Snappy: Efficient Fuzzing with Adaptive and Mutable Snapshots

Elia Geretto
Vrije Universiteit Amsterdam
Amsterdam, the Netherlands
e.geretto@vu.nl

Cristiano Giuffrida
Vrije Universiteit Amsterdam
Amsterdam, the Netherlands
giuffrida@cs.vu.nl

Herbert Bos
Vrije Universiteit Amsterdam
Amsterdam, the Netherlands
herbertb@cs.vu.nl

Erik van der Kouwe
Vrije Universiteit Amsterdam
Amsterdam, the Netherlands
vdkouwe@cs.vu.nl

ABSTRACT

Modern coverage-oriented fuzzers play a crucial role in vulnerability finding. While much research focuses on improving the core fuzzing techniques, some fundamental speed bottlenecks, such as the *redundant computations* incurred by re-executing the target for every input, remain. Prior solutions mitigate the impact of redundant computations by instead fuzzing a program snapshot, such as the one placed by a fork server at the program entry point or generalizations for annotated APIs, drivers, networked servers, etc. Such snapshots are *static* and, as such, cannot adapt to the characteristics of the target and the input, missing opportunities to further reduce redundancy and improve fuzzing speed.

In this paper, we present Snappy, a new approach to speed up fuzzing by aggressively pruning redundant computations with *adaptive and mutable* snapshots. The key ideas are to: (i) push the snapshot as deep in the target execution as possible and also end its execution as early as possible, according to how the target processes the relevant input data (*adaptive placement*); (ii) for each identified placement, cache snapshots across different inputs by patching the snapshot just-in-time with the relevant input data (*mutable restore*). We propose a generic design applicable to both branch-agnostic and branch-guided input mutation operators and demonstrate Snappy on top of Angora (supporting both classes of operators). Our evaluation shows that, while general, Snappy scores gains even compared to a fork server with hand-optimized static placement such as in FuzzBench, for instance obtaining up to ≈1.8x speedups across benchmarks.

CCS CONCEPTS

• **Security and privacy → Software security engineering**; • **Software and its engineering → Software testing and debugging**.

KEYWORDS

fuzzing, checkpointing, dynamic taint analysis, program analysis

This work is licensed under a Creative Commons Attribution International 4.0 License.

ACSAC '22, December 5–9, 2022, Austin, TX, USA
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9759-9/22/12.
https://doi.org/10.1145/3564625.3564639

ACM Reference Format:
Elia Geretto, Cristiano Giuffrida, Herbert Bos, and Erik van der Kouwe. 2022. Snappy: Efficient Fuzzing with Adaptive and Mutable Snapshots. In *Annual Computer Security Applications Conference (ACSAC '22), December 5–9, 2022, Austin, TX, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3564625.3564639

1 INTRODUCTION

All modern greybox fuzzers [37] share the same underlying principle: repeatedly running the same program with different test inputs to trigger bugs. As such, for any fuzzer, executing more test cases per unit of time increases the performance of the fuzzing campaign by either achieving the same coverage faster, saving time and energy, or achieving better coverage given the same resources.

When fuzzing, the time spent for a single execution of a target user program can be roughly divided in: (i) the time spent in the operating system to start the program, (ii) the time spent in userland to run the program, and (iii) the time spent parsing the results in the fuzzer. Speeding up any of these steps (as done in, e.g., [21, 35]) allows one to reduce the execution time and thus increase the speed.

Looking at the userland execution, a common source of overhead is *redundant computation* (i.e., execution that is always the same across inputs). Examples include the dynamic loader and, on larger programs, the initialization of the target itself. To mitigate the loader overhead, AFL [37] introduced snapshots: a fork server taking a reusable snapshot before the program starts executing. This works for all programs, but does not eliminate all the redundant computation. Some other tools, such as AFL++ [14], allow the user to manually move the snapshot further down into the execution. This, however, requires knowledge of the target, manual work, and only allows one to prune redundancy shared among all executions, not subsets of similar executions. Similar *static* snapshot policies have been applied to specific APIs [2], drivers [32], kernels [28], hypervisors [27], servers [29], etc.

In this paper, we propose a performance optimization based on *dynamic* and more extensive snapshot policies to prune much more redundant computation than possible before. Our approach, which we term Snappy, is based on *adaptive* and *mutable* snapshots. The former allows Snappy to automatically push the snapshot placement as deep in the execution as possible and also terminate its execution as early as possible, according to how the target processes the relevant input data. The latter allows Snappy to automatically cache snapshots across different inputs by patching the snapshot just-in-time with the relevant input data.

To develop these ideas, we propose two novel dynamic taint analysis-based [1] techniques catering to both common branch-agnostic (a la AFL [37]) and branch-guided (a la VUzzer [26]) input mutation operators: First Tainted File operation (FTF) and First Tainted Memory read (FTM) snapshots (respectively). FTF automatically places a snapshot when the program first loads any input data, automating the process of pruning redundant computation across all executions in a branch-agnostic fashion. FTM, in turn, automatically places a snapshot when the program loads input data controlling a branch that a branch-guided input mutator seeks to flip to improve coverage. This prunes even redundant computation that is not present in all executions and only shared among subsets of them. Moreover, we can operate an *early exit* optimization by ending the snapshot execution as soon as the target branch is reached. Since FTM, despite potentially producing higher gains, trades off faster executions against more frequent (and thus costly) snapshots, we also developed an algorithm to dynamically switch between FTM and FTF, selecting the most promising technique at any given time while fuzzing.

While our ideas are generally applicable to a broad range of fuzzers, we prototyped Snappy on top of Angora [11], since it combines both branch-agnostic and branch-guided input mutation operators, allowing us to test our system with both. Most other fuzzers (e.g. AFL++ [14], libFuzzer [2] and RedQueen [5]) lack one or the other of these features, making them less suitable for testing FTF and FTM collectively. In addition, we also backported a host of common optimizations present in more recent works to Angora to make it the best baseline possible, given our requirements. We evaluated our approach on FuzzBench [19], a well-known benchmarking suite which already includes hand-optimized (static) snapshot placements to achieve high speeds. Despite such manual optimizations, our experiments showed Snappy can significantly outperform the baseline in 53% of the benchmarks, with an average speedup of 1.2x (and 1.8x in one case). Moreover, we present an evaluation on two real-world programs, `sqlite3` and `objdump`, to better represent the performance of Snappy without manual optimizations. In this scenario, Snappy reported coverage increases of 3% and 31%.

Summarizing:

- We present a generic approach to automatically prune redundant computation while fuzzing by means of *adaptive* and *mutable* snapshots.
- We show our approach can be instantiated in practice with two novel (FTF and FTM) automatic snapshot techniques. Our techniques cover the two common classes of input mutation operators and are applicable to a broad range of fuzzers.
- We present Snappy, an open-source prototype of our design, publicly available at https://github.com/vusec/snappy.
- An evaluation of Snappy to demonstrate the effectiveness of our approach and which types of fuzzing campaigns benefit the most from our speedups.

2 BACKGROUND

Snappy uses dynamic taint analysis (DTA) [1] to automatically place a snapshot and prune redundant execution for different input mutators. We now present background information on snapshots and input mutation for fuzzing.

2.1 Snapshots for fuzzing

To eliminate redundant computation while the fuzzing target starts up, researchers have devised several snapshot mechanisms for recent fuzzers. The goal is to bring the target to a state where it can be executed as quickly as possible.

The general concept of snapshots can be implemented in very different ways, depending on the fuzzer's needs. One major distinction is between fuzzers that are emulation or virtualization-based [27–29, 32] and fuzzers that are process-based [11, 26, 37]. Virtualization is relatively slow [14, 37], but can support privileged and binary-only code more easily. Process-based approaches do not incur emulation overhead, and are typically used to run instrumented user-space programs. Our prototype is process-based, but our design is equally applicable to virtualization-based fuzzing.

Our prototype builds on top of a *fork server*, a common snapshot mechanism for process-based fuzzers popularized by AFL [37] to avoid the cost of the `execve` syscall and instrumentation-related initialization for each run. The idea is to initialize the program once and then clone the initialized process at a *statically* determined program point for each run using the `fork` syscall. This approach effectively relies on the operating system's copy-on-write features to retain the original memory state in the snapshot, but shared state, such as file descriptors, requires special handling. Unlike the traditional (static) model, Snappy's snapshots are *adaptive* and *mutable*, allowing the fork server to be automatically initialized to a state where input data is already in memory and will be adjusted after the snapshot is restored.

