SnapFuzz: High-Throughput Fuzzing of Network Applications

Anastasios Andronidis
Imperial College London
London, United Kingdom
a.andronidis@imperial.ac.uk

Cristian Cadar
Imperial College London
London, United Kingdom
c.cadar@imperial.ac.uk

ABSTRACT
In recent years, fuzz testing has benefited from increased computational power and important algorithmic advances, leading to systems that have discovered many critical bugs and vulnerabilities in production software. Despite these successes, not all applications can be fuzzed efficiently. In particular, stateful applications such as network protocol implementations are constrained by a low fuzzing throughput and the need to develop complex fuzzing harnesses that involve custom time delays and clean-up scripts.

In this paper, we present SnapFuzz, a novel fuzzing framework for network applications. SnapFuzz offers a robust architecture that transforms slow asynchronous network communication into fast synchronous communication, snapshots the target at the latest point at which it is safe to do so, speeds up file operations by redirecting them to a custom in-memory filesystem, and removes the need for many fragile modifications, such as configuring time delays or writing clean-up scripts.

Using SnapFuzz, we fuzzed five popular networking applications: LightFTP, TinyD TLS, Dnsmasq, Ll V E 5 5 5 and Dcmqrscp. We report impressive performance speedups of 62.8x, 41.2x, 30.6x, 24.6x, and 8.4x, respectively, with significantly simpler fuzzing harnesses in all cases. Due to its advantages, SnapFuzz has also found 12 extra crashes compared to AFLNet in these applications.

CCS CONCEPTS
- Software and its engineering → Software testing and debugging.
- Security and privacy → Systems security.

KEYWORDS
Fuzzing, network protocol implementations, stateful applications

ACM Reference Format:

1 INTRODUCTION
Fuzzing is an effective technique for testing software systems, with popular fuzzers such as AFL and LibFuzzer having found thousands of bugs in both open-source and commercial software. For instance, Google has discovered over 25,000 bugs in their products and over 22,000 bugs in open-source code using greybox fuzzing [18].

Unfortunately, not all software can benefit from such fuzzing campaigns. One important class of software, network protocol implementations, is difficult to fuzz. There are two main difficulties: the fact that in-depth testing of such applications needs to be aware of the network protocol they implement (e.g., FTP, DICOM, SIP), and the fact that they have side effects, such as writing data to the file system or exchanging messages over the network.

There are two main approaches for testing such software in a meaningful way. One approach, adopted by Google’s OSS-Fuzz, is to write unit-level test drivers that interact with the software via its API [21]. While such an approach can be effective, it requires significant manual effort, and does not perform system-level testing where an actual server instance interacts with actual clients.

A second approach, used by AFLNet [30], performs system-level testing by starting actual server and client processes, and generating random message exchanges between them which nevertheless follow the underlying network protocol. Furthermore, it does so without needing a specification of the protocol, but rather by using a corpus of real message exchanges between server and clients. AFLNet’s approach has significant advantages, requiring less manual effort and performing end-to-end testing at the protocol level.

While AFLNet makes important advances in terms of fuzzing network protocols, it has two main limitations. First, it requires users to add or configure various time delays in order to make sure the protocol is followed, and to write clean-up scripts to reset the state across fuzzing iterations. Second, it has poor fuzzing performance, caused by asynchronous network communication, various time delays, and expensive file system operations, among others.

SnapFuzz addresses both of these challenges through a robust architecture that transforms slow asynchronous network communication into fast synchronous communication, speeds up file operations and removes the need for clean-up scripts via an in-memory filesystem, and improves other aspects such as delaying and automating the forkserver placement, correctly handling signal propagation and eliminating developer-added delays.

These improvements significantly simplify the construction of fuzzing harnesses for network applications and dramatically improve fuzzing throughput in the range of 8.4x to 62.8x (mean: 30.6x) for a set of five popular server benchmarks.

2 FROM AFL TO AFLNET TO SNAPFUZZ
In this section, we first discuss how AFL and AFLNet work, focusing on their internal architecture and performance implications, and then provide an overview of SnapFuzz’s architecture and main contributions.
2.1 American Fuzzy Lop (AFL)

AFL [27] is a greybox fuzzer that uses an effective coverage-guided genetic algorithm. AFL uses a modified form of edge coverage to efficiently identify inputs that change the target application’s control flow.

In a nutshell, AFL first loads user-provided initial seed inputs into a queue, picks an input, and mutates it using a variety of strategies. If a mutated input improves coverage, it is added to the queue and the cycle is repeated.

At a systems level, AFL’s simplest mode (called dumb mode) is to restart the target application from scratch by forking first and then creating a fresh process via execve. When this happens, the standard sequence of events to start a process is taking place, with the OS loader first initialising the target application and its libraries into memory. AFL then sends to the new process the fuzzed input through a file descriptor that usually points to an actual file or stdin. Lastly, AFL waits for the target to terminate, but kills it if a predefined timeout is exceeded. These steps are repeated for every input AFL wants to provide to the target application.

AFL’s dumb mode is rather slow as too much time is spent on loading and initialising the target and its libraries (such as libc) for every generated input. Ideally, the application would be restarted after all these initialisation steps are done, as they are irrelevant to the input provided by AFL. This is exactly what AFL’s forkserver mode offers, as shown in Figure 1.

In this mode, AFL first creates a child server called the forkserver (step 1 in Figure 1), which loads the target application via execve and freezes it just before the main function is about to start.

Then, in each fuzzing iteration, the following steps take place in a loop: AFL requests a new target instance from the forkserver (step 2), the forkserver creates a new instance (step 3). AFL sends fuzzed input to this new instance (step 4), and the forkserver checks the target instance for crashes (step 5).

With this forkserver snapshotting mechanism, AFL replaces the loading overhead by a much less expensive fork call, while guaranteeing that the application will be at its initial state for every freshly generated input from AFL.

