Netfuzzlib: Adding First-Class Fuzzing Support to Network Protocol Implementations

Jeroen Robben[0000−0001−7566−1194] and Mathy Vanhoef[0000−0002−8971−9470]

DistriNet, KU Leuven

Abstract. This paper introduces **Netfuzzlib**, a framework that emulates network input and output in user space, allowing more efficient fuzz testing and symbolic execution of Linux network programs. We first systemize common requirements and caveats for efficient fuzzing and symbolic execution of socket-based network software. Building on top of the collected requirements, we design and implement a framework that seamlessly integrates with state-of-the-art fuzzing tools, as well as symbolic execution engines, and requires no knowledge of the program's source code. Moreover, Netfuzzlib enables a range of speed optimizations, where in our evaluations it improved fuzzing speed by a factor ranging from roughly 10 to 600. By using this framework it becomes significantly easier to fuzz and symbolically analyze Linux network programs.

Keywords: Network security · Fuzzing · Dynamic testing.

1 Introduction

Fuzzing is a popular dynamic testing technique that has gained significant traction due to its effectiveness in uncovering software bugs and vulnerabilities. A particularly interesting fuzzing target is network programs, since they process input received over a remote connection, making them a common target for adversaries. Recent research has introduced a plethora of approaches for generating fuzz test cases for network software, and while doing so revealed various types of errors, ranging from common issues like buffer overflows to logic bugs such as illegal protocol state transitions [16,52,29].

Although past research has made strong progress on how to fuzz network programs, we observed that fuzzing over network connections is still slow and inefficient in practice. Using symbolic execution to test network programs is an equally challenging task: even state-of-the-art symbolic execution engines cannot, by default, handle network-related kernel interactions [10]. The current ad hoc approach to circumvent these problems is to write custom test harnesses that test specific components of the program in isolation. However, this has several disadvantages: it requires access to the source code of the System Under Test (SUT), might not cover certain code paths and thus miss bugs, and places a substantial workload on the tester in terms of engineering effort.

In this paper, we design, implement, and evaluate a framework called Netfuzzlib that seamlessly integrates into state-of-the-art fuzzers and symbolic execution engines, with as goal to more easily and efficiently test network programs.

To accomplish this, we first systemize common requirements and caveats when testing or fuzzing network software. Building on top of the collected requirements, we then design a framework for Linux network programs that emulates all network-related kernel interactions in user space with as goal to simultaneously make fuzzing easier and more efficient.

We improve fuzzing speed through several optimizations. First, by synchronizing network I/O with the fuzzer, we eliminate the need for delays, timeouts, and circumvent rate-limiting and dropped packets. Second, our framework allows one to easily replace socket I/O with a more efficient Inter-Process Communication (IPC) mechanism, such as shared memory. Third, Netfuzzlib allows the System Under Test (SUT) to notify or transfer execution from the SUT to the fuzzer on important network I/O-related events. Finally, we use the framework to provide a liveness heuristic, allowing the fuzzed program to automatically self-terminate when all fuzz input data has been processed, avoiding costly timeouts or delays that are used by existing network-based fuzzers.

We evaluate our framework on a set of network applications. This demonstrates an increased performance compared to other state-of-the-art fuzzers. Additionally, our evaluations illustrate the ability to more seamlessly symbolically analyse network services. All combined, this led to the discovery of three new vulnerabilities, which were each assigned a CVE identifier.

To summarize, our main contributions are:

- We systemize the requirements that fuzzing and symbolic execution tools must fulfill in order to seamlessly and efficiently support network software (Section 3).
- We propose and implement a framework to facilitate the fuzzing of Linux network programs (Section 4 and 5).
- We evaluate our framework with state-of-the-art fuzzing and symbolic execution tools, resulting in the discovery of three new vulnerabilities (Section 6).
- We publicly release our framework as well as documentation and examples.

Finally, Section 2 provides a background on fuzzing network software, and we compare to related work in Section 7 and conclude in Section 8.

2 Background: fuzzing network software

2.1 Modern fuzzing approaches

Fuzzing renaissance Fuzzing is an umbrella term for dynamic testing methods where invalid or unexpected inputs are repeatedly evaluated on the System Under Test (SUT) to find an input that triggers a bug. In the last decade, we have seen a revival in fuzzing interest and effectiveness, initiated by the advent of feedback driven grey-box fuzzing [50,46] and advancements in white-box fuzzing with constraint solvers [13,22]. There is an increasing trend to integrate fuzzing into the continuous development process of software systems, which has proven to be effective in finding bugs and improving programs' robustness. For

instance, Google found over 6,000 security-related bugs in its Chrome browser using coverage-guided fuzzing [2] and Microsoft announced that during the development of Windows 7, roughly 33% of all bugs identified were uncovered through the use of symbolic execution [22].

Applying fuzzing Unit-level, harness-based fuzzing is often regarded as the most effective and generic way to apply fuzzing [17]. The tester manually writes one or more test harness(es), which set-up the testing environment, invoke some target function with fuzz input and reset the program's state for every fuzz test. This approach allows the tester to pinpoint the entrypoint of the code to be fuzzed, allowing to test parts of the program in isolation which can otherwise be challenging to cover thoroughly.

Another advantage is that unit-level test harnesses can often be implemented to support *in-process* or *persistent mode* fuzzing [3]. Here, the same process is reused for various fuzz inputs, enabling a lightweight reset of the program state for processing subsequent inputs without the requirement of starting a new process, significantly enhancing fuzzing-throughput.

Network protocols often require an exchange of multiple messages in order to fulfill the intended operation, usually leading to implementations which maintain state over some session. While unit-level fuzzing is also effective to find bugs in network protocol implementations, this approach generally focuses on testing the handling of a single message within a particular state of the program. This forces the tester to identify and make assumptions on which states are globally reachable, and the requirements that will be met when executing certain parts of the code. Moreover, harness-based fuzzing typically requires an understanding of (and access to) the program's source code. It demands additional engineering effort and can be complex to implement correctly, especially due to the intertwining of I/O handling logic within other code. There is also a risk of missing bugs if the implemented fuzzing harnesses do not adequately cover all the program's code and states. Finally, some types of vulnerabilities for stateful systems, such as illegal state transitions, can not be detected by fuzzing each state individually.