2.2 Input mutation for fuzzing

Many modern fuzzers [11, 20, 26] augment the set of general-purpose (or *branch-agnostic*) input mutation operators as offered by fuzzers such as AFL [37] with specialized (or *branch-guided*) input mutation operators based on dynamic taint analysis (DTA). Specifically, DTA is used to link branches in a program under test to bytes in the input, so that the fuzzer can focus its mutation efforts on exactly those bytes to flip the corresponding branches. While solving path constraints to flip branches can also in principle be done with concolic execution, doing so is generally more expensive.

Fuzzers such as Angora [11] implement support for both classes of operators. Branch-agnostic mutation operators do not use DTA and simply modify test cases (inputs) by flipping bits, injecting interesting values, or splicing test cases together. Branch-guided mutation operators, on the other hand, use DTA information to flip particular branches with specialized search strategies. Examples include an operator which explores all the values of a single byte (used when only a single input byte taints the target condition), and a gradient descent operator (which treats the tainted bytes as inputs for a gradient descent algorithm that searches for the right values). DTA information can also be used to track test case sizes, allowing the creation of DTA-based operators that increase or decrease the size of the test case. For each run, Snappy considers the mutation operator currently in use to determine the potential set of inputs for which snapshots can be reused.

It is also possible to reduce the performance impact of DTA by compiling the target program into two different instrumented binaries: a slow one to track taint, and a fast one with only lightweight

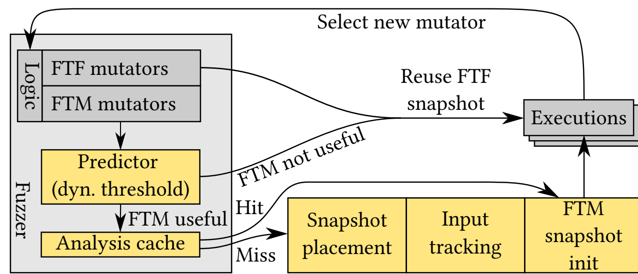

Figure 1: Overview of Snappy, illustrating how its components (in yellow) integrate in the fuzzing logic (in grey).

instrumentation that is used for repeated executions of the target program with different test cases [11]. The fast binary reports to the fuzzer the operands of only one specific target condition, which the fuzzer uses to decide whether it should continue targeting the current branch or switch to a new one. Snappy adopts a similar approach to optimize its DTA-based instrumentation, but uses tainting to find the ideal snapshot position for both classes of input mutation operators, as well as to create mutable snapshots in which part of the test input can be patched just-in-time in memory.

3 OVERVIEW

While fuzzing, we tend to do the same work over and over again. Examples include running the dynamic linker and initializing the target program itself. While the first operation can be trivially eliminated with mechanisms such as the AFL fork server, the latter requires manual work. In addition to these known sources of redundant computation, similar test cases tend to share the initial portion of their execution trace because the program will use the same code regions to handle them—under the assumption common to fuzzers that the execution is mostly deterministic. As an example, all JSON test cases that contain only arrays will visit array-related code, regardless of the size or content of such arrays.

To reduce redundant computation, we first create snapshots of the target program and then, rather than executing the program from the start each time, we recover an appropriate snapshot to skip the redundant part of the execution. To achieve the best possible performance, one needs to take a snapshot at the latest possible moment to skip as much redundant computation as possible. However, it is also important to reuse snapshots as often as possible to reduce snapshot creation overhead. These are conflicting requirements. Any input data processed before taking the snapshot cannot be mutated by the fuzzer, making later snapshots less reusable.

Snappy offers a design for snapshot-based fuzzing that automatically chooses the optimal snapshot placement, allowing the fuzzer to perform more runs per time unit, reducing compute time and power usage needed to reach results. To achieve this, we present two novel techniques and an algorithm that dynamically selects the best one throughout the fuzzing campaign. The first technique, First Tainted File operation snapshots (FTF) places the snapshot at the first read of the content of the test case into memory, automatically pruning redundant program initialization. This is our baseline technique, which Snappy can apply to all the input mutation operators. The second technique, First Tainted Memory read snapshots (FTM), pushes the snapshot even further into the execution, to where the input bytes that are relevant for specific branches are first *used*, also pruning the redundant computation between similar test cases. However, this technique only applies to branch-guided input mutation operators.

As shown in Figure 1, Snappy dynamically selects either FTF or FTM snapshots, depending on the mutation operator. For arbitrary operators, Snappy can rely on FTF to create and then reuse a single, global snapshot at the first I/O operation on the input file. In case the (branch-guided) mutation operator supports FTM snapshots, rather than applying them blindly, Snappy uses a *predictor* to decide if doing so is beneficial. If the predictor confirms that an FTM snapshot is needed, it builds one in three steps, each using a different instrumented version of the program. The first step determines where to *place the snapshot* based on the first tainted memory read identified by DTA. The second step *locates all tainted (input) bytes* that will exist in the memory snapshot we are generating—to allow the fuzzer to modify them easily for each test case using the snapshot. The third step *initializes the snapshot* itself. The first two steps in the process benefit from aggressive caching and Snappy applies the full three-step pipeline only if the results are not cached. Snapshots and caching will be explained in detail in Section 4.

4 DESIGN

To speed up fuzzing, we use snapshots to reduce the amount of code executed redundantly. This section describes the techniques we use to achieve this. We first describe the First Tainted File operation (FTF) policy, which is a lightweight and widely applicable approach using I/O wrappers. Subsequently, we discuss First Tainted Memory read (FTM), which uses more heavyweight dynamic taint analysis to push the snapshot point even deeper into the execution. Finally, we discuss our early exit optimization that reduces execution time by terminating executions that achieved their goals.

4.1 First Tainted File Operation

Fuzzers explore program code by varying the program's inputs, typically provided through an input file. By definition, any code executed prior to operating on this file is input-independent and behaves identically (for deterministic programs) across runs. In other words, if we consider the input to be tainted, a snapshot made on the First Tainted File operation (FTF) can be reused as-is for all runs. This automates a process that is typically performed by hand to prune redundant computation due to program initialization.

Finding the FTF requires only lightweight program analysis and instrumentation: by wrapping all I/O functions, we can detect the first file operation executed by the fuzzer. If the operation obtains information that the fuzzer may change, such as reading the file's contents or querying the file's size, we place the snapshot inside the I/O wrapper, just before the file is read or otherwise queried. For example, when considering Listing 1, this technique would position the snapshot at line 6, inside the wrapper for `fread`.

Using FTF, a single snapshot can be reused for the whole fuzzing campaign, as it happens for (static) snapshots taken before the beginning of the execution of the target program. In addition, it leaves the fuzzer free to use any mutation operator, even one that performs test case size changes, as long as the functions used to perform size

```
1  int main(int argc, char* argv[argc + 1]) {
2    initialization();
3    FILE* file = fopen(argv[1], "rb");
4    ...
5    uint8_t buffer[10];
6    fread(buffer, 1, sizeof(buffer), file);
7    if (buffer[0] != 42) { exit(1); }
8    ...
9    uint8_t buffer2[sizeof(buffer)];
10   memcpy(buffer2, buffer, sizeof(buffer));
11   other_operation();
12   if (buffer2[1] == 10) { abort(); }
13   ...
14 }
```

Listing 1: Example with positions of FTF and FTM snapshots.

checks are also wrapped. To this end, Snappy generalizes the FTF approach to the first tainted I/O operation beyond reads, wrapping functions such as `fstat` (commonly used to perform size checks) and conservatively taking snapshots before the first read as needed.

4.2 First Tainted Memory Read

The fuzzer's input data is often not immediately used after reading it from the input file. It is stored in a buffer, possibly copied around, and only later used in a computation, conditional branch, or as a syscall argument. In addition, after certain mutation operators are used, most of the input data typically remains the same. Therefore, FTF still leaves considerable redundant computation, allowing us to safely delay snapshot creation up to the point where the mutated data is actually used. With FTM, we taint the data that is to be mutated, run the program up to the point where the target tainted data is loaded from memory, and create a snapshot. We can use this snapshot for any run that only mutates the tainted part of the input, as long as we perform the mutation in-memory after restoring the snapshot. This approach skips all redundant computation up to the point where the executions diverge.

