision loss. We implemented Arvin and demonstrated how a partial dynamic CFG enables rich but sufficiently cheap run-time context information to prioritize inputs, and we showed how effective our approach could be in attaining better coverage with fewer iterations. Arvin is open-source [5] and outperforms different popular state-of-the-art fuzzers in terms of coverage, speed, and finding bugs.

REFERENCES

[1] 2010. Replacing ptrace(). https://lwn.net/Articles/371501.
[2] 2016. The Cyber Grand Challenge. https://blogs.grammatech.com/the-cyber-grand-challenge.
[3] 2022. American Fuzzy Lop. https://lcamtuf.coredump.cx/afl/.
[4] 2022. Pin. https://software.intel.com/content/www/us/en/develop/articles/pin-a-dynamic-binary-instrumentation-tool.html.
[5] 2023. Arvin. https://github.com/0xsirus/arvin.
[6] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *Symposium on Network and Distributed System Security (NDSS)*.
[7] Boris Beizer. 1995. *Black-Box Testing: Techniques for Functional Testing of Software and Systems*. Wiley.
[8] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-Based Greybox Fuzzing as Markov Chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*.
[9] Ella Bounimova, Patrice Godefroid, and David Molnar. 2012. Billions and Billions of Constraints: Whitebox Fuzz Testing in Production. https://www.microsoft.com/en-us/research/publication/billions-and-billions-of-constraints-whitebox-fuzz-testing-in-production/.
[10] Peng Chen, Hao Chen, and Jianzhong Liu. 2018. Angora: Efficient Fuzzing by Principled Search. In *39th IEEE Symposium on Security and Privacy*.
[11] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Tao Wei, and Long Lu. 2020. Savior: Towards bug-driven hybrid testing. In *2020 IEEE SP*. IEEE.
[12] Jared D. DeMott, Richard J. Enbody, and William F. Punch. 2007. Revolutionizing the field of grey-box attack surface testing with evolutionary fuzzing. In *Black hat USA*.
[13] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *Proceedings of the USENIX ATC*.
[14] S. Embleton, S. Sparks, and R. Cunningham. 2006. "Sidewinder": An Evolutionary Guidance System for Malicious Input Crafting. In *Black hat USA*.
[15] Google, Inc. 2022. HonggFuzz. https://github.com/google/honggfuzz.
[16] Google, Inc. 2022. HonggFuzz Trophies. https://honggfuzz.dev/#trophies.
[17] Adrian Herrera, Hendra Gunadi, Shane Magrath, Michael Norrish, Mathias Payer, and Antony L. Hosking. 2021. Seed Selection for Successful Fuzzing. In *Proceedings of the 30th ACM SIGSOFT ISSTA*.
[18] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. 2017. Steelix: Program-State Based Binary Fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*.
[19] Valentin J.M. Manes, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. 2019. The Art, Science, and Engineering of Fuzzing: A Survey. In *ACM Computing Surveys*.
[20] G. J. Myers, C. Sandler, and T. Badgett. 2011. *The Art of Software Testing*. Wiley.
[21] Stefan Nagy and Matthew Hicks. 2019. Full-Speed Fuzzing: Reducing Fuzzing Overhead through Coverage-Guided Tracing. In *2019 IEEE SP*.
[22] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2020. ParmeSan: Sanitizer-guided Greybox Fuzzing. In *USENIX Security*.
[23] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In *NDSS*.
[24] Alexander Rebert, Cha Sang Li, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. 2014. Optimizing Seed Selection for Fuzzing. In *Proceedings USENIX ATC*.
[25] Kostya Serebryany. 2017. OSS-Fuzz - Google's continuous fuzzing service for open source software. USENIX Association.
[26] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. 2016. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *2016 SP*.
[27] Nick Stephens, John Grosen, Christopher Salls, Audrey Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna.

2016. Driller: AugmentingFuzzing Through Selective Symbolic Execution. In *NDSS*.
[28] Ari Takanen and Charlie Miller. 2008. *Fuzzing for Software Security Testing and Quality Assurance.* Artech House.
[29] Yanhao Wang, Xiangkun Jia, Yuwei Liu, Kyle Zeng, Tiffany Bao, Dinghao Wu, and Purui Su. 2020. Not All Coverage Measurements Are Equal: Fuzzing by Coverage Accounting for Input Prioritization. In *NDSS*.
[30] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. 2013. Scheduling Black-Box Mutational Fuzzing. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security* (Berlin, Germany).
[31] Mingyuan Wu, Ling Jiang, Jiahong Xiang, Yanwei Huang, Heming Cui, Lingming Zhang, and Yuqun Zhang. 2022. One Fuzzing Strategy to Rule Them All. In *Proceedings of the 44th International Conference on Software Engineering*.

