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 widely used and effective technique to test software. Unfortunately, certain types of systems, including network protocols, are more challenging to fuzz than others. An important complication with fuzzing network protocols is that this tends to be a slow process, which is problematic as fuzzing involves many test inputs. This article presents an analysis of the root causes behind the inefficiency of fuzzing network protocols and of strategies to avoid them. It extends earlier work we did on network protocol fuzzers which explored some of this strategies, to give a more comprehensive overview of overheads in fuzzing and ways to reduce 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++ [23] and LibFuzzer [2] have found thousands of bugs in both open-source and commercial software. For example, Google has discovered over 25,000 bugs in their software (e.g., Chrome) and over 36,000 bugs in over 550 open-source projects [4]. Fuzzing involves sending many – tens or hundreds of thousands – (semi)automatically generated inputs to the System-Under-Test (SUT), so the speed of generating and processing many inputs is important.

Unfortunately, not all software can benefit from such fuzzer performance. For instance, network protocols fuzzers struggle to achieve high speeds. Whereas a typical fuzzing campaign with a modern fuzzer like AFL++ [23] on, say, a graphics library will produce thousands of inputs per second [32], a fuzzer like AFLNet [14] for fuzzing network protocols produces only a few dozens of inputs per second. One of the reasons is the overhead of network stacks. To fuzz a network protocol it is common to modify the code to by-pass the network stack [26]. In addition to this, network protocols are often stateful, and statefulness also influences fuzzer performance [10]: to fuzz a stateful network SUT, a fuzzer

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

is required to send *sequences* of messages (also called traces), instead of single message. Moreover, context switches between the fuzzer and the SUT can further slow down the fuzzing speed here.

In this paper the focus is on generic techniques that can be implemented in any fuzzer to increase its speed, but we also explore techniques – used in the state-of-the-art tools – that require ad-hoc modification of the SUT.

In an earlier article [31], we reported results of two different strategies (implemented in a library Desock+ and in a fuzzer called Green-Fuzz) to improve the efficiency of fuzzing network protocols. Desock+ reduces the communication overheads (root cause O1, network stack overhead, in Section 3.1). Green-Fuzz reduces the context switches between fuzzer and SUT (root cause O2, context switching, in Section 3.2). This article extends the previous paper by providing a comprehensive analysis of the root causes of performance overhead in fuzzing network protocols and strategies to tackle them.

The contribution of the paper is three-fold:

- We analyze all the root causes of overheads in network protocol fuzzing.
- We provide all the strategies that we know of to tackle these root causes, including the two presented in the earlier article [31].
- We provide a comprehensive analysis of the strategies and their impact, including experimental data to quantify the impact.

Section 2 presents the background. Section 3 delves into the types of overheads encountered in fuzzing network protocols. Section 4 discusses strategies to overcome the communication overheads that slow down the fuzzing network protocols. Section 5 focuses on strategies to reduce the overhead caused by context switching between the fuzzer and the SUT. Section 6 explores strategies to address the initialization and termination overheads. Section 7 analyses and compares these strategies. In Section 8, we review the related work and in Section 9 we discuss future research directions and the limitations of the article. Finally, in Section 11, 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, as dynamic code testing technique, is very useful for identifying such vulnerabilities. As fuzzing relies on sending *many* inputs to the SUT, performance is very important.

Time is also critical when it comes to integrating fuzzing in the CI/CD [1] pipelines. As mentioned in [28], 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 speed of fuzzing tends to be very poor when fuzzing network protocol implementations. In fact, fuzzing regular command line software is, on average, 100

[1] Continuous Integration/Continuous Deployment.

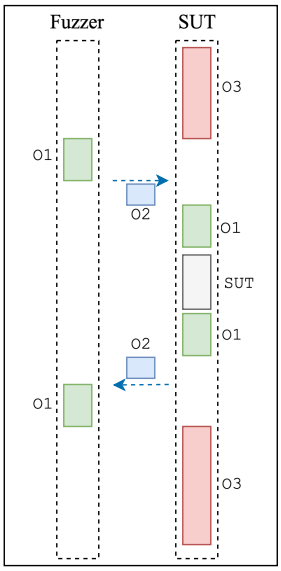


Fig. 1: Overheads of fuzzing a network protocol for a trace with one input message. The green color refers to overhead root cause O1, blue is for root cause O2 and red is for root cause O3.

times faster than fuzzing network protocol implementations[2]. This motivated us to more research and find hurdles in efficiently fuzzing network protocol implementations.

