On the (in)Efficiency of Fuzzing Network Protocols

Seyed Behnam Andarzian(✉), Cristian Daniele, and Erik Poll*

Radboud Universiteit, Nijmegen, Netherlands
seyedbehnam.andarzian@ru.nl

Abstract. Fuzzing is a great technique to find software bugs. However, certain types of systems, notably network protocols, are still challenging to fuzz because of in-efficiency. As an extended version of our previous article, we identify the root causes for overhead in fuzzing network protocols. After that, we categorize and discuss strategies for fuzzing network protocols. As a result, this article presents a comprehensive analysis of all the root causes behind the inefficiency of fuzzing network protocols and all the strategies to avoid them.

Keywords: Testing · Fuzzing · Software Security · Network Protocol Fuzzing.

1 Introduction

Fuzzing (a.k.a. fuzz testing) is an effective technique for testing software systems. Popular fuzzers such as AFL++ [22] and LibFuzzer [1] have found thousands of bugs in both open-source and commercial software. For instance, Google has discovered over 25,000 bugs in their software (e.g., Chrome) and over 36,000 bugs in over 550 open-source projects [3]. Fuzzing involves sending many – tens or hundreds of thousands – (semi)automatically generated input to the System-Under-Test (SUT), so the speed of generating and processing many inputs is important. Fuzzers go to extreme lengths to increase the speed [28].

Unfortunately, not all software can benefit from such fuzzing campaigns. For network protocols, achieving high speeds in fuzzing is tricky. Whereas a typical fuzzing campaign with a modern fuzzer like AFL++ [22] on, say, a graphics library will produce thousands of inputs per second [31], a fuzzer like AFLNet [13] for fuzzing network protocols produces a few dozens of inputs per second. One important reason is the overhead of network stacks. However, network protocols are often stateful protocols, and this statefulness is then another reason that slows down the fuzzing [9] as it means that each test case is a *sequence* of input messages – is then another reason that slows down the fuzzing. Another reason is context switching between the fuzzer and the SUT that further slows down the fuzzing speed.

* This research is funded by NWO as part of the INTERSCT project (NWA.1160.18.301)

Often in a campaign to fuzz some network protocol implementation, the code of the SUT will be modified in an ad-hoc way to remove performance overheads, for instance, by removing the network stack or letting the fuzzer by-passing the network stack when interacting with the SUT (for example, see [25]). We are primarily interested in generic techniques that can implemented in a fuzzer (or some library used by the fuzzer) that will speed fuzzing up for any network protocol implementation. Still, our analysis sheds light on what can be achieved by ad-hoc modification of the SUT.

In an earlier article [30], we reported results of two strategies to improve the efficiency of fuzzing network protocols that we implemented, namely Desock+ to reduce communication overheads (root cause network stack in section 3.1) and Green-Fuzz to reduce the context switches between fuzzer and SUT (root cause context switching in section 3.2). As an extended version of the earlier article [30], this article provides a more comprehensive analysis of the root causes of performance overhead in fuzzing network protocols and strategies to tackle them. Some of the results of this analysis are also relevant for fuzzing other stateful protocols and even fuzzing in general. Our contributions are as follows:

- We present all root causes of overheads in network protocol fuzzing.
- We provide all strategies to avoid or eliminate root causes of overhead in fuzzing network protocols, including the two presented in the earlier article.
- We provide a comprehensive analysis of the strategies and their impact on the root causes of overhead in fuzzing network protocols.

Section 2 explains why fuzzing performance is important. Section 3 delves into the types of overheads encountered while fuzzing network protocols. Moving on to Section 4, we discuss strategies to overcome the communication overheads that slow down the fuzzing network protocols. Section 5 is focused on strategies to reduce the overhead caused by context switching between the fuzzer and the SUT. Section 6 explores strategies to address the challenges of initialization and termination overheads. Section 7 is dedicated to analyzing and comparing these strategies to determine which works best. We then shift our focus in Section 8 to review related work in this field. Section 9 looks ahead, discussing future research directions and acknowledging the limitations of our current article. Finally, in Section 10, we conclude our article by summarizing our findings and their implications for improving fuzzing performance in network protocol testing.

2 Background

In the realm of software security, one of the major challenges is ensuring the robustness and safety of software against malicious inputs. Fuzzing, a dynamic code testing technique, is a useful way of identifying vulnerabilities in software. There are many factors considered for effective fuzzing of software. The main ones are code-coverage, performance, and applicability. Each of these factors is essential for effectively fuzzing and finding vulnerabilities. Performance, as one

of these factors, is critical in fuzzing because more fuzzing speed means we need less computing resources and energy.

For example, Google is spending a lot of computing resources for OSS fuzz [26] to find bugs. By having an efficient fuzzer, these companies can spend less time and resources on fuzzing. Furthermore, time is critical when it comes to integrating fuzzing in the CI/CD [1] pipelines for software. As mentioned in [27], the reasonable amount of time that should be spent on fuzzing in the CI/CD pipeline is around 10 minutes per day, which is very short.

The issues mentioned above get worse when it comes to fuzzing network protocol implementations. When fuzzing regular command line software[2], on average, we are 100 times faster than fuzzing network protocol implementations. This observation led us to do more research and find hurdles in efficiently fuzzing network protocol implementations. After addressing these hurdles, we believe this is the maximal amount of speed gain we can have when doing out-process fuzzing.