2.2 Fuzzing of stateful systems

Recent research in stateful fuzzing has shown that fuzz testing stateful systems as a *whole* allows to uncover new bugs and vulnerabilities, by integrating *awareness* of the interactions/transitions between several program states into the design of the fuzzer. A categorization on stateful fuzzing techniques is available in the surveys by Daniele et al. and Zhang et al. [16,52].

The first end-to-end fuzzers for stateful systems used a generative black-box approach [5,21]. That is, these fuzzers did not use any form of feedback, but only used a model of the implemented protocol, e.g., in the form of a grammar or sample traces of valid messages, in order to generate meaningful test cases. In contrast, more modern feedback-guided state-aware fuzzers make use

of heuristics to track state transitions, for example, by extracting the semantics of received network messages [37] or by inspecting the internal memory at certain I/O related events, such as when sending or receiving data [34,31,9,8]. More concretely, three notable types of state-guided network-based fuzzers are:

- In *stateful coverage-guided fuzzing* (SCGF), the combination of AFL-style edge coverage and state feedback is used in order to guide the mutation of a test corpus [37,9,34,49,25].
- *Protocol state fuzzing* uses active automata learning to fuzz the order of protocol messages to detect deviations from the protocol specification [42,15,20].
- PISE and MACE use symbolic execution and automata learning to infer the state machine and message formats of a network protocol, and to find possible security vulnerabilities such as memory corruption errors [30,14].

2.3 Fuzzing over network I/O

Fuzzing programs over network I/O poses some unique requirements and caveats compared to testing programs which process input over some other channel, such as reading a file or via command-line arguments. In addition, grey-box feedback methods for tracking state changes require new kinds of interactions with the SUT *during* the processing of a test case, as was explained in Section 2.2. To provide a better understanding of these challenges, we first review the most common API used for a program to perform network I/O.

Sockets On most general purpose operating systems such as Linux and Microsoft Windows, the kernel requires user space programs to do network I/O using an interface based on the Berkeley sockets API [40,27,33]. Sockets are an abstraction provided by the kernel meant to depict a *communication endpoint*. Apart from Internet connections, sockets are also used for other types of interprocess communication, e.g, UNIX or Netlink sockets. All socket state is held in the kernel and programs only make use of a *handle* to refer to a socket. Following common Unix-philosophy, on Linux, socket instances are referred to in user space using *file descriptors*. The Linux IP sockets implementation closely follows the POSIX specification [24]. Common C standard library implementations for Linux, such as glibc, aim to implement the C POSIX specification and provide a set of wrapper functions which invoke the correct syscall(s) for interacting with sockets.

3 Requirements for end-to-end network fuzzing

In this section, we identify, systemize, and discuss specific requirements for applying recent fuzzing techniques to network programs and how existing fuzzers try to meet these requirements. As we will explain, some of the following requirements only apply to certain fuzzing techniques, such as symbolic execution.

3.1 Requirement 1: Determinism

Feedback-guided fuzzing systems require some sense of determinism on the feedback received from the SUT on the fuzzing input to ensure that each test is consistent and not influenced by earlier inputs. For example, active automata learning algorithms such as L* and TTT, expect the same message type to be returned for the same trace of message types that were sent as input [7]. If this requirement is not met, e.g., due to the influence from a previous fuzzing cycle or an occasionally dropped packet, the learned state model could be erroneous.

Existing approaches The required degree of determinism in stateful fuzzing varies widely and can refer to both external observations as well as internal processes of the target depending on the used feedback method(s). Some fuzzing techniques only require a new *protocol session* to be started, e.g, by closing the underlying TCP tunnel and initiating a new connection [42]. Fuzzers that require a more profound state reset, e.g., coverage-guided fuzzers, usually restart the SUT from its *main* entrypoint for every test case. Recent research proposes to use *snapshots* to revert the target state. [43,6,28,25,4,48]. Section 3.5 further elaborates on existing optimizations for resetting state and their impact on fuzzing speed.

3.2 Requirement 2: Data readiness detection

Classic network fuzzers are agnostic on the exact moment when the program is ready to receive network data. For TCP, the fuzzer typically uses a busy loop to try to connect to the SUT [36,37,34]. If the target is fuzzed over a connectionless network layer, such as when using UDP or raw IP packets, existing fuzzers cannot distinguish between the kernel dropping a packet because no socket was yet configured to receive the message, or because the tested program has received the packet but not (yet) transmitted a reply.

Existing approaches Most network fuzzers ensure that the SUT has passed the initialization step and is ready to receive messages by introducing a fixed *delay* before sending the first packet. Fine-tuning the duration of this delay is hard and often done in an ad-hoc fashion. If the delay is set too long, time is lost for the duration between the SUT being ready to receive input and the fuzzer sending a message. If the delay is too short, packets might be dropped [36,37,34,15]. Recent work has proposed methods to make network I/O synchronous for effective fuzzing, i.e., by notifying the fuzzer when the SUT is ready to receive or send network data [43,6,28,39].

3.3 Requirement 3: Liveness detection

When fuzzing a specific function in isolation, control of execution typically returns to the caller once the function has finished processing the test case. On the contrary, network services usually do not terminate after processing input, but wait indefinitely for new incoming requests. After transmitting a test case,

the fuzzer cannot differentiate between the SUT still processing the submitted message(s), being stuck in an infinite loop due to a bug, or having processed the submitted data without sending a reply. For state-aware fuzzing, a premature reset can cause the failure to detect new states, for example, due to the tested program not having enough time to send a reply [20,15].

Existing approaches In some protocols, the SUT explicitly terminates the session by sending a protocol-specific response message or by closing the underlying communication channel, e.g., a TCP tunnel. Most fuzzers consider the target to have completed a test case after a fixed *delay* has elapsed following the transmission of the final network message [36,37,20]. The processing time required by the SUT can vary widely depending on the specific fuzz input. Messages covering more code take longer to process and are likelier to exhibit interesting behavior. Therefore, the delay should be set sufficiently high to allow the target to fully process these test cases. Conversely, malformed data is often rejected early in the handling process, as is likely the case with most fuzzed messages. Therefore, a long delay can result in considerable idle time for the majority of fuzz inputs.

