Green-Fuzz: Efficient Fuzzing for Network Protocol Implementations

No Author Given

No Institute Given

Abstract. Recent techniques have significantly improved fuzzing, discovering many vulnerabilities in various software systems. However, certain types of systems, such as network protocols, are still challenging to fuzz. This article presents two enhancements that allow efficient fuzzing of network protocols. The first is Desock+, which simulates a network socket and supports different POSIX options to make Desock+ suitable for faster network protocol fuzzing. The second is Green-Fuzz, which sends input messages in one go and reduces the system-call overhead while fuzzing network protocols. We applied this modification to AFLNet, but it could be applied to any fuzzer for stateful systems. This is the maximum overhead we can avoid, when doing out-process fuzzing on stateful systems. Our evaluation shows that these enhancements make AFLNet up to four times faster.

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

1 Introduction

Fuzzing (a.k.a. fuzz testing) is an effective technique for testing software systems, with popular fuzzers such as AFL++ [22] and LibFuzzer [1] having found thousands of bugs in both open-source and commercial software. For instance, Google has discovered over 25,000 bugs in their own software (e.g. Chrome) and over 36,000 bugs in over 550 open source projects [3].

Unfortunately, not all software can benefit from such fuzzing campaigns. One important class of software is network protocols which are challenging to fuzz [9] [6]. One of the challenges in fuzzing network protocols is performance overheads caused by the network stack, and context switching between the fuzzer and Software Under Test (SUT).

There are two main approaches for testing such software. One approach is in-process fuzzing [1] to exercise different parts or APIs of the SUT. Although this method is fast and can yield positive results, it needs considerable manual effort (including source code modification), and system-level testing still needs to be performed. Another approach is out-process fuzzing [13], where the fuzzer runs outside of SUT, generates random input messages, and sends them to the SUT. While this approach requires less manual effort and no source code modification, it is very slow and takes a lot of computation power.

We propose two enhancements to have an efficient fuzzer:

- We offer a new method that leverages a simulated socket library named Desock+, a modified version of `preeny`. Unlike `preeny` [2], Desock+ works with a wide range of SUTs, allowing for efficient fuzzing of network protocols.
- We present the Green-Fuzz fuzzer, which sends input message traces in one go to the SUT. Therefore, it can avoid some system-call and context-switching overhead between the fuzzer and SUT.

These enhancements are different: Desock+ is a simulated socket library that can work with any fuzzer, and Green-Fuzz is a network protocol fuzzer that uses a specific version of Desock+ named Fast-desock+.

Our evaluation of the execution speed on ProFuzzBench [12] shows that Green-Fuzz is up to four times faster than AFLNet [13]. We also compared our approach with related work, which shows that our solution has the advantage of supporting more types of SUT for fuzzing, which use complex socket functionalities.

This paper is structured as follows: section 2 presents the background and our motivation for this research. In section 3, we talk about the issue of network communication overhead and how we solve it with Desock+. Section 4 presents our new approach used in Green-Fuzz and its architecture to reduce the overhead in fuzzing. In section 5, we discuss the related work. In section 6, we discuss the limitations and future work, and finally, in section 7, we provide the conclusion of this paper.

2 Background and Motivation

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

[1] Continuous Integration / Continuous Deployment

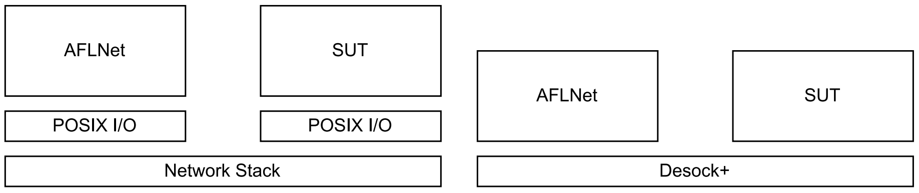
[2] This is just an estimation based on our experience with out-process fuzzing using AFL fuzzer

average, we are 100 times faster than fuzzing network protocol implementations. This observation led us to do more research and find different 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 (see section 4.3).

