SnapFuzz: An Efficient Fuzzing Framework for 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 their low fuzzing throughput and the need to develop fuzzing harnesses that reset their state and isolate their side effects.

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 based on UNIX domain sockets, speeds up all file operations by redirecting them to an in-memory filesystem, and removes the need for many fragile modifications, such as configuring time delays or writing cleanup scripts, together with several other improvements.

Using *SnapFuzz*, we fuzzed five popular networking applications: LightFTP, Dnsmasq, LIVE555, TinyDTLS and Dcmqrscp. We report impressive performance speedups of 72.4x, 49.7x, 24.8x, 23.9x, and 8.5x, respectively, with significantly simpler fuzzing harnesses in all cases. Through its performance advantage, *SnapFuzz* has also found 12 previously-unknown crashes in these applications.

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 [1].

Unfortunately, not all software can benefit from such fuzzing campaigns. One important class of software, networking protocol implementations, are 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 [25]. 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* [31], 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 cleanup 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 thorough a robust architecture that transforms slow asynchronous network communication into fast synchronous communication based on UNIX domain sockets, speeds up file operations and removes the need for cleanup scripts via an in-memory filesystem, and improves several other aspects such as delaying and automating the placement of the forkserver, improving signal propagation and server exit 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 8.5x to 72.4x (mean: 24.8x) 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 [28] 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 covers a new state, 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 Linux process is taking place, with the loader being the first step that loads and initializes the target application in 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

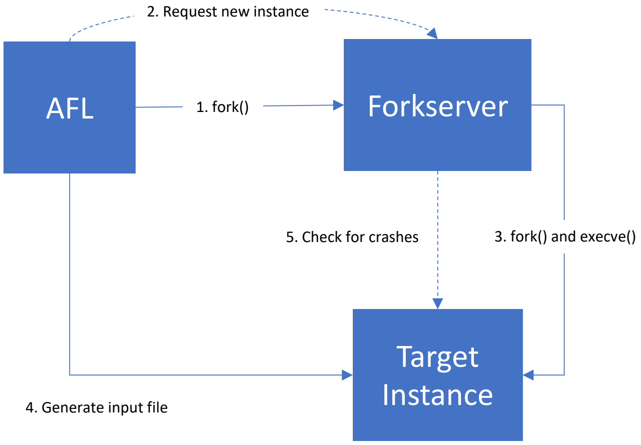

Figure 1: Architecture of *AFL*'s forkserver mode.

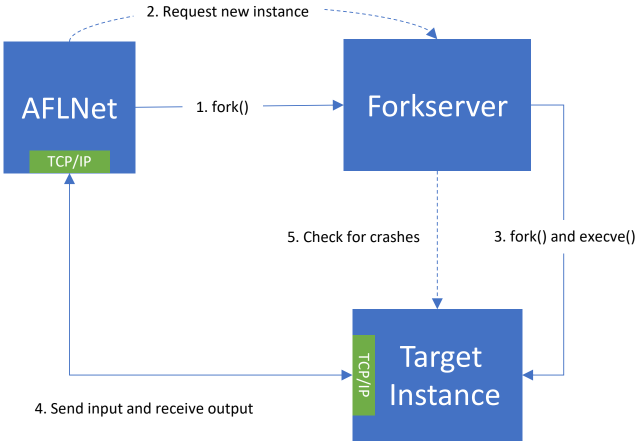

Figure 2: Architecture of *AFLNet*.

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, initialising libc and the target, and exiting the target application 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 and receives back the output (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*. In the most recent versions of *AFL*, this is implemented as an LLVM pass, but other methods that do not require access to the source code are also available.

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 the internals of the target application in order to place the deferred call at the correct stage of execution. As we will explain later, 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 type of file). This means that it is not directly applicable to network applications, as they expect inputs to arrive through network sockets and to follow an underlying *network protocol*.