Similarity sets. Since divergence in the execution path originates from the differences in the content of each test case, we define a similarity set, thus a set of similar test cases, by selecting a test case size, a property commonly checked in programs, and a set of bytes that can vary given that size. As an example, all test cases that have size 42 and differ only in their 12th byte are a similarity set. It is possible to observe that a similarity set so defined includes test cases that share a common execution prefix because, as long as the (nontainted) data being used is the same, the execution path will also stay the same. For each similarity set, the variable bytes serve as taint sources, and we use dynamic taint analysis to keep track of where these bytes end up in memory. When one of them is loaded into a register to perform a computation, we conservatively take a snapshot before the executions can diverge. Operations such as copies can be ignored because they taint new memory locations, but they do not generate new values using tainted bytes.

Looking now at Listing 1, we define a set of similar test cases that have size 12 and vary only by their second byte. Both these parameters are determined by the mutation operator and the test case selected by the fuzzer. The FTM policy will start tracking the bytes read into `buffer` at line 6. The check at line 7 will have the same result for all test cases in the similarity set and thus will not trigger a snapshot. The copy at line 10 will also not trigger a snapshot because the tainted bytes are copied, but this operation by itself cannot alter the result of the execution. The check at line 12 will finally trigger the snapshot because the program loads a tainted byte to check an equality.

Mutable snapshots. When compared to FTF snapshots, one additional step is required by the FTM policy after the snapshot is placed: since the tainted (mutated) bytes are already in memory when the snapshot is taken, we need to replace them with the ones in the new test case each time the snapshot is restored. This means that, to use this method, simply taking the snapshot is not sufficient, it is also necessary to provide information about the position of all tainted bytes in memory when the snapshot is taken. This strategy allows us to implement the mutable snapshots that we need.

Mutation operators and similarity sets. Our FTM policy requires one snapshot to be taken during the fuzzing campaign for each similarity set. How many snapshots we need depends on how many test cases are considered similar enough to reuse the same snapshot. We need to balance between snapshots deep in the execution, but taken too often, and snapshots taken rarely, but too early in the program. Indeed, snapshot creation is an expensive operation and performing it too often may actually hurt the general performance of the fuzzer. In Snappy, we have chosen to define similarity sets as all the possible mutations the current operator can perform given a specific test case, which determines the test case size. Although more complex aggregations are possible, we favored simplicity and predictability in our current Snappy prototype.

The definition of similarity given before implies a few restrictions on the mutation operators that a fuzzer can use: first of all, they cannot modify the size of a test case; secondly, they can use snapshots more efficiently if they produce test cases that fall within a small similarity set, instead of a large one. Trying to modify all bytes in a test case will produce a very large similarity set and thus a shallow snapshot, while trying to modify only a few will produce a small set and a deep snapshot. In the former case, the FTF policy is likely to provide better performance than FTM. In general, FTF is preferable for mutators that offer no guarantees on the mutations they perform, while FTM is better suited for common branch-guided mutators that have a high degree of locality.

FTM snapshot pipeline. Snappy generates FTM snapshots in three steps, as shown in Figure 1: (1) snapshot placement, (2) input tracking, and (3) snapshot initialization. The snapshot placement step instruments the program with byte-granularity dynamic taint analysis (DTA), tainting the bytes that are variable in the current similarity set, and aborting the first time tainted data is loaded from memory. This yields the ideal (that is, latest possible) snapshot position for the similarity set. The input tracking step also uses byte-granularity DTA, runs up to the snapshot position, and yields the locations of the bytes that need to be mutated when the snapshot is restored. This cannot be done in the first step because the actual snapshot location may be earlier than the ideal snapshot location due to implementation limitations (see Section 5.3 for details). The third step runs the program up to the snapshot location,

to be used as a fork server by the fuzzer. The third binary is built without DTA to ensure it can execute quickly.

Caching. Each snapshot requires three program executions, the first two of which use dynamic taint analysis. As such, creating a snapshot is an expensive operation. SNAPPY uses caching to cut down snapshot cost. To reduce snapshot creation time, we eliminate redundant computation even during snapshot creation, taking snapshots for each of the three binaries. This allows us to perform expensive analyses only once, as long as their results are based only on static properties of the binaries, such as DWARF symbols.

Caching many FTM snapshots for use by the fuzzer is impractical due to the fork server model fuzzers typically use. Instead, we cache analysis results from the first two steps. We reuse a single snapshot when fuzzing within the same similarity set and we quickly construct a new snapshot using the cached analyses when the fuzzer switches to another similarity set. In this way, when restoring a cached snapshot, SNAPPY needs to rerun only the third binary, which is faster because it does not use dynamic taint analysis.

4.3 FTM benefit predictor

The FTM policy is beneficial only when each snapshot is used at least for a certain number of executions. The worst case is that a snapshot is created and then used only for one execution, after which the fuzzer decides that it is not worth using anymore. Unfortunately smart fuzzers, such as Angora, may exhibit this behavior. Having a success condition, such as solving a specific comparison, allows the fuzzer to decide to change target early if this condition is met. Rather than imposing restrictions on the fuzzer, which should simply try to increase coverage as fast as possible, we use a predictor to decide if an FTM snapshot is likely to be beneficial.

Although more advanced predictors are possible, our prototype is based on a simple dynamic threshold. Specifically, within a similarity set, the fuzzer will run the target binary without snapshot for a number of times equal to the threshold. If the fuzzer has not changed target before the threshold is reached, it predicts that a snapshot will indeed be beneficial.

This solution allows us to try and snapshot at the moment when doing so provides the greatest benefit. For example, if the fuzzer usually performs either 1 or 100 executions before changing target and a snapshot can be amortized in 10 executions, the best solution is to snapshot after the first execution. A lower threshold would make the system waste time each time the fuzzer performs only one execution per target, while a higher one would forfeit some of the gains obtainable with the snapshot. It is also possible that, on certain binaries, the gain provided by the snapshots is so reduced that it is not possible to amortize the snapshots; in that case, snapshots are effectively disabled, falling back to FTF snapshots.

We position the threshold dynamically using a benefit metric, as illustrated by Algorithm 1. The computation is based on runtime statistics collected by the fuzzer and triggered every 60 seconds. The statistics collected include data for: the time taken to perform an execution using FTF snapshots (*plain_time*); the time taken to perform an execution using FTM snapshots (*delayed_time*); the time taken to prepare a snapshot (*snap_time*); and the number of executions for which a snapshot has been used (*execs_hist*).

Algorithm 1 Predictor: Dynamic threshold placement

```
function GET_BENEFIT(thres, execs_hist, amort_execs)
    benefit ← 0
    amort_thres ← thres + amort_execs
    for bin ∈ execs_hist|bin.value > amort_execs do
        col_benefit ← (bin.value − amort_thres) · bin.count
        benefit ← benefit + col_benefit
    end for
    return benefit
end function
function GET_THRESHOLD(plain_time, delayed_time, snap_time, execs_hist)
    gain ← MDN(plain_time) − MDN(delayed_time)
    am_ex ← MDN(snap_time)/gain
    best_thres ← +∞
    max_benefit ← 0
    for all thres|thres < MAX(execs_hist.values) do
        benefit ← GET_BENEFIT(thres, execs_hist, am_ex)
        if benefit > max_benefit then
            best_thres ← thres
        end if
    end for
    return best_thres
end function
```

The frequencies collected for the last item are decayed by 15% every 15 minutes so that temporal locality in the fuzzing process is preserved (empirically chosen values that worked well across applications). It is common for the fuzzer to perform deterministic mutations at the beginning of the run, which exhibit one pattern, and then switch to random ones later, which may exhibit another.

The intuition behind this algorithm is that it finds the threshold value (number of runs before taking a snapshot) for which it is most likely that the cost of taking a snapshot can be amortized. We calculate a benefit function for various candidate thresholds and then pick the most beneficial one. The benefit formula considers the distribution of past number of executions per snapshot taken. We multiply the number of samples in each bin in the histogram of this distribution (*bin.count*) by the gain or loss that would result from using the new snapshot for the amount of executions represented by that bin (*bin.value*), given a certain threshold (*thres*). Summing up all the benefits for each bin, both positive and negative, gives the total benefit for a threshold.

The advantage of this benefit function is that, given a snapshot amortizable in 10 executions, the algorithm is aware that performing 9 executions with it causes a loss that is significantly lower than performing only 1 execution. In addition, if the fuzzer commonly runs 9 times with the same snapshot, but rarely 1000 times, the algorithm realizes that snapshots may still be beneficial: commonly losing little time may be offset by rarely gaining a lot.