A MUTATION MODERATION AND CFG SET PRUNING

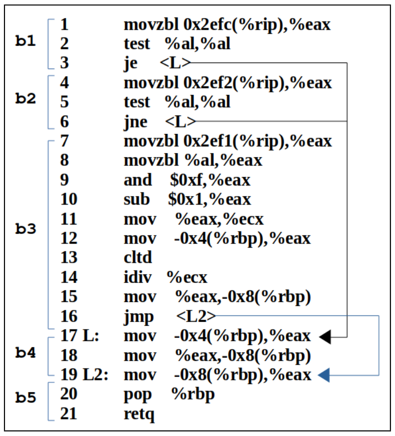
This is an example of how Arvin moderates the usage of mutation functions and why it is important.

While having an optimal order for inputs is helpful, the fuzzer can still perform inefficiently if a poor mutation policy is practiced. We need to *avoid the generation* of new inputs that are unlikely to exercise new execution paths. Toward this end, we designed a *Performance Check* in Arvin to dynamically change the mutation pattern of inputs based on the context information we gathered from previous CFGs. For this check, "performance" means the time it takes to evaluate the input queue, which for the most part consists of mutated inputs: the goal is to progress through the queue more quickly. Because each input can take from a few minutes to a few hours to evaluate, having an effective, accurate, and technically sound policy to avoid low-potential mutations has a dramatic effect on the performance. Different parts of an input have different roles and meanings to the PUT; as a result, the mutation in different parts have different potentials to generate better execution paths in terms of coverage and depth. Since we analyze all CFGs (even those that have been created due to mutation of less effective parts of an input), unnecessary trials, significantly impact the fuzzing speed because cycles are wasted if the mutated part exercises previously visited blocks and no crash in the PUT happens.

```
if (input[0x248] && !input[0x249]){
  result=num/(unsigned char)((input[0x248] & 0x0F)-1);
}else{
  result=num;
}
```

Listing 1: Example code snippet of a program

We defined three different performance-check modes (MD, ME1 and ME2), which specify how aggressively the deterministic mutation functions (like iterative bitflipping functions) should be carried out for each input. The more the mutation functions are used for each input, the more the fuzzer has to spend time on each single input. MD makes the highest use of the deterministic functions and ME1 and ME2 spend less time on some mutation functions and speed up the evaluation of each input with ME2 limiting per-input evaluation quota more aggressively. Both ME1 and ME2 try to change the usage pattern of mutation functions dynamically to reduce the overhead of deterministic checks and evaluate a bigger number of inputs within a specific period of time with ME2 being more restrictive in choosing mutation functions. ME1 and ME2 are reserved for faster coverage growth and MD is more focused on finding bugs. The logic is that there are potentially multiple (in some cases an unlimited number of) different inputs that may lead to the execution of a specific set of basic blocks. When Arvin works in restrictive performance mode (i.e ME1 or ME2), it gives a higher priority to finding new basic blocks rather than crashing the PUT. Although finding the correct set of basic blocks that take part in a crash in the PUT is *necessary*, it is not enough. This is why in the restrictive modes, we have better coverage but poorer crash-finding ability. We can work around this inconsistency by performing automatic multi-pass fuzzing.