One additional optimisation that AFL offers is the deferred forkserver mode. In this mode, the user can manually add in the target’s source code a special call to an internal function of AFL in order to instruct it to create the forkserver at a later stage in the execution of the target application. This can provide significant performance benefits in the common case where the target application needs to perform a long initialisation phase before it is able to consume AFL’s input. Unfortunately though, this mode requires the user not only to have access to the source code of the target application, but also knowledge of its internals in order to place the deferred call at the correct stage of execution. As we will explain in §3.4, the forkserver placement has several restrictions (e.g., it cannot be placed after file descriptors are created) and if these restrictions are violated, the fuzzing campaign can waste a lot of time exploring invalid executions.

2.2 AFLNet

AFL essentially targets applications that receive inputs via files (with stdin a special file type). This means that it is not directly applicable to network applications, as they expect inputs to arrive through network sockets and follow an underlying network protocol.

AFLNet [30] extends AFL to work with network applications. Its most important contribution is that it proposes a new algorithm on how to generate inputs that follow the underlying network protocol (e.g., the FTP, DNS or SIP protocols). More specifically, AFLNet infers the underlying protocol via examples of recorded message exchanges between a client and the server.

AFLNet also extends AFL by building the required infrastructure to direct the generated inputs through a network socket to the target application, as shown in Figure 2. More precisely, from a systems perspective, AFLNet acts as the client application. After a configurable delay waiting for the server under fuzzing to initialise, it sends inputs to the server through TCP/IP or UDP/IP sockets, with configurable delays between those deliveries (we describe the various time delays needed by AFLNet in §3.2). AFLNet consumes the replies from the server (or else the server might block) and also sends to the server a SIGTERM signal after each exchange is deemed complete, as usually network applications run in infinite loops.

As shown in Figure 2, the architecture of AFLNet is similar to that of AFL’s deferred forkserver mode, except that communication takes place over the network instead of via files.

Network applications like databases or FTP servers are often stateful, keeping track of their state by storing information to various files. This can create issues during a fuzzing campaign because when AFLNet restarts the application, its state might be tainted by information from a previous execution. To avoid this problem,
"AFLNet requires the user to write custom clean-up scripts that are invoked to reset any filesystem state.

We use the term fuzzing harness to refer to all the code that users need to write in order to be able to fuzz an application. In AFLNet, this includes the client code, the various time delays that need to be manually added, and the clean-up scripts. One important goal of SnapFuzz is to simplify the creation of fuzzing harnesses for network applications.

2.3 SnapFuzz

SnapFuzz is built on top of AFLNet by revamping its network communication architecture as shown in Figure 3, without any modifications to AFLNet’s fuzzing algorithm.

SnapFuzz’s main objectives are (1) to improve the performance (throughput) of fuzzing network applications, and (2) lower the barrier for testing network applications by simplifying the construction of fuzzing harnesses, in particular by eliminating the need to add manually-specified time delays and to write clean-up scripts.

At a high level, SnapFuzz achieves its significant performance gains by: optimising all network communications by eliminating synchronisation delays (the SnapFuzz protocol); automatically injecting AFL’s forkserver deeper into the application than otherwise possible and without the user’s intervention (smart deferred forkserver); performing binary rewriting-enabled optimisations which eliminate additional delays and inefficiencies; automatically resetting any filesystem state; and optimising filesystem accesses by redirecting them into an in-memory filesystem.

SnapFuzz also makes fuzzing harness development easier and in some cases trivial by completely removing the need for manual code modifications. Such manual changes are often required to: reset the state of either the target or its environment after each fuzzing iteration; terminate the target, as usually servers run in infinite loops; pin the CPU for threads and processes; and add deferred forkserver support to the target.

Figure 3 shows the architecture of SnapFuzz. While at a high-level it resembles that of AFLNet, there are several important changes. First, SnapFuzz intercepts the external actions of the target application using binary rewriting (§3.1). It then monitors the behaviour of both the target application and the AFLNet client in order to eliminate synchronisation delays using its SnapFuzz protocol (§3.2). Second, a custom in-memory filesystem is added, to improve performance and facilitate resetting the state after each fuzzing iteration (§3.3). Third, the forkserver is replaced by a smart deferred forkserver, which automates and optimizes the forkserver placement (§3.4). We describe the main components of SnapFuzz in detail in the next section.

3 DESIGN

SnapFuzz has two main goals: significantly increase fuzzing throughput, and simplify the construction of fuzzing harnesses. SnapFuzz accomplishes these goals by intercepting all the communication between the target application and its environment via binary rewriting (§3.1). By controlling this communication, SnapFuzz can then:

1. Implement an efficient network fuzzing protocol which notifies the fuzzier when the target application state is ready to accept a new request or when a response is ready to be consumed (§3.2). This improves fuzzing throughput and eliminates the need for all the custom delays that AFLNet users need to insert in order to synchronise the communication between the fuzzier and the target application. SnapFuzz also replaces Internet sockets by UNIX domain sockets, which improves performance, and implements an efficient server termination strategy.

2. Redirect all file operations to use an in-memory filesystem (§3.3). This improves the performance of filesystem operations, and obviates the need for user-provided clean-up scripts, as SnapFuzz can automatically clean up after each fuzzing iteration by simply discarding the in-memory state.

3. Automatically place and defer the forkserver (“smart deferred forkserver”) to the latest safe point (§3.4). This improves performance and eliminates the need for manual annotations.

4. Eliminate custom delays, unnecessary system calls and potentially expensive clean-up routines that are part of the target application, correctly propagate signals from child processes, and better control CPU affinity (§3.5).

3.1 Binary Rewriting

SnapFuzz implements a load-time binary rewriting subsystem that dynamically intercepts both the OS loader’s and the target’s functionalities in order to monitor and modify all external behaviours of the target application.

Applications interact with the external world via system calls, such as read() and write() in Linux, which provide various OS services. As an optimisation, Linux provides some services via vDSO (virtual Dynamic Shared Object) calls. vDSO is essentially a small shared library injected by the kernel in every application in order to provide fast access to some services. For instance, gettimeofday() is typically using a vDSO call on Linux.

The main goal of the binary rewriting component of SnapFuzz is to intercept all the system calls and vDSO calls issued by the application being fuzzed, and redirect them to a system call handler. §4.1 presents the implementation details.