3.4 Requirement 4: Transfer of execution on I/O events

Some network-based grey-box fuzzers interact with the SUT upon certain I/O-related events, such as when receiving or sending a network message. For instance, *StateAFL* tracks the iterations of request/reply exchanges by hooking into send() and receive() calls. It utilizes the moment the SUT attempts to receive a packet to take snapshots of memory associated with a protocol session in order to track state changes [34]. As another example, *NSFuzz* adds compile-time instrumentation to raise a SIGSTOP signal upon entry into the network event loop, thereby pausing the target's process. This pause is used to notify the fuzzer that the SUT is ready to receive a new message and to process possible changes of tracked state variables [39]. Lastly, snapshot-based fuzzers typically use the moment when the SUT attempts to receive a message to create a snapshot [43,6,28].

Existing approaches Programs typically perform network I/O using standard library functions, such as *poll()*, and *recv()*, which in turn invoke the corresponding kernel syscall(s). Various methods are employed to execute fuzzer-controlled code when calling network I/O-related methods. StateAFL and NSFuzz add instrumentation during compile-time [34,39]. Nyx-net uses LD_PRELOAD to have calls to socket I/O-related functions linked to their runtime library instead of the normal standard library implementation [43]. Additionally, some fuzzers insert hooks at the system call level: SnapFuzz performs binary rewriting of syscall invocations in the target and its dependencies during runtime, while FitM patches QEMU's syscall translation layer [6,28].

3.5 Requirement 5: high fuzzing throughput

The input space that fuzzers sample test cases from is vast. A single network packet with a size of 100 bytes already encompasses more than 10^{240} combinations. In addition to finding smarter methods for generating fuzz inputs, an obvious strategy to uncover more bugs is to conduct more fuzz cycles. Beyond throwing more compute power at the problem, one can increase the number of fuzzing cycles by reducing the amount of time spent on executing each test case. By studying existing network-based fuzzers, we identified a set of key causes that contribute to low fuzzing throughput in state-aware and network fuzzers:

Expensive state reset and work repetition As was explained in Section 3.1, some form of state reset of the SUT needs to be perfomed for each fuzzing cycle. In the case where the SUT is completely restarted, a new process must be created, and the initialization step has to be repeated for every test case. Furthermore, when targeting a deeper state repeatedly, the same set of prefix messages must be exchanged to bring the SUT to the desired state. This *work repetition* means that the portion of time spent on computations dependent on the fuzz input might be limited compared to the total time used for each fuzzing cycle. Finally, when doing a reset through a complete restart, termination is usually handled by sending a termination signal such as a SIGTERM. Sending and handling such termination signals is slow, which further delays the fuzzing cycle.

Existing approaches *AFL* introduced the use of a *forkserver* for fuzzing. In this approach, instrumentation is added to suspend and create a clone of the target process just before its *main* entry point for each fuzz case. This eliminates the need to repeat the execve() call, linking, and libc initialization steps, but still requires the creation of a new (child) process for each test input [50].

Recent work uses various snapshot mechanisms to revert the SUT's state to a prior execution point, either at the process level or at the system level. The goal of this is to replace work repetition with a presumably less expensive checkpoint and restore mechanism, such as by making a snapshot when having reached some deeper state. *SnapFuzz* uses an AFL-style *deferred* forkserver placed at the *"latest safe point"*, e.g., just before creating the first child thread or receiving a packet. *FitM* and *SNPSFuzzer* use CRIU user space snapshots [28,25]. Xu et. al. propose to add process-level snapshot support to the OS kernel for efficient fuzzing [48]. *Nyx-net* and *Snapchange* provide a lightweight KVM-based snapshot and restore mechanism, both utilizing copy-on-write [43,4].

Delays Another source of time loss in each fuzz cycle is due to the use of delays, either introduced by the fuzzer or by the application developer. Sections 3.2 and 3.3 explain the use of delays to wait for the SUT to become ready to receive input, or to give the target enough time to process a test case before performing a reset. As was already mentioned in Snapchange, for some targets, the fuzzing throughput is also reduced due to developer-added delays [4], for example, by adding sleep intervals while busy polling for new data.

Existing approaches Recent research proposed methods to make network I/O synchronous by adding a mechanism to notify the fuzzer that the SUT is ready to receive data, eliminating the need for most fuzzer-added delays [43,28,39]. In addition, Snapchange uses binary rewriting during runtime to remove developer-added delays [6]. Unfortunately, the implementation of these approaches is specific to each fuzzer, and currently requires substantial work to also incorporate in other network-based fuzzers.

Slow I/O When transferring data using network sockets, the message payload is copied from user space to kernel space and vice versa for every message sent. Compared to other IPC methods, IP sockets are relatively slow [23]. Furthermore, to exchange multiple packets, various context switches are needed between the processes of the fuzzer, the SUT, and the kernel, which slows down fuzzing.

Existing approaches A common approach to speedup network I/O is to add hooks to replace IP with UNIX sockets [6,44,51]. Linux uses an efficient implementation based on remapping memory pages for transferring data over UNIX sockets, avoiding the need for byte-per-byte copying of messages. The semantics and operations of UNIX sockets differ from those of IP sockets. For instance, the auxiliary data returned by *recvmsg()* or the options that can be configured using *setsockopt()*, requiring the inserted hooks to translate between UNIX and IP sockets semantics in order to test most real-world network programs.

3.6 Requirement 6: handling kernel interaction

The last requirement we identified is specific to symbolic execution. Most symbolic execution engines can only trace symbolic values in user space, which poses some challenges since operations on network sockets invoke the kernel [10].

First, the symbolic execution engine cannot deduce the relationship between the passed (possibly symbolic) input arguments and the returned data. For example, on Linux, *send()* returns the amount of bytes that were transmitted or -1 in case of an error. When attempting to send data with a (bounded) symbolic length, an appropriate value must be returned. Some symbolic execution engines handle external calls by executing the intractable code with concrete arguments, and consequently limiting the *concretized* symbolic variables to a single value, which might lead to unexplored branches or missed bugs.