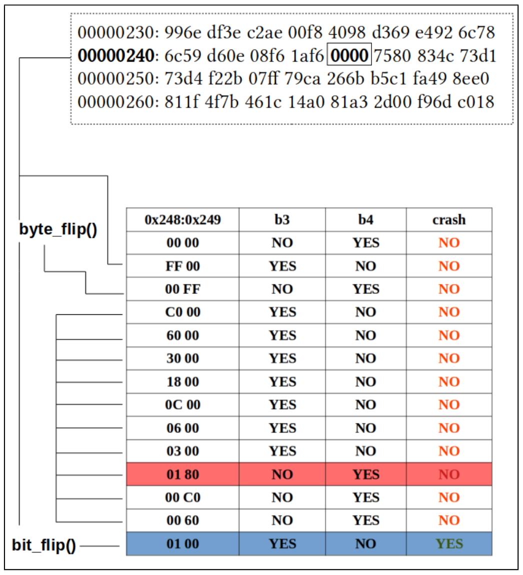


```
b1   1        movzbl 0x2efc(%rip),%eax
     2        test   %al,%al
     3        je     <L>
b2   4        movzbl 0x2ef2(%rip),%eax
     5        test   %al,%al
     6        jne    <L>
b3   7        movzbl 0x2ef1(%rip),%eax
     8        movzbl %al,%eax
     9        and    $0xf,%eax
     10       sub    $0x1,%eax
     11       mov    %eax,%ecx
     12       mov    -0x4(%rbp),%eax
     13       cltd
     14       idiv   %ecx
     15       mov    %eax,-0x8(%rbp)
     16       jmp    <L2>
b4   17 L:    mov    -0x4(%rbp),%eax
     18       mov    %eax,-0x8(%rbp)
     19 L2:   mov    -0x8(%rbp),%eax
b5   20       pop    %rbp
     21       retq
```

Figure 14: Assembly equivalent of the code above with involded basic blocks

We illustrate the aforementioned natural inconsistency issue and our solution using an example. In Listing 1, a code sample with an obvious *divide-by-zero* bug, has been shown. There is a conditional check in the first line which translates into multiple branch instructions in the compiled binary. The body of the `if-else` block is compiled into two basic blocks shown as b3 and b4 in Figure 14. The conditional check reads the input at offsets `0x248` and `0x249` and checks their values. If `input[0x249]` is zero and `input[0x248]` is non-zero, it divides *num* (`[%rbp - 9]`) by the *right half of the value minus one* and stores the result in the *result* variable (`[%rbp - 8]`). Any value like `0xn1` in `input[0x248]` where n is an arbitrary hex digit, can crash this program if `input[0x249]==0`. An example seed input has been shown in Figure 15. Initially `input[0x248]` and `input[0x249]` are both zero.

Suppose that in a deterministic mutation function, Arvin flips the byte in `input[0x248]` and observes that basic block b3 is executed. Suppose that the next mutation function flips two neighboring bits at a time for each iteration until all adjacent two-bit groups are tested. The result of this second mutation function for all the two-bit groups in `input[0x248]` have been shown in Figure 15. None of them can crash the program because when the mutation generates `0x01` for `input[0x248]`, it also sets the left-most bits of `input[0x249]` to 1 (and hence the whole byte to `0x80`) which violates the `if` condition to enter b3. The result is that Arvin observes that no new basic block is visited after trying all two-bit tests for `data[0x248]`. This is when the performance check kicks in and limits the use of the next deterministic functions for ME1 and ME2



```
00000230: 996e df3e c2ae 00f8 4098 d369 e492 6c78
00000240: 6c59 d60e 08f6 1af6 0000 7580 834c 73d1
00000250: 73d4 f22b 07ff 79ca 266b b5c1 fa49 8ee0
00000260: 811f 4f7b 461c 14a0 81a3 2d00 f96d c018
```

0x248:0x249	b3	b4	crash
00 00	NO	YES	NO
FF 00	YES	NO	NO
00 FF	NO	YES	NO
C0 00	YES	NO	NO
60 00	YES	NO	NO
30 00	YES	NO	NO
18 00	YES	NO	NO
0C 00	YES	NO	NO
06 00	YES	NO	NO
03 00	YES	NO	NO
01 80	NO	YES	NO
00 C0	NO	YES	NO
00 60	NO	YES	NO
01 00	YES	NO	YES

Figure 15: Example input for the code in Figure 14 showing the values of bytes `0x248` and `0x249` along with the respective status of execution of `b1` and `b2` and whether or not it crashes. When performance check is active in restrictive mode (`ME1` or `ME2`), the fuzzer disables the `bit_flip` function for bytes `0x248` and `0x249` and the correct deterministic check to trigger the bug is missed. On the other hand in performance check mode zero, the bug is triggered and the fuzzer detects the crash.

and then another mutation function *single-bit-flip* which is able to crash the program gets disabled to improve speed. However in `MD`, this restriction is not applied and *single-bit-flip* functions can find the correct input by flipping the right-most bit in `data[0x248]` (without touching `input[0x249]`) and crash the program. But it also evaluates many other tests for the remaining bits, which may not either increase the coverage or trigger a crash.

B DCG COMPLEMENTARY GRAPHS

In this part, we have reported the visual results of the effect of DCG on coverage growth and path discovery of various programs.