3 Types of Overhead in Network Protocol Fuzzing

In fuzzing network protocols, we identified three kinds of overhead, as shown in Figure 1:

- **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.
- **Context switching overhead (O2)** happens when there is a context switch between the fuzzer and the SUT.
- **SUT initialization and termination overhead (O3)** happens every time the SUT is initialized (i.e. started up) and terminated.

3.1 Network Stack Overhead (O1)

When testing an application that we can run on the same machine as the fuzzer, the whole network stack is obviously not really needed. All the overhead that

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

the network stack introduces only makes the fuzzing slower and less effective. The network-related system calls also add overhead here. For example, to send a message and receive a response, the fuzzer would need to call the *setsockopt*, *sendto* and *recvfrom* system-calls.

3.2 Context Switching Overhead (O2)

In fuzzing network protocols, the context switching overhead in operating systems becomes important. Fuzzing involves rapidly sending and receiving many inputs and outputs to/from the SUT, which requires frequent context switches 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 can consume significant system resources, especially if there is a high frequency of switches.

Furthermore, the cache invalidation caused by these switches — where previously loaded cache data becomes irrelevant after a context-switch — can lead to additional memory reads, and hence more overhead. This can significantly affect the efficiency of fuzzing.

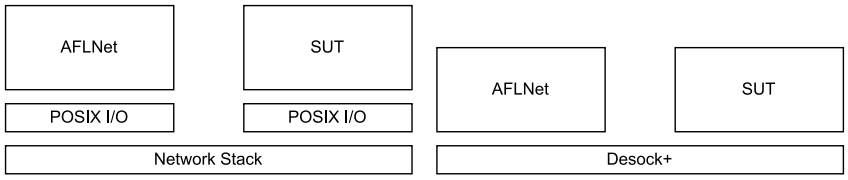
3.3 SUT Initialization and Termination (O3)

In fuzzing network protocols, a new SUT process must be created for each input trace. That process needs to be initialized, which will involve running constructors and initialization functions. This leads to some *initialization overhead*. After processing each input trace, the SUT must be terminated. The termination process involves the *kill* system call. This leads to *termination overhead*. This can be expensive as the operating system needs to ensure all resources allocated to the process are properly released.

The cumulative effect of repeatedly initializing and terminating the SUT for each input trace can significantly slow down the fuzzing process. This overhead is not specific to network protocols and is present for fuzzing all types of SUT. Instead of starting a process from scratch for each input, some fuzzers (including AFL) use the *fork* system-call to clone the SUT process to reduce the initialization overhead. That does leave the overhead of the *fork* system call of course, and does not avoid the overhead of terminating the old SUT process.

4 Mitigating Network Stack Overhead (O1)

As discussed above, using real network communication to fuzz network protocols introduces overhead. Despite this drawback, it remains a popular choice among many fuzzers, including AFLNet [14], AFLNwe [23] and StateAFL [22]. These fuzzers work by sending inputs and receiving responses through real network sockets and thus suffer the overhead of the — slow — network stack. Strategies to avoid this overhead include 1) using a simulated network stack, 2) using shared-memory between the fuzzer and SUT (which still run as separate processes) and



(a) AFLNet (b) AFLNet using Desock+

Fig. 2: Comparison between plain AFLNet and AFLNet using Desock+.

3) using in-memory communication (which requires fuzzer and SUT to be run as a single process). These three strategies all reduce O1 for every input message. We discuss them in more detail below.

Simulated network stack (Desock+) Our Desock+ library [31] provides a simulated network socket that works with any fuzzer to avoid network communication overhead. Desock+ does not require emulation or any modification of the source code of the SUT. It is a modified version of the *preeny* library [3], which communicates with the SUT via the standard I/O.