Second, when handling exernal calls by concretizing symbolic values, the symbolic execution engine will not be able to track side effects. Branching symbolic executors, such as KLEE, do not restart the SUT when finishing a program path, but continue the exploration process from a previous branch point [13]. For example, if path A closes a socket which is shared with path B, path B will encounter an error when sending data on the (now closed) socket, making the analysis unsound.

Finally, incoming network data must be marked as *symbolic* using the API provided by the symbolic execution engine, ideally without having to modify the SUT's source code.

Existing approaches The usual approach to apply symbolic execution to network programs is to patch the program's source code, remove socket-related calls and manually declare the correct variables and buffers as symbolic. Some symbolic execution engines such as KLEE and Angr ship with an environment model for certain OS calls, but these do not support network I/O [13,45]. To the best of our knowledge, no sound environment models for Linux networking exist.

4 Motivation

In this section, we elaborate on the shortcomings of current network-based fuzzers from the perspective of both testers/developers and researchers. This provides a direct motivation for the creation of our framework, namely, to better facilitate the fuzzing and testing of network-based software.

4.1 Tester perspective

Fuzzing has proven to be an effective and useful bug detection method, and fuzz testing a program for the first time is likely to reveal bugs, even with relatively few computing resources [12]. In order to spread its adoption and integrate fuzz testing into the day-to-day development process, ideally, application developers should not require knowledge of fuzzer internals nor have to modify the source code of the tested program. Although effective end-to-end fuzzers for certain targets, such as command-line programs exist, automated fuzzing over other input methods remains an open problem [11]. Additionally, fuzzer usability is still not sufficient [11,38].

Setting up and configuring recent network fuzzers often requires the tester to have in-depth knowledge of the used fuzzing system. Various methods for speeding up fuzzing have been proposed, but they often lack direct portability across different fuzzers or are complex to use, such as when requiring a custom hypervisor for snapshot fuzzing. Applying symbolic execution to network software almost always requires patching source code. Consequently, in practice, testers of network programs often default to using simpler, blind, black-box, or grammar-based fuzzers due to their ease of use.

4.2 Fuzzing researcher perspective

Network protocol fuzzing is an active research topic, with numerous publications and advancements in recent years [16,52]. To evaluate a new idea for improving a fuzzer building block, such as a new mutation algorithm or state tracking method, researchers typically implement a prototype, and then compare certain metrics such as code coverage or bugs found over a given time frame with other fuzzers [32,35]. To soundly compare new fuzzing techniques with existing fuzzers, it is important that SUT-specific optimizations, such as (not) avoiding delays when doing network I/O, are identical between the compared fuzzers [41,19]. Apart from this, the methods used to transfer execution control from the SUT

to the fuzzer on I/O events are technically complex and often reimplemented, possibly in slightly different ways, which again can make it difficult to soundly isolate and evaluate specific fuzzing improvements (see Section 3.4). These issues create the need for a solution that can support/optimize the fuzzing of network programs which can seamlessly integrate with different fuzzers, so that different fuzzers can more easily be compared on equal footing.

All combined, to better support the use of, and research into, network-based fuzzers, we provide a framework for fuzzing of Linux network software in user space. The framework combines existing methods and adds new ones to increase fuzzing throughput, allows symbolic execution without altering source code, and provides an API for transferring control to the fuzzer on network I/O.

5 Overview of Netfuzzlib

In this section, we first present the approach of Netfuzzlib. We then elaborate on its design, covering the fuzzing module interface, selection of kernel APIs and how we ensured consistency with the host's system configuration, and introduce a new liveness heuristic. Finally, we describe the implementation and usage of Netfuzzlib in practice.

5.1 Approach

The Netfuzzlib framework consists of a *main library*, which implements a set of networking-related functions and syscall wrappers in the GNU C library entirely in user space. The main library is accompanied with a fuzzer-specific *fuzzing module*, which itself is another library that, just like the main library, runs in the same process(es) as the SUT and defines the outcome of certain I/O events. When a new IP packet is to be received or sent, or the target attempts to open a new TCP connection, the corresponding handler in the module is called. The fuzzing module then performs fuzzer specific tasks, such as returning the next message in a test case or inspecting the SUT's internal memory. To communicate with the fuzzer, any IPC-mechanism of choice, e.g., shared memory, can be used.

5.2 Fuzzing module interface

The fuzzing module is a dynamic or static library that exports the handler functions defined in *<netfuzzlib/module_ api.h>*. The functions that are defined herein, which the user must provide a (stub) implementation for, are:

- `nfl_initialize()` This function is automatically called during initialization of the framework, just before the SUT's main() entrypoint.
- `nfl_tcp_connect()`: Called when the SUT attempts to initiate a new outbound TCP connection, i.e., when invoking *connect()* on a TCP connect.
- `nfl_tcp_accept()`: Used when the framework requires knowledge of a pending inbound TCP connection, for example, when the SUT calls *accept()* or *poll()* on a TCP socket in listening mode.

- `nfl_send()`: Invoked when the SUT attempts to send data, e.g, using *sendmsg()*.
- `nfl_receive()`: Called when the framework needs information about the *next* packet to be received, e.g., when using *poll()* or *recv()*. Messages received from the fuzzing module are added to a queue until consumed by the SUT.
- `nfl_end()`: Invoked when the liveness heuristic indicates that all received packets have been processed, see section 5.5.

5.3 Included kernel APIs

Methodology To decide which kernel functionalities to include in the *main library*, a list of Linux kernel APIs related to networking was made, using the Linux man-pages project and the latest POSIX specification [24,27]. Then, the usage of these APIs in real-world network programs was examined. Particular attention was made to allow sound symbolic execution, as discussed in Section 3.6.