3 Removing Network Communication Overhead with Desock+

In this section, we discuss our approach to avoid network communication overhead. We provide 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 approach 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 approach, 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+.



(a) AFLNet (b) AFLNet using Desock+

Fig. 1: Removing network communication overhead using Desock+.

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

3.1 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 1. 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. Figure 2 shows AFLNet fuzzing using Desock+ as a simulated socket-library.

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

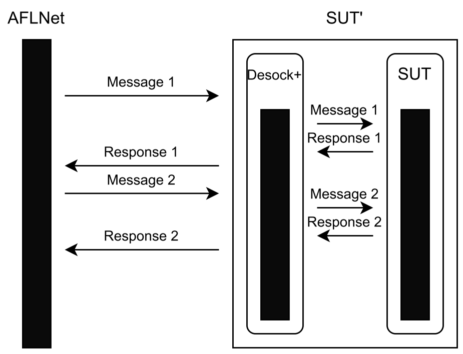


Fig. 2: AFLNet fuzzing using Desock+, which is a simulated socket library.

3.2 Extending Desock+

Although Desock+ supports many SUTs for fuzzing network protocols, there are corner-case SUTs that it does not support, because of system-calls that are not simulated. To address this issue, we must identify which system-calls and their input arguments are causing the errors (EAGAIN, EBADF, etc.) and simulate them correctly. However, manually identifying these error-prone system-calls (for

Table 2: 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%

example, `epoll` or `select`) and their arguments among thousands of system-calls is not possible.

To solve this problem, we have developed an automated system-call filtering module. As shown in Figure 3, when the fuzzer starts fuzzing the SUT, the system-call filtering module begins monitoring the SUT by using the Ptrace to intercept system-calls between the operating system and the SUT. The module then filters the socket-related system-calls and looks for the ones that have returned -1 as an error. Then, it extracts the system-call arguments using `GDB` debugger. The error-prone system-calls and their arguments are then saved as the output of this module. Therefore, the user of Desock+ can simulate these system-calls into Desock+ to support different SUTs.

3.3 Evaluation of the AFLNet Fuzzing Speed using Desock+ 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 2 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+.

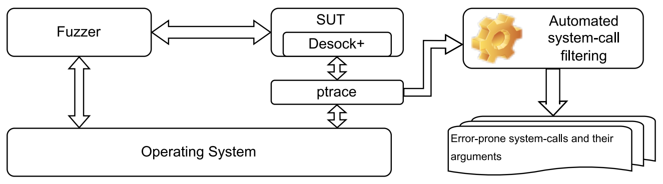

Fuzzer
SUT
Desock+
ptrace
Automated system-call filtering
Operating System
Error-prone system-calls and their arguments

Fig. 3: Extracting the error-prone POSIX system-calls and their arguments.

4 Green-Fuzz Fuzzer

Monitoring the execution of AFLNet reveals that specific system-calls and context-switches between the fuzzer and SUT impose much overhead in the fuzzing process. In this section, we present Green-Fuzz, a new fuzzer to reduce the number of `sendto`, `setsockopt`, `recvfrom` system-calls, and also the context-switches between the fuzzer and SUT in the fuzzing process.

As seen in section 2, 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 approach can be applied to any network protocol fuzzer, assuming the fuzzer can decide on the input trace in advance. In our case, we applied it to AFLNet. In Figure 4, we can see how Green-Fuzz has reduced this overhead compared to AFLNet.

4.1 Design

To apply our approach to AFLNet, we had to make a slight change to it. We call the new fuzzer Green-Fuzz, which sends a trace of the input messages to the SUT in one go. For this purpose, 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.

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. We send the list of responses as a list of tuples 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 5-a shows the AFLNet interaction with the SUT, where the fuzzer sends each input message one by one. Figure 5-b shows the Green-Fuzz interaction with the SUT, which sends a trace of the input messages to the SUT in one go.