We also considered simpler approaches to select the threshold, such as choosing a fixed value for all programs. However, the optimal value is very sensitive to the program under test and to the current state of the campaign. We thus opted for a dynamic solution that takes such factors into account.

4.4 Early exit

So far we considered redundant computation at the start of executions, but there is also a potential for optimization at the end. It is quite common for fuzzers to hit always the same error paths. As an example, while trying to produce a valid ELF, a fuzzer may hit the error path for an invalid magic number a huge amount of times.

When using a branch-targeted mutation operator, a fuzzer uses mutations that are specifically targeted towards solving a certain path condition. If that path condition is not solved in the current execution, these targeted mutations are unlikely to trigger other interesting behavior. Based on this assumption, we make the program exit early in such cases, cutting off the redundant computation after the target condition has been encountered. When the condition is eventually solved, the program is allowed to finish its execution, so that the coverage increase can be recorded. If any coverage increase is ever lost due to this technique, it can be quickly regained when the condition protecting the new portion of code is targeted.

5 IMPLEMENTATION

To evaluate our design, we built a prototype of Snappy on top of the Angora fuzzer [11]. Angora supports various mutation operators with a wide range of mutation patterns. It includes both AFL-like (branch-agnostic) mutation operators, which tend to modify large portions of the test case at once, but also very targeted (branch-guided) operators that can be used to focus all mutation effort on a single condition. As such, it is well-suited to assess the efficacy of our approach for both large and small similarity sets. To ensure that our results are relevant to the current state of the art, though, we integrated various performance improvements developed since the publication of Angora. In this section, we describe the improvements that we made to the Angora baseline, as well as relevant implementation details for the snapshot generation pipeline. Limitations of the current implementation are discussed in Section 7.

5.1 Improvements to Angora

After porting Angora to LLVM 11 and fixing an issue that prevented it from applying the AFL mutation operators correctly, we significantly optimized its performance (by around one order of magnitude on our benchmarks). These changes were necessary to remove common and unnecessary bottlenecks that would have otherwise dominated the execution time. To ensure fair benchmarking, all improvements are included in the baseline in Section 6.

The first performance improvement is to use kernel-based snapshots [35] as implemented in AFL++ with a kernel module[1], but optimized by us. These fast snapshots replace the slow AFL-like fork server in the original Angora. Our version features faster restores, the ability to handle large memory maps, such as the DFSan shadow map, and the ability to restore file descriptors.

The second major performance improvement integrated into Angora is AVX2-accelerated coverage map parsing. The algorithm was taken from the AFL++ project [14]. This change helps in improving performance when fuzzing small and fast target programs, given the large coverage map (1 MB) used by Angora.

[1] https://github.com/AFLplusplus/AFL-Snapshot-LKM

5.2 FTF snapshots

We implemented FTF snapshots by using our kernel-based snapshotting system to take a checkpoint at the first tainted file operation and restore the snapshot upon termination. We hook into termination events from the kernel module. To hook into tainted file operations, our instrumentation wraps all the relevant calls (`mmap`, `*fscanf`, `*getc*`, `*gets*`, `get{line,delim}`, `*read*`, `*stat`, `ftell`, `*seek`) and places the snapshot at the first wrapper invocation.

5.3 FTM snapshots

In this section, we describe the implementation of the pipeline used to build FTM snapshots, which was described in Section 4.2. This pipeline consists of three steps, as shown in Figure 1: (1) snapshot placement, (2) input tracking, and (3) snapshot initialization. The first two steps perform dynamic taint analysis using LLVM DataFlowSanitizer (DFSan) [1], while all three use LLVM XRay's code patching capabilities [6] to dynamically enable/disable snapshot-related instrumentation at each function entry/exit.

Snapshot placement. The first step in the pipeline decides where the snapshot should be placed. We wrap I/O calls and taint reads from the input generated by the fuzzer according to the current similarity set. We use taint analysis to follow the bytes in memory and, when one is loaded to perform a computation, we select the most recently encountered instrumentation position.

Instrumentation positions are defined by the instrumentation method currently being employed. In the case of Snappy, snapshots can be taken either at the beginning or at the end of a function, due to the limitations imposed by LLVM XRay. In Listing 1, the last valid instrumentation position prior to line 14 is at the end of function `other_operation`, called at line 13. In practical situations, such function calls are frequent, so the instrumentation point is typically close to the desired snapshot position.

Finally, we inform the fuzzer about the chosen instrumentation point, encoding it with its associated XRay identifier and the number of times it has been encountered before taking the snapshot.

Input tracking. The second step in the pipeline retrieves the location of individual tainted bytes present in memory at the snapshot position. We use byte-level dynamic taint analysis, exploiting the same I/O wrappers as the first instrumentation, but tracking each tainted byte in the input test case separately. Given that we know the instrumentation point selected to place the snapshot in advance, once it is encountered, all tainted bytes can be located in memory.

Snappy parses the DFSan shadow memory to find which memory locations are tainted. It then maps these addresses to address-independent metadata to be able to find them in the third binary in the pipeline. We split allocations based on their allocation type: stack, heap, or global. Respectively, we identify stack allocations using LLVM StackMaps, heap allocations using a serial identifier, and global allocations using DWARF information. Once we know the base of an allocation, we can locate single bytes using an offset. Other taints, such as the ones present in deallocated stack frames or freed heap chunks, are discarded.

Snapshot initialization. The last step in the pipeline prepares the snapshot for use by the fuzzer. Snappy uses a binary with the instrumentation needed by the fuzzer to function correctly (i.e. for

coverage tracking) and XRay instrumentation, which minimizes the overhead introduced by the snapshotting system when disabled.

Given the information provided by the previous analyses, the binary runs up to the snapshot position. There, it records a snapshot and it converts the metadata about the tainted bytes into valid addresses for the current binary. Later, each time the fuzzer restores the snapshot, we use the addresses pointing to the tainted bytes to update them with the values found in the new test case. We make this process faster by mapping test cases in a memory region shared between the fuzzer and the instrumented binary.

6 EVALUATION

We conducted our experiments on two servers, both fitted with an AMD Ryzen Threadripper 2990WX and 128 GB of RAM. We ran the experiments in Docker containers running on Ubuntu 20.04.

We completed our first experiment on FuzzBench, a well-known benchmarking suite which includes various benchmarks that have been hand-optimized to remove all the unnecessary initialization code; this renders our FTF policy not very effective. The goal of this experiment is to prove that, even in such a difficult scenario, Snappy is still able to provide significant speed improvements. We then tested Snappy on two complete programs, `sqlite3` and `objdump` using simpler harnesses. Our goal, in this case, is to provide a more realistic representation of the speed achievable using Snappy in a more realistic scenario. Finally, we examined how the speed improvements generated by Snappy correlate to coverage improvements in our benchmarks.

Each configuration in our experiments has been tested 16 times, enough to reach statistical significance for most of the benchmarks, with campaigns lasting 24 hours, following the recommendations of [17]. Our total CPU budget thus amounts to around 18 months.

6.1 FuzzBench

To show that Snappy can improve fuzzing speed, we measured the number of executions achieved in 24 hours by our prototype, and compared against our optimized version of Angora (see Section 5) as a baseline. Since Snappy is a performance optimization, it can only be compared against the fuzzer on which it was applied; as a consequence, we have not included other fuzzers in the comparison.

We conducted this experiment on all suitable benchmarks proposed by FuzzBench [19]. We excluded benchmarks that use features not supported by Angora and thus our prototype (particularly assembly and multi-threading), leaving 15 out of the 27 code coverage benchmarks. This is similar to other fuzzers with non-trivial source-level instrumentations, such as SymCC [24]. We extended the FuzzBench infrastructure to add support for forwarding host devices to be able to use the kernel-based snapshots we integrated.

All FuzzBench benchmarks are based on a libFuzzer harness, so they have been manually crafted to minimize the initialization phase while still allowing the fuzzer to run properly: they open the test case file, read it, perform some initialization if required, and then use the test case. With this structure, apart from one single case to be discussed later, FTF snapshots cannot produce any benefit; all speedups can be attributed almost entirely to FTM snapshots.