POSIX sockets The socket interface provided on Linux closely follows the POSIX standard and contains well known functions operating on sockets, such as *bind()* and *recv()* [24]. Common C standard library implementations for Linux, aim to implement the C POSIX API. Other programming languages often provide runtime libraries which have strong similarities with this interface, e.g., the Python socket module, the classes under java.net.* and the Rust std::net module, which are often in their turn wrappers for the functions provided by the C standard library. All functions defined in *<sys/socket.h>* were included.

Other operations on file descriptors Following common Unix-philosophy, on Linux, socket instances are referred to in user space using file descriptors. Linux provides a unified set of operations to interact with file descriptors, regardless of the associated I/O resource, e.g, (e)poll(), fcntl() and close(). All operations that apply to IP socket file descriptors were included in the model.

Rtnetlink Netlink is a socket-based API used for transferring information between various kernel interfaces and user space processes. Rtnetlink, is an interface that is accessible through Netlink and is used to retrieve or change networking related configurations [26]. Rtnetlink is used in the implementation of various library functions, such as the *getifaddrs()* function. The same consideration as with Internet sockets for symbolic execution apply, see section 3.6. All *getters* of Rtnetlink were included in the framework.

5.4 Sound co-operation with non-emulated environment

The SUT's behavior should remain unchanged regardless of Netfuzzlib's use. This section outlines approaches employed to ensure the framework's soundness. First, the handling of operations on file descriptors not owned by the framework is discussed. Then, the approach to ensure consistent behavior with the host's network configuration is covered.

Forwarding non-modelled file descriptors Many operations used on IP and Rtnetlink sockets are also applicable to other sorts of I/O resources (e.g., an ordinary file) represented by a file descriptor. For example, *close()* terminates any type of file descriptor and *poll()* can listen for events on multiple file descriptors at once.

The framework maintains a separate file descriptor table which acts as a mask over the one from the kernel. When the SUT performs an action on a file descriptor that is not managed by the framework, the operation is forwarded to the normal C standard library. For functions operating on multiple file descriptors, such as select() and (e)poll(), the given set of file descriptors is split. The kernel implementation is invoked in non-blocking mode for the file descriptors not managed by the framework. Next, the outcome from the kernel is merged with that from the framework and returned to the caller. To prevent the framework from using the same (integer) file descriptor values as the kernel, we preemptively reserve file descriptors used by the framework by creating a dummy file handle which points to */dev/null*.

Consistent network environment As explained in Section 5.3, the framework implements certain functionality which requires knowledge of the underlying network interfaces and environment. For example, the *SIOCGIFHWADDR* ioctl returns the MAC address of a given network interface. To remain consistent with the host system, the framework makes a copy of the host system's configuration, i.e., present network interfaces and routing table, during startup. Optionally, the fuzzing module can alter this behaviour or add new network interfaces using an API provided by the main library, defined in <netfuzzlib/api.h>.

5.5 Liveness heuristic

Netfuzzlib provides an optional liveness heuristic intended to end the fuzzing cycle if the target is ready with processing a test case (recall Section 3.3). If the incoming data buffer is empty, the fuzzing module provides no new messages and the program performs a call that blocks on the availability of incoming network data or checks a predetermined number of times for new network data (e.g., with *poll()* or *recv()*), then the SUT is considered to have completely consumed a test case. The corresponding handler in the fuzzing module is invoked, which can then perform a fuzzer-specific action to prepare the target for the next test input or indicate that the SUT should terminate.

5.6 Netfuzzlib usage

The functions implemented in the framework can be used as a drop-in replacement for the ones provided by the system's C standard library, usually glibc. The *main library* can either be statically linked to the target, or the dynamic loader can be instructed to use the relevant functions in the *main library* instead of the C standard library during the runtime linking process. For GNU/Linux-based

systems, this can be done by setting the *LD_PRELOAD* environment variable to the framework's *main library* path. In theory, the framework could also be activated by placing hooks at the syscall level, for example, to test malware that directly invokes syscalls as an obfuscation technique.

5.7 Implementation

To obtain compatibility with common symbolic execution engines, Netfuzzlib was implemented in C, with the standard library as its only dependency. The *main library* was implemented in 15511 LoC, including 10949 LoC for unit tests.

6 Evaluation and Discussion

In this section, we first explain our approach to checking the *soundness* of Netfuzzlib. We then demonstrate and discuss the practicality and benefits of the framework using three experiments. In the first one, we show the speed improvements when fuzzing over native socket I/O versus Netfuzzlib for an existing network fuzzer, AFLNet. In the second experiment, we explain how we used the framework to add network fuzzing support to AFL++, without altering its source code. In the third, we use KLEE to symbolically execute a set of network servers without creating a test harness.

6.1 Checking soundness

General approach Netfuzzlib essentially re-implements a set of interfaces provided by the Linux kernel in user space. In order to do a sound analysis, a network program should behave the same regardless the use of Netfuzzlib. The standards implemented by the framework (recall Section 5.3) are only available in lengthy *prose* specifications, and are spread or duplicated through multiple documents and authorities, e.g., POSIX, Linux man-pages, and RFCs. No complete test-suite for the implemented APIs exists. To assure the soundness of our framework, we therefore took an engineering approach. We first developed a suite of unit tests while investigating the specifications of the APIs to implement, and used these to test the Linux kernel (v6.1) and the Netfuzzlib framework. When a specification was ambiguous or the kernel's implementation did not exactly follow the specification, we chose to mimick the kernel's behaviour.

Second, to ensure the correct behaviour of the network programs used in the following experiments with the framework, we *replayed* a prerecorded list of packets of a valid protocol exchange while using Netfuzzlib, and manually compared the responses to the ones received without the framework.

6.2 Experiment 1: Speeding up fuzzing

In this experiment, the fuzzing throughput is compared of an existing network fuzzer, AFLNet, when used with or without the framework. AFLNet is a state-aware coverage-guided fuzzer with support for fuzzing over UDP or TCP [37].

Table 1: Average time to reach one million executions.