By intercepting the target application’s interactions with its outside environment at this level of granularity, SnapFuzz can significantly increase fuzzing throughput and eliminate the need for custom delays and scripts, as we discuss in the next subsections.
3.2 SnapFuzz Network Fuzzing Protocol: Eliminating Communication Delays

Network applications often implement multistep protocols with multiple requests and replies per session. One of AFLNet’s main contributions is to infer the network protocol starting from a set of recorded message exchanges. However, AFLNet cannot guarantee that during a certain fuzzing iteration the target will indeed respect the protocol. Deviations might be possible for instance due to a partly-incorrect protocol being inferred, bugs in the target application, or most commonly due to the target not being ready to send or receive a certain message.

Therefore, AFLNet performs several checks and adds several user-specified delays to ensure communication is in sync with the protocol. These communication delays, which can significantly degrade the fuzzing throughput, are:

1. A delay to allow the server to initialise before AFLNet attempts to communicate.
2. A delay specifying how long to wait before concluding that no responses are forthcoming and instead try to send more information, and
3. A delay specifying how long to wait after each packet is sent or received.

These delays are necessary, as otherwise the OS kernel will reject packets that come too fast while the target is not ready, and AFLNet will desynchronise from its state machine. But they cause a lot of time to be wasted, essentially because AFLNet does not know whether the target is ready to send or receive information.

SnapFuzz overcomes this challenge through a simple but effective network fuzzing protocol. The protocol keeps track of the next action of the target, and notifies AFLNet about it. Figure 4 shows the messages exchanged between SnapFuzz and AFLNet on each recv (for receiving data) and send (for sending data) system calls. Essentially, to avoid the need for the communication delays discussed above, SnapFuzz informs AFLNet when the target is about to issue a recv or a send. This is performed by introducing an additional control socket (implemented via an efficient UNIX domain socket), which is used as a send-only channel from the SnapFuzz plugin to AFLNet.

The SnapFuzz network fuzzing protocol additionally implements the following two optimisations:

**UNIX Domain Sockets.** The standard Internet sockets (TCP/IP and UDP/IP) used by AFLNet to communicate to the target and send

---

that using tmpfs would not solve this issue—as far as we know, there is no way to duplicate a tmpfs filesystem in a copy-on-write way. This sharing of pages between the new instance and the fork-server is problematic, as now any modifications to the in-memory files by the fuzzed application instance will persist even after the instance finishes execution. So in the next iteration, when the fork-server creates a new instance, this new instance will inherit those modifications too.

SnapFuzz solves this issue as follows. First, note that SnapFuzz knows whether the application is executing before or after the fork-server’s snapshot, as it intercepts all system calls, including fork. While the target application executes before the fork-server’s snapshot, SnapFuzz allows all file interactions to be handled normally. When a new instance is requested from the fork-server, SnapFuzz recreates in the new instance all in-memory files registered in the in-memory filesystem and copies all their contents by using the efficient sendfile system call once per in-memory file.

### 3.4 Smart Deferred Forkserver

As discussed in §2.1, the deferred forkserver can offer great performance benefits by avoiding initialisation overheads in the target. Such overheads include loading the shared libraries used by the target, parsing configuration files and cryptographic initialisation routines. Unfortunately, for the deferred forkserver to be used, the user needs to manually modify to source code of the target. Furthermore, the deferred forkserver cannot be used after the target has created threads, child processes, temporary files, network sockets, offset-sensitive file descriptors, or shared-state resources, so the user has to carefully decide where to place it: do it too early and optimisation opportunities are missed, do it too late and correctness is affected.

SnapFuzz makes two important improvements to the deferred forkserver: first, it makes it possible to defer it much further than usually possible with AFL’s architecture, and second, it does so automatically, without any need for manual source modifications.

The two components which enable SnapFuzz to place the fork-server after many system calls which normally would have caused problems are: (1) its custom network fuzzing protocol which allows it to skip network setup calls such as socket and accept (§3.2) and (2) its in-memory filesystem, which transforms filesystem operations into in-memory changes (§3.3).

Via binary rewriting, SnapFuzz intercepts each system call, and places the forkserver just before it encounters either a system call that spawns new threads (clone, fork), or one used to receive input from a client. The reason SnapFuzz still has to stop before the application spawns new threads is that the forkserver relies on fork to spawn new instances to be fuzzed, and fork cannot reconstruct existing threads—in Linux, forking a multi-threaded application creates a process with a single thread [15]. As a possible mitigation, we tried to combine SnapFuzz and the pthsem / GNU pth library [33]—a green threading library that provides non-preemptive priority-based scheduling, with the green threads executing inside an event-driven framework—but the performance overhead was too high.

In particular, we used pthsem with LightFTP, as this application has to execute two clone system calls before it accepts input. With pthsem support, SnapFuzz’s forkserver can skip these two clone calls, as well as 37 additional system calls, as now SnapFuzz can place the forkserver just before LightFTP is ready to accept input. However, despite this gain, the overall performance was 10% lower than in the version of SnapFuzz without pthsem, due to the overhead of this library. Ideally, SnapFuzz should implement a lightweight thread reconstruction mechanism to recreate all dead threads, but this is left as future work.

### 3.5 Additional Binary Rewriting-enabled Optimisations

In this section, we discuss several additional optimisations performed by SnapFuzz, which are enabled by its binary rewriting-based architecture. They concern developer-added delays, writes to stdout/stderr, signal propagation, and CPU affinity, and highlight the versatility of SnapFuzz’s approach in addressing a variety of challenges and inefficiencies when fuzzing network applications.

#### 3.5.1 Eliminating developer-added delays

Occasionally, network applications add sleeps or timeouts in order to avoid high CPU utilisation when they poll for new connections or data. SnapFuzz removes these delays via binary rewriting, making those calls use a more aggressive polling model.

We also noticed that in some cases application developers deliberately choose to add sleeps in order to wait for various events. For example, LightFTP adds a one second sleep in order to wait for all its threads to terminate. This might be fine in a production environment, but during a fuzzing campaign such a delay is unnecessary and expensive. SnapFuzz completely skips such sleeps by intercepting and then not issuing this family of system calls at all.

