Nyx-Net: Network Fuzzing with Incremental Snapshots

Sergej Schumilo[1], Cornelius Aschermann[1], Andrea Jemmett[2], Ali Abbasi[1], and Thorsten Holz[3]
[1]Ruhr-Universität Bochum, [2]Vrije Universiteit Amsterdam
[3]CISPA Helmholtz Center for Information Security

Abstract

Coverage-guided fuzz testing ("fuzzing") has become mainstream and we have observed lots of progress in this research area recently. However, it is still challenging to efficiently test network services with existing coverage-guided fuzzing methods. In this paper, we introduce the design and implementation of Nyx-Net, a novel snapshot-based fuzzing approach that can successfully fuzz a wide range of targets spanning servers, clients, games, and even Firefox's Inter-Process Communication (IPC) interface. Compared to state-of-the-art methods, Nyx-Net improves test throughput by up to 300x and coverage found by up to 70%. Additionally, Nyx-Net is able to find crashes in two of ProFuzzBench's targets that no other fuzzer found previously. When using Nyx-Net to play the game *Super Mario*, Nyx-Net shows speedups of 10-30x compared to existing work. Moreover, Nyx-Net is able to find previously unknown bugs in servers such as Lighttpd, clients such as MySQL client, and even Firefox's IPC mechanism—demonstrating the strength and versatility of the proposed approach. Lastly, our prototype implementation was awarded a $20.000 bug bounty for enabling fuzzing on previously unfuzzable code in Firefox and solving a long-standing problem at Mozilla.

CCS Concepts: • **Security and privacy → Systems security**; • **Software and its engineering → Software verification and validation**.

Keywords: Testing, Fuzzing, Software Security

ACM Reference Format:
Sergej Schumilo, Cornelius Aschermann, Andrea Jemmett, Ali Abbasi, and Thorsten Holz. 2022. Nyx-Net: Network Fuzzing with Incremental Snapshots. In *Seventeenth European Conference on Computer Systems (EuroSys '22), April 5–8, 2022, RENNES, France.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3492321.3519591

EuroSys '22, April 5–8, 2022, RENNES, France

ACM ISBN 978-1-4503-9162-7/22/04.
https://doi.org/10.1145/3492321.3519591

1 Introduction

In the last years, we have seen a lot of research progress in the field of fuzzing—both in academia [3, 12, 17, 28, 31, 45, 53, 57, 69, 71] and in industry [18, 34, 48, 70]. The majority of improvements made in the last years have focused on improving fuzzing algorithms themselves. However, it is slowly becoming apparent that improvements for fuzzing on the algorithmic level have less impact in practice compared to improvements to the ability to fuzz new targets. Any application that is fuzzed for the first time is likely to result in many security-relevant findings [7]. As a result, we are currently observing a shift towards making it feasible (and even easy) to target new applications or systems.

The two most common approaches to achieve this goal are generating function-style harnesses for persistent mode fuzzers such as libFuzzer [5, 29, 31, 48] or fuzzing based on snapshots taken at the start of the test case [16, 53, 56]. While these approaches address some of the problems of harnessing targets, none of them allows proper harnessing of systems with many messages that are being passed back and forth (e.g., network services). Such services pose a unique set of challenges distinct from "parse a binary blob" fuzzing targets: the target applications are typically much more stateful, more complex (and hence slower), and the message formats are often more complex than individual file formats.

Such applications make up a significant fraction of interesting attack surface. Yet, little has been done to allow effectively fuzzing and harnessing such targets with coverage-guided fuzzers. The notable exception is AFLNet [47], which runs the target and creates new connections for each test input. While this approach has long been used to fuzz network services in blind fuzzers, its drawbacks has made coverage-guided network fuzzing exceedingly difficult. First, using network connections is significantly slower than reading from a file. Second, as the service is running persistently, the fuzzer has no clear point at which the service is ready to receive a test case; such fuzzers require waiting for manually-specified, fixed periods of time during startup, and use similar timeouts when handling each test case. Third, reusing the same process (without fully restarting the server) is also "noisy": for example, background threads in the service can randomly get scheduled independently of the test cases the fuzzer sent. These seemingly random code paths still affect the fuzzer's coverage and introduce pointless inputs into the