	AFLNet	**AFLNet+Netfuzzlib**	**Speedup**
Bftpd	37h 15m	5h 45m	23.5x
Dcmtk	6hr 52m	4h 34m	1.5x
Dnmasq	41h 58m	3h 56m	10.66x
LightFTP	31h 15m	3h 15m	9.58x
OpenSSH	14h 50m	37m	23.5x
OpenSSL	45h 1min	3h 54m	11.5x
TinyDTLS	140h 53m	14m	593.7x

Porting AFLNet to Netfuzzlib AFLNet only has support for fuzzing over network I/O, hence we had to slightly alter the fuzzer's source code to work with Netfuzzlib. We created a new fuzzing module and changed AFLNet's *net_ recv()* function, to transfer test case message(s) and response(s) between the fuzzer and the fuzzing module using a shared memory region. We configured the fuzzing module to immediately terminate the SUT when the liveness heuristic indicates that all messages received by the SUT were processed, by invoking *exit()*.

Note that in this design, the fuzzer transfers all messages and responses to and from the shared memory region at once, respectively before the SUT is started and after it is terminated. This eliminates the need for context switches between the fuzzer and the SUT when receiving/sending consequent messages.

Experimental setup For our tests, we selected seven real-world network programs that implement a protocol supported by AFLNet. All experiments experiment were run on a laptop with an 8-core AMD 5850U CPU and 16GiB RAM, using Ubuntu 20.04. All stated measurements are an average of four fuzz runs.

Results Table 1 provides an overview of the average time required to reach one million executions, with or without the framework. We find that usage of Netfuzzlib increases fuzzing throughput for every tested program, with a typical speed increase of an order of magnitude. To show the soundness of the fuzzing process, we include coverage plots in Appendix A. All coverage measurements were conducted without the use of Netfuzzlib.

6.3 Experiment 2: Adding network fuzzing support

AFL++ is a continuation of the original AFL fuzzer, and aims to combine new insights and recent research in coverage-guided fuzzing in a single fuzzer implementation [18].

Fuzzing module The fuzzing module for AFL++ reads a test case using the fuzzer's shared memory API, and presents it to the SUT as a single UDP packet or in a TCP stream. The module is configurable via environment variables, for example, to select the destination port.

Results For this experiment, the same test subjects and configuration as in experiment 1 was used. It was possible to fuzz all seven network programs without creating a test harness. Edge coverage plots were included in Appendix A.

6.4 Experiment 3: Symbolic execution

In the third experiment, we use KLEE, a state-of-the-art symbolic execution engine that can be used to detect a wide range of bugs such as memory corruption errors and assertion errors or to generate a set of test inputs for every explored program path [13]. KLEE ships with a module to simulate (symbolic) interactions with the filesystem and standard I/O, but cannot (symbolically) handle networking operations. In the first test, we examine whether it is possible to use Netfuzzlib to detect a known vulnerability in Dnsmasq with KLEE, without creating a test harness. Finally, we use symbolic execution to discover three new vulnerabilities across two publicly available network servers.

Test setup In order to introduce symbolic network packets, we created a Netfuzzlib module which marks memory regions containing incoming network data as symbolic using KLEE's API (`klee_make_symbolic()`). KLEE operates on the intermediate representation used in the LLVM compiler framework, hence both Netfuzzlib's *main library* and corresponding module were compiled to LLVM IR and added using KLEE's `-link-llvm-lib` option.

Dnsmasq CVE 2017-14492 and 2017-14493 are respectively a heap and stack-based buffer overflow that were present in Dnsmasq up to version 2.77, and can be triggered by a malformed ICMPv6/DHCPv6 message. While symbolically executing Dnsmasq 2.76, KLEE reported the second vulnerability when introducing a symbolic packet with a size of at least 120 bytes within 5 minutes. We were not able to detect the first vulnerability using KLEE within 12 hours.

New vulnerabilities Finally, we use KLEE to detect three vulnerabilities across two publicly available network programs, with assigned CVE's 2023-50432, 2023-50433 and 2023-50434, demonstrating the utility of the framework for symbolic execution. Two vulnerabilities are in an open-source DHCP server [1] and can be abused to perform a Denial-of-Service attack: a NULL-pointer dereference and a type confusion bug causing a failed memory allocation and subsequent crash. A third vulnerability was found in an open-source DNS server for embedded systems [47], where a stack-based buffer over-read can lead to the potential disclosure of sensitive information.

7 Related work

Nyx-net [43]. Nyx-Net uses hypervisor-based fuzzing with incremental snapshots, combined with selective emulation of IP sockets. To reduce work repetition, it uses snapshots with an efficient copy-on-write mechanism in order to

revert the SUT to a prior state. The system first creates a normal IP socket, and emulates socket I/O when the first data is received, which can not be used when fuzzing over multiple connections. In contrast, we emulate network APIs entirely in user-space to improve usability and performance, and do not require a hypervisor. Netfuzzlib could be used in combination with Nyx-Net to further improve performance and to handle more complex scenarios.

SnapFuzz [6]. The SnapFuzz framework proposes a set of methods and optimizations for increasing the efficiency of network fuzzers. It replaces IP sockets with faster UNIX sockets, uses a *forkserver* placed at a later point in execution to reduce work repetition, and uses an in-memory filesystem to enable a faster state reset and increase file I/O throughput. SnapFuzz also adds a mechanism to eliminate fuzzer-added delays by notifying the fuzzer when the SUT is ready to receive data. Finally, it uses binary-rewriting to add various other optimization, for example, to remove developer-added delays. The concepts introduced in SnapFuzz are mainly orthogonal to our work, and could be used in combination with Netfuzzlib to further increase fuzzer performance.

LibAFL [19]. LibAFL is a modular framework for building fuzzers, which architecturally decouples the building blocks which make up a fuzzer. The authors identify that in current research, new fuzzing technologies are often implemented by forking and altering an existing fuzzer (often AFL), making it difficult to build upon or assess the combination of separate advancements in fuzzing. Netfuzzlib could be used to add (efficient) network fuzzing support to LibAFL by implementing a LibAFL *Executor* and corresponding Netfuzzlib fuzzing module.