#### 3.5.2 Avoiding stdout/stderr writes

By default, AFL redirects stdout and stderr to /dev/null. This is much more performant than actually writing to a file or any other medium, as the kernel optimizes those operations aggressively. SnapFuzz goes one step further and saves additional time by completely skipping any system call that targets stdout or stderr.

#### 3.5.3 Signal Propagation

Some applications use a multi-process rather than a multi-threaded concurrency model. In this case, if a subprocess crashes with a segfault, the signal might not be propagated properly to the forkserver and the crash missed. We stumbled upon this case with the Dcmqrscp server (§6.5) where a valid new bug was manifesting, but AFLNet was unable to detect the issue as the main process of Dcmqrscp never checked the exit status of its child processes.

As SnapFuzz has full control of the system calls of the target, whenever a process is about to exit, it checks the exit status of its child processes too. If an error is detected, it is raised to the forkserver.

#### 3.5.4 Smart affinity

AFL is designed to work with single-threaded applications that receive input via files, with the fuzzer creating a file and then sending it to the target for processing. Therefore, AFL pins the fuzzer and the target to the same CPU core. SnapFuzz can detect when a new thread or process is about to be spawned as both clone and fork system calls are intercepted. This creates the opportunity for SnapFuzz to take control of thread scheduling.
by pinning threads and processes to available CPUs. SnapFuzz implements a very simple algorithm that pins every newly created thread or process to the next available CPU, and also places the fuzzer on a separate CPU.

4 IMPLEMENTATION

SnapFuzz is implemented on top of AFLNet, and targets the Linux platform. However, the ideas in SnapFuzz could be implemented using other fuzzers and operating systems. Below, we provide implementation details related to binary rewriting (§3.1), our in-memory filesystem (§4.2), and the use of UNIX domain sockets (§4.3).

4.1 Binary Rewriting

Binary rewriting in SnapFuzz employs two major components: 1) the rewriter module, which scans the code for specific functions, vdso and system call assembly opcodes, and redirects them to the plugin module, and 2) the plugin module where SnapFuzz resides.

Rewriter. SnapFuzz is an ordinary dynamically linked executable that is provided with a path to a target application together with the arguments to invoke it with. When SnapFuzz is launched, the expected sequence of events of a standard Linux operating system is taking place, with the first step being the dynamic loader that loads SnapFuzz and its dependencies in memory.

When SnapFuzz starts executing, it inspects the target’s ELF binary to obtain information about its interpreter, which in our implementation is always the standard Linux ld loader. SnapFuzz then scans the loader code for system call assembly opcodes and some special functions in order to instruct the loader to load the SnapFuzz plugin. In particular, the rewriter: (1) intercepts the dynamic scanning of the loader in order to append the SnapFuzz plugin shared object as a dependency, and (2) intercepts the initialisation order of the shared libraries in order to prepend the SnapFuzz plugin initialisation code (in the .preinit_array).

After the SnapFuzz rewriter finishes rewriting the loader, execution is passed to the rewritten loader in order to load the target application and its library dependencies. As the normal execution of the loader progresses, SnapFuzz intercepts its mmap system calls used to load libraries into memory, and scans these libraries in order to recursively rewrite their system calls and redirect them to the SnapFuzz plugin. The SnapFuzz rewriter is based on the open-source load-time binary rewriter SaBRe [1].

Plugin. After the loader completes, execution is passed to the target application, which will start by executing SnapFuzz’s initialisation function. Per the ELF specification, execution starts from the function pointers of .preinit_array. This is a common ELF feature used by LLVM sanitisers to initialise various internal data structures early, such as the shadow memory [36, 38]. SnapFuzz is using the same mechanism to initialise its subsystems like its in-memory filesystem before the execution starts.

After the initialisation phase of the plugin, control is passed back to the target and normal execution resumed. At this stage, the SnapFuzz plugin is only executed when the target is about to issue a system call or a vdso call. When this happens, the plugin checks if the call should be intercepted, and if so, it redirects it to the appropriate handler, and then returns back control to the target.

4.2 In-memory Filesystem

As discussed in §3.3, SnapFuzz redirects all file operations to use a custom in-memory filesystem. This reduces the overhead of reading and writing from a storage medium, and eliminates the need for manually-written clean-up scripts.

SnapFuzz implements a lightweight in-memory filesystem, which uses two distinct mechanisms, one for files and the other for directories. For files, SnapFuzz’s in-memory filesystem uses the recent memfd_create() system call, introduced in Linux in 2015 [26]. This system call creates an anonymous file and returns a file descriptor that refers to it. The file behaves like a regular file, but lives in memory. Under this scheme, SnapFuzz only needs to specially handle system calls that initiate interactions with a file through a pathname (like the open and mmap system calls). All other system calls that handle file descriptors are compatible by default with the file descriptors returned by memfd_create.

When a target application opens a file, the default behavior of SnapFuzz is to check if this file is a regular file (e.g. device files are ignored), and if so, create an in-memory file descriptor and copy the whole contents of the file in the memory address space of the target. SnapFuzz keeps track of pathnames in order to avoid reloading the same file twice. This is not only a performance optimisation but also a correctness requirement, as the application might have changed the contents of the file in memory.

For directories, SnapFuzz employs the Libsqlfs library [17], which implements a POSIX-style file system on top of the SQLite database and allows applications to have access to a full read/write filesystem with its own file and directory hierarchy. Libsqlfs simplifies the emulation of a real filesystem with directories and permissions. SnapFuzz uses Libsqlfs for directories only, as we observed better performance for files via memfd_create.

4.3 UNIX Domain Sockets

AFLNet uses the standard Internet sockets (TCP/IP and UDP/IP) to communicate to the target and send it fuzzed inputs. The Internet socket stack includes functionality—such as calculating checksums of packets, inserting headers, routing—which is unnecessary when fuzzing applications on a single machine.