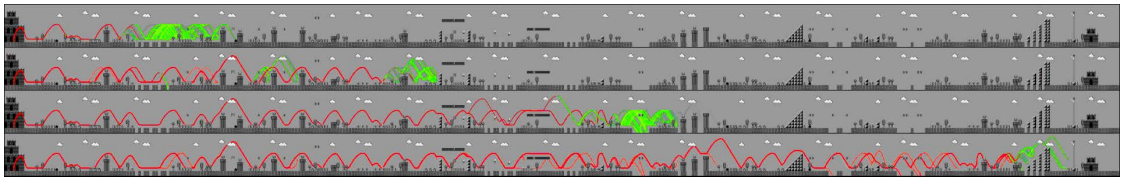

Figure 1. Visualizing Nyx-Net' use of incremental snapshots to solve a hard Super Mario Bros. level: red paths are taken once to create an incremental snapshot. For each snapshot, a number of test cases (green paths) are performed. Note how Nyx-Net also uses "old" positions to create snapshots.

queue. Similarly, great care has to be taken to ensure that the target is not operating based on state introduced by previous test cases. To this end, AFLNet requires a user to write a "cleanup" script that is solely responsible for ensuring that all changes to the file system, databases, etc. are rolled back after each test case. Building such a script based on spurious, non-reproducible inputs can require a significant effort.

In this paper, we present the design and implementation of Nyx-Net, a fuzzing method that is able to test complex, stateful message-passing systems such as network services. Our approach is based on the following design principles: first, we use *hypervisor-based snapshot fuzzing* [53, 54, 56] to ensure noise-free fuzzing and to speed up resetting to a clean state. Second, we propose *selective emulation* of network functionality to avoid the heavy cost of real network traffic. This also allows Nyx-Net to accurately control when packets are consumed to avoid guessing the right timeouts.

Our prototype implementation of Nyx-Net is built on top of Nyx [53], which it enhances with a set of new capabilities. Most importantly, we enable Nyx to target network connections and add support for handling the network stack. We also extend Nyx's snapshot capabilities by introducing incremental, whole-VM snapshots. The snapshot mechanism is agnostic of the OS running in the VM. Using such snapshots increases performance and ensures that all state is properly reset between each test case. Furthermore, using such snapshots allows us to emulate a significant fraction of the network APIs: we run the target until a snapshot is taken without major interference (i.e., close to native speed). When the hooks detect that the target is about to receive the first bytes of fuzzer-supplied input, we take a whole-system snapshot. Following this snapshot, we now *emulate* the network interactions of the target connection to further increase speed. As our evaluation shows, precisely emulating all network interactions is a difficult task. By only emulating the (few) operations on the target connection, we reduce the need to emulate all I/O functionality faithfully. While we tested our prototype implementation only on two different Linux VM setups (*busybox* and *Ubuntu*) as well as targets running inside a Docker container inside of a Ubuntu VM, Nyx-Net's network emulation layer should be compatible with any POSIX-compliant system.

Note that Nyx-Net is not limited to network interface fuzzing. Instead, our approach can fuzz any complex, stateful message-based target such as Inter-Process Communication (IPC) interfaces. For example, Firefox splits safety-critical parts of its codebase into isolated sandbox processes that communicate via various IPC methods such as Unix domain sockets and shared memory. We show that Nyx-Net can efficiently and effectively test these IPC interfaces. In fact, after seeing that Mozilla was looking for a tool like Nyx-Net, we reached out to them and they decided to integrate Nyx-Net into their testing pipeline after finding multiple security issues using. Excerpt from Mozilla's blogpost on the challenges of IPC fuzzing [32] and their statement regarding Nyx-Net are shown in Figure 2.

> *"Hence, we believe that a system testing approach is the only viable solution for [...] IPC testing. One [...] approach [...] could be to [...] perform a snapshot of the parent [..] and then replace [..] child messages [...]"*
> — **Blogpost** - Security Team at Mozilla

Figure 2. Excerpt taken from Mozilla's blogpost [32] on IPC-Fuzzing.

As our evaluation shows, Nyx-Net drastically improves upon the state-of-the-art: compared to AFLNet on their own benchmark ProFuzzBench [40], we are able to improve test throughput by up to 300x and coverage found by up to 70%. Compared to Agamotto [56], the state-of-the-art in snapshot fuzzing kernel modules, Nyx-Net is able to perform both snapshot reload and creation operations almost 10x faster. Additionally, Nyx-Net is able to find crashes in two of ProFuzzBench's targets that no other fuzzer could detect previously. In an evaluation with the game *Super Mario Bros.*, we show that Nyx-Net is able to solve most levels about 10-30x faster than the state-of-the-art (AFL + Ijon [3]) — demonstrating it's ability to improve the performance of message-based targets unrelated to networking (see Figure 1 for a visualization). In fact, when running in parallel, Nyx-Net is able to solve some levels "faster than light": solving the level takes less wall-clock time than playing the level perfectly even once. Nyx-Net is even able to exploit a glitch to solve a level that the authors of Ijon believed to be unsolvable. Lastly, Nyx-Net is not only able to find previously unknown bugs in servers such as Lighttpd, but also network clients such as MySQL client, and even Firefox's IPC.