Table 1 shows the results of this experiment: 53% of the benchmarks experience a statistically significant improvement in the total number of executions performed in 24 hours, reaching a maximum of 1.76x, while the remaining 47% do not experience any significant variation. No statistically significant performance regressions were observed. We report the contribution of the early exit optimization separately, highlighting the cases in which enabling it produces a significant improvement. We normalized over the reported global improvement to hint to the real contribution, but the concrete values are unstable. We also show additional statistics concerning execution times of FTF-based and FTM-based executions, measuring them from the moment the fuzzer requests a new program instance from the snapshotting system to the moment the instance concludes its execution. These values are aggregated using their median but, since the distribution is often multimodal, they are not fully representative. Despite this, it is evident how FTM snapshots produce executions that are 37% faster than FTF on median. In addition, the cost of snapshots remains mostly in the order of 10 executions and can even cost less than one execution for slower benchmarks, such as `sqlite3_ossfuzz`, thanks to our caching system. Snappy produces up to 36,370 different snapshots in this experiment; these are often constituted by a small number of unique positions in the code, but a large number of different test cases reaching them. Finally, the column showing the "mean threshold" across each run allows to see for which benchmarks our dynamic threshold system was triggered, limiting the usage of FTM snapshots. A higher than average snapshot cost tends to correspond to benchmarks with high thresholds, which limit the use of FTM snapshots and thus the advantages of the caching system.

We performed further analysis into the benchmarks which experienced the highest speedups and confirmed that for most benchmarks, performance gains are only due to FTM snapshots. FTF snapshots do not bring major gains because the first operation performed by the libFuzzer wrappers is reading the test case. However, `libxslt_xpath` is an exception. This benchmark uses an initialization function which reads an XML file from disk, so it benefits from both FTM and FTF snapshots.

In Figure 2, we present an illustrative selection of the plots obtained from our FuzzBench evaluation, showing speed over time; the remaining plots are in Figure 4. The `lcms-2017-03-21` plot shows the best result obtained in our evaluation; once the fuzzers have covered all the fast error paths in the first few hours, the speed settles down on a fairly stable value for both fuzzers, with Snappy showing almost twice the speed when compared to the baseline. This behavior is similar in other well performing benchmarks, such as `mbedtls_dtlsclient` and `openthread-2019`. The `libxslt_xpath` plot shows instead how Snappy has a significant advantage in the first few hours, thanks to faster FTF and FTM executions, but then settles down on the same speed as baseline when the executions start to slow down, making the advantage of the FTF snapshot less significant. This progressive slowdown, present in other benchmarks as well, can be attributed to the fuzzers picking progressively less efficient (i.e. slower) test cases they have in their queue. This behavior is visible in the `re2-2014-12-09` plot as well, albeit here the advantage in the first few hours can be attributed exclusively to FTM snapshots. At the beginning of the run, both fuzzers tend to use many FTM-snapshottable mutation operators, when they can find many conditions to solve; after these conditions are solved or discarded, they switch to doing mostly AFL-like

Benchmark	Speedup (execs)	Exit opt.	FTF exec. time	FTM exec. time	Snap. time	Unique snap.	Mean thres.
freetype2-2017	1.18x	(+0.18x)	132 µs	118 µs	16 767 µs	722	396
harfbuzz-1.3.2	1.17x	(+0.24x)	168 µs	150 µs	3679 µs	28830	11
jsoncpp_fuzzer	1.04x	(+0.01x)	99 µs	72 µs	1427 µs	1474	75
lcms-2017-03-21	1.76x	(+0.37x)	915 µs	498 µs	2031 µs	2206	0
libpng-1.2.56	1.06x	(−0.02x)	91 µs	66 µs	1563 µs	450	70
libxml2-v2.9.2	1.24x	(+0.07x)	122 µs	85 µs	1991 µs	36370	10
libxslt_xpath	1.38x	(+0.07x)	90 µs	86 µs	13 855 µs	498	339
mbedtls_dtlsclient	1.62x	(+0.26x)	923 µs	97 µs	2199 µs	1539	0
openthread-2019	1.29x	(+0.04x)	246 µs	154 µs	1887 µs	1300	0
proj4-2017-08-14	0.82x	(+0.03x)	123 µs	115 µs	1391 µs	1030	203
re2-2014-12-09	1.26x	(+0.28x)	108 µs	100 µs	1195 µs	26198	166
sqlite3_ossfuzz	1.20x	(+0.22x)	10 431 µs	6143 µs	2823 µs	2244	0
vorbis-2017-12-11	0.81x	(−0.11x)	2367 µs	1323 µs	2687 µs	1036	0
woff2-2016-05-06	0.98x	(−0.20x)	95 µs	98 µs	2727 µs	1327	330
zlib_uncompress	1.10x	(−0.15x)	72 µs	57 µs	917 µs	206	119

Table 1: Statistics obtained after 24 hours on FuzzBench, statistically significant results (p-value < 0.05) for speedup are highlighted. We tested for statistical significance with the Mann-Whitney U test, as proposed by [17]. Execution times show the data used by the benefit predictor (FTM-supported operators only). All data except the speedup is aggregated using its median. The contribution of the early exit optimization, in parenthesis, is included in the overall speedup value.

mutations, which are only FTF-snapshottable, making our system still useful but less performant. Lastly, the vorbis-2017-12-11 plot shows a situation in which, albeit not statistically significant, Snappy performs slightly worse than the baseline; in this specific case, the variance in speed, which leads to overlapping confidence intervals, does not allow our dynamic threshold predictor to revert back to FTF executions, and the threshold is always left to 0.

6.2 Real-world applications

Given the highly optimized construction of FuzzBench, which nullifies the advantages provided by FTF snapshots most of the time, we decided to consider two real-world applications, fuzzing them with generic wrappers. We chose objdump, since its parent project, binutils, is commonly used in the literature, and sqlite3, which allows for a direct comparison with the wrapper present in FuzzBench. As we shall see later, these two programs are also useful to showcase how FTF snapshots can provide more (or less) benefits compared to FTM depending on the characteristics of the target application—which Snappy can adapt to automatically.

These experiments also show whether Snappy can be used to save the manual effort to construct a harness for the target program. In this context, FTF should be generally more useful than on FuzzBench. Therefore, we fuzzed objdump without major changes, and sqlite3 using a generic wrapper, fuzzershell, provided by the project itself. This wrapper mimics the original program but simplifies the fuzzing setup, requiring no integration work.

The plots we discuss in our analysis are aggregated in Figure 3.

objdump. Snappy shows a peculiar behavior for this benchmark: after an initial 6 hours period, its execution speed drops down to around 1 execution per second. The cause of this behavior lies in the seeds used for our runs: in order to ensure fairness, we reused the same seeds provided by OSS-Fuzz [30], which go up to 7 MB in size. Seeds this large make the fuzzers produce test cases that are likely to time out and are thus discarded. In Snappy, though, the increased speed generates 5x fewer timeouts, retaining such large seeds in the queue. As a consequence, the total amount of executions decreases by 84%, but the benefit in terms of coverage is significant because, as discussed later, Snappy is able to use such large seeds to explore additional portions of the program.

Although not manually optimized like FuzzBench, objdump also shows improvements mostly due to FTM snapshots; the predictor's dynamic threshold is set to 0 for around 97% of the time, which means the program uses FTM on each run. Manual analysis shows that the program checks the size of the test case early on, triggering the FTF snapshot right after main is called. This shows that FTM snapshots are not only useful for optimized wrappers, but also to limit the shortcomings of FTF snapshots on normal programs.

sqlite3. In this benchmark, the total amount of executions increases by 1.35x, which is statistically significant and higher than the 1.20x obtained on sqlite3_ossfuzz. This benchmark is characterized by high variations in speed, probably associated with different regions of code. Despite this, Snappy is able to perform better both in the slow portions, in the first few hours, and in the faster ones, for the remainder of the time. As discussed later, the advantage is clearly reflected in coverage, where the higher initial speed introduces a lag between Snappy and our baseline.

The predictor's snapshot threshold is set to $+\infty$ for 95% of the time, so it almost exclusively uses FTF snapshots, with FTM executions concentrated exclusively in the first few hours. This behavior is completely opposite to the sqlite3_ossfuzz benchmark in FuzzBench, for which the threshold was instead always set to 0. This shows that, with less carefully crafted harnesses, FTF snapshots can be extremely useful, even on the same target application.