AFLNet [31] 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 initialize, 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). *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 *cleanup 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 cleanup scripts. One important

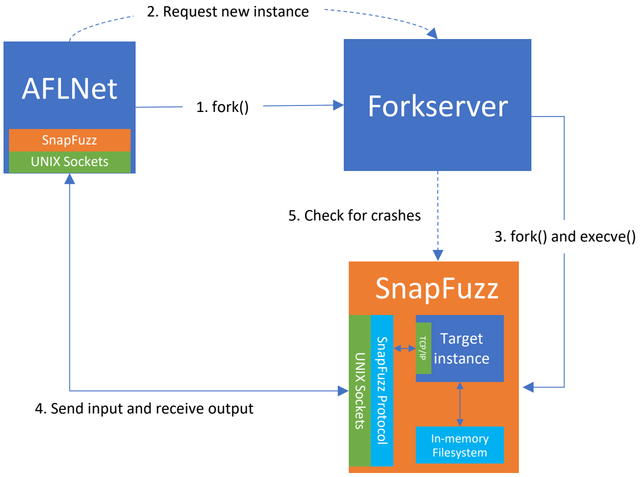

Figure 3: Architecture of *SnapFuzz*.

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 networking communication architecture as shown in Figure 3, without any modifications to *AFLNet*'s fuzzing algorithm.

SnapFuzz's main goals 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 cleanup scripts. At the same time, it is not a goal of *SnapFuzz* to improve in any way *AFL*'s and *AFLNet*'s fuzzing algorithms or mutation strategies.

At a high level, *SnapFuzz* achieves its significant performance gains through optimising filesystem writes by redirecting them into an in-memory filesystem; automatically injecting *AFL*'s forkserver deeper into the application than otherwise possible and without the user's intervention; automatically resetting any filesystem state; and optimising all networking communications by eliminating synchronisation delays.

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* monitors both the behaviour of the target application and of the AFLNet client, in order to eliminate synchronisation delays. Second, Internet sockets are replaced by UNIX domain sockets. Finally, an in-memory filesystem is added, to improve performance and facilitate resetting the state after each fuzzing iteration. We describe *SnapFuzz*'s main components in more detail in the next section.

3 DESIGN AND IMPLEMENTATION

SnapFuzz has two main goals: significantly increase fuzzing throughput, and simplify the construction of fuzzing harnesses.

At a high-level, *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) Redirect all file operations to use an in-memory filesystem (§3.2). This improves the performance of these 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.
(2) Replace internet sockets by UNIX domain sockets, which improves performance, as UNIX domain sockets are faster (§3.5).
(3) Notify the fuzzer when the target application is ready to accept a new request or when a response is ready to be consumed (§3.6). This eliminates all the custom delays that users need to insert in order to synchronise the communication between the fuzzer and the target application. This improves fuzzing throughput and also makes it easier to write fuzzing harnesses, as users don't need to worry about these delays anymore.
(4) Eliminate custom delays, unnecessary system calls and potentially expensive clean-up routines that are part of the target application (§3.7.1). This improves the fuzzing throughput, with the only downside of missing bugs in the clean-up routines themselves.
(5) Automatically defer the forkserver to the latest point at which it is safe to start it (§3.3). This eliminates the need for manual annotations and achieves the best possible performance.
(6) Propagate signals from child processes (§3.7.3). This ensures no crashes are missed during fuzzing.
(7) Pin threads and processes to available CPUs (§3.7.5). This improves performance, without the need for any user intervention.

SnapFuzz is implemented on top of *AFLNet*, and targets the Linux platform. As such, it makes use of various OS facilities provided by Linux. However, *SnapFuzz* could in principle be implemented on other platforms too.

3.1 Binary Rewriting

SnapFuzz implements a load-time binary rewriting subsystem that dynamically intercepts both the 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 optimization, 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*. The rest of this subsection presents how this interception is realised and can be skipped by readers less interested in the technical details involved.

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.

3.1.1 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 are 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. At this stage, all system calls and some specific loader functions are monitored. 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 [14].

3.1.2 Plugin. The *SnapFuzz* plugin implements a simple API that exposes three important functions: (1) An initialisation function that will be the first code to run after loading is done (this function is prepended dynamically to the *.preinit_array* of our target's ELF binary), (2) a function to which all system calls will be redirected, which we call the *system call handler*, and (3) a similar function to which all vDSO calls will be redirected, the *vDSO handler*.

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 sanitizers to initialise various internal data structures early, such as the shadow memory [33, 35]. *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 application and normal execution is 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 further redirects it to the appropriate handlers, after which control is returned back to the target.

The *SnapFuzz* plugin is also responsible to handle and guard against recursive calls and vDSO. For example, the plugin itself is allowed to issue system calls through the use of the target's *libc*. To achieve this, *SnapFuzz*'s plugin is guarding every jump from the target application to the plugin with a thread local flag.