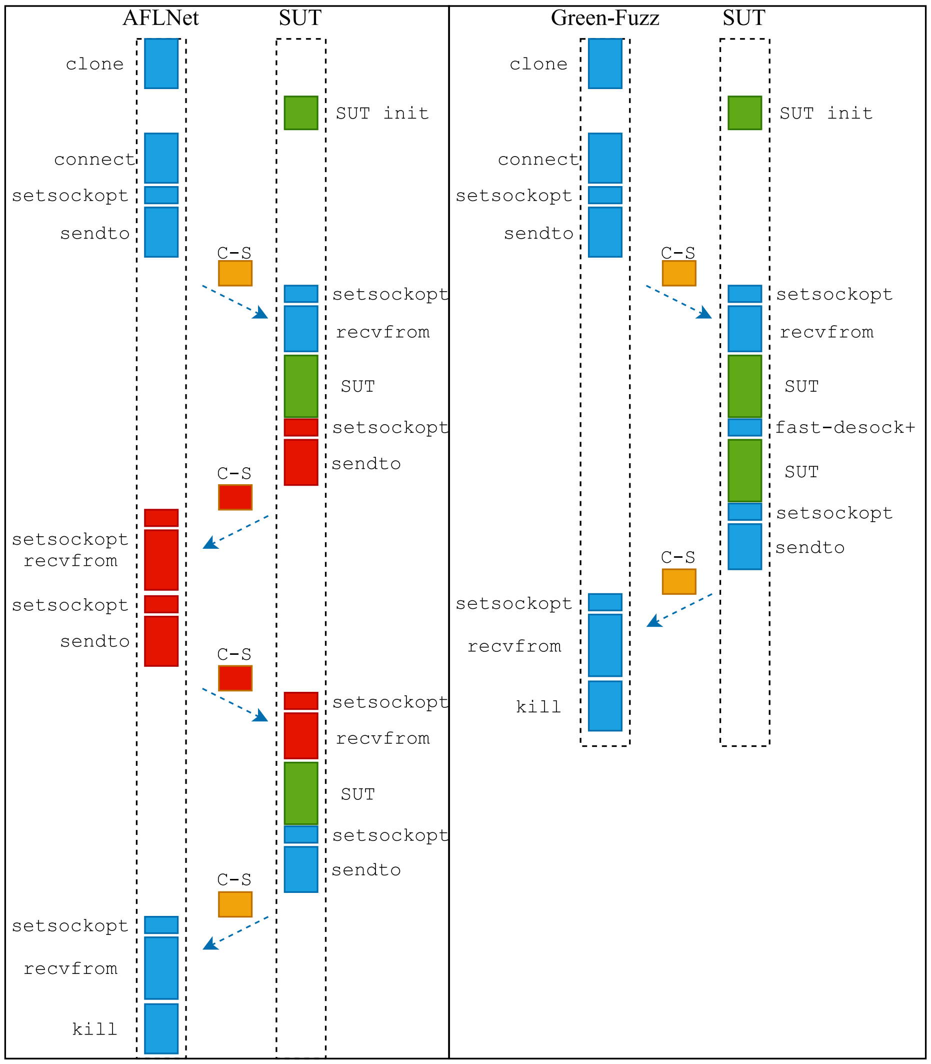
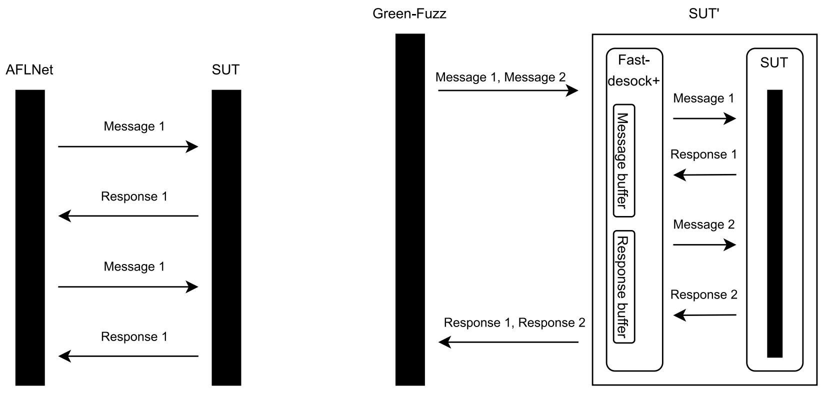


Fig. 4: Comparison of fuzzing overheads in AFLNet (left) and Green-Fuzz (right). Green-Fuzz avoids the overheads colored in red; as shown in Table 3, this can be from 0% to 80%. C-S stands for a context-switch, shown in orange. The time spent by SUT in processing input is shown in green and other system-calls are shown in blue.