In summary, we make the following key contributions:

- We introduce Nyx-Net, an efficient fuzzing method that uses hypervisor-based snapshot fuzzing and selective emulation of network functionality to avoid the heavy cost of handling the full network traffic.
- We study the concept of incremental snapshots in fuzzing for complex and slow targets, and show that our approach is able to efficiently test different types of client and server systems.
- In our evaluation, we show that Nyx-Net outperforms state-of-the-art fuzzing tools by more than an order of magnitude in many benchmarks. Furthermore, our prototype implementation found multiple unknown bugs in complex, real-world software.

To foster research, we release Nyx-Net under an open-source license at https://github.com/RUB-SysSec/nyx-net.

2 Technical Background

We now discuss the technical difficulties of fuzzing real-world network services. We specifically focus on the approach used by existing methods to perform network fuzzing. Since Nyx-Net is based on Nyx, we also address the technical aspects of Nyx that are relevant to this work.

2.1 Network Service Fuzzing

Many widely used network services and servers are still written in memory unsafe languages for performance reasons. This puts such software at significant risk: Most of the complexity is part of some very public (often Internet-wide) attack surface. To make the matter worse, getting memory safety right in languages that do not enforce memory safety is notoriously difficult. Over the last years, fuzzing has become one of the primary tools for finding complex memory safety bugs in an automated way. Of course, this includes network service fuzzing. For example, AFLNet [47] removed AFL's focus on single files and enabled it to send packets to target sockets. There have also been various efforts to remap socket-based I/O to file-based I/O [55]. When successful, this approach allows to use pre-existing file-based fuzzers such as AFL++ [18] to fuzz network services.

However, effectively fuzzing network services remains challenging: AFL and most of its derivatives assume that the target is fast and spawns only a single process that runs until the input is consumed. Afterwards, the target is supposed to terminate immediately. Also, two sub-sequential executions should be (mostly) independent of each other. Unfortunately, none of these assumptions hold for most network services. They are often designed with little regard for startup time, persist across connections (often by spawning threads or sub-processes), and maintain a significant state.

AFLNet forces a user to write clean-up scripts that need to reset the environment to avoid contaminating results of later tests. It also employs fixed sleep times to ensure servers are online and in a good state. Overall, this makes it difficult to test new software and drastically reduces the performance. We found that it is not uncommon for AFLNet to only be able to achieve single digit test executions per second.

To make matters worse, network APIs provided by current operating systems (OSs) are notoriously slow. Establishing a connection and reading data from it is far slower than reading from a file. A common workaround is to avoid network interfaces entirely: libpreeny [55] introduced a "de-sock" hook that returns the file descriptor of `stdin` instead of network sockets when a new connection is established (this idea is now also part of AFL++). This massively improves the performances, but fidelity is low: the vast majority of operations possible on sockets are not supported by `stdin`. As such, it will simply not work with most real-world software. Note that libpreeny also contains a more advanced `stdin`-to-socket connection that uses a real socket and a new thread to move data from `stdin` to the network socket used by the target. This makes fuzzing of more complex network targets possible, but also losses the performance gains coming from network emulation.

2.2 Protocol Fuzzing

Fuzzing network services is further complicated by the fact that they are much more interactive than software that processes static file formats. Network services also often incorporate features such as compression, encryption, sequence numbers, and checksums that greatly hinder fuzzing efforts.

Historically, this problem has been addressed by *blind generator-style fuzzers*: the user simply writes a program that connects to the target and sends random, but (almost) valid protocol runs. This makes it easy to fix the aforementioned problems, but requires expert knowledge of the specific protocol and significant efforts.

The main success criterion of AFL was that only a superficial or even no understanding of the format being fuzzed is necessary to use the tool to find bugs. Using AFL requires a set of start inputs (so-called *seeds*). While AFL is often able to work even with empty seeds, it usually is more effective if sensible seeds are provided. This is mostly due to the mutation-based fuzzing of AFL and its coverage feedback. However, to use these advantages, in addition to writing a good generator, the user would also have to write a good mutator—an additional hurdle to jump that reduces usage of fuzzing. Nonetheless, there quite a few fuzzers exist that allow the user to specify detailed formats used for coverage-guided fuzzing. Usually, these fuzzers allow the user to provide a grammar or format specification [2, 46].