Desock+ can be used by the SUT instead of the standard POSIX library to fuzz more efficiently. The overview of a fuzzer working with Desock+ is shown in Figure 2. In this case, the fuzzer is a slightly modified version of AFLNet which sends and receives input messages through standard I/O instead of network sockets. The SUT remains unchanged; the only thing that changes is the underlying socket library which the SUT would load instead of the real socket library.

The difference between *preeny* and Desock+ is that *preeny* can not support specific socket-related system-calls and arguments. Table 1 lists the system calls that Desock+ supports and that allow it to deal with SUTs that:

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

The modifications concern the *socket* system-call, which is responsible for creating the socket file descriptor. More in detail, we added a function named *setup* to modify 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.

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

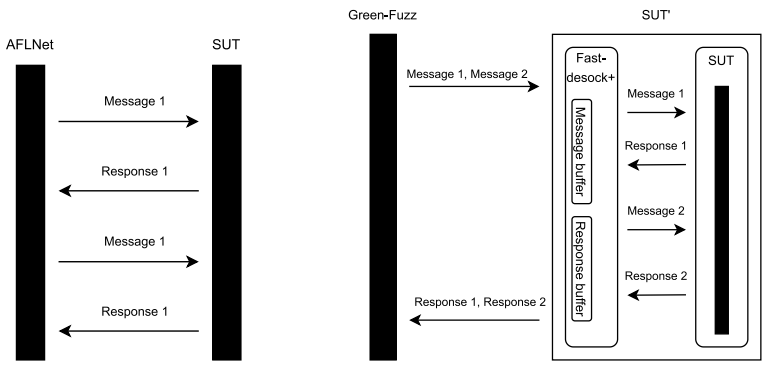
System-Call	Arguments	System-Call	Arguments
socket()	*AF_LOCAL*	*connect3()*	*SOCK_NONBLOCK*
	AF_INET		*SOCK_CLOEXEC*
	AF_INET6		*SOCK_SEQPACKET*
	SOCK_STREAM		*SOCK_DGRAM*
	SOCK_DGRAM		*SOCK_STREAM*
	SOCK_SEQPACKET	*dup3()*	*SOCK_NONBLOCK*
	SOCK_RAW		*SOCK_CLOEXEC*
	SOCK_RDM	*recv()*	*MSG_CMSG_CLOEXEC*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

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[3]. 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.

Shared memory Using shared memory [34] for fuzzing network protocols increases fuzzing network protocols performance because communications take place 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. It eliminates the overheads associated with network stack or file-based communication. So there is potential to further enhance a library like Desock+ by adapting it to use shared memory instead of files.

In-memory If we integrate the SUT and fuzzer into a single process — a strategy called in-process fuzzing — then the overhead to pass data between them can be reduced further still. In-process fuzzers can simply mutate a variable in memory in each fuzzing round. This is called in-memory fuzzing. This differs from shared-memory, where the inputs are sent from the fuzzer process to the

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



(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.

SUT process via shared-memory (pipes in Linux). In-memory communication is the fastest strategy for fuzzer and SUT to communicate, as it avoids the overhead of a real network stack, a simulated network stack, or shared memory.

As mentioned above, this approach only works with in-process fuzzing, which involves compiling the SUT code in such a way that the fuzzer is directly embedded within it. LibFuzzer [2] uses this approach. In-process fuzzing not only reduces the overhead of communication over the network stack (O1): it also provides ways to tackle the overhead of context switches (O2) and the initialization and termination of the SUT (03), as discussed in Section 6.

5 Mitigating Context Switching Overhead (O2)

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

1. Sending input messages one by one.
2. Sending a sequence of messages in one go (reduces overhead O2, for every input trace).

Sending input messages one by one The classical approach of network protocol fuzzers (e.g., AFLNet, AFLNwe, StateAFL) is to send 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.

Sending input messages in one go (GreenFuzz) In our previous article [31], we presented Green-Fuzz, a new strategy 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 $T =< m_1, m_2, ..., m_i >$, and send the input message m_i 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 [14].