3.2 In-memory filesystem

As mentioned before, *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 to be run after each fuzzing operation, as the clean-up can be done automatically by simply discarding the in-memory state.

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 [8]. 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 application through the memory file descriptor. *SnapFuzz* keeps track of pathnames in order to avoid reloading the same file twice. This is not only a performance optimization but also a correctness requirement, as the application might have changed the contents of the file in memory.

Implementing an in-memory filesystem from scratch with anonymous mapping through *mmap* and rewriting all I/O system calls to become function calls operating on the in-memory files would be even more efficient than the current *SnapFuzz* implementation which is still issuing regular system calls and thus paying the associated context-switch overhead. But developing such a subsystem that is compatible with the large diversity of system call options available on Linux is a laborious and difficult task, which is why we have opted for a `memfd_create`-based approach.

For directories, *SnapFuzz* takes advantage of the *Libsqlfs* library [5], 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*.

With the in-memory filesystem in-place, *SnapFuzz* can already implement two important optimizations: a smart deferred forkserver (§3.3) and an efficient state reset (§3.4).

3.3 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 of a binary, 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. This is made possible through *SnapFuzz*'s binary rewriting mechanism together with its in-memory filesystem and custom network communication mechanism.

There are two reasons for which *SnapFuzz* can place the forkserver after many system calls which normally would have caused problems: (1) its use of an in-memory filesystem in the case of file operations, as it transforms external side effects into in-memory changes; and (2) its custom network communication mechanism which allows it to skip network setup calls such as `socket` and `accept` (see §3.5).

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 [4]. As a possible mitigation, we tried to combine *SnapFuzz* and the *pthsem / GNU pth* library [12]—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.

3.4 Efficient state reset

To use *AFLNet*, users typically have to write a clean-up script to reset the application state after each iteration. For instance, LightFTP under *AFLNet* requires a Bash script to be invoked in every iteration in order to clean up any directories or files that have been created in the previous iteration. Under *SnapFuzz*, there is no need for such a cleanup script, which simplifies the test harness construction, and improves performance by avoiding the invocation of the cleanup script.

In the simplest case where *AFL* checkpoints the target application before `main`, no filesystem modifications have happened at the point where the forkserver is placed. So when a fuzzing iteration has finished, the target application process just exits and the OS discards its memory, which includes any in-memory filesystem modifications made during fuzzing. Then, when the forkserver spawns a new instance of the target application, the filesystem is brought back to a state where all initial files of the actual filesystem are unmodified.

The situation is more complicated when the deferred forkserver is placed after the target application has already created some files. When the forkserver creates a new instance to be fuzzed, the Linux kernel shares the memory pages associated with the newly-created in-memory files between the new instance and the forkserver. This is problematic for *SnapFuzz*, 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 forkserver 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 forkserver's checkpoint, as it intercepts all system calls, including `fork`. While the target application executes before the forkserver's checkpoint, *SnapFuzz* allows all file interactions to be handled normally. When a new instance is requested from the forkserver, *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.

3.5 UNIX domain sockets

AFLNet uses the standard Internet sockets (TPC/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, *SnapFuzz* replaces Internet sockets with UNIX domain sockets. More specifically, *SnapFuzz* uses Sequenced Packets 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 Packets type, and passes one

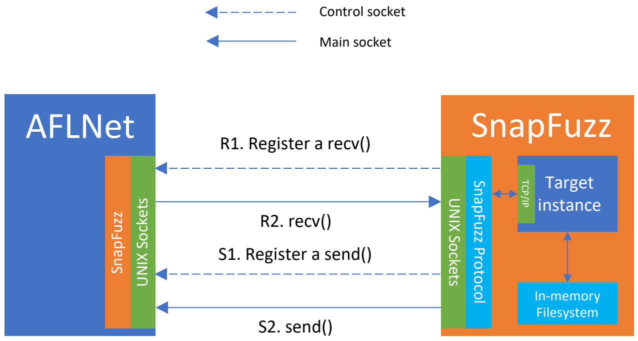

Figure 4: Messages exchanged for each `recv` and `send`.

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 networking communications 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 the different synchronous and asynchronous network communication system calls like `poll`, `epoll` 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 (§3.2) 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.

SnapFuzz also needs to handle asynchronous networking communications as the base socket will not receive any prompt for incoming connections. *SnapFuzz* intercepts all asynchronous system calls and the first time the base socket is passed to an asynchronous system call like `poll`, it sets its state to be notified. This informs the target application that an incoming connection is waiting to be handled, making it progress to execute an `accept` system call.