3 Types of Overhead in Network Protocol Fuzzing

There are types of overheads for fuzzing network protocols. There are three kinds of overhead, which we will refer to as O1-O3:

- **Network stack overhead** (O1) happens when the fuzzer sends the input to the SUT or the SUT sends back a response using the network stack.
- There is also **context switching overhead** (O2), which happens when there is a context switch between the fuzzer and the SUT.
- There is also SUT **initialization and termination overhead** (O3), which happens every time the SUT is initialized and terminated.

In the remainder of this section, we will take a deeper look into these overheads. Figure 1 shows the overheads while fuzzing a network protocols.

3.1 Root cause 1: network stack(O1)

When testing applications on a single machine using fuzzing, the network stack includes functions that are not needed. These include calculating checksums, adding headers, and routing. These tasks are important for network communication but are unnecessary for fuzzing on one machine. They use extra computer resources without any benefit and make the fuzzing process slower and less effective. This shows the need for a more suitable strategy for fuzzing on a single machine, which would use resources better and improve the process of finding vulnerabilities.

On top of previously mentioned overheads in the paragraph above, our experiments show that when using network stack for fuzzing, network related system

[1] Continuous Integration / Continuous Deployment

[2] This is just an estimation based on our experience with out-process fuzzing using AFL fuzzer

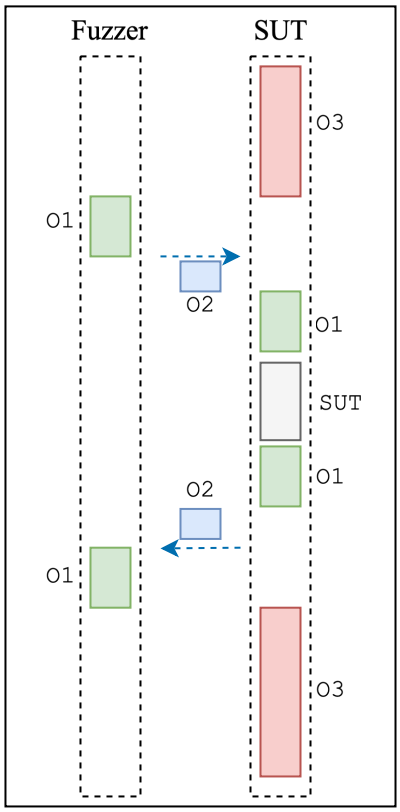


Fig. 1: Overheads of fuzzing a network protocol for a trace with one input message. The green color refers to overhead root cause one, blue is for root cause two and red is for root cause three.

calls also add overheads. Network stack overhead (O1) is composed of sending and receiving overheads. For example, in order to send an input to the SUT, the fuzzer would call `setsockopt` and `sendto` system-calls and when SUT want to recieve the input, it has to call the `recvfrom` system-call.

3.2 Context Switching between Fuzzer and SUT - root cause 2

In fuzzing network protocols, the context switching overhead(O2) in operating systems becomes important. Fuzzing involves rapidly sending and receiving many inputs and outputs to/from the SUT, which requires frequent context switches as the operating system switches the execution between the fuzzer and the SUT. Each context switch entails the operating system saving the state of the currently active process (the fuzzer or the SUT) and loading the state of the other. This process consumes significant system resources. This is especially resource-intensive due to the high frequency of switches, as the state information (including register data, program counters, and memory allocations) must be continuously stored and restored.

Furthermore, the cache invalidation caused by these switches, where previously loaded cache data becomes irrelevant after a context-switch, leads to additional memory reads, thereby more overhead. This substantial overhead can significantly affect the efficiency of fuzzing processes, slowing down the fuzzing and potentially impacting the detection of vulnerabilities.

3.3 SUT Initialization and Termination - root cause 3

In fuzzing network protocols, a new SUT process must be created for each input trace. This involves repeatedly invoking `fork` system-call, leading to overhead. On top of that, the SUT needs to be initialized for each input, which includes running constructors and initialization functions specific to that SUT. These initializations, which set up the execution environment for the SUT, need to be executed for every input trace, which leads to SUT initialization overhead (O3).

After processing each input trace, the SUT must be terminated. The termination process involves the `kill` system call, which needs to be called multiple times to ensure the process is completely terminated and this leads to SUT termination overhead (O3). This is particularly expensive in terms of system resources, as the operating system needs to ensure all resources allocated to the process are properly released and the process state is fully cleared. The cumulative effect of repeatedly initializing and terminating the SUT for each input trace significantly slows down the fuzzing process and increase overhead.

4 Communication strategies (tackling root cause 1)

An essential part of a network protocol fuzzer is sending the inputs to the SUT and receiving the respective outputs for each input. Using real network communication to test network protocols through fuzzing introduces significant overhead, making it less efficient. Despite this drawback, it remains a popular choice

among many fuzzers like AFLNet [13], AFLNwe [22], and StateAFL [21], mainly because of its simplicity. These tools work by sending inputs and receiving responses through actual network sockets. This process, while straightforward, involves the network stack, which adds overhead. Essentially, while real network communication is the path of least resistance for many developers, the associated overheads suggest exploring alternative strategies that could offer improved efficiency and performance. There are strategies for this purpose as follows:

1. S1A: Using simulated network stack.
2. S1B: Using shared-memory.
3. S1C: Using In-memory communication.

The strategies listed above, reduce overhead O1, for every input message. We discussed these strategies in more detail below.