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 and sends it to the SUT in one go. Likewise, it intercept and save all the SUT responses into a response buffer before forwarding them to the fuzzer.

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 Mitigating SUT Initialization and Termination (O3)

When it comes to the overhead of initializing and termination the SUT (O3), we can distinguish four strategies:

1. Out-process fuzzing;
2. Snapshotting and forking(reduces overhead O3, for every input trace);
3. Persistent mode fuzzing (reduces overhead O3, for every input trace);
4. In-process fuzzing (reduces overhead O1 and O3, for every input message and trace).

Out-process fuzzing Here the fuzzer and SUT run as separate processes. Most fuzzers, including most network protocol fuzzers, take this approach, for example AFLNet [14] or StateAFL [22].

Advantage of this approach is that it does not require any modification of the SUT, unlike the persistent mode and in-process strategies below, which do require manual work to change the code of the SUT. So it is suited to situations where the source code is unavailable or SUT modification would be complex.

Downsides are the overheads: in the communication between the two processes, i.e. O1 (using one of the techniques discussed in Section 4), in the context

switches between the processes, i.e. O2, and the initialization and termination of the SUT process, i.e. O3.

Snapshotting Snapshotting consists of saving (or *snapshotting*) the SUT at a certain point in order to avoid the initialization and termination overheads.

For stateless systems, it is usually useful to snapshot the SUT after the initialization phase and just before the handling of the inputs to avoid unnecessary initialization and termination overhead.

For the stateful network SUT, snapshotting can do more. In fact, fuzzers like Nyx-Net [1] use snapshots to capture the program state at a particular point and then avoid the costs of re-sending the entire trace to reach a certain state. Unfortunately, the snapshotting mechanisms itself is often quite expensive.

Persistent fuzzing The basic idea behind persistent fuzzing is to avoid the overhead of initializing and terminating the SUT. To avoid these overheads, persistent fuzzing allows the SUT to process multiple inputs before shutting down and restarting.

This idea was initially applied only to fuzz stateless SUTs, like graphic routines that process a single JPEG and then terminate. Note that even if for such SUTs we might not have the overhead of a network connection (O1), we do have the overhead of initializing and terminating SUT processes (O3). Enabling the persistent mode for such systems requires the SUT modification. This is typically done by adding a while loop around the code that accepts and then processes inputs. This allows having the overhead O3 only once in a while (i.e., when the loop ends), and not for each input. Persistent fuzzing is known to give a huge speed-up. The AFL++ documentation claims that it is easily 10 or 20 times faster[4] – and hence the method of choice for the most fuzzing campaign.

Things are slightly different when we apply the idea of persistent fuzzing in network protocols. In fact, network protocols already do *some* persistent fuzzing. In fact, they usually process several messages (a trace) without restarting the SUT. The persistent mode is still useful for stateful network SUT when we want to process several traces without restarting the SUT. In this case, we need a *soft* reset command to bring the SUT back to the initial state.

For example, AFL* [33] is a network protocol fuzzer based on AFL++ [23] persistent mode. By leveraging AFL++ persistent mode, AFL* avoids the overheads associated with initialization and termination. However, this strategy introduces the need to reset the SUT after sending each trace. Such 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 network SUT in scenarios lacking a soft reset, reintroducing significant overhead. This variance underscores the importance of understanding the specific requirements

[4] `https://github.com/AFLplusplus/AFLplusplus/blob/sable/instrumentation/README.persistent_mode.md`

and capabilities of the protocol being fuzzed to optimize the efficiency of the AFL* strategy.

In-process fuzzing The idea behind in-process fuzzing is to run the fuzzer and the SUT into a single process. This reduces or even eliminates a lot of overheads: there are no context switches between SUT and fuzzer and communication can happen in-memory[5].

In-process fuzzing typically goes hand-in-hand with persistent fuzzing as is ideal not to start a new process for each input (in the case of stateless SUT) or each trace (in the case of stateful network SUT). A critical distinction between persistent mode and in-process fuzzing lies in the way they integrate with the SUT. In fact, in-process fuzzing allows the fuzzing and the SUT to operate as a single process, avoiding context switches.