In Figure 16 we have tested Arvin's path discovery at three levels (L_1, L_5 and L_{10}) for four programs. It should be noted that the reported number of paths is the lower bound of actual paths executed because of the approximation used in the fuzzing library. In the test, we wanted to evaluate the effect of DHT levels on path discovery and coverage. For bigger programs (the assembler in this test) the difference between lower and higher levels is more significant due to the big speed difference between the levels. In all tests, lower levels tend to spend more time on finding new paths *whose majority of basic blocks have been already visited in previous iterations* while higher levels tend to find paths *with new basic blocks*. This can be directly inferred from looking at Figure 17 which shows the number of basic blocks covered in the same configurations as in Figure 16. Generally, higher levels have better coverage Figure 17. Interestingly, for `size`, while L_1 has the best path discovery, it got the poorest basic block coverage among the three tested levels.

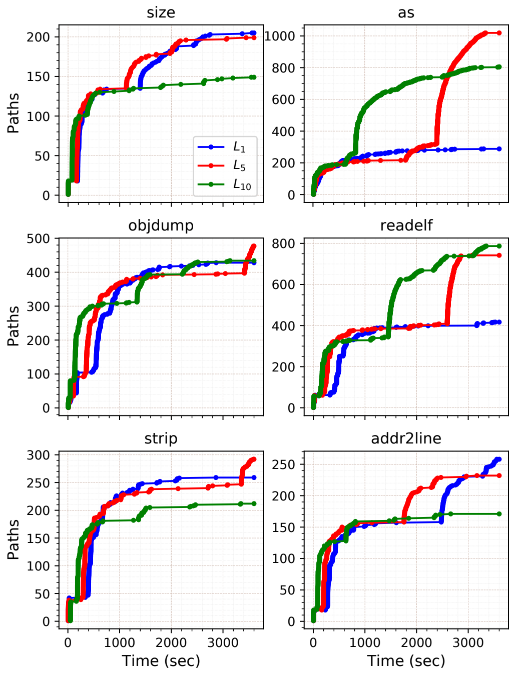


Figure 16: The effect of DHT on path discovery.

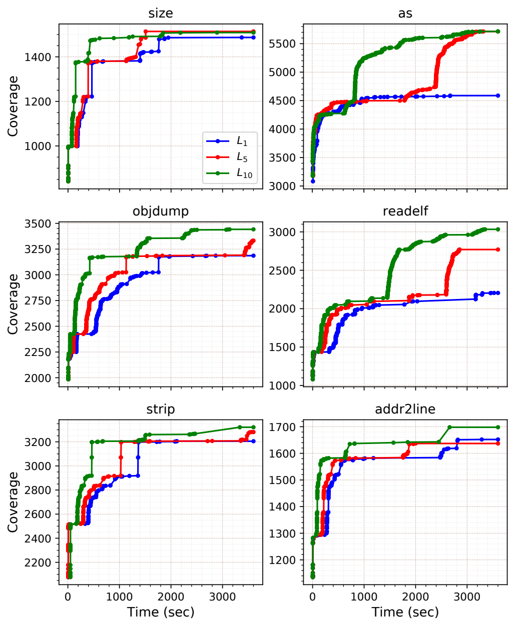


Figure 17: The effect of DHT on coverage growth.

As mentioned before, DCG shows better results as time passes. In Figure 18 we repeated the same test for `as` and `size` in a longer session. Although L_5 and L_{10} have similar coverage in the shorter session shown in Figure 17, L_{10} has obviously better coverage for `size` in the second test. More tests showed that `as` reaches a local saturation point relatively early when it is fuzzed by Arvin in L_5

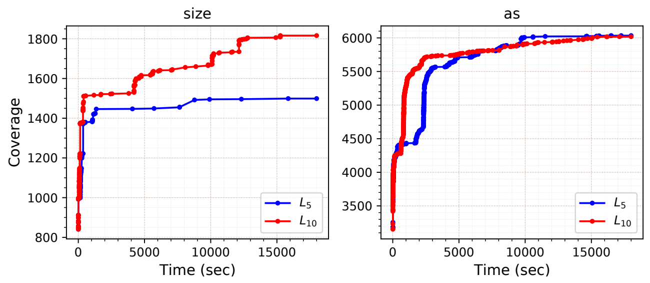


Figure 18: The effect of DCG Hit Threshold on coverage growth of size and as in a 5-hour session.

and L_{10}. This saturation point lasts for many hours and this is why both levels show similar coverage for this program.