In summary, *SnapFuzz*'s use of UNIX domain sockets provides two key advantages: it simplifies the construction of test harnesses, which don't need to specify fragile delays between network communications anymore; and speeds up fuzzing by eliminating these delays and the unnecessary overhead of Internet sockets.

3.6 Eliminating protocol-related 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 protocol-related 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 Linux kernel will reject packets that come too fast while the target is not ready, and *AFLNet* will desynchronize 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 by notifying *AFLNet* about the next action of the target. It does this by introducing a second UNIX domain socket, namely the *control socket*, which is used as a send-only channel from the *SnapFuzz* plugin to *AFLNet* to inform the latter if the target will next execute a `recv` or a `send` system call, as shown in Figure 4.

In summary, *SnapFuzz* makes it possible to remove three important communication delays, which simplifies fuzzing harnesses and leads to significant performance gains, as we will show in the evaluation.

3.7 Additional optimizations

In this section, we discuss several additional optimizations performed by *SnapFuzz*. They concern developer-added delays, writes to `stdout/stderr`, signal propagation, server termination and thread pinning, and highlight the versatility of *SnapFuzz*'s approach in addressing a variety of challenges and inefficiencies when fuzzing network applications.

3.7.1 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, 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.

3.7.2 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`.

3.7.3 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 (§4.5) 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.

3.7.4 Efficient server exit. Network servers usually run in a loop. This loop is terminated either via a special protocol-specific keyword or an OS signal. Since *AFLNet* cannot guarantee that each fuzzing iteration will finish via a termination keyword, if the target does not terminate, it sends it a `SIGTERM` signal and waits for it to terminate.

Signal delivery is slow and also servers might take a long time to properly terminate execution. In the context of fuzzing, proper termination is not so important, while fuzzing throughput is. *SnapFuzz* implements a simple mechanism to terminate the server: when it receives an empty string, this signals that the fuzzer has no more inputs to provide and the application is instantly killed. This obviously has the downside that it could miss bugs in the termination routines, but these could be tested separately.

3.7.5 Smart affinity. *AFL* assumes that its targets are single-threaded and thus tries to pin the fuzzer and the target to two free CPUs. Unfortunately, there is no mechanism to handle multi-threaded applications, other than just turning off *AFL*'s pinning mechanism. *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.

4 EVALUATION

We demonstrate the benefits of *SnapFuzz* using five popular servers that were previously used in evaluating *AFLNet* [31]: LightFTP (§4.4), Dcmqrscp (§4.5), LIVE555 (§4.7) and TinyDTLS (§4.8). Our experiments show that *SnapFuzz* can significantly improve fuzzing throughput, while at the same time reducing the effort needed to create fuzzing harnesses. *SnapFuzz* has also demonstrated the ability to discover previously-unknown bugs in applications that were subjected to fuzzing campaigns in the past, due to its significant performance benefit.

4.1 Methodology

Since *SnapFuzz*'s contribution is in increasing the fuzzing throughput, our main comparison 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 16 to 36 hours). We repeated the execution of each campaign three times and report averages.

For bug finding, we left *SnapFuzz* to run for 24 hours, three times for each benchmark. We then accumulated all discovered crashes in a single repository. To uniquely categorise the crashes found, we recompiled all benchmarks under *ASan* and *UBSan*, and then grouped the crashing inputs based on the reports from the sanitizers.

4.2 Experimental Setup

All of our experiments were conducted on a 3.50 GHz Intel Xeon E3-1280 CPU (4 physical cores, 8 logical) and 16 GB RAM running 64-bit Ubuntu 18.04 LTS (kernel version 5.4.0-67) with spinning disks.

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 few exceptions. For the Dcmqrscp server, two changes were required: 1) we had to include a cleanup Bash script to reset the state of a data directory of the server, and 2) we had to add a wait time between requests of 5 milliseconds as we observed *AFLNet* to desynchronise from its target.

These changes further emphasise the fact that the cleanup scripts and delays that users need to specify when building a fuzzing harness are fragile and may need adjustment when using different machines, thus *SnapFuzz*'s ability to eliminate their need is important.

In TinyDTLS we have decided to decrease the wait time between requests from 30 to 2 milliseconds, as we noticed the performance of *AFLNet* was seriously suffering due to this large delay. Again, this shows that choosing the right values for these time delays is difficult.