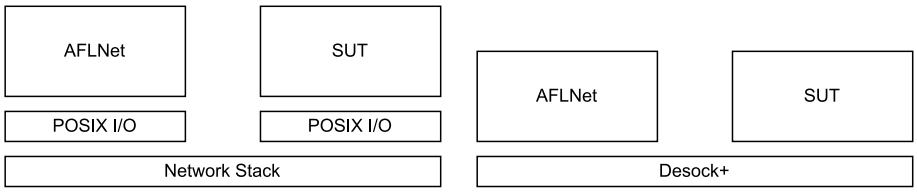
4.1 Simulated network stack

In our previous article we presented Desock+ as a simulated socket library that works with any fuzzer to avoid network communication overhead. Existing fuzzers for network protocols, such as AFLNet [13], rely on network communication to send inputs to the SUT. However, this strategy has two drawbacks. The fuzzer sends an input message to the SUT and gets a response. Each round of fuzzing [3] is done by sending a sequence of input messages, which we call a trace of input messages. For each trace of input message $T =< m_1, m_2, ..., m_n >$, the fuzzer must create a new connection, which adds overhead. Additionally, sending each input message m_n through the network also incurs overhead due to the time-consuming steps in the network stack, which are unnecessary for the fuzzing.

To reduce this overhead, we propose using a simulated socket instead of sending inputs through the network stack. By taking this strategy, we do not have to use emulation or modify the source code of the SUT, and it is faster. We accomplish this by using a modified version of the simulated socket library called `preeny`, which communicates with the SUT via the standard I/O. However, we found that `preeny` does not work out of the box. We addressed this issue by modifying `preeny` and introducing a new simulated socket library named Desock+.

Network Protocol Fuzzing using Desock+: Desock+ can be used by the SUT instead of the standard POSIX library to fuzz network protocols more efficiently. The overview of a fuzzer working with Desock+ is shown in Figure 2. In this case, the fuzzer is the slightly modified AFLNet which sends and receives input messages through standard I/O instead of network sockets. As we can see, the SUT is intact, and the only thing that is changed is the underlying socket library, which the SUT would load instead of the real socket library.

[3] One round of fuzzing consists of sending one input to the SUT to test it, and refreshing the SUT for the next input



(a) AFLNet (b) AFLNet using Desock+

Fig. 2: Removing network communication overhead using Desock+.

The difference between `preeny` and Desock+ is that `preeny` can not support specific socket-related system-calls and arguments. However, by modifying `preeny`, we have provided Desock+, which can handle any types of SUT that use POSIX network I/O. The arguments in socket-related system-calls that Desock+ supports are listed in Table 1. The advantage of Desock+ over `preeny` is that it can also support SUTs that:

- Contain socket system-calls using blocking or non-blocking network I/O.
- Receive the input messages as datagram, streams, sequenced, connectionless, and raw.
- Use `connect` and `accept4` system-calls.

The modifications made to `preeny` to make Desock+ are implemented in the `socket` system-call, which is responsible for creating the socket file descriptor. We have added a function named `setup`, which modifies the socket file descriptor by considering different arguments provided to the `socket` system-call. Based on the arguments passed to the `socket` system-call, Desock+ uses `fcntl` and `setsockopt` to set different arguments on the socket file descriptor. This way, other socket-related system-calls can use this socket file descriptor without resulting in an error. In `preeny`, these arguments are ignored while creating the socket file descriptor, resulting in an error when other socket-related system-calls try to use different arguments inside the SUT.

Desock+ is only helpful for fuzzing network protocols, whereas `preeny` is also intended to be used for SUT interaction with other services on the system or using a loopback address[4]. To be able to set different arguments on the socket file descriptor, Desock+ avoids assigning an IP address and port number to the socket file descriptor (setting arguments on a simulated file descriptor with assigned IP and port results in an EINVAL error). However, since `preeny` is meant to be used for many other purposes, this can break its functionality. Therefore, we made Desock+ a separate library for use by fuzzers.

[4] A loopback address is a unique IP address, that is used to refer to the localhost.

Table 1: Socket-related POSIX system-calls and their arguments supported by Desock+.

System-Call	Arguments	System-Call	Arguments
socket()	AF_LOCAL	connect3()	SOCK_NONBLOCK
	AF_INET		SOCK_CLOEXEC
	AF_INET6		SOCK_SEQPACKET
	SOCK_STREAM		SOCK_DGRAM
	SOCK_DGRAM		SOCK_STREAM
	SOCK_SEQPACKET	dup3()	SOCK_NONBLOCK
	SOCK_RAW		SOCK_CLOEXEC
	SOCK_RDM	recv() recvfrom() recvmsg()	MSG_CMSG_CLOEXEC
	SOCK_PACKET		SCM_RIGHTS
accept4()	SOCK_NONBLOCK		MSG_DONTWAIT
	SOCK_CLOEXEC		MSG_ERRQUEUE
	SOCK_SEQPACKET	send() sendto() sendmsg()	SOCK_STREAM
	SOCK_STREAM		SOCK_SEQPACKET
bind()	AF_INET		MSG_CONFIRM
	AF_INET6		MSG_DONTWAIT

4.2 Shared-memory

Using shared memory [33] for fuzzing network protocols increases fuzzing network protocols performance because communication occurs directly through memory rather than actual or simulated network sockets. Unlike Desock+, which relies on files to mimic network communication, shared memory offers a more direct and faster strategy. This speed improvement is attributed to eliminating the overheads associated with network stack or file-based communication strategies. Moreover, there is potential to further enhance tools like Desock+ by adapting them to use shared memory instead of files.