6.3 Influence on coverage

The metric most commonly used in the literature to evaluate new coverage-oriented fuzzers is edge coverage. The reason is that coverage is considered a proxy metric for the actual goal of fuzzers, which is finding bugs. Trivially, if the code containing a bug cannot be reached, the bug cannot be discovered. The advantage of using

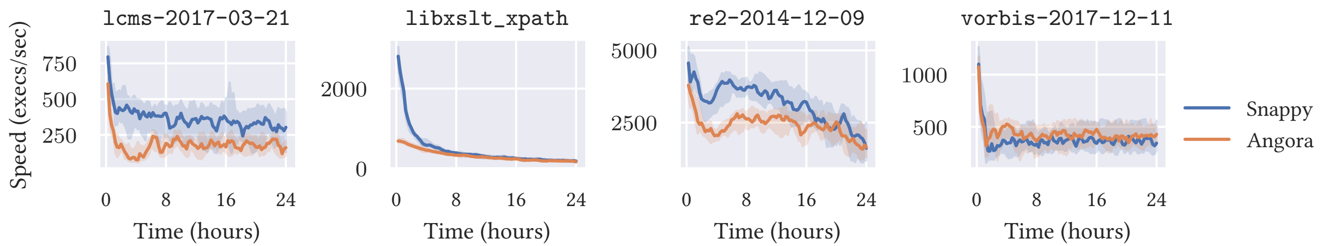

Figure 2: A selection of the speed plots obtained from our FuzzBench evaluation. The line plots show the median value in our 16 runs and the confidence intervals are 95% on the median. The plots have been smoothed with a 1-hour window for readability.

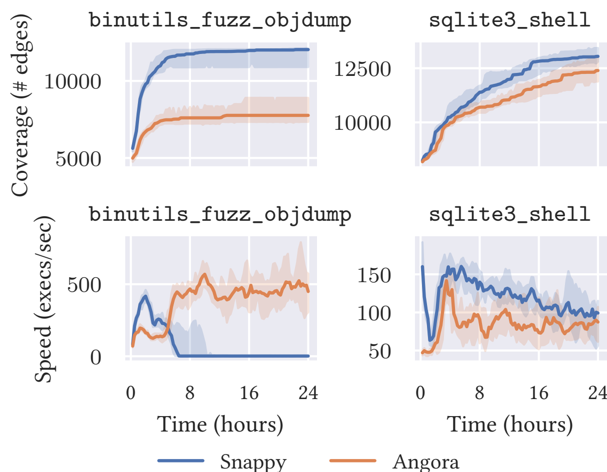

Figure 3: Coverage and speed plots for `objdump` and `sqlite3`. The line plots show the median value in our 16 runs with a 95% confidence interval. Interestingly, the speed results for `objdump` are influenced by Snappy being too fast, allowing large seeds to not time out but instead yield more coverage.

coverage over the number of unique bugs found is that it is clearly defined, more granular, and also more easily measurable.

Snappy improves fuzzing speed. As such, compared to the baseline, the same coverage can be achieved in less time (and consequently, using less power). Fuzzers achieve much coverage early on, when exploring the main paths in the target, and progress slowly as they need to flip the remaining more difficult branches until a plateau is reached (i.e., the fuzzer is no longer able to make significant progress in bounded time). As such, our approach is most effective in increasing coverage during the progression towards the plateau, but cannot improve it after the plateau is reached.

FuzzBench. Most of the benchmarks are small libraries which reach their coverage plateau very quickly with a large initial increase; they thus cannot strongly benefit from increases in speed. Only one of the benchmarks with significant speed improvements also exhibited significant coverage differences at the end of the 24 hours. In 38% of the cases with significantly increased speed, though, Snappy was able to gain a statistically significant coverage advantage at some point during the 24 hours, before the baseline was able to reach the plateau; in the remaining 68%, the speed increase does not seem to have significantly influenced coverage progress. Among the cases in which no significant speed difference could be obtained, there are a few benchmarks (38%) in which the baseline performs instead better at some point, mainly due to Snappy performing slightly worse, albeit not significantly. Among these, `harfbuzz-1.3.2` is peculiar because the baseline reaches a higher coverage plateau; we attribute this to a worst case interaction between our speed changes and the scheduling algorithm of the fuzzers. In the remaining 62% of the cases without significant speed differences, coverage does not appear to significantly change.

Real-world programs. Both `objdump` and `sqlite3` present significant coverage increases, 31% for the former and 3% for the latter, due to an increased execution speed. In Figure 3, the correlation is clearly visible for `sqlite3`, where higher speed introduces a lag between Snappy and our baseline. For `objdump`, the effect is stronger but less evident. Fewer timeouts allow to retain more interesting test cases and thus explore the program more in depth.

7 DISCUSSION

In this section, we discuss limitations of our current Snappy prototype and possible future improvements that we believe would further increase the performance of our system in specific cases.

Similarity set awareness. Since the snapshot policies proposed by Snappy support all common mutation operators without modifications, they can be adapted to other fuzzers as well. We believe, however, that making DTA-agnostic fuzzers (e.g., AFL++) aware of similarity sets will allow them to partially benefit from FTM snapshots. Since taint information is not available, it is necessary to define a new policy for the creation of similarity sets; the DTA-agnostic mutation operators should then be modified to use them. The use of pseudo-DTA methods, such as the one proposed by RedQueen [5], should allow to define similarity sets that provide comparable performance. Another possibility is to divide the test case arbitrarily (e.g., in blocks) and then use or discard snapshots based on their performance or on their position. Both these policies are less precise, but should still provide performance improvements.

Limitations of the current implementation. Our prototype uses compiler instrumentation, and therefore requires source code for the program to be fuzzed, as is common among greybox fuzzers.

In addition, our prototype supports linking against uninstrumented libraries, but, since it is based on DFSan [1], it inherits its limitation of being able to track taint for these libraries only when appropriate wrappers are provided. When this is not the case, the FTM placement step may misplace the snapshot if the first tainted memory read is performed inside an uninstrumented function.

In practice, support for tainting in external library functions is mainly an issue for `libc`, and, following DFSan, we added appropriate wrappers for the most relevant functions. For example, `memcpy` is wrapped to propagate taint, and our `memcmp` wrapper determines whether the result depends on tainted bytes from its input buffers, and triggers a snapshot if it is. Only functions that may take pointers to tainted data are affected, and only if they either propagate tainted data or can semantically change the flow of the program.

Other uninstrumented external libraries are also supported, but receive no special handling in our evaluation. Our rationale is that greybox fuzzers typically require instrumentation to track coverage, so that such libraries should not be the fuzzing target and are therefore unlikely to handle input data in ways that are relevant for the result. A missed snapshot may lead to a partially corrupted program execution, but this is not harmful in the context of fuzzing as, at worst, it wastes a reduced number of executions (and only if not immediately detected). When such failed snapshots are detected, Snappy simply falls back on FTF snapshots. Finally, if an external library represents a worst-case scenario for this issue, it will be easily detected due to spurious crashes and reduced coverage progress, hinting that additional wrappers should be added. This has not been the case during our evaluation.

While Angora is affected by implicit flows, FTM snapshots are not. In order to generate an implicit flow, tainted data need to be loaded from memory; this operation, according to our FTM policy, triggers a snapshot, so no implicit flow is possible.

8 RELATED WORK

Much recent work in the field of fuzzing focuses on mutation-based (greybox) fuzzers, which commonly rely on a feedback function to select and mutate a set of test cases deemed interesting. The most commonly used feedback function is code coverage [37], but several efforts also propose to direct fuzzers towards specific locations in the program using distance metrics as feedback [8, 10, 13, 22]. Among coverage-guided fuzzers, prior work focuses on improving various elements of the fuzzing loop. For instance, there is work on improving both test case [9, 15] and mutation operator [18] scheduling. Most efforts, however, focus on improving the quality of the produced test cases by changing the mutation operators themselves. For instance, some solutions focus on improving performance for highly structured grammars [4, 7], others rely on machine learning to help guide the mutation operators towards solving branch conditions [31]. Moreover, several solutions propose to improve mutations by means of dynamic taint analysis (DTA), using either blackbox [5] or whitebox [11, 12, 26, 34] flavors of DTA. Our work also relies on whitebox DTA, but exploits it for the purpose of placing snapshots, instead of improving mutation operators. Beyond DTA, several efforts propose to integrate more heavyweight techniques, such as concolic execution [24, 25, 33, 36] or program transformations [23] to improve fuzzing performance.

Speed is one of the most important properties in a fuzzer and thus, like Snappy, other solutions focus on improving it. The operating system primitive proposed by [35] aims at reducing the time spent in the operating system before starting a new execution; we build upon that idea by integrating its modern reincarnation [14] in our prototype. Nonetheless, our snapshot placement technique is orthogonal to the snapshot system used and thus could further benefit from additional work in this direction. In the context of directed fuzzing, Beacon [16] proposes a solution that removes useless computation by cutting executions when they stray away from the target specified. This solution, however, is tightly coupled with directed fuzzing and thus cannot be adapted to coverage-guided fuzzers. Moreover, our system could be used in combination with Beacon (and other similar fuzzers) in a directed fuzzing context as well, allowing it to achieve even better results.