In-process fuzzing [8][2] (like persistent mode) requires the users to modify the SUT code manually. After the modification, the SUT is capable of sending multiple messages avoiding the overhead of initialization and termination. This results in a significantly faster fuzzing process. 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 Analyzing and comparing strategies

This section analyzes the effectiveness of each strategy presented in Sections 4,5, 6. For each strategy, we measure the overheads of the individual system-calls, context-switching, and time spent on the process. Moreover, we compare these results with the ones present in our previous paper about Desock+ and Green-Fuzz.

7.1 Evaluation criteria

We evaluate and compare the strategies discussed before to measure the potential reduction in overhead by each strategy. In our analysis we:

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

[5] Be aware that in-process and in-memory are not synonyms.

Table 2: Comparing strategies presented in Sections 4,5, 6 and their performance gain on LightFTP protocol. 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**
Simulated network stack	50%	15 ms	-	-	-	-	+
Shared-memory	70%	21 ms	-	-	-	-	++
All-in-one-go	-	-	78%	8 ms	-	-	++
Persistent mode	-	-	-	-	96%	45 ms	+++
In-memory	92%	28 ms	100%	10 ms	96%	45 ms	++++

Table 3: Speed in message per second, of AFLNet with and without Desock+ on ProFuzzBench [13].

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%

Table 2 presents the reduction in overhead for processing a single input trace for each specific overhead root cause. We expect the performance gain to grow along with the number of traces taken into account. 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. According to the information in Table 2, the in-memory and in-process strategy demonstrates the highest overhead reduction among all the strategies evaluated. However, as outlined in Section 4, this significant performance gain requires modifications to the SUT, which might not always be feasible.

7.2 Evaluation of Simulated Network Stack (Desock+)

We used AFLNet with and without Desock+ to evaluate the effectiveness of the technique. Both sets of fuzzing experiments have been done with an identical setup on the five SUTs from ProFuzzBench[13], a benchmark framework for for stateful systems.

We ran our experiment five times for an hour. Table 3 shows that using Desock+ increases the fuzzing speed up to four times.

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 on LightFTP protocol. 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, which is 5 in this experiment.

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%

7.3 Evaluation of Sending Multiple Messages in One Go (GreenFuzz)

Table 4 shows the execution speed of Green-Fuzz and AFLNet using Desock+. Five of the ten SUTs included in ProFuzzBench use the socket options our fuzzer supports. We fuzzed the SUTs for an hour and repeated our experiment to ensure the numbers were reliable. The results show that the trace of input messages fuzzed per second is higher when using Green-Fuzz than AFLNet using Desock+.

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.

8 Related Work

Using grey-box fuzzing solutions to test network services has become a popular research topic. One example is Peach* [19], which combines code coverage feedback with the original Peach [20] fuzzer to test Industrial Control Systems (ICS).

It collects code coverage information during fuzzing and uses Peach's capabilities to generate more effective test cases.

IoTHunter [21] applies grey-box fuzzing for network services in IoT devices. It uses code coverage to guide the fuzzing process and implemented a multi-stage testing approach based on protocol state feedback.

AFLNet [14] is a grey-box fuzzer for protocol implementations which uses state feedback to guide fuzzing. It acts as a client which mutates and forwards messages to the SUT.

StateAFL [22] is a variation of AFLnet that uses a memory state to represent the service state. It instruments the target server during compilation and determined the current protocol state at runtime. It gradually builds a protocol state machine to guide the fuzzing process.

8.1 Related work with Desock+

Zeng et al. [18] 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. [12] 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. Since FitM has emulation overhead, it is slower than our approach but has the capability to fuzz both the client and server of a network protocol as the SUT.

There are also ad-hoc approaches [5][6] [26] that manually modify the SUT to get rid of the network stack.

8.2 Related work with Green-Fuzz

Nyx-Net [15] uses 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. 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 speed.