4.3 In-memory (requires in-process fuzzing strategy S3C)

In-process fuzzers usually mutate a variable within the program in each fuzzing round, called in-memory fuzzing. This differs from shared-memory, where the inputs are sent from the fuzzer process to the SUT process via a shared-memory (pipes in Linux). In-memory communication is the fastest strategy for fuzzing network protocols, primarily due to its direct interaction with the SUT memory. This strategy is significantly faster because it eliminates the overhead of processing inputs from real networks, files, or shared memory. This strategy only works with in-process fuzzing (see section 6.3). The process involves compiling the SUT code with a compiler like LLVM [1] and embedding the fuzzer directly within the SUT, sidestepping the need for external data transmission. Notably, LibFuzzer [1] employs this strategy, leveraging in-process fuzzing to optimize the efficiency and speed of the fuzzing operation. This strategy not only simplifies the communication mechanism but also significantly boosts the performance of the

fuzzing process, demonstrating the effectiveness of in-memory communication in conjunction with in-process fuzzing.

5 Trace sending strategies (tackling root cause 2)

There are strategies to send input traces to the SUT that can affect the overheads associated with context switching. These strategies are:

1. S2A: Sending input messages one by one.
2. S2B: Sending input messages in one go (reduces overhead O2, for every input trace).

In this section, first, we introduce the strategy used by many fuzzers for network protocols. After that, we discuss the strategy we introduced in our previous article.

5.1 Sending input messages one by one

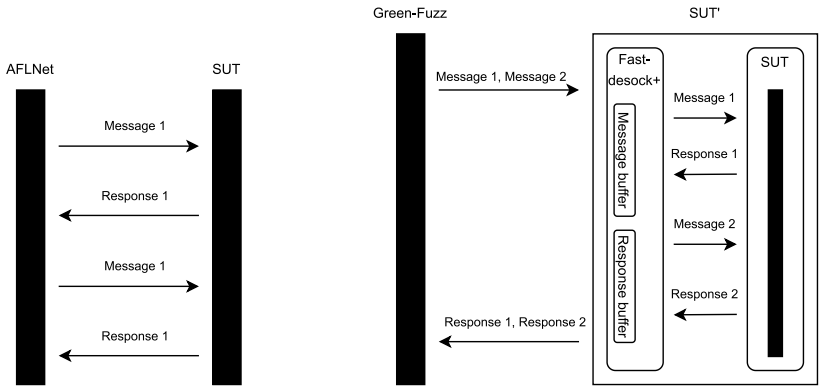
The baseline for the network protocol fuzzers (e.g., AFLNet, AFLNwe, StateAFL) sends input messages one by one to the SUT and receives the respective responses. However, this strategy adds overhead for context switching between the fuzzer and the SUT. This overhead can be avoided by applying more sophisticated strategies like the one in section 5.2.

5.2 Sending input messages in one go

In our previous article [30], we have presented Green-Fuzz, a new fuzzer to reduce the context-switches between the fuzzer and SUT in the fuzzing process.

Current fuzzers for network protocols consider a trace of input messages $T =< m_1, m_2, ..., m_n >$, and send the input message m_n one by one to fuzz the SUT. By using the Green-Fuzz, we do not send input messages one by one but as a trace. We do this because when the fuzzer sends input messages one by one, the fuzzer has to call two (or more) system-calls for each input message and call the same number of system-calls to receive the respective response from the SUT. However, by sending the entire trace of input messages in one go, the number of system-calls is reduced: for a trace of input messages T with n messages, we only have the overhead once, instead of n times. This strategy can be applied to any network protocol fuzzer. However, because Green-Fuzz sends the whole trace in one go, there is a limitation where we assume that the fuzzer can decide on the input trace in advance. We applied this strategy on AFLNet [13].

Design: To apply our strategy to AFLNet, we implemented another simulated socket library named Fast-desock+. Fast-desock+ intercepts and buffers the trace of input messages T sent by Green-Fuzz fuzzer. After that, it takes each message m_n from trace T and sends it to the SUT. Consequently, the SUT finishes processing and sends back a response r_j, which Fast-desock+ intercept and save into a response buffer.



(a) AFLNet (b) Green-Fuzz

Fig. 3: Sending a trace of input messages in AFLNet (a) vs Green-Fuzz (b). By sending all messages/entire trace in one go, unlike one by one in AFLNet, we save overhead from context switches and system-calls.

When Fast-desock+ has sent all input messages and saved all the respective responses into the buffer, it sends the responses back to Green-Fuzz in one go. These responses are a list of responses. Because an individual input message m_n can produce several responses or none. Therefore, we must associate each input message with its responses, and so we need a list of pairs, including input and its respective response in each pair. We send the list of responses as a list of pairs to the Green-Fuzz. This list of tuples would be in the form of $\{(n, r)|r \in R = \{r_1, ..., r_j\}\}$, where n is the index of the input message and r is the respective response to input message m_n. This way, the Green-Fuzz can relate the input messages and their respective response (or responses).

The difference between Fast-desock+ and Desock+ is that Fast-desock+ also hooks `sendto`, `recvfrom`, and `setsockopt` to intercept and buffer trace of input messages and responses between the fuzzer and SUT.

Figure 3-a shows the AFLNet interaction with the SUT, where the fuzzer sends each input message one by one. Figure 3-b shows the Green-Fuzz interaction with the SUT, which sends a trace of the input messages to the SUT in one go.

6 Process integration strategies (tackling root cause 3)

This section delves into strategies by which fuzzers can integrate with the operating system, which are as follows:

1. S3A: Out-process fuzzing.
2. S3B: Persistent mode fuzzing (reduces overhead O3, for every input trace).

3. S3C: In-process fuzzing (reduces overhead O1 and O3, for every input message and trace).

The strategy S3C, tackles root cause 3, but can also tackle root cause 2, because in-process fuzzing usually comes with in-memory fuzzing. Each strategy has its advantage and disadvantages, which we discuss in detail in this section.