(a) AFLNet (b) Green-Fuzz

Fig. 5: 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.

4.2 Evaluation of Green-Fuzz on ProFuzzBench

To show the benefits of Green-Fuzz, in this section, 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 3 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 different system-calls that are a source of the overhead while fuzzing. Table 4 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.

4.3 Comparison with In-Process Fuzzing

Fuzzers are broadly classified into out-process and in-process fuzzers. Like AFLNet [13], out-process fuzzing involves forking, (i.e., duplicating a process by calling `clone` system-call) and killing, (i.e., terminating a process by calling `kill` system-call) the SUT for each input. Although this approach imposes overhead, it does not require patching or modification of the SUT's source code. Moreover, out-process fuzzing can be applied to closed-source programs, which is not feasible using in-process fuzzing.

Table 3: 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 4: 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%

In contrast, for in-process fuzzing [7][1], we manually modify the SUT so that instead of processing a single input, it can process multiple. To do this, we introduce a loop, where after processing one input message, we jump back to the program point where it begins processing an input message. This method avoids frequent forking, initialization, and killing of the SUT, resulting in a much faster fuzzing speed. In-process fuzzing is also known as in-memory fuzzing in some publications [15][8].

In this section, we compare the speed of sending input message traces per second between Green-Fuzz (out-process fuzzer) and in-process fuzzers [7][1]. For this purpose, we make a hypothetical comparison based on our experiment in section 3.2 and the expected overheads that in-process fuzzing can save. First, we discuss the overhead difference between the two types of fuzzers. After that, we present our comparison based on previous experiments (see section 3.2) and our expectations from the in-process fuzzing overheads. Figure 6 shows the fuzzing overhead occurring while doing out-process and in-process fuzzing for a network protocol.

Table 5 shows the comparison between the Green-Fuzz (out-process fuzzer) and an in-process fuzzer. Since in-process fuzzing only fork and kills the SUT once, it does not have overheads regarding `clone` and `kill` system-calls. Furthermore, the input messages are mutated inside the SUT (in-memory) and not sent through simulated sockets, so it does not have overhead regarding

Table 5: comparison of Green-Fuzz and in-process fuzzing based on different overheads for fuzzing n trace of m input messages. The times are in milliseconds (from an example SUT) and shown in the format of $n \times m \times time$.

Source of Overhead	**Green-Fuzz**	**In-process Fuzzer**	**Overhead Difference**
`clone`	$n \times 6.5$	6.5	$\approx -100\%$
`kill`	$n \times 8.7$	8.7	$\approx -100\%$
`recvfrom`	$n \times m \times 1.2$	0	-100%
`sendto`	$n \times m \times 1.3$	0	-100%
`setsockopt`	$n \times m \times 0.2$	0	-100%
`connect`	$n \times 11$	0	-100%
Context-switching	$n \times m \times 0.8$	2×0.8	$\approx -100\%$
SUT initialization	$n \times 5.1$	5.1	$\approx -100\%$

`recvfrom, sendto, setsockopt` and `connect` system-calls. There is also the context-switching overhead between the fuzzer and SUT, which the in-process fuzzer saves. Finally, the overhead of SUT initialization differs between the two types of fuzzing. Because in-process fuzzing patches the start of the fuzzing loop right after the initialization of the SUT and right before the processing of a new input. However, the out-process fuzzing has to go through the SUT initialization each time it forks the SUT to send a new input.