As explained in Section 6, in-process fuzzing [2][8] allows not to restart or fork the SUT for each trace or messages and to mutate the messages in-memory, avoiding the network communication overhead. Although this methods perform better than our approach (around 200 to 300 times in our experiments), it involves manual work to specify the exact position of variables inside the memory. 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 specific system-call. Since part of Green-Fuzz is based on Fast-desock+, these limitations also apply to Green-Fuzz. In the future, we plan to improve Green-Fuzz in order to be able to fuzz network protocols such as OPC-UA [24] and Modbus [25] 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.

In this article, we applied Desock+ and Green-Fuzz to AFLNet. However, these are general solutions that can be applied to any fuzzer that uses network sockets and does not need feedback after every single message, as [11] does. Good candidates for future integration are SGPFuzzer [17] and Nyx-net [15].

10 Recommendations for software developers

In order to reduce the overhead and complexity of the SUT, developers might consider to:

- incorporating a restart message: this feature would allow 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 restart;
- disabling encryption mechanisms: this would help fuzzers fuzzing cryptographic protocols without the need of dealing with encrypted messages, keys, or cryptographic checks;
- supporting standard I/O communication: enabling standard inputs/outputs communication channels (along with the network stack one) will allow for faster and more direct data transmission between the fuzzer and the SUT.

11 Conclusions

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 strategies to reduce or avoid these strategies. We explored and categorized 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.

12 Conflicts of Interest

Not applicable

References

1. Schumilo, Sergej, et al. "Nyx-net: network fuzzing with incremental snapshots." Proceedings of the Seventeenth European Conference on Computer Systems. 2022.
2. Libfuzzer. 2023. A library for coverage-guided fuzz testing. Retrieved Feb 2, 2023 from `https://llvm.org/docs/LibFuzzer.html`
3. Zardus. 2023. preeny. Retrieved Jan 6, 2023 from `https://github.com/zardus/preeny`
4. Google. 2022. ClusterFuzz Trophies. Retrieved Feb 12, 2023 from `https://google.github.io/clusterfuzz/#trophies`
5. Nicola Tuveri. 2021. Fuzzing open-SSL. Retrieved Feb 6, 2023 from `https://github.com/openssl/openssl/blob/master/fuzz/README.md`
6. 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`
7. 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.
8. Ba, Jinsheng, Marcel Böhme, Zahra Mirzamomen, and Abhik Roychoudhury. "Stateful greybox fuzzing." In 31st USENIX Security Symposium (USENIX Security 22), pp. 3255-3272. 2022.
9. 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.
10. Daniele, Cristian, Seyed Behnam Andarzian, and Erik Poll. "Fuzzers for stateful systems: Survey and Research Directions." arXiv preprint arXiv:2301.02490 (2023).
11. 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.
12. 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.
13. 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.

14. Pham, Van-Thuan, Marcel Böhme, and Abhik Roychoudhury. "AFLNet: a grey-box fuzzer for network protocols." In 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST), pp. 460-465. IEEE, 2020.
15. 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.
16. Sutton, Michael, Adam Greene, and Pedram Amini. Fuzzing: brute force vulnerability discovery. Pearson Education, 2007.
17. 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.
18. 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.
19. 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.
20. Mozilla Security. 2021. Peach. Retrieved Feb 2, 2023 from `https://github.com/MozillaSecurity/peach`
21. 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.
22. Natella, Roberto. "StateAFL: Greybox fuzzing for stateful network servers." Empirical Software Engineering 27, no. 7 (2022).
23. 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.
24. The OPC foundation 2023. The OPC Unified Architecture (UA). Retrieved April 2, 2023 from `https://opcfoundation.org/about/opc-technologies/opc-ua/`
25. Modbus organization. 2023. Modbus data communications protocol . Retrieved April 2, 2023 from `https://modbus.org/`
26. 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/`
27. Serebryany, Kostya. "OSS-Fuzz-Google's continuous fuzzing service for open source software." (USENIX 2017).
28. 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.
29. 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)
30. 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.
31. 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).

32. 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.
33. Anonymous. "AFL*: A Simple Approach to Fuzzing Stateful Systems." OpenReview Preprint. 2024.
34. POSIX shared memory. Retrieved Feb 6, 2024 from `https://man7.org/linux/man-pages/man7/shm_overview.7.html`