Other solutions have sought to remove initialization overhead when fuzzing complex targets through the use of a hypervisor. Nyx [27, 29] uses lightweight snapshots to restore VMs quickly; these snapshots, however, are taken before its agent operating system starts to parse the input. This solution is thus positionally equivalent to taking a snapshot after the dynamic loader has completed its tasks when fuzzing a program. Agamotto [32] goes further and proposes to move snapshots just before the first mutation performed in a test case formed by a series of system calls, removing redundant computation, as we do. This technique, however, is strongly dependent on the fact that system calls are executed in sequence when testing operating systems; this is not the case for general programs and thus our novel snapshot positioning policies do not rely on this assumption. Finally, FIRM-AFL [38] and SnapFuzz [3] propose to take snapshots wrapping networking functions, making them similar to our FTF snapshot policy. Our policy, however, does not snapshot on any interaction, as FIRM-AFL does, but only on those that copy data produced by the fuzzer into memory or check the test case size. Furthermore, FIRM-AFL and SnapFuzz do not track tainted data in memory, as our FTM snapshots do.

9 CONCLUSION

We presented Snappy, a system to speed up fuzzing by aggressively pruning redundant computation using adaptive and mutable snapshots. Snappy uses two novel snapshot positioning policies, FTF and FTM snapshots, to place snapshots late in the execution trace; the most convenient policy is selected dynamically, by examining run-time statistics.

We evaluated Snappy using FuzzBench and two real world programs: `sqlite3` and `objdump`. Our evaluation demonstrated that both our snapshot policies are beneficial—well complementing each other on programs with different characteristics—and that our system can dynamically adapt to select the best technique and score gains for common fuzzers and input mutation operators.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their feedback. This work was supported by Ministry of Economic Affairs and Climate Policy (EZK) through the AVR "Memo" project and by the Dutch Research Council (NWO) through projects "TROPICS", "Theseus", and "INTERSECT".

REFERENCES

[1] [n.d.]. DataFlowSanitizer. https://clang.llvm.org/docs/DataFlowSanitizer.html.
[2] [n.d.]. libFuzzer. https://llvm.org/docs/LibFuzzer.html.
[3] Anastasios Andronidis and Cristian Cadar. 2022. SnapFuzz: High-Throughput Fuzzing of Network Applications. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2022)* (Virtual Event).
[4] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. Nautilus: Fishing for Deep Bugs with Grammars. In *Proceedings of the 2019 Network and Distributed System Security Symposium (NDSS)*.
[5] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *Proceedings of the 2019 Network and Distributed System Security Symposium (NDSS)*, Vol. 19. 1–15.
[6] Dean Michael Berris, Alistair Veitch, Nevin Heintze, Eric Anderson, and Ning Wang. 2016. *XRay: A Function Call Tracing System.* Technical Report. A white paper on XRay, a function call tracing system developed at Google.
[7] Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. 2019. GRIMOIRE: Synthesizing Structure while Fuzzing. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1985–2002. https://www.usenix.org/conference/usenixsecurity19/presentation/blazytko
[8] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 2329–2344. https://doi.org/10.1145/3133956.3134020
[9] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2019. Coverage-Based Greybox Fuzzing as Markov Chain. *IEEE Transactions on Software Engineering* 45, 5 (2019), 489–506. https://doi.org/10.1109/TSE.2017.2785841
[10] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. 2018. Hawkeye: Towards a Desired Directed Grey-Box Fuzzer. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 2095–2108. https://doi.org/10.1145/3243734.3243849
[11] Peng Chen and Hao Chen. 2018. Angora: Efficient Fuzzing by Principled Search. In *2018 IEEE Symposium on Security and Privacy (SP)*. 711–725. https://doi.org/10.1109/SP.2018.00046
[12] Peng Chen, Jianzhong Liu, and Hao Chen. 2019. Matryoshka: Fuzzing Deeply Nested Branches. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 499–513. https://doi.org/10.1145/3319535.3363225
[13] 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 Symposium on Security and Privacy (SP)*. 1580–1596. https://doi.org/10.1109/SP40000.2020.00002
[14] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++ : Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association. https://www.usenix.org/conference/woot20/presentation/fioraldi
[15] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. 2018. CollAFL: Path Sensitive Fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*. 679–696. https://doi.org/10.1109/SP.2018.00040
[16] Heqing Huang, Yiyuan Guo, Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. 2022. BEACON: Directed Grey-Box Fuzzing with Provable Path Pruning. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE.
[17] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 2123–2138. https://doi.org/10.1145/3243734.3243804
[18] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. MOPT: Optimized Mutation Scheduling for Fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1949–1966. https://www.usenix.org/conference/usenixsecurity19/presentation/lyu
[19] Jonathan Metzman, László Szekeres, Laurent Maurice Romain Simon, Read Trevelin Sprabery, and Abhishek Arya. 2021. 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 (ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 1393–1403. https://doi.org/10.1145/3468264.3473932
[20] Xianya Mi, Sanjay Rawat, Cristiano Giuffrida, and Herbert Bos. 2021. LeanSym: Efficient Hybrid Fuzzing Through Conservative Constraint Debloating. In *24th International Symposium on Research in Attacks, Intrusions and Defenses* (San Sebastian, Spain) *(RAID '21)*. Association for Computing Machinery, New York, NY, USA, 62–77. https://doi.org/10.1145/3471621.3471852
[21] Stefan Nagy and Matthew Hicks. 2019. Full-Speed Fuzzing: Reducing Fuzzing Overhead through Coverage-Guided Tracing. In *2019 IEEE Symposium on Security and Privacy (SP)*. 787–802. https://doi.org/10.1109/SP.2019.00069
[22] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2020. ParmeSan: Sanitizer-guided Greybox Fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2289–2306. https://www.usenix.org/conference/usenixsecurity20/presentation/osterlund
[23] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. 2018. T-Fuzz: Fuzzing by Program Transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*. 697–710. https://doi.org/10.1109/SP.2018.00056
[24] Sebastian Poeplau and Aurélien Francillon. 2020. Symbolic execution with SymCC: Don't interpret, compile!. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 181–198. https://www.usenix.org/conference/usenixsecurity20/presentation/poeplau
[25] Sebastian Poeplau and Aurélien Francillon. 2021. SymQEMU: Compilation-based symbolic execution for binaries. In *Proceedings of the 2021 Network and Distributed System Security Symposium (NDSS)*.
[26] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In *Proceedings of the 2017 Network and Distributed System Security Symposium (NDSS)*, Vol. 17. 1–14. https://download.vusec.net/papers/vuzzer_ndss17.pdf
[27] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. 2021. Nyx: Greybox Hypervisor Fuzzing using Fast Snapshots and Affine Types. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2597–2614. https://www.usenix.org/conference/usenixsecurity21/presentation/schumilo
[28] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. 2017. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 167–182. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/schumilo
[29] Sergej Schumilo, Cornelius Aschermann, Andrea Jemmett, Ali Abbasi, and Thorsten Holz. 2022. Nyx-Net: Network Fuzzing with Incremental Snapshots. In *EuroSys*.
[30] Kostya Serebryany. 2017. OSS-Fuzz - Google's continuous fuzzing service for open source software. USENIX Association, Vancouver, BC.
[31] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. 2019. NEUZZ: Efficient Fuzzing with Neural Program Smoothing. In *2019 IEEE Symposium on Security and Privacy (SP)*. 803–817. https://doi.org/10.1109/SP.2019.00052
[32] Dokyung Song, Felicitas Hetzelt, Jonghwan Kim, Brent ByungHoon Kang, Jean-Pierre Seifert, and Michael Franz. 2020. Agamotto: Accelerating Kernel Driver Fuzzing with Lightweight Virtual Machine Checkpoints. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2541–2557. https://www.usenix.org/conference/usenixsecurity20/presentation/song
[33] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting fuzzing through selective symbolic execution. In *Proceedings of the 2016 Network and Distributed System Security Symposium (NDSS)*, Vol. 16. 1–16.
[34] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. 2010. TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection. In *2010 IEEE Symposium on Security and Privacy*. 497–512. https://doi.org/10.1109/SP.2010.37
[35] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. 2017. Designing New Operating Primitives to Improve Fuzzing Performance. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 2313—-2328. https://doi.org/10.1145/3133956.3134046
[36] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 745–761. https://www.usenix.org/conference/usenixsecurity18/presentation/yun
[37] Michał Zalewski. 2013. American fuzzy lop.
[38] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. 2019. FIRM-AFL: High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1099–1114. https://www.usenix.org/conference/usenixsecurity19/presentation/zheng