To eliminate this overhead, similarly to prior work [45], SnapFuzz replaces Internet sockets with UNIX domain sockets. More specifically, SnapFuzz uses Sequenced Packet sockets (SOCK_SEQPACKET). This configuration offers performance benefits and also simplifies the implementation. Sequenced Packets are quite similar to TCP, providing a sequenced, reliable, two-way connection-based data transmission path for datagrams. The difference is that Sequenced Packets require the consumer (in our case the SnapFuzz plugin running inside the target application) to read an entire packet with each input system call. This atomicity of network communications simplifies corner cases where the target application might read only parts of the fuzzer’s input due to scheduling or other delays. By contrast, AFLNet handles this issue by exposing manually defined knobs for introducing delays between network communications.

Our modified version of AFLNet creates a socketpair of UNIX domain sockets with the Sequenced Packet type, and passes one end to the forkserver, which later passes it to the SnapFuzz plugin. The SnapFuzz plugin initiates a handshake with the modified AFLNet,
after which AFLNet is ready to submit generated inputs to the target or consume responses.

Translating network communication from Internet sockets to UNIX domain sockets is not trivial, as SnapFuzz needs to support the two main IP families of TCP and UDP which have a slightly different approach to how network communication is established. In addition, SnapFuzz also needs to support different types of synchronous and asynchronous communication such as (e)poll and select.

For the TCP family, the socket system call creates a TCP/IP socket and returns a file descriptor which is then passed to bind, listen and finally to accept, before the system is ready to send or receive any data. SnapFuzz monitors this sequence of events on the target and when the accept system call is detected, it returns the UNIX domain socket file descriptor from the forkserver. SnapFuzz doesn’t interfere with the socket system call and intentionally allows its normal execution in order to avoid complications with target applications that perform advanced configurations on the base socket. This strategy is similar to the one used by the in-memory file system via the memfd_create system call (§4.2) in order to provide compatibility by default.

The UDP family is handled in a similar way, with the only difference that instead of monitoring for an accept system call to return the UNIX domain socket of the forkserver, SnapFuzz is monitoring for a bind system call.

5 LIMITATIONS

This section summarises the main limitations of SnapFuzz.

As discussed in §3.4, the smart deferred forkserver cannot be placed after the application has spawned threads, because our implementation relies on fork, and in Linux, forking a multi-threaded application creates a process with a single thread [15]. To address this, we envision a lightweight thread reconstruction mechanism to recreate all dead threads, but this is not a trivial task and would require understanding the performance trade-offs involved.

The effectiveness of the smart forkserver is also inhibited by early unimportant network communication, such as a handshake. When during such early communication the target application is the first to send some data, we take a snapshot after this completes; unfortunately, we cannot do so once the fuzzer starts sending input.

SnapFuzz’s smart deferred forkserver is typically ideal for target applications that retrieve data from files early in their execution, just before network communication starts. In this way, the reads have to be executed only once rather than on every fuzzing iteration. However, if the files are large, performance can degrade. For every new forked process spawned from the smart deferred forkserver, SnapFuzz has to memory copy all the in-memory loaded files—as memfd_create uses shared memory to store the in-memory files—in order to avoid different forked processes to interfere with each other. For large files, this might entail a significant overhead. In future work, we plan to investigate approaches for better handling such large files.

SnapFuzz might also subtly change the program behaviour. We discuss three such instances: (1) SnapFuzz is emulating Internet sockets with UNIX domain sockets, and the native file system with an in-memory one. While we strive for maximum compatibility, subtle differences might exist. For example network communication in SnapFuzz uses Sequenced Packet sockets (SOCK_SEQPACKET, see §4.3) that provide atomicity of sending and receiving, which can be violated by Internet sockets. (2) If the snapshot taken by the smart deferred forkserver contains time-sensitive state (such as a certificate with a short time expiration), the state might become invalid when new processes are forked during later fuzzing iterations. (3) The target application might rely on timed waits to explore various corner cases, and with SnapFuzz removing all sleeps, these corner cases might be missed. We have not directly observed such issues in our experiments, but if they occur, they might lead to both false positives and false negatives.

Finally, as discussed in §3.2, SnapFuzz instantly terminates the application when the fuzzer has no more inputs to provide. While this approach increases performance, it may miss bugs in the termination routines.

6 EVALUATION

We demonstrate the benefits of SnapFuzz using five popular servers that were previously used in evaluating AFLNet [30]: Light FTP (§6.4), Dcmqrscp (§6.5), Dnsmasq (§6.6), LIVE555 (§6.7) and TinyDTLS (§6.8). Our experiments show that SnapFuzz significantly improves fuzzing throughput, while at the same time reducing the effort needed to create fuzzing harnesses. As a result of its significant performance benefit, SnapFuzz also found 12 extra crashes compared to AFLNet in these applications.

6.1 Methodology

Our main performance metric is the number of fuzzing iterations per second. Note that each fuzzing iteration may include multiple message exchanges between the fuzzer and the target. A fuzzing campaign consists of a given number of fuzzing iterations.

During a fuzzing campaign, the fuzzer’s speed may vary across iterations, sometimes significantly, due to different code executed by the target. To ensure a meaningful comparison between SnapFuzz and AFLNet, rather than fixing a time budget and counting the number of iterations performed by each, we instead fix the number of iterations and measure the execution time of each system. We monitored standard fuzzing metrics including bug count, coverage, stability, path and cycles completed, to make sure that the SnapFuzz and AFLNet campaigns have the same (or very similar) behaviour.

We chose to run each target for one million iterations to simulate realistic AFLNet fuzzing campaigns (ranging from approximately 15 h to 36 h). We repeated the execution of each campaign 10 times.

6.2 Experimental Setup

All of our experiments were conducted on a 3.0 GHz AMD EPYC 7302P 16-Core CPU and 128 GB RAM running 64-bit Ubuntu 18.04 LTS (kernel version 4.15.0-162) with an SSD disk. Note that using a slower HDD instead of an SSD disk would likely lead to larger gains for SnapFuzz’s in-memory filesystem component.

SnapFuzz is built on top of AFLNet revision 0f51f9e from January 2021 and SaBRe revision 7a94f83. The servers tested and their workloads were taken from the AFLNet paper and repository at the revision mentioned above.

We used the default configurations proposed by AFLNet for all benchmarks, with a couple of exceptions. For the Dcmqrscp server,
Table 1: Average time in minutes, with standard deviation in parenthesis, to complete one million fuzzing iterations in AFLNet vs SnapFuzz, across 10 repetitions.