8 Conclusion

This paper identified a set of requirements and caveats for end-to-end fuzzing of network software and presented `Netfuzzlib`, a novel framework that removes the main obstacles for end-to-end fuzzing over network I/O in Linux. The framework provides a simple API to seamlessly facilitate the fuzzing and testing of network-based programs and typically results in a fuzzing throughput increase of at least 10x. This speedup is accomplished by eliminating the need for I/O delays, using a faster IPC mechanism to transfer network data, reducing the number of required context switches, automatically terminating the target when finishing a test case and various other improvements.

To enable the reproduction of our results, and encourage research into fuzzing network programs, we make Netfuzzlib open source under the Apache 2.0 and MIT license. It is available online at:

`https://netfuzzlib.com`

Acknowledgements

This research is partially funded by the Research Fund KU Leuven, and by the Cybersecurity Research Programme Flanders.

References

1. simple-dhcp-server. https://code.google.com/archive/p/simple-dhcp-server/
2. ClusterFuzz (2023), https://google.github.io/clusterfuzz/, accessed: 19-11-23
3. libFuzzer – a library for coverage-guided fuzz testing (2023), https://llvm.org/docs/LibFuzzer.html, accessed: 17-11-23
4. Snapchange – lightweight fuzzing of a memory snapshot using kvm. (2023), https://github.com/awslabs/snapchange
5. Sulley (2023), https://github.com/OpenRCE/sulley
6. Andronidis, A., Cadar, C.: SnapFuzz: high-throughput fuzzing of network applications. In: International Symposium on Software Testing and Analysis. p. 340–351. ISSTA 2022, ACM (2022), https://doi.org/10.1145/3533767.3534376
7. Angluin, D.: Learning regular sets from queries and counterexamples. Information and Computation **75**(2), 87–106 (1987), https://doi.org/10.1016/0890-5401(87)90052-6
8. Aschermann, C., Schumilo, S., Abbasi, A., Holz, T.: IJON: Exploring deep state spaces via fuzzing. In: IEEE Symposium on Security and Privacy (S&P). pp. 1597–1612 (2020). https://doi.org/10.1109/SP40000.2020.00117
9. Ba, J., Böhme, M., Mirzamomen, Z., Roychoudhury, A.: Stateful greybox fuzzing. In: USENIX Security 22. pp. 3255–3272. USENIX Association (2022), https://www.usenix.org/conference/usenixsecurity22/presentation/ba
10. Baldoni, R., Coppa, E., D'elia, D.C., Demetrescu, C., Finocchi, I.: A survey of symbolic execution techniques. ACM Comput. Surv. **51**(3) (2018), https://doi.org/10.1145/3182657
11. Boehme, M., Cadar, C., Roychoudhury, A.: Fuzzing: Challenges and reflections. IEEE Software **38**(3), 79–86 (2021). https://doi.org/10.1109/MS.2020.3016773
12. Böhme, M., Falk, B.: Fuzzing: on the exponential cost of vulnerability discovery. In: ACM SIGSOFT Symposium on the Founda- tions of Software Engineering. p. 713–724. FSE 2020, ACM (2020), https://doi.org/10.1145/3368089.3409729
13. Cadar, C., Dunbar, D., Engler, D.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: USENIX Conference on Operating Systems Design and Implementation. p. 209–224. OSDI'08, USENIX Association (2008), https://dl.acm.org/doi/10.5555/1855741.1855756
14. Cho, C.Y., Babić, D., Poosankam, P., Chen, K.Z., Wu, E.X., Song, D.: MACE: Model-inference-Assisted concolic exploration for protocol and vulnerability discovery. In: USENIX Security 11. USENIX Association (2011), https://www.usenix.org/conference/usenix-security-11/mace-model-inference-assisted-concolic-exploration-protocol-and
15. Daniel, L.A., Poll, E., de Ruiter, J.: Inferring OpenVPN state machines using protocol state fuzzing. In: 2018 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW). pp. 11–19 (2018). https://doi.org/10.1109/EuroSPW.2018.00009
16. Daniele, C., Andarzian, S.B., Poll, E.: Fuzzers for stateful systems: Survey and research directions (feb 2024). https://doi.org/10.1145/3648468
17. Ding, Z.Y., Le Goues, C.: An empirical study of OSS-Fuzz bugs. In: IEEE/ACM International Conference on Mining Software Repositories (MSR). pp. 131–142. IEEE (2021), https://doi.ieeecomputersociety.org/10.1109/MSR52588.2021.00026