6.1 Out-process fuzzing

One strategy used in fuzzing is out-process fuzzing, which involves the fuzzer working outside the SUT as a separate process. This strategy is more straightforward because it does not need SUT modification and can be done without much manual work. However, it's not very fast and uses much computational power. Tools like AFLNet [13] and StateAFL [21] use out-process fuzzing strategy, which includes creating a copy of the SUT process (process forking) for every new input trace and then stopping the process when needed with termination (killing process). This strategy is beneficial for fuzzing SUTs where the source code is unavailable, or SUT modification is complex, which cannot be done with in-process fuzzing.

6.2 Persistent mode fuzzing

AFL++ persistent mode offers a take on out-process fuzzing by embedding a large while loop within SUT. This loop begins after the SUT is initialized and finishes just before the SUT terminates, effectively avoiding the overhead caused by initialization and termination of the SUT. This strategy speeds up the fuzzing process by getting rid of the overhead linked to initialization and termination. However, making this work requires modification of the SUT, which is a drawback.

AFL* [32] is a fuzzer based on AFL++ [22] persistent mode. By leveraging AFL++ persistent mode, AFL* effectively bypasses the overheads associated with initialization and termination. However, this strategy introduces the need to reset the SUT after sending each trace. Resets can be soft or hard, depending on the nature of the SUT options. For example, a soft reset can be used if the SUT supports a "quit" command, eliminating reset-related overhead. A hard reset becomes mandatory for fuzzing stateful systems in scenarios lacking a soft reset, reintroducing significant overhead. This variance underscores the importance of understanding the specific requirements and capabilities of the protocol being fuzzed to optimize the efficiency of the AFL* strategy.

6.3 In-process fuzzing

For the in-process fuzzing strategy [7][1], users must modify the SUT manually as it incorporates a loop within the SUT similar to persistent mode (see section 6.2). After the SUT processes each input message, it jumps back to the start point for processing another input, thus avoiding the overhead of initialization

and termination. This results in a significantly faster fuzzing process. A critical distinction between AFL++ persistent mode and in-process fuzzing lies in integrating the fuzzer within the SUT, enabling it to mutate inputs directly. This integration ensures that the fuzzing and the SUT operate as a single process, avoiding context switches. While in-process fuzzing often uses in-memory communication to achieve efficiency, it is not always the case. However, it is important to note that employing an in-memory fuzzing strategy depends on using an in-process strategy, highlighting the intertwined nature of these strategies for enhancing fuzzing effectiveness.

.

7 Analysing and comparing all Strategies

This section analyzes all strategies employed for fuzzing network protocols listed in sections 4-6 to understand their performance gain. We have two sets of data:

- We measured overheads of individual system-calls, context-switching, and time spent on the processes while using each fuzzing strategy.
- We implemented strategy S1A (desock+) and S1A+S2B (Green-Fuzz) and have data from experiments using this implementation.

7.1 Comparing all strategies

We evaluate and compare all strategies discussed in sections 4-6, using data from our experiments to measure the potential reduction in overhead by each strategy. The data is gathered using the following approach:

- We considered the time taken by network-related system calls from fuzzer and SUT.
- We considered the context switching between the SUT and compared the time taken for Green-Fuzz with its baseline.
- We added break points after initialization and termination to see how long it takes for the SUT to process the input messages.
- We monitored the system-calls and execution time for both in-process and out-process fuzzers.

The percentages shown in Table 2 represent the reduction in overhead for processing a single input trace (for each specific overhead root cause) rather than an entire fuzzing campaign comprised of many traces. Consequently, the performance gain differences are expected to amplify as the number of traces increases. In front of each performance gain percentage, we have also shown the time gained by each strategy. We have also shown the impact of each strategy on the performance gain (more + means more performance gain), which relates to the percentage of the performance gain for each overhead root cause. Strategy S1B and S2B have almost the same impact but in different overheads. These two strategies are orthogonal and can be used together.

According to the information in Table 2, the in-memory and in-process strategy demonstrates the highest overhead reduction among all strategies evaluated. However, as outlined in section 6.3, this significant performance gain requires modifications to the SUT, which might not always be feasible. It is important to note that we have omitted any strategies that failed to show a performance improvement.

Table 2: Comparing all strategies and their performance gain. The performance gain percentage is per overhead root cause. For example simulated network stack reduces the network stack overhead by 50%. We also show the absolute gained time is in milliseconds. The dash (-) means no performance gain.

Strategy	**Reduction in O1**		**Reduction in O2**		**Reduction in O3**		**Impact**
S1A: simulated network stack	50%	15 ms	-	-	-	-	+
S1B: shared-memory	70%	21 ms	-	-	-	-	++
S2B: All-in-one-go	-	-	78%	8 ms	-	-	++
S3B: persistent mode	-	-	-	-	96%	45 ms	+++
S1C+S3C: in-memory + in-process	92%	28 ms	100%	10 ms	96%	45 ms	++++

7.2 Evaluation of the strategy S1A on ProFuzzBench

We have used AFLNet with and without Desock+ to evaluate the fuzzing speed. Both sets of fuzzing experiments have been done with an identical setup on the five SUTs from ProFuzzBench[12]. ProFuzzBench is a benchmark that is used for the evaluation of fuzzers for stateful systems.