<table>
<thead>
<tr>
<th></th>
<th>AFLNet</th>
<th>SnapFuzz</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dcmqrscp</td>
<td>1055 (138)</td>
<td>127 (15)</td>
<td>8.4 x</td>
</tr>
<tr>
<td>Dnsmasq</td>
<td>917 (111)</td>
<td>30 (1)</td>
<td>30.6 x</td>
</tr>
<tr>
<td>TinyDTLS</td>
<td>1401 (297)</td>
<td>34 (11)</td>
<td>41.2 x</td>
</tr>
<tr>
<td>LightFTP</td>
<td>2135 (327)</td>
<td>34 (2)</td>
<td>62.8 x</td>
</tr>
<tr>
<td>LIVE555</td>
<td>1547 (218)</td>
<td>63 (2)</td>
<td>24.6 x</td>
</tr>
</tbody>
</table>

Note that `SnapFuzz` can place the forkserver later than it could be placed manually. For the deferred forkserver to work properly, recall that no file descriptor must be open before the forkserver snapshots the target. This is because the underlying resource of a file descriptor is retained after a fork happens. This limits the area where the deferred forkserver can be placed manually. `SnapFuzz` overcomes this challenge with its in-memory file system as described in §4.2 and thus it is able to place the forkserver after the whole initialisation process has finished.

The one million iterations for LightFTP take on average 35 h 35 min under `AFLNet`, while only 34 min under `SnapFuzz`, providing a 62.8 x speedup.

6.5 Dcmqrscp

Dcmqrscp [12] is a multi-threaded DICOM image archive server that manages a number of storage areas and allows images to be stored and queried. The fuzzing harness instructs the DICOM server to echo connection information back to the client, and to store, find and retrieve specific images into and from its database.

Dcmqrscp heavily exercises `SnapFuzz`'s in-memory filesystem as on every iteration the probability to read or create files is high. Dcmqrscp also benefits from the smart deferred forkserver, as it has a long initialisation phase in which the server dynamically loads the `libnss` library and also parses multiple configuration files that dictate the syntax and capabilities of the DICOM language.

Our signal propagation subsystem (§3.5.3) was able to expose a bug in Dcmqrscp which was also triggered by `AFLNet` but was missed because signals were not properly propagated.

The one million Dcmqrscp iterations take on average 17 h 35 min to execute under `AFLNet`, while only 2 h 7 min under `SnapFuzz`, providing a 8.4 x speedup.

6.6 Dnsmasq

Dnsmasq [13] is a single-threaded DNS proxy and DHCP server designed to have a small footprint and be suitable for resource-constrained routers and firewalls. The fuzzing harness instructs Dnsmasq to query various bogus domain names from its configuration file and then report results back to its client.

Dnsmasq is an in-memory database with very little interaction with the filesystem. Therefore, it doesn’t benefit from the in-memory filesystem, but it profits from the `SnapFuzz` protocol and the optimisations of §3.5. Furthermore, it highly benefits from the smart deferred forkserver, as it has a long initialisation process: Dnsmasq performs approximately 1,200 system calls before it is ready to accept input.

As for other benchmarks, a manually-placed forkserver under `AFLNet` could not snapshot the application at the same depth as `SnapFuzz`’s smart deferred forkserver. This is because Dnsmasq needs to execute a sequence of system calls to establish a network connection with `AFLNet`. This sequence includes creating a socket, binding its file descriptor, calling `listen`, executing a `select` to check for incoming connections, and finally accepting the connection. Therefore, under `AFLNet`, the latest possible placement of the forkserver would be just before this sequence. Under `SnapFuzz`, network communications are translated into UNIX domain socket communications that don’t require any of the above, and thus the
smart deferred forkserver can snapshot the target right before reading the input from the fuzzer, saving a lot of initialisation time.

The one million Dnsmasq iterations take on average 15 h 17 min under AFLNet, while only 30 min under SnapFuzz, providing a 30.6 x speedup.

6.7 LIVE555

LIVE555 [24] is a single-threaded multimedia streaming server that uses open standard protocols like RTP/RTCP, RTSP and SIP. The fuzzing harness instructs the LIVE555 server to accept requests to stream the content of a specific file, and the server replies to these requests with information and the actual streaming data.

LIVE555 only reads files and thus no state reset script is required. It has a relatively slim initialisation phase with the main overhead coming from the many writes to stdout with welcoming messages to users. LIVE555 benefits from the SnapFuzz protocol and the elimination of stdout writes.

LIVE555 reads its files only after the forkserver performs its snapshot. As a result, those files are not kept in the in-memory filesystem of SnapFuzz, and are read from the actual filesystem in each iteration. We leave as future work the optimisation of pre-defining a set of files to be loaded in the in-memory file system when the smart deferred forkserver kicks in, so the target could read these files from memory rather than the actual file system.

The one million LIVE555 iterations take on average 25 h 47 min under AFLNet, while only 63 min under SnapFuzz, providing a 24.6 x speedup.

6.8 TinyDTLS

TinyDTLS [40] is a DTLS 1.2 single-threaded UDP server targeting IoT devices. In the fuzzing harness, TinyDTLS accepts a new connection and then the DTLS handshake is initiated in order for communication to be established.

The protocol followed by AFLNet has several steps, and progress to the next step is accomplished either by a successful network action or after a timeout has expired. TinyDTLS supports two cipher suites, one Elliptic Curve (EC)-based, the other Pre-Shared Keys (PSK)-based. EC-based encryption is slow, requiring the use of a full buffer delivery from and to the target—without these delays, network packets might be dropped and AFLNet’s state machine desynchronised. Due to TinyDTLS’s processing delays, network buffers might fill up if AFLNet sends too much data in a short time period. To deal with this, AFLNet checks on every send and receive if all the bytes are sent, and retries if not.

SnapFuzz handles all these issues through its network fuzzing protocol. (We also note that TinyDTLS exercises SnapFuzz’s UDP translation capabilities, unlike the other servers which use TCP.)