Commonly, the inputs all have to be valid for parsing. As a consequence, AFLSmart [46] (which uses Peach's [58] pit file format to specify inputs) has to take great care to handle broken or otherwise unparseable inputs. Hence, AFLSmart only parses the seed inputs as it is computationally infeasible to parse new inputs found during fuzzing.

```
d_bytes = s.data_vec("bytes", s.data_u8("u8"))
n_con = s.node_type("connection", outputs=[e_con])
n_pkt = s.node_type("pkt", borrows=[e_con], d_bytes)
```

Listing 1. A (hypothetical) specification for multi-connection network emulation.

Some fuzzers avoid this problem by simply making it impossible to use seed inputs. For example, NAUTILUS [2] uses context-free grammars, but does not allow to provide seeds. As such, arbitrary grammars including those that are hard to parse can be used. Similarly, NYX and SYZKALLER [59] follow the purely generative approach and allow to specify input formats as sequences of typed function calls (or opcodes), but forgo the option to provide seeds.

Lastly, in a far less principled, but just as effective approach, most fuzzers support registering custom mutators. They usually parse inputs on a best effort basis (e.g., by splitting the input at newlines or matching parenthesis), perform some mutations, and then recreate the input. This unloads all the work to the user, but can be highly effective, particularly for formats where most structural information can be easily inferred (e.g., line based formats). AFLNET follows a similar approach: it uses mutators based on a handful of rudimentary packet boundary parsers for the supported formats.

NYX's Affine Typed Bytecode. As mentioned before, NYX follows the generative approach: the user specifies a set of opcodes that can be chained by NYX. While the authors only used the tool to fuzz hypervisors, the opcode-based approach can potentially be used to fuzz a wide variety of interactive targets. All the user needs to do is to implement a set of different opcodes (with their respective inputs and outputs).

For example, a network specification handling multiple connections at the same time is shown in Listing 1. First, we define a data type (`d_bytes`) that contains the payload of actual packets. Then, we define a new opcode (or "node" in NYX's terminology) that creates a new connection. It takes no inputs and returns a new connection handle (`e_con`). Lastly, we define an opcode that emits/sends a single packet via a given connection. To this end, we create a node that borrows a connection and contains a vector of bytes (the actual content of a single TCP or UDP packet).

Note that the specification that we use for network targets in this paper is even simpler: we usually hook the first connection established via a given port and address. Our agent then delivers packets to each function call that attempts to read data from this connection (e.g. `recv()` or `read()`). Similarly, the agent signals readiness when functions such as `epoll()` or `select()` try to wait for more data on the given connection. All that remains is to fill out the two opcode handlers with actual C code that establishes a connection and sends the packet. The fuzzer auto-generates a bytecode format and a custom VM that executes the bytecode by calling the corresponding handlers, as well as custom post-processing mutators.

The actual fuzzing mutations are implemented by the fuzzer running on the host. NYX-NET reuses the same set of mutators provided by NYX:

- **Generate**: generate a new input sequence from scratch.
- **GenerateTail**: generate a new packet sequence and append it to the end of the current input sequence.
- **Splice**: merge the current input with a random input from the input queue. The input from the input queue is appended to the end of the input sequence.
- **SpliceRandom**: merge the current input with a random input from the input queue and merge both sequences at a random offset in the packet sequence.
- **Repeat**: repeat the current input sequence n times.
- **DataOnly**: perform various AFL-style mutations on one specific packet from the current input sequence.

2.3 Hypervisor-Based Snapshot Fuzzing

As explained before, many network applications maintain state between individual test cases or have expensive startup routines. The former reduces reproducibility, while the latter reduces test throughput. It turns out that both problems can be largely avoided by a clever trick: by obtaining a snapshot of the system's state directly *before* executing the test case, we can reset the system to a deterministic state after each test. The cost of this reset is independent of startup complexity and only determined by the size of the changes to the state of the system caused by executing the test. For example, while starting Firefox requires to load hundreds of megabyte of code into memory and to initialize all kinds of system APIs, handling a handful of IPC packets will typically only dirty a few hundred kilobytes of memory. AFL++ contains a Linux kernel module that is able to reset the memory and some limited kernel state of target ring-3 processes to increase performance. AGAMOTTO [56] and NYX both implement such a mechanism to create a snapshot of a whole VM and to reset back to this snapshot after each test. This allows efficient and deterministic fuzzing of a whole OS and even hypervisors. As NYX-NET is based on NYX, we now give a short introduction on how NYX captures and reapplies VM snapshots.