We ran our experiment five times to ensure the speed is consistent. Each time the fuzzing went on for an hour. Table 3 shows the execution speed of AFLNet with and without Desock+. We see that the speed of fuzzing traces of input messages per second is up to four times faster using Desock+.

Table 3: Speed in message per second, of AFLNet with and without Desock+ on ProFuzzBench [12].

SUT	**AFLNet**	**AFLNet with Desock+**	**Speed up**
lightFTP	12	49	+308%
dnsmasq	15	19	+26%
live555	14	29	+107%
dcmqrscp	17	21	+23%
tinydtls	12	19	+58%

7.3 Evaluation of the strategy S1A+S2B (Green-Fuzz) on ProFuzzBench

To show the benefits of Green-Fuzz, we evaluate it on ProFuzzBench [12]. After that, we compare the absolute fuzzing overhead and its difference between AFLNet using Desock+ and Green-Fuzz.

Table 4 shows the execution speed of Green-Fuzz compared to the AFLNet using Desock+. Five of the ten SUTs included in ProFuzzBench use the socket options our tool supports. We fuzzed the SUTs for an hour and repeated our experiment to ensure the numbers are reliable. The results show that the trace of input messages fuzzed per second is higher when using Green-Fuzz than AFLNet using Desock+, but not that much.

We used ptrace to monitor system-calls that are a source of the overhead while fuzzing. Table 5 shows the absolute overhead difference, where we can see Green-Fuzz decreases overhead in `recvfrom`, `sendto`, `setsockopt`, and `connect` system-calls. There is no change in overhead regarding the `kill` and `clone` system-calls because both AFLNet and Green-Fuzz are out-process fuzzers and have to use these system-calls for each trace of input messages.

Table 4: Speed in message per second, of AFLNet with Desock+ and Green-Fuzz on ProFuzzBench.

SUT	**AFLNet with Desock+**	**Green-Fuzz**	**Speed up**
lightFTP	49	64	+30%
dnsmasq	19	19	0%
live555	29	31	+6%
dcmqrscp	21	25	+19%
tinyDTLS	19	34	+78%

Table 5: Comparison of absolute system-call overhead between AFLNet and Green-Fuzz. The times are in milliseconds (from an example SUT) and shown in the format of $n \times m \times time$ where n is the number of traces and m is the number of messages in one trace.

System-call	**AFLNet**	**Green-Fuzz**	**Overhead Difference**
`clone`	$n \times 6.5$	$n \times 6.5$	0%
`kill`	$n \times 8.7$	$n \times 8.7$	0%
`recvfrom`	$n \times m \times 1.2$	$n \times 1.2$	−80%
`sendto`	$n \times m \times 1.3$	$n \times 1.3$	−80%
`setsockopt`	$n \times m \times 0.1$	$n \times 0.1$	−80%
`connect`	$n \times 11$	$n \times 4$	−63%

8 Related Work

Using grey-box fuzzing solutions to test network services has become a popular research topic. One example is Peach* [18], which combined code coverage feedback with the original Peach [19] fuzzer to test Industrial Control Systems (ICS) protocols. It collected code coverage information during fuzzing and used Peach's capabilities to generate more effective test cases.

IoTHunter [20] applied grey-box fuzzing for network services in IoT devices. It used code coverage to guide the fuzzing process and implemented a multi-stage testing approach based on protocol state feedback.

AFLNet [13] is a grey-box fuzzer for protocol implementations which uses state feedback to guide fuzzing. It acts as a client and replayed different variations of the original message sequence sent to the server. It kept the variations that increased code or state space coverage effectively.

StateAFL [21] is a variation of AFLnet that utilizes a memory state to represent the service state. It instrumented the target server during compilation and determined the current protocol state at runtime. It gradually built a protocol state machine to guide the fuzzing process.

8.1 Related work with Desock+

Zeng et al. [17] also made a simulated socket library, named Desockmulti, to avoid network communication overhead when fuzzing network protocols. However, compared to Desock+, Desockmulti does not support `connect` and `accept4` system-calls, which limits its applicability.

Maier et al. [11] introduced the Fuzzer in the Middle (FitM) for fuzzing network protocols. Instead of using a simulated socket library, FitM intercepts the emulated system-calls inside the QEMU emulator and sends the input messages to the SUT without the network communication overhead. Because FitM has emulation overhead, it is slower than our approach. Compared to our approach, FitM has the capability to fuzz both the client and server of a network protocol as the SUT.

There are also ad-hoc approaches [4][5] [25] where by manually modifying the SUT, the fuzzer would send the input messages to the SUT without network communication. These approaches change the source code of SUT to read the inputs from a file or argument variables to avoid network communication. These approaches require manual effort for each SUT, which is not ideal, but are more stable because of SUT specific fuzzing harnesses that are built per SUT.

8.2 Related work with Green-Fuzz

Nyx-Net [14] utilizes hypervisor-based snapshot fuzzing incorporated with the emulation of network functionality to handle network traffic. Nyx-Net uses a customized kernel module, a modified version of QEMU and KVM, and a custom VM configuration where the target applications are executed. Nyx-Net also contains a custom networking layer miming certain POSIX network functionalities,

which currently needs more support for complicated network targets. In contrast, Green-Fuzz adopts a user-mode approach that avoids complexity. Green-Fuzz is also an orthogonal approach to be added on top of Nyx-Net, to speed up the fuzzing.