The end result is that all these delays are eliminated: AFLNet doesn’t need to guess the state of the target anymore, as SnapFuzz explicitly informs AFLNet about the next action of the target. Similarly, the issue of dropped packets disappears, as AFLNet is always informed when it is the right time to send more data. Finally, SnapFuzz’s UNIX domain sockets eliminate the need for send and receive retries, as full buffer delivery from and to the target is guaranteed by the domain socket protocol. TinyDTLS writes a lot of data to stdout, so it also benefits from SnapFuzz’s ability to skip these system calls.

The one million TinyDTLS iterations take on average 23 h 21 min under AFLNet, while only 34 min under SnapFuzz, providing a 41.2 x speedup.

We remind the reader that in TinyDTLS we decided to decrease the manually-added inter-request time delay from 30ms to 2ms, as we noticed the performance of AFLNet was seriously affected by it. Without this change, AFLNet would take significantly longer to complete one million iterations.

6.9 Performance Breakdown

In §6.4–§6.8 we discuss which components of SnapFuzz are likely to benefit each application the most. Those conclusions were reached by investigating the system calls issued by the applications, using the estimates provided by strace about how much each system call takes in the kernel. To have a better understanding of the contribution of each components, we perform an ablation study in which we run different versions of SnapFuzz for a short number of 10k iterations. We chose a much smaller number of iterations because running so many experiments with 1m iterations was prohibitive on our computing infrastructure. This means that our speedups sometimes differ from those achieved by 1m iterations. However, the main goal of these experiments is to gain additional insights into the impact of different components and their interaction.

Due to various dependencies among components, we start with a version of SnapFuzz containing only the network fuzzing protocol, and keep adding components one by one. However, it is important to understand that the order in which we add components matters, as their effect is often multiplicative. In particular, this means that the additional impact of components added earlier can be significantly diminished compared to the case where the same component is added later. We give two examples:

1. **SnapFuzz protocol and smart affinity.** The SnapFuzz protocol is a performant non-blocking protocol that polls the fuzzer and the application for communication. Under the default restricted CPU affinity of AFLNet, the protocol is underperforming, because the polling model requires independent CPU cores to get the expected performance benefit. At the same time, the smart CPU affinity component depends on whether the SnapFuzz protocol is enabled or not, as the protocol changes what is executed on the CPU.

2. **In-memory filesystem and smart deferred forkserver.** The smart deferred forkserver performs better when the in-memory filesystem is enabled, because with an in-memory filesystem it can delay the forkserver past filesystem operations. On the other hand, the in-memory filesystem also performs better when the smart deferred forkserver is enabled. This is because the in-memory filesystem has a fixed overhead of loading and storing the files the target is reading in the beginning of its execution. This initial overhead might degrade performance, especially for short executions. When the deferred forkserver is enabled, this overhead is bypassed, as these files are loaded only once in memory and consecutive operations will be only in-memory.

One option would be to try all possible orderings. However, the full number is large (6! = 720) and some orderings are difficult to
run due to engineering limitations (e.g., the SnapFuzz protocol is deeply embedded into SnapFuzz and disabling it would require a major engineering overhaul). Nevertheless, we believe the ordering we present here is still useful in providing insights into the impact of each SnapFuzz component.

Table 2 shows our results. Note that all components have a significant impact on at least one benchmark. Furthermore, the SnapFuzz protocol, the smart affinity, and the smart deferred forkserver always lead to gains, while eliminating developer-added delays (no sleeps), avoiding stdout/stderr writes (no stdout) and the in-memory file system make no difference in some benchmarks. Removing writes to stdout/stderr is the least impactful component, benefiting only LIVE555. Note that while the smart affinity has the highest overall gains, these are partly due to the fact that the SnapFuzz protocol is enabled. As explained above, the smart affinity makes the polling model used by the SnapFuzz protocol work efficiently.

The performance numbers for other benchmarks also largely agree with our expectations. For instance, the main benefits of LightFTP come from the SnapFuzz protocol (1.90 x) which removes synchronisation and server termination delays; from smart affinity (1.79 x), especially since LightFTP is multi-threaded; from removing developer-added delays, which are present in LightFTP (2.76 x); from the smart deferred forkserver (2.39 x), as it has a long initialisation phase; and from the in-memory filesystem (2.23 x), as it makes heavy use of the filesystem. While LightFTP has writes to stdout, removing them does not make a noticeable difference.

The performance numbers for other benchmarks also largely agree with our expectations. For instance, the in-memory filesystem brings no benefits to Dnsmasq, which is an in-memory database with little filesystem interaction; but it highly benefits from the smart deferred server (4.79 x), given that it has a long initialisation with over 1,200 system calls issued before it is able to accept input.

Finally, all benchmarks required code modifications to add support for the deferred forkserver. As discussed in §3.4, locating the optimal position in the target’s code can be difficult and error-prone. By contrast, SnapFuzz places the smart deferred forkserver automatically, and often at a later stage than it would have been possible with AFLNet.

### 6.11 Detected Crashes

In addition to the experiments above, we have run both SnapFuzz and AFLNet for 24 h on each benchmark, with three repetitions. We then accumulated all discovered crashes in a single repository. To deduplicate the crashes found, we recomplied all benchmarks under Asan and UBSan, and then grouped the crashing inputs based on the reports from the sanitisers. SnapFuzz, as expected, was able to find all the crashes discovered by AFLNet. Due to its advantages, it also found additional crashes in 3 of the 5 benchmarks. More precisely, it found 4 crashes in the Dcmqrs scp benchmark while AFLNet was not able to find any. In Dnsmasq, SnapFuzz found 7 crashes, while AFLNet found only 1, and in LIVE555 it found 4 crashes, while AFLNet only 2. Both tools found 3 crashes in TinyDTLS. Overall, SnapFuzz found 18 deduplicated crashes, 12 more than AFLNet. (But note that with the exception of the crash discussed in §3.5.3, we expect AFLNet to find the other 11, but after a significantly longer time.)

The crashes are caused by heap overflows, stack overflows, use-after-free bugs, and other types of undefined behaviours. Fortunately, they seem to have been fixed in the latest versions of these applications. We plan to rerun SnapFuzz on the latest versions.