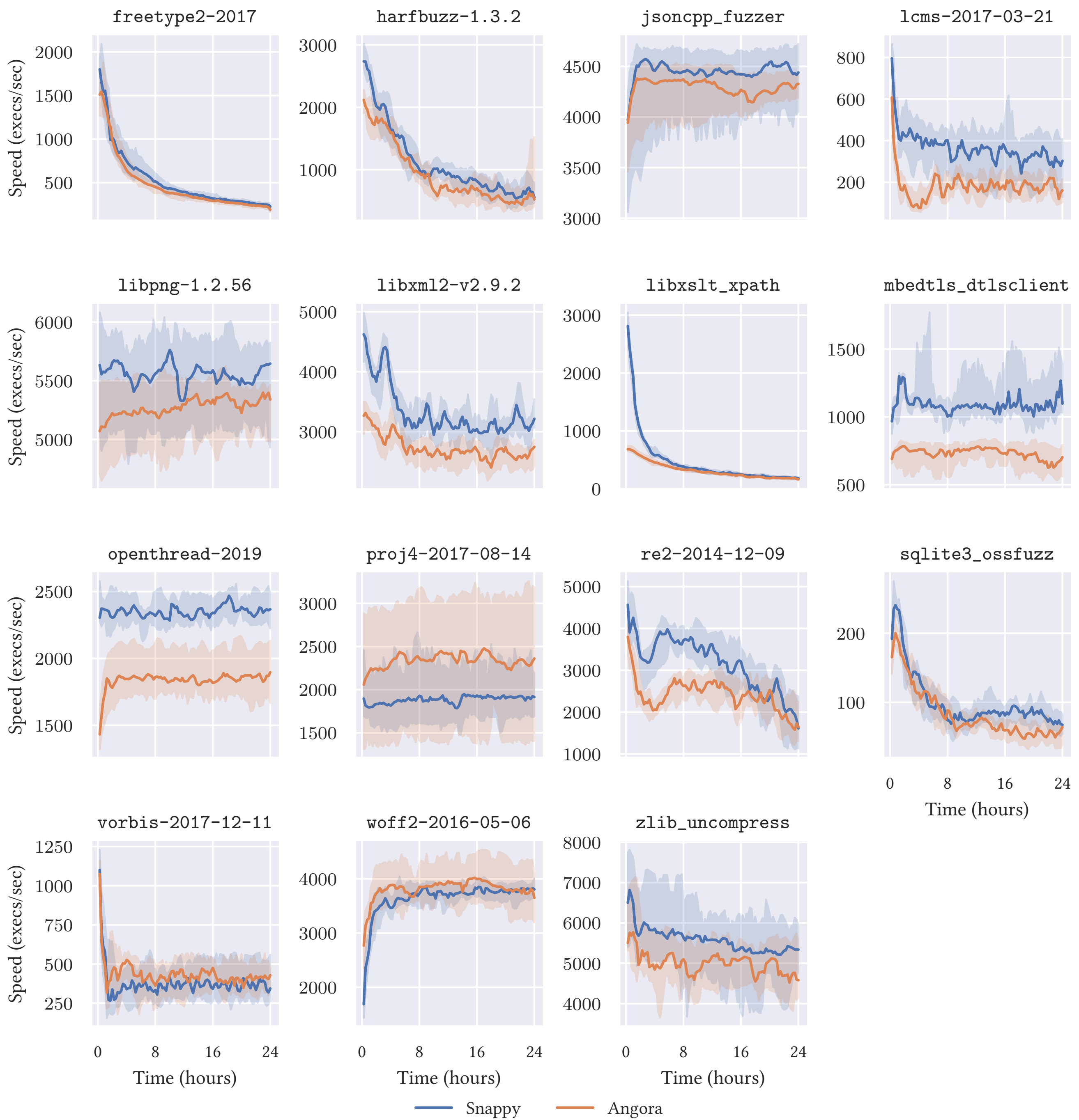

Figure 4: Speed plots obtained from our FuzzBench evaluation. The line plots show the median value in our 16 runs and the confidence intervals are 95% on the median. The plots have been smoothed with a 1-hour window for readability.

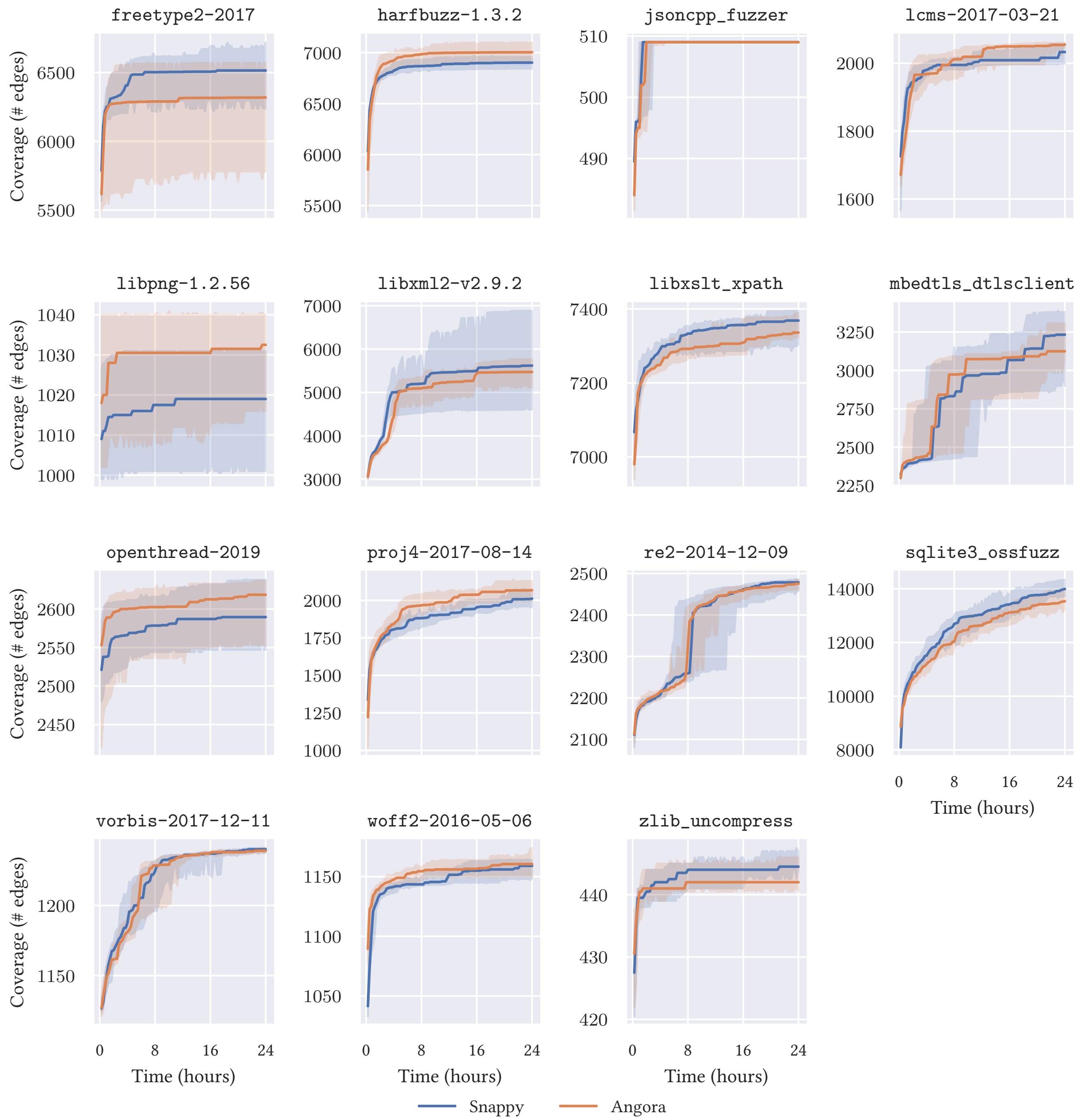

Figure 5: Coverage plots obtained from our FuzzBench evaluation. The line plots show the median value in our 16 runs and the confidence intervals are 95% on the median.