In-process (a.k.a in-memory) fuzzing [1][7] is an approach where a fuzzer does not restart or fork the SUT for each trace of input messages, and the fuzzing is done within the same process. The input values are mutated inside the memory. Therefore, it also avoids network communication overhead. However, these methods involve manual work to modify a piece of code as the SUT and specifying the exact position of variables inside the memory. Using a simulated socket library, Green-Fuzz does not require these manual steps. However, in-process fuzzing is faster (around 200 to 300 times in our experiments) than our approach because it has less fuzzing overhead. Another issue of in-process fuzzing is that usually it can not test the whole system, because of the fuzzing loop that is defined for the harness.

9 Limitations and Future Work

Currently, Desock+ only works with the SUTs using system-calls and their arguments shown in in Table 1. Some SUTs use other socket options. For example, input arguments for `epoll` system-call must be simulated in Desock+ to work correctly if the SUT is using this system-call. Because part of Green-Fuzz is based on Fast-desock+, these limitations also apply to Green-Fuzz. Since the non-simulated options in Desock+ and Fast-desock+ can be complex. As our future work, we would like to complete these engineering efforts and use Green-Fuzz to fuzz network protocols such as OPC-UA [23] and Modbus [24] protocols. To Fuzz protocols that require a handshake, the Green-Fuzz needs a minor modification to do the handshake before sending the whole trace in one go.

Desock+ and Green-Fuzz are general solutions to fuzzers for stateful systems, so we plan to apply them to other fuzzers for stateful systems. In this article, we applied our improvements to AFLNet. However, any fuzzer that sends the inputs to a network protocol via network sockets or sends the input messages from a trace of messages can be upgraded by using our solutions, except if it needs feedback after sending each input message, as [10] does. For example, SGPFuzzer [16] and Nyx-net [14] are network protocol fuzzers that can be upgraded by Green-Fuzz, to have an efficient fuzzer. The modification for applying Green-Fuzz to other fuzzers is relatively simple. The user has to modify the harness to send and receive the messages in a trace format to the SUT.

9.1 Recommendations for software developers to make fuzzer friendly network protocol implementations

We propose that software developers implementing network protocols consider reducing the overhead and complexities typically associated with the SUT when the tester wants to fuzz the SUT. By adopting these strategies, implementers

can enhance the efficiency and effectiveness of fuzzing, leading to quicker and more thorough vulnerability detection.

- **Incorporate a Restart Message:** Implement a 'restart' message within the protocol. This feature allows the fuzzer to send a specific command to reset the state of the protocol and refresh all variables. It is beneficial in stateful fuzzing, as it permits quick resetting without requiring complete process reinitialization, saving time and resources.
- **Disable Encryption Mechanisms:** Develop a mechanism to turn off all encryption during fuzzing temporarily. This enables fuzzers to avoid the complexities of handling encrypted messages, managing encryption keys, or performing cryptographic handshakes, thus simplifying the fuzzing process and focusing on fuzz testing.
- **Support for Alternate Input/Output Mechanisms:** Provide support for receiving and sending inputs/outputs through operating system pipes or standard input/output instead of the network interface. This alternative startegy can significantly reduce the overhead involved in the network stack, allowing for faster and more direct data transmission between the fuzzer and the SUT.

10 Conclusion

In conclusion, fuzzing emerges as a powerful method for uncovering bugs and security flaws within software systems. Yet, its application to network protocols has faced limitations, primarily due to reduced throughput. This article delves into the root causes of overhead in fuzzing network protocols, thoroughly examining all strategies to reduce or avoid these strategies. We explored and categorized all strategies, assessing the advantages and disadvantages of each to offer a comprehensive view. Our analysis, including insights from our prior article and additional research, indicates that in-memory and in-process fuzzing strategies are the fastest fuzzing strategy. However, this efficiency often requires modifications to the SUT, which may not always be desirable or feasible. For scenarios where modifying the SUT is not an option, employing an out-process fuzzer, particularly one that utilizes sending a whole trace in one go and using shared memory, presents the next best strategy for enhancing fuzzing speed. Our overview provides better insight into choosing the appropriate strategies for fuzzing network protocols. This paves the way for more effective and efficient identification of vulnerabilities in network protocols.