### 7 RELATED WORK

SnapFuzz focuses on creating an efficient fuzzing platform for network applications and helps algorithmic research to be built on top of a strong foundation. SnapFuzz builds on top of AFLNet [30], and reuses its ability to infer network protocols. However, AFLNet has various inefficiencies and requires fragile manual delays and clean-up scripts in its fuzzing harnesses. Our comprehensive evaluation against AFLNet shows how SnapFuzz can address both problems, resulting in impressive speedups in the range of 8.4 x-62.8 x.

Besides AFLNet, a popular way of fuzzing network applications is via the de-socketing functionality of Preeny [32]. Preeny intercepts networking functions such as connect and accept and makes them return sockets that are synchronised with stdin and stdout, essentially allowing AFL to continue to fuzz files and redirecting their contents over network sockets, as expected by the network applications being tested. Synchronisation is done in a hacky way: Preeny...
implements a small server thread that is continuously polling AFL’s generated input file and then forwards the read data to the appropriate network calls through a UNIX domain socket to the target [31]. While a direct comparison with AFLNet and SnapFuzz is not easily possible because a meaningful fuzzing campaign requires the network protocol inferred by AFLNet, we expect a rewrite of AFLNet on top of Preeny to be slower than vanilla AFLNet, due to the extra overhead imposed by file-based fuzzing and the additional thread server used by Preeny.

We are also aware of other tools for fuzzing network applications, such as jbirdwell/afl [19] and WinAFL [42]. These tools have not been published, so trying to understand their operation by using their documentation or reverse-engineering the code is difficult. Nevertheless, by briefly doing so, our understanding is that they use a similar architecture to AFLNet, based on Internet sockets and custom delays. So we would expect their performance to be comparable to that of AFLNet. Their documentation also confirms our understanding of the code. For instance, WinAFL discusses that: “this mode [network fuzzing] is considered as experimental since we have experienced some problems with stability and performance.” These performance problems are likely due to the AFLNet-style architecture observed in the code. As a second example, jbirdwell/afl discusses that: “The user needs to experimentally determine a timeout delay (in milliseconds) that produces a sufficiently low percentage of hangs (exits forced by expiration of the delay) while allowing the input to the target from afl-fuzz to be completely processed.” This type of manual effort and experimentation is one of the key problems that SnapFuzz solves.

MultiFuzz [45] presents a more advanced de-socketing library called Desockmulti, which is similar to Preeny, but optimised in various ways, e.g., by removing the use of threads and adding the ability to initiate multiple connections to the target. MultiFuzz is specifically designed for publish/subscribe protocols and the evaluation does not include the benchmarks used by AFLNet and us. For the two benchmarks used, libcoap and Mosquitto, the paper reports throughput increases of two to three orders of magnitude on top of AFLNet. We expect SnapFuzz to perform even better due to its network fuzzing protocol, smart deferred forkserver and its memory file system (MultiFuzz uses tmpfs, see §3.3) but unfortunately, MultiFuzz is not available as open source (only its Desockmulti library is available), so a direct comparison is not possible.

Nyx-Net [35] uses hypervisor-based snapshot fuzzing with selective emulation of network functionality to handle network traffic. While Nyx-Net achieves impressive speedups similar to SnapFuzz, its architecture is fundamentally different. Nyx-Net requires a custom kernel module, a modified version of QEMU and KVM, and also a custom VM build in which the target applications are executed. Nyx-Net also implements a custom networking layer that emulates some POSIX network functionality which currently does not support complex network targets. While Nyx-Net is able to support more advanced fuzzing use cases like fuzzing hypervisors, SnapFuzz’s user-mode approach avoids many layers of additional complexity.

Xu et al. [43] propose new operating systems primitives for fuzzing. These include, for instance, a new snapshot system call, which aims to address the same goal as SnapFuzz with respect to efficiently snapshotting the target. As for Nyx-Net, the main disadvantage of this approach is that it requires kernel support; by contrast, SnapFuzz runs in user mode, using an unmodified OS.

Most work on testing network protocol implementations has targeted algorithmic rather than platform-level improvements, focusing in particular on inferring network protocol implementa-
tions [3, 8, 11, 30, 44]. This work is orthogonal to SnapFuzz and could be combined with it, as we have done with AFLNet’s protocol inference algorithm. More broadly, greybox fuzzing is an active area of research [5] with recent work on improving its effectiveness by directing exploration toward interesting program parts [6, 7], combining it with symbolic execution [10, 29, 39], inferring input grammars [2, 41] or specialising it to various application domains [20, 22, 46].

Besides greybox fuzzing, other forms of fuzzing have been used to test network applications, such as blackbox fuzzing [4, 14, 16], fault injection [25, 28] and symbolic execution [9, 34, 37].

8 CONCLUSION

Fuzzing stateless applications has proven extremely successful, with hundreds of bugs and security vulnerabilities being discovered. Recently, in-depth fuzzing of stateful applications such as network servers has become feasible, due to algorithmic advances that make it possible to generate inputs that follow the application’s network protocol. Unfortunately, fuzzing such applications requires clean-up scripts and manually-configured time delays that are error-prone, and suffers from low fuzzing throughput. SnapFuzz addresses these challenges through a robust architecture, which combines a synchronous communication protocol with an in-memory filesystem and the ability to delay the forkserver to the latest safe point, as well as other optimisations. As a result, SnapFuzz simplifies fuzzing harness construction and improves the fuzzing throughput significantly, between 8.4 x and 62.8 x on a set of popular network applications, allowing it to find additional crashes.

SnapFuzz is made available to the community as open-source, with the hope that it will help improve the security and reliability of network applications and facilitate further research in this space: https://srg.doc.ic.ac.uk/projects/snapfuzz/.

Acknowledgements. We would like to thank the AFLNet authors, Van-Thuan, Marcel and Abhik, for making their excellent system available to the community and for answering our questions. We would also like to thank Frank Busse for his feedback on the paper. This research has received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme (grant agreement 819141) and from the Engineering and Physical Sciences Research Council (EPSRC) under the HiPEDS Centre for Doctoral Training.

REFERENCES


[23] LightFTP repository 2022. https://github.com/hife0x6/LightFTP.