Table 1: Time (in minutes) to complete one million fuzzing iterations in AFLNet vs Snapfuzz.

	AFLNet	*SnapFuzz*	Speedup
Dcmqrscp	1081	128	8.5x
Dnsmasq	944	19	49.7x
TinyDTLS	1196	50	23.9x
LightFTP	2172	30	72.4x
LIVE555	1312	53	24.8x

4.3 Summary of Results

Table 1 shows a summary of the results. In particular, it compares the average time needed by *AFLNet* and by *SnapFuzz* to complete one million iterations. As can be seen, *AFLNet* takes between 15 hours 44 minutes to 36 hours 12 minutes to complete these iterations, with *SnapFuzz* taking only a fraction of that time, between 50 minutes and 2 hours 8 minutes. The speedups are impressive in each case, varying between 8.5x for Dcmqrscp and 72.4x for LightFTP.

4.4 LightFTP

LightFTP [6] is a small server for file transfers that implements the FTP protocol. The fuzzing harness instructs LightFTP to log in a specific user, list the contents of the home directory on the FTP server, create directories, and execute various other commands for system information.

LightFTP exercises a large set of *SnapFuzz*'s subsystems. First, it heavily utilises the filesystem, as the probability to create directories is quite high on every iteration. Second, it has verbose logging and writing to `stdout`. Third, it has a long initialisation phase, because it parses a configuration file and then undergoes a heavyweight process of initialising x509 certificates. And lastly, LightFTP is a multi-threaded application and has a hardcoded sleep to make sure that all of its threads have terminated gracefully.

SnapFuzz optimises all the above functionalities. All directory interactions are translated into in-memory operations, thus avoiding context switches and device (hard drive) overheads. *SnapFuzz* cancels `stdout` and `stderr` writes. *SnapFuzz*'s smart deferred forkserver snapshots the LightFTP server after its initialisation phase and thus fuzzing under *SnapFuzz* pays the initialisation overhead only once. And lastly, *SnapFuzz* cancels any calls to `sleep` and similar system calls.

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 §3.2 and thus it is able to place the forkserver after the whole initialisation process has finished.

The one million iterations run for LightFTP take on average 36 hours 12 minutes under *AFLNet*, while only 30 minutes under *SnapFuzz*, providing a 72.4x speedup. We observed identical coverage statistics and stability numbers.

4.5 Dcmqrscp

Dcmqrscp [2] is 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 quite high. Dcmqrscp has a long initialisation phase because the server has to load and parse multiple configuration files that dictate the syntax and capabilities of the DICOM language, and the server also executes a *dlopen()* to dynamically load the *libnss* library.

Furthermore, Dcmqrscp is a multi-process server, which poses a unique challenge to *AFLNet*. As discussed in §3.7.3, if a parent process receives a SIGSEGV signal from its child and does not manually raise again the signal error, then *AFLNet* won't be able to detect the crash.

The Dcmqrscp benchmark benefits from (1) *SnapFuzz*'s in-memory file system, as all file interactions become memory operations, (2) *SnapFuzz*'s smart deferred forkserver which makes Dcmqrscp's initialization phase execute only once, and (3) better signal handling which enables it to catch crashes of child processes, as described above.

The one million Dcmqrscp iterations take on average 18 hours 1 minute to execute under *AFLNet*, while only 2 hours 8 minutes under *SnapFuzz*, providing a 8.5x speedup. We observed identical coverage statistics and stability numbers.

Our signal propagation subsystem (§3.7.3) was able to expose a bug in Dcmqrscp which was also triggered by *AFLNet* but was missed because signals were not properly propagated. We debugged the crash with *ASan* and determined that the bug was due to a message header reporting a much larger size than that of the actual message. This bug was already fixed on the latest Dcmqrscp version, via commit `48d00326a03d0`.

4.6 Dnsmasq

Dnsmasq [3] 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 mainly benefits from the networking subsystems of *SnapFuzz* and its additional optimizations of §3.7. Furthermore, it highly benefits from the smart deferred forkserver, as it has a long initialisation process which uses *dlopen()* and performs various network-related configurations. Dnsmasq requires approximately 1,200 system calls before the process is ready to accept input.

As for other benchmarks, manually adding support for the deferred forkserver would not be able to 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 networking 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. Because of all these, 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, which saves a lot of initialisation time.