11 Conflicts of Interest

Not applicable

References

1. Libfuzzer. 2023. A ibrary for coverage-guided fuzz testing. Retrieved Feb 2, 2023 from `https://llvm.org/docs/LibFuzzer.html`
2. Zardus. 2023. preeny. Retrieved Jan 6, 2023 from `https://github.com/zardus/preeny`
3. Google. 2022. ClusterFuzz Trophies. Retrieved Feb 12, 2023 from `https://google.github.io/clusterfuzz/#trophies`
4. Nicola Tuveri. 2021. Fuzzing open-SSL. Retrieved Feb 6, 2023 from `https://github.com/openssl/openssl/blob/master/fuzz/README.md`
5. Wayne Chin Yick Low. 2022. Dissecting Microsoft IMAP Client Protocol. Retrieved Feb 6, 2023 from `https://www.fortinet.com/blog/threat-research/analyzing-microsoft-imap-client-protocol`
6. Aschermann, Cornelius, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. "Ijon: Exploring deep state spaces via fuzzing." In 2020 IEEE Symposium on Security and Privacy (SP), pp. 1597-1612. IEEE, 2020.
7. Ba, Jinsheng, Marcel Böhme, Zahra Mirzamomen, and Abhik Roychoudhury. "Stateful greybox fuzzing." In 31st USENIX Security Symposium (USENIX Security 22), pp. 3255-3272. 2022.
8. Cui, Baojiang, Fuwei Wang, Yongle Hao, and Xiaofeng Chen. "WhirlingFuzzwork: a taint-analysis-based API in-memory fuzzing framework." Soft Computing 21 (2017): 3401-3414.
9. Daniele, Cristian, Seyed Behnam Andarzian, and Erik Poll. "Fuzzers for stateful systems: Survey and Research Directions." arXiv preprint arXiv:2301.02490 (2023).
10. Isberner, Malte, Falk Howar, and Bernhard Steffen. "The TTT algorithm: a redundancy-free approach to active automata learning." In Runtime Verification: 5th International Conference, September 22-25, 2014. Proceedings 5, pp. 307-322. Springer, 2014.
11. Maier, Dominik, Otto Bittner, Marc Munier, and Julian Beier. "FitM: Binary-Only Coverage-Guided Fuzzing for Stateful Network Protocols." In Workshop on Binary Analysis Research (BAR), vol. 2022.
12. Natella, Roberto, and Van-Thuan Pham. "Profuzzbench: A benchmark for stateful protocol fuzzing." In Proceedings of the 30th ACM SIGSOFT international symposium on software testing and analysis, pp. 662-665. 2021.
13. Pham, Van-Thuan, Marcel Böhme, and Abhik Roychoudhury. "AFLNet: a greybox fuzzer for network protocols." In 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST), pp. 460-465. IEEE, 2020.
14. Schumilo, Sergej, Cornelius Aschermann, Andrea Jemmett, Ali Abbasi, and Thorsten Holz. "Nyx-net: network fuzzing with incremental snapshots." In Proceedings of the Seventeenth European Conference on Computer Systems, pp. 166-180. 2022.
15. Sutton, Michael, Adam Greene, and Pedram Amini. Fuzzing: brute force vulnerability discovery. Pearson Education, 2007.
16. Yu, Yingchao, Zuoning Chen, Shuitao Gan, and Xiaofeng Wang. "SGPFuzzer: A state-driven smart graybox protocol fuzzer for network protocol implementations." IEEE Access 8 (2020): 198668-198678.
17. Zeng, Yingpei, Mingmin Lin, Shanqing Guo, Yanzhao Shen, Tingting Cui, Ting Wu, Qiuhua Zheng, and Qiuhua Wang. "Multifuzz: A coverage-based multiparty-protocol fuzzer for iot publish/subscribe protocols." Sensors 20, no. 18 (2020): 5194.

18. Luo, Zhengxiong, Feilong Zuo, Yuheng Shen, Xun Jiao, Wanli Chang, and Yu Jiang. "ICS protocol fuzzing: Coverage guided packet crack and generation." In 2020 57th ACM/IEEE Design Automation Conference (DAC), pp. 1-6. IEEE, 2020.
19. Mozilla Security. 2021. Peach. Retrieved Feb 2, 2023 from `https://github.com/MozillaSecurity/peach`
20. Yu, Bo, Pengfei Wang, Tai Yue, and Yong Tang. "Poster: Fuzzing IoT firmware via multi-stage message generation." In Proceedings of the 2019 ACM SIGSAC conference on computer and communications security (CCS 2019), pp. 2525-2527. 2019.
21. Natella, Roberto. "StateAFL: Greybox fuzzing for stateful network servers." Empirical Software Engineering 27, no. 7 (2022).
22. Fioraldi, Andrea, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. "AFL++: Combining incremental steps of fuzzing research." In 14th USENIX Workshop on Offensive Technologies (WOOT 20). 2020.
23. The OPC foundation 2023. The OPC Unified Architecture (UA). Retrieved April 2, 2023 from `https://opcfoundation.org/about/opc-technologies/opc-ua/`
24. Modbus organization. 2023. Modbus data communications protocol . Retrieved April 2, 2023 from `https://modbus.org/`
25. Cheremushkin, Temnikov. OPC UA security analysis 2023. Technical report, Kaspersky. Retrieved April 14, 2023 from `https://ics-cert.kaspersky.com/publications/reports/2018/05/10/opc-ua-security-analysis/`
26. Serebryany, Kostya. "OSS-Fuzz-Google's continuous fuzzing service for open source software." (USENIX 2017).
27. Klooster, Thijs, Fatih Turkmen, Gerben Broenink, Ruben Ten Hove, and Marcel Böhme. "Continuous Fuzzing: A Study of the Effectiveness and Scalability of Fuzzing in CI/CD Pipelines." In 2023 IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT), pp. 25-32. IEEE, 2023.
28. Gorter, Floris, Enrico Barberis, Raphael Isemann, Erik van der Kouwe, Cristiano Giuffrida, and Herbert Bos. "FloatZone: How Floating Point Additions can Detect Memory Errors." (USENIX 2023)
29. Andronidis, Anastasios, and Cristian Cadar. "Snapfuzz: high-throughput fuzzing of network applications." In Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 340-351. 2022.
30. Andarzian, Seyed Behnam, Cristian Daniele and Erik Poll. "Green-Fuzz: Efficient Fuzzing for Network Protocol Implementations" In Proceedings of the 16th International Symposium on Foundations and Practice of Security (FPS – 2023).
31. Geretto, Elia, Cristiano Giuffrida, Herbert Bos, and Erik Van Der Kouwe. "Snappy: Efficient Fuzzing with Adaptive and Mutable Snapshots." In Proceedings of the 38th Annual Computer Security Applications Conference, pp. 375-387. 2022.
32. Anonymous. "AFL*: A Simple Approach to Fuzzing Stateful Systems." OpenReview Preprint. 2024.
33. POSIX shared memory. Retrieved Feb 6, 2024 from `https://man7.org/linux/man-pages/man7/shm_overview.7.html`