C DCG CODE EXAMPLE

In Listing 2, a sample sanitization code from `readelf` is shown that checks the value of `e_phnum` from the function `get_program_headers`.

```
if (filedata->file_header.e_phnum
  * (is_32bit_elf ? sizeof (Elf32_External_Phdr) :
    sizeof (Elf64_External_Phdr))
  >= filedata->file_size)
{
  error (_(...
  return FALSE;
}
```

Listing 2: Example header sanitization in readelf

In another function `get_64bit_program_headers` we have the following loop:

```
for (i = 0, internal = pheaders, external = phdrs;
  i < filedata->file_header.e_phnum;
  i++, internal++, external++)
{
  internal->p_type= BYTE_GET (external->p_type);
  ...
  internal->p_align= BYTE_GET (external->p_align);
}
```

Listing 3: Example loop in readelf

In Listing 3, the program updates two variables in each loop iteration. The loop variable itself `i` is protected against overflow by checking the value of `e_phnum` which consequently prevents the two linked pointers `internal` and `external` from exceeding their expected boundaries. For a small 20KB ELF file as the input of `readelf`, the maximum allowed value for `e_phnum`, is $20 * 1024/sizeof(Elf64\_Externalphdr) = 365$. This means that the initial basic block of this loop alone will produce 365 different paths, and this is just one of several similar loops in `readelf.c`. Multiplying the number of additional paths in each loop by the number of paths discovered from other parts of the program will result in hundreds of thousands of paths that can be generated just by tweaking the `e_phnum` in the input header. Each of these paths incurs graph analysis cost without triggering a crash in the PUT or increasing coverage. This is a small example of why DCG is notably effective to moderate the analysis cost.

D LOW-LEVEL DETAILS OF TESTED PROGRAMS

The complexity and time cost of dynamic CFG analysis in Arvin depends on the number of basic blocks and type of basic blocks in the target program. In Table 4 we have reported basic blocks details and the total number of instructions in each tested program.

The are two types of basic blocks that we have defined for Arvin play an important role in our analysis and their relative number also affects fuzzing speed. The size of each control flow graph is directly proportional to the number of *independent* blocks that Arvin identifies. In Figure 19-top we have shown the ratio of nested blocks to independent blocks for each program. When this ratio for each program gets smaller, the dynamic CFGs of the program will grow bigger in Arvin. In Figure 19-bottom, we have shown the ratio of number of basic blocks to total number of instructions in each program. Transition to some of these basic blocks happens due to indirect jumps which Arvin can precisely capture during dynamic tracing of the program under fuzz. Naturally for a bigger executable (i.e an executable with more instructions) we'll have more basic blocks. However, as we see in Figure 19-bottom the relation between these two metrics is significantly different for some programs. This ratio for the majority of programs is on average about 0.25 and gzip and unxz (which uses the same binary as xz) had respectively the highest and lowest ratios in the set of our tested programs.

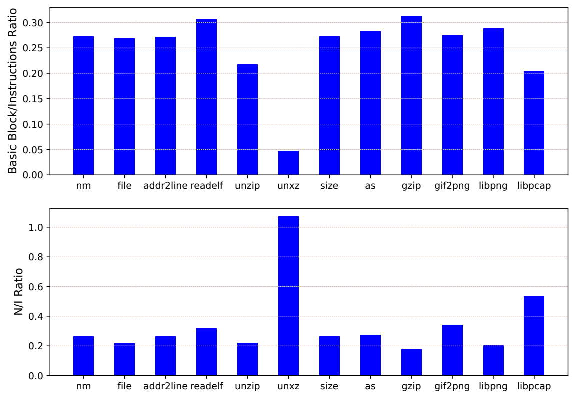


Figure 19: Ratio of basic blocks to all instructions (top) and nested blocks to independent blocks (bottom)

Table 4: Low-level static details of the tested programs

Program	Basic Blocks	Independent Blocks	Nested Blocks	Instructions
nm	51082	40432	10650	187406
objdump	17939	12777	5162	58925
file	7131	5853	1278	26512
addr2line	50436	39937	10499	185350
readelf	35176	26661	8515	115015
unzip	9461	7761	1700	43576
unxz	18481	8914	9567	388009
size	50524	40002	10522	185645
as	72231	56708	15523	255542
gzip	4407	3745	662	14059
gif2png	929	692	237	3383
libpng	8254	6851	1403	28608