The one million Dnsmasq iterations take on average 15 hours 44 minutes under *AFLNet*, while only 19 minutes under *SnapFuzz*, providing a 49.7x speedup. As expected, we observed identical coverage statistics, bug counts (these are the bugs already reported by *AFLNet*), and stability numbers.

4.7 LIVE555

LIVE555 [7] 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 serve the content of a specific file in a streaming fashion, 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 mainly benefits from *SnapFuzz*'s networking subsystem 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 predefining 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 the actual filesystem.

The one million LIVE555 iterations take on average 21 hours 52 minutes under *AFLNet*, while only 53 minutes under *SnapFuzz*, providing a 24.8x speedup. As expected, we observed identical coverage statistics, bug counts and stability numbers.

4.8 TinyDTLS

TinyDTLS [13] is a DTLS 1.2 single-threaded UDP server targetting 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 Eliptic Curve (EC)-based, the other Pre-Shared Keys (PSK)-based. EC-based encryption is slow, requiring the use of a large timeout between requests, which slows down fuzzing with *AFLNet* considerably.

In addition, *AFLNet* includes some hardcoded delays between network interactions so that it doesn't overwhelm the target—without these delays, network packets might be dropped and *AFLNet*'s state machine will get desynchronized.

Due to TinyDTLS's processing delays, network buffers might fill up if *AFLNet* sends too much data within 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.

Table 2: Number of unique crashes discovered by *AFLNet* and *SnapFuzz* during 24-hour fuzzing campaigns.

	AFLNet	*SnapFuzz*
Dcmqrscp	0	4
Dnsmasq	1	7
TinyDTLS	3	3
LightFTP	0	0
LIVE555	2	4
Total	6	18

SnapFuzz handles all these issues through its use of UNIX domain sockets and its ability to eliminate protocol-related delays. 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 disappeared, 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.

We also note that TinyDTLS exercises *SnapFuzz*'s UDP translation capabilities, unlike the other servers which are based on TCP.

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 19 hours 56 minutes under *AFLNet*, while only 50 minutes under *SnapFuzz*, providing a 23.9x speedup. As expected, we observed identical coverage statistics, bug count and stability numbers.

We also remind the reader that in TinyDTLS we have decided to decrease the manually-added time delay between requests from 30ms to 2ms, as we noticed the performance of *AFLNet* was seriously suffering due to this large delay. Without this change, *AFLNet* would take significantly longer to complete one million iterations, and the speedup achieved by *SnapFuzz* would be significantly higher (while we haven't completed all iterations, we estimate a speedup of around 110x). We decided to make this change as we thought the value chosen was way too large, but this shows that choosing the right values for these time delays is difficult, and thus *SnapFuzz*'s ability to eliminate them extremely important.

4.9 Unique Crashes Found

SnapFuzz, as expected, was able to replicate all *AFLNet* discovered crashes. Because *SnapFuzz* is significantly faster than vanilla *AFLNet*, *SnapFuzz* has found additional, previously-unknown crashes in 3 of the 5 benchmarks. As Table 2 shows, during 24-hour fuzzing campaigns, *SnapFuzz* found 4 bugs in the Dcmqrscp benchmark while *AFLNet* was not able to find any. For Dnsmasq, *SnapFuzz* was able to find 7 crashes while *AFLNet* found only 1, and for the LIVE555 benchmark, *SnapFuzz* was able to find 4 crashes while *AFLNet* found 2. Overall, *SnapFuzz* found 18 unique crashes, 12 more than *AFLNet*.

The bugs are a variety of heap overflows, stack overflows, use-after-free bugs, segfaults and undefined behaviours. We are currently reaching out to the project developers to assess the security severity of these issues.

5 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. We envision that this separation of concerns will help future research to progress faster by alleviating the laborious task of building performant fuzzers for network and other stateful applications.

SnapFuzz builds on top of *AFLNet* [31], and reuses its ability to infer network protocols. However, *AFLNet* has various inefficiencies and requires fragile manual delays and cleanup 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 8.5x-72.4x.

Besides *AFLNet*, a popular way of fuzzing network applications is via the *de-socketing* functionality of Preeny [11]. 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 [10]. 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.