Green-Fuzz, as an out-process fuzzer, does not require changing the program's source code and can also be applied to closed-source SUT. Overall, there is a trade-off between the fuzzing speed and modifying the source code of the SUT. For SUTs in which the time of SUT execution (the green part in Figure 6) is very high, using in-process fuzzing does not save much overhead relatively. Using in-process fuzzing also introduces the risk of missing some parts of the SUT behavior because we have to introduce a loop inside the SUT, where only the code inside that loop would be exercised. Finally, since Green-Fuzz avoids any out-process fuzzing overhead in network protocol fuzzing possible, it is the most efficient out-process fuzzer that we could have for network protocol implementations.

5 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

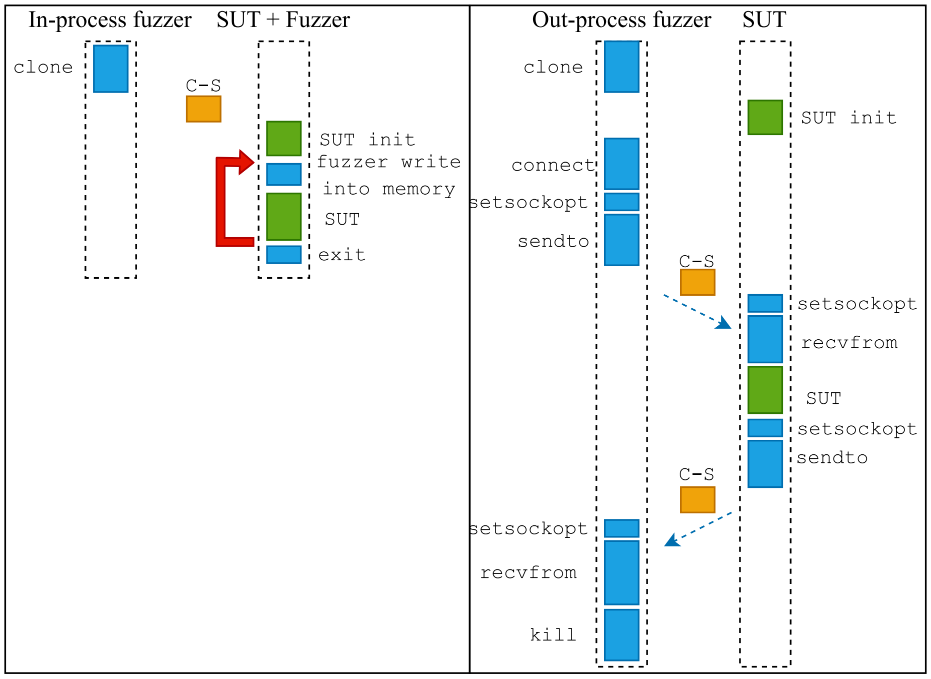


Fig. 6: In-process fuzzer (left) and out-process fuzzer (right) overheads while fuzzing network protocols. The blue colors are system-call overhead, and the green ones are the time spent by the SUT in processing input message. C-S stands for context-switching between the fuzzer and the SUT, shown in orange.

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.

5.1 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.

5.2 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.

6 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.

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 paper, 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.

7 Conclusion

Fuzzing is an effective technique for identifying bugs and security vulnerabilities in software systems. However, it's application to network protocols has been challenging because of low fuzzing throughput.

This work proposes a solution to improve the efficiency of network protocol fuzzing by introducing a simulated socket library, Desock+, that enables efficient fuzzing without modifying the source code of the SUT. The study also presents Green-Fuzz, a novel approach that utilizes a trace-based input message-sending method to increase efficiency further. Green-Fuzz can be easily applied to other fuzzers for stateful systems to gain more performance.

Our evaluation shows that the proposed method outperforms AFLNet by being up to four times faster and can be applied to a broader range of SUTs. Green-Fuzz removes as much overhead as possible without resorting to in-process fuzzing, which requires non-trivial manual changes to the SUT and introduces the risk of missing parts of the SUT behavior (as discussed in section 3.3). Therefore, we can achieve this maximum performance gain with a generic solution that works for any network protocol. While it substantially improves, we may still want to do the extra work to move to in-process fuzzing.

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, 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. 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." (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.