18. Fioraldi, A., Maier, D., Eißfeldt, H., Heuse, M.: AFL++: combining incremental steps of fuzzing research. In: Proceedings of the 14th USENIX Conference on Offensive Technologies. WOOT'20, USENIX Association (2020), https://www.usenix.org/system/files/woot20-paper-fioraldi.pdf
19. Fioraldi, A., Maier, D.C., Zhang, D., Balzarotti, D.: Libafl: A framework to build modular and reusable fuzzers. In: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security. p. 1051–1065. CCS '22, ACM (2022), https://doi.org/10.1145/3548606.3560602
20. Fiterau-Brostean, P., Jonsson, B., Merget, R., de Ruiter, J., Sagonas, K., Somorovsky, J.: Analysis of DTLS implementations using protocol state fuzzing. In: USENIX Security 20. pp. 2523–2540. USENIX Association (Aug 2020), https://www.usenix.org/conference/usenixsecurity20/presentation/fiterau-brostean
21. Gascon, H., Wressnegger, C., Yamaguchi, F., Arp, D., Rieck, K.: Pulsar: Stateful black-box fuzzing of proprietary network protocols. In: Thuraisingham, B., Wang, X., Yegneswaran, V. (eds.) Security and Privacy in Communication Networks. pp. 330–347. Springer International Publishing (2015), https://doi.org/10.1007/978-3-319-28865-9_18
22. Godefroid, P., Levin, M.Y., Molnar, D.: SAGE: Whitebox fuzzing for security testing. Queue **10**(1), 20–27 (jan 2012), https://doi.org/10.1145/2090147.2094081
23. Goldsborough, P.: ipc-bench (2023), https://github.com/goldsborough/ipc-bench
24. IEEE and The Open Group: The Open Group Base Specifications Issue 7 (2018)
25. Li, J., Li, S., Sun, G., Chen, T., Yu, H.: SNPSFuzzer: A fast greybox fuzzer for stateful network protocols using snapshots. IEEE Transactions on Information Forensics and Security **17**, 2673–2687 (2022). https://doi.org/10.1109/TIFS.2022.3192991
26. Linux Programmer's Manual: rtnetlink(7) - Linux man page, 6.04 edn. (2023), https://man7.org/linux/man-pages/man7/rtnetlink.7.html
27. Linux Programmer's Manual: socket(7) - Linux man page, 6.04 edn. (2023), https://man7.org/linux/man-pages/man7/socket.7.html
28. Maier, D., Bittner, O., Munier, M., Beier, J.: FitM: Binary-only coverage-guided fuzzing for stateful network protocols. In: Workshop on Binary Analysis Research (BAR), 2022 (2022), https://dx.doi.org/10.14722/bar.2022.23008
29. Manès, V.J., Han, H., Han, C., Cha, S.K., Egele, M., Schwartz, E.J., Woo, M.: The art, science, and engineering of fuzzing: A survey. IEEE Transactions on Software Engineering **47**(11), 2312–2331 (2021). https://doi.org/10.1109/TSE.2019.2946563
30. Marcovich, R., Grumberg, O., Nakibly, G.: PISE: Protocol inference using symbolic execution and automata learning. In: Workshop on Binary Analysis Research. BAR 2023 (2023), http://dx.doi.org/10.14722/bar.2023.23002
31. McMahon Stone, C., Thomas, S.L., Vanhoef, M., Henderson, J., Bailluet, N., Chothia, T.: The closer you look, the more you learn: A grey-box approach to protocol state machine learning (2022), https://doi.org/10.1145/3548606.3559365
32. Metzman, J., Szekeres, L., Simon, L., Sprabery, R., Arya, A.: Fuzzbench: an open fuzzer benchmarking platform and service. In: ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. p. 1393–1403. ESEC/FSE 2021, ACM (2021), https://doi.org/10.1145/3468264.3473932
33. Microsoft: Windows Sockets 2 (01 2021), https://learn.microsoft.com/en-us/windows/win32/WinSock/windows-sockets-start-page-2, accessed: 2023-11-26

34. Natella, R.: StateAFL: Greybox fuzzing for stateful network servers. Empirical Softw. Engg. **27**(7) (dec 2022), https://doi.org/10.1007/s10664-022-10233-3
35. Natella, R., Pham, V.T.: ProFuzzBench: a benchmark for stateful protocol fuzzing. p. 662–665. ISSTA 2021, ACM (2021), https://doi.org/10.1145/3460319.3469077
36. Pham, T.: AFLnwe (2020), https://github.com/thuanpv/aflnwe/commits/master
37. Pham, V.T., Böhme, M., Roychoudhury, A.: AFLNET: A greybox fuzzer for network protocols. In: IEEE International Conference on Software Testing, Validation and Verification (ICST). pp. 460–465 (2020). https://doi.org/10.1109/ICST46399.2020.00062
38. Plöger, S., Meier, M., Smith, M.: A usability evaluation of afl and libfuzzer with cs students. CHI '23, ACM (2023), https://doi.org/10.1145/3544548.3581178
39. Qin, S., Hu, F., Ma, Z., Zhao, B., Yin, T., Zhang, C.: NSFuzz: Towards efficient and state-aware network service fuzzing. ACM Trans. Softw. Eng. Methodol. **32**(6) (sep 2023), https://doi.org/10.1145/3580598
40. Quarterman, J.S., Silberschatz, A., Peterson, J.L.: 4.2BSD and 4.3BSD as examples of the UNIX system. ACM Comput. Surv. **17**(4), 379–418 (dec 1985), https://doi.org/10.1145/6041.6043
41. Rizzi, E.F., Elbaum, S., Dwyer, M.B.: On the techniques we create, the tools we build, and their misalignments: A study of KLEE. In: IEEE/ACM International Conference on Software Engineering. pp. 132–143. ICSE '16 (2016). https://doi.org/10.1145/2884781.2884835
42. de Ruiter, J., Poll, E.: Protocol state fuzzing of TLS implementations. In: USENIX Security 15. pp. 193–206. USENIX Association (Aug 2015), https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/de-ruiter
43. Schumilo, S., Aschermann, C., Jemmett, A., Abbasi, A., Holz, T.: Nyx-net: network fuzzing with incremental snapshots. p. 166–180. EuroSys '22, ACM (2022), https://doi.org/10.1145/3492321.3519591
44. Shoshitaishvili, Y.: preeny (2021), https://github.com/zardus/preeny
45. Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., Vigna, G.: SOK: (state of) the art of war: Offensive techniques in binary analysis. In: IEEE Symposium on Security and Privacy (S&P). pp. 138–157 (2016). https://doi.org/10.1109/SP.2016.17
46. Swiecki, R.: Honggfuzz (2023), https://github.com/google/honggfuzz
47. Tsarev, M.: emdns (2020), https://github.com/mtsarev/emdns
48. Xu, W., Kashyap, S., Min, C., Kim, T.: Designing new operating primitives to improve fuzzing performance. p. 2313–2328. CCS '17, ACM (2017), https://doi.org/10.1145/3133956.3134046
49. Yu, Y., Chen, Z., Gan, S., Wang, X.: SGPFuzzer: A state-driven smart graybox protocol fuzzer for network protocol implementations. IEEE Access **8** (2020). https://doi.org/10.1109/ACCESS.2020.3025037
50. Zalewski, M.: AFL (2021), https://github.com/google/AFL
51. Zeng, Y., Lin, M., Guo, S., Shen, Y., Cui, T., Wu, T., Zheng, Q., Wang, Q.: Multifuzz: A coverage-based multiparty-protocol fuzzer for iot publish/subscribe protocols. Sensors **20**(18) (2020). https://doi.org/10.3390/s20185194
52. Zhang, Z., Zhang, H., Zhao, J., Yin, Y.: A survey on the development of network protocol fuzzing techniques. Electronics **12**(13) (2023). https://doi.org/10.3390/electronics12132904

A Coverage plots