Most work on testing network protocol implementations has focused on algorithmic rather than platform-level improvements, focusing in particular on inferring network protocol implementations [19, 21, 31, 39]. 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 with recent work on improving its effectiveness by directing exploration toward interesting program parts [17, 18], combining it with symbolic execution [20, 30, 36], inferring input grammars [15, 38] or specialising it to various application domains [16, 24, 26, 40].

Besides greybox fuzzing, other forms of fuzzing have been used to test network applications, in particular blackbox fuzzing [22, 23] and fault injection [9, 27].

In addition to fuzzing, network protocols can also be tested via other techniques, such as symbolic execution [32, 34], static analysis [37] and model checking [29].

6 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 cleanup scripts and manually-configured time delays that are error-prone, and suffers from low fuzzing throughput. *SnapFuzz* addresses both challenges through a robust architecture, which combines an in-memory file system with synchronous communication via UNIX domain sockets and various mechanism for eliminating unnecessary time delays. As a result, *SnapFuzz* simplifies fuzzing harness construction and improves the fuzzing throughput significantly, between 8.5x and 72.4x on a set of popular network applications, allowing it to find new crashes.

SnapFuzz will be made available to the community shortly after publication, with the hope that it will help improve the security and reliability of network applications and facilitate further research in this space.

REFERENCES

[1] [n.d.]. ClusterFuzz trophies. https://google.github.io/clusterfuzz/#trophies. Accessed: 2021-05-10.

[2] [n.d.]. dcmqrscp: DICOM image archive (central test node). https://support.dcmtk.org/docs/dcmqrscp.html. Accessed: 2021-03-12.

[3] [n.d.]. dnsmasq. https://thekelleys.org.uk/dnsmasq/. Accessed: 2021-03-12.

[4] [n.d.]. fork(2) — Linux manual page. https://man7.org/linux/man-pages/man2/fork.2.html. Accessed: 2021-05-10.

[5] [n.d.]. GitHub page of Libsqlfs library. https://github.com/guardianproject/libsqlfs. Accessed: 2021-05-06.

[6] [n.d.]. LightFTP repository. https://github.com/hfiref0x/LightFTP. Accessed: 2021-03-12.

[7] [n.d.]. LIVE555 repository. https://github.com/rgaufman/live555. Accessed: 2021-03-12.

[8] [n.d.]. memfd_create(2) — Linux manual page. https://man7.org/linux/man-pages/man2/memfd_create.2.html. Accessed: 2021-05-06.

[9] [n.d.]. network-emulator repository. https://github.com/guidovranken/network-emulator. Accessed: 2021-05-10.

[10] [n.d.]. Preeny documentation. https://github.com/zardus/preeny/issues/10. Accessed: 2021-05-10.

[11] [n.d.]. Preeny repository. https://github.com/zardus/preeny. Accessed: 2021-05-10.

[12] [n.d.]. pthsem / GNU Pth. https://www.auto.tuwien.ac.at/~mkoegler/index.php/pth. Accessed: 2021-05-10.

[13] [n.d.]. tinydtls-fuzz repository. https://github.com/assist-project/tinydtls-fuzz. Accessed: 2021-03-12.

[14] Paul-Antoine Arras. 2020. SaBRe: Load-time selective binary rewriting (FOSDEM presentation). https://archive.fosdem.org/2020/schedule/event/sabre/.

[15] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for Deep Bugs with Grammars. In *Proc. of the 26th Network and Distributed System Security Symposium (NDSS'19)*.

[16] Greg Banks, Marco Cova, Viktoria Felmetsger, Kevin Almeroth, Richard Kemmerer, and Giovanni Vigna. 2006. SNOOZE: toward a Stateful NetwOrk prOtocol fuzZEr. In *International conference on information security*. Springer, 343–358.

[17] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *Proc. of the 24th ACM Conference on Computer and Communications Security (CCS'17)*.

[18] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-Based Greybox Fuzzing as Markov Chain. In *Proc. of the 23rd ACM Conference on Computer and Communications Security (CCS'16)*.

[19] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. 2007. Polyglot: Automatic Extraction of Protocol Message Format Using Dynamic Binary Analysis. In *Proc. of the 14th ACM Conference on Computer and Communications Security (CCS'07)*.

[20] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Taowei, and Long Lu. 2020. SAVIOR: Towards Bug-Driven Hybrid Testing. In *Proc. of the 29th USENIX Security Symposium (USENIX Security'20)*.

[21] Weidong Cui, Marcus Peinado, Karl Chen, Helen J. Wang, and Luis Irun-Briz. 2008. Tupni: Automatic Reverse Engineering of Input Formats. In *Proc. of the 15th ACM Conference on Computer and Communications Security (CCS'08)*.

[22] Rong Fan and Yaoyao Chang. 2017. Machine learning for black-box fuzzing of network protocols. In *International Conference on Information and Communications Security*. Springer, 621–632.

[23] Hugo Gascon, Christian Wressnegger, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. 2015. Pulsar: Stateful black-box fuzzing of proprietary network protocols. In *International Conference on Security and Privacy in Communication Systems*. Springer, 330–347.

[24] Hyungsub Kim, M. Ozgur Ozmen, Antonio Bianchi, Z. Berkay Celik, and Dongyan Xu. 2021. PGFUZZ: Policy-Guided Fuzzing for Robotic Vehicles. In *Proc. of the 27th Network and Distributed System Security Symposium (NDSS'21)*.

[25] libfuzzer [n.d.]. LibFuzzer. http://llvm.org/docs/LibFuzzer.html.

[26] Daniel Liew, Cristian Cadar, Alastair Donaldson, and J. Ryan Stinnett. 2019. Just Fuzz It: Solving Floating-point Constraints Using Coverage-guided Fuzzing. In *Proc. of the Joint Meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE'19)*.

[27] Paul Dan Marinescu, Radu Banabic, and George Candea. 2010. An Extensible Technique for High-Precision Testing of Recovery Code. In *Proc. of the 2010 USENIX Annual Technical Conference (USENIX ATC'10)*.

[28] Michal Zalewski. [n.d.]. Technical "whitepaper" for afl-fuzz. http://lcamtuf.coredump.cx/afl/technical_details.txt.

[29] Madanlal Musuvathi and Dawson R. Engler. 2004. Model Checking Large Network Protocol Implementations. In *Proc. of the 1st USENIX Symposium on Networked Systems Design and Implementation (NSDI'04)*.

[30] Saahil Ognawala, Thomas Hutzelmann, Eirini Psallida, and Alexander Pretschner. 2018. Improving Function Coverage with Munch: A Hybrid Fuzzing and Directed Symbolic Execution Approach. (April 2018).

[31] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. 2020. AFLNet: A Greybox Fuzzer for Network Protocols. In *Proc. of the IEEE International Conference on Software Testing, Verification, and Validation – Testing Tools Track (ICST'20)*.

[32] Raimondas Sasnauskas, Olaf Landsiedel, Muhammad Hamad Alizai, Carsten Weise, Stefan Kowalewski, and Klaus Wehrle. 2010. KleeNet: discovering insidious interaction bugs in wireless sensor networks before deployment. In *Proc. of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN'10)*.

[33] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *Proc. of the 2012 USENIX Annual Technical Conference (USENIX ATC'12)*.

[34] JaeSeung Song, Tiejun Ma, Cristian Cadar, and Peter Pietzuch. 2011. Rule-based Verification of Network Protocol Implementations using Symbolic Execution. In *Proc. of the 20th International Conference on Computer Communication Networks (ICCCN'11)*.

[35] Evgeniy Stepanov and Konstantin Serebryany. 2015. MemorySanitizer: fast detector of uninitialized memory use in C++. In *Proc. of the International Symposium on Code Generation and Optimization (CGO'15)*.

[36] 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 *Proc. of the 23rd Network and Distributed System Security Symposium (NDSS'16)*.

[37] Octavian Udrea, Cristian Lumezanu, and Jeffrey S Foster. 2008. Rule-based static analysis of network protocol implementations. *Information and Computation* 206, 2-4 (2008), 130–157.

[38] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: Grammar-Aware Greybox Fuzzing. In *Proc. of the 41st International Conference on Software Engineering (ICSE'19)*.

[39] Yingchao Yu, Zuoning Chen, Shuitao Gan, and Xiaofeng Wang. 2020. SGPFuzzer: A State-Driven Smart Graybox Protocol Fuzzer for Network Protocol Implementations. *IEEE Access* 8 (2020), 198668–198678.

[40] Rui Zhong, Yongheng Chen, Hong Hu, Hangfan Zhang, Wenke Lee, and Dinghao Wu. 2020. SQUIRREL: Testing Database Management Systems with Language Validity and Coverage Feedback. In *Proc. of the 27th ACM Conference on Computer and Communications Security (CCS'20)*.