NYX Agents. NYX runs the fuzz target inside of a VM controlled by a modified version of QEMU, and executed by a modified build of KVM. QEMU sets up the VM state and emulates devices as needed, while KVM uses hardware virtualization extensions provided by modern CPUs to run the guest OS inside of the VM natively. This setup provides high performance virtualization. NYX integrates with both QEMU and KVM to take control of the VM, and to reset the state to a given snapshot. The fuzzer uses an agent component within the VM to control the fuzzing cycle: the agent indicates that the target is ready to receive an input and to

create a snapshot. Then, the agent passes the input to the target and lastly the agent notifies the fuzzer that the test case was finished. To achieve this, the agent uses so-called hypercalls. Hypercalls are like syscalls but for VMs: they leave the VM context and pass the control to the hypervisor. QEMU then reacts to those events and creates or restores a snapshot.

Nyx Snapshots. To take a snapshot, a copy of the physical memory and all device state is created. To revert back to a snapshot, all device state is overwritten by the old state. Similarly, the VM's physical memory is overwritten by the original memory. Since the physical memory is often large (4GB and upwards), its prohibitively expensive to reset the whole memory. To accelerate this process, both Nyx and Agamotto use a variety of optimizations: both fuzzers track which pages in the VM's memory have been altered since the start of the execution. This way they avoid overwriting the whole memory in favor of the (few) modified pages.

Modern CPUs provide hardware acceleration features to support efficient tracking of the set of modified pages, the CPU tracks when a page is dirtied during execution. Once a certain amount of pages have been dirtied (typically up to 512 pages), the CPU exits the VM context and informs the hypervisor of the pages that were affected. The hypervisor, in this case KVM, maintains a bitmap of pages that were written to. Both Nyx and Agamotto use this bitmap to selectively reset the VM's memory. However, Nyx's extension to KVM also maintains a stack of pages that need to be reset. This allows Nyx to avoid searching the bitmap for pages to be reset after the execution. For some reason, KVM uses 1 byte in the bitmap for each page in the physical memory. As a consequence, for a 4GB VM, Nyx's stack of dirty pages saves approximately 1MB of memory bandwidth per test case over KVM's approach. Additionally, Nyx implements a custom reset mechanism for the state of emulated devices that is much faster than QEMU's native device serialization/deserialization routine.

3 Design

In the following, we describe the design of Nyx-Net and the rationale behind each choice we made when designing and building it. First, we present a brief threat model that describes the attack scenarios and surfaces we are concerned with. Then, we present the architecture of Nyx-Net, Figure 3 provides a high-level overview.

3.1 Threat Model

Any network interface is usually a clear security boundary. As such, we mostly target various socket style interfaces (e.g., TCP, UDP, and Unix domain sockets). However, in some context (such as sandboxes), this boundary sometimes also includes shared memory. We assume that the attacker has full control over all data that is being sent to the corresponding

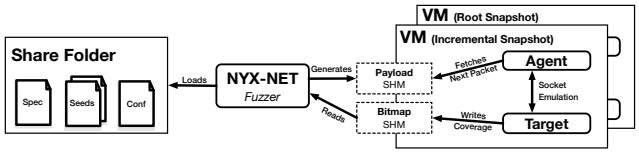

Figure 3. High-level overview of Nyx-Net's architecture. The fuzzer maintains two snapshots of the same VM, in which the agent component hooks the relevant connection of the target.

interfaces. With real networks, this is obvious: an attacker is usually able to send arbitrary data to any TCP/UDP server or client. In the case of Unix domain sockets and shared memory, we assume that the attacker has gained full control of the sandboxed process and attempts to exploit the higher privileged process controlling the sandbox. As such, the attacker is also able to send arbitrary data via Unix domain sockets or shared memory.

3.2 Towards Efficient Network Fuzzing

Our approach uses *hypervisor-based snapshot fuzzing*. Hence, the target is running in a customized VM. At the same time, the fuzzer is running outside of the VM to ensure that the fuzzer has full control over the environment. Additionally, this method allows to take a snapshot of the target. During fuzzing, the snapshot is used to quickly reset the whole virtual machine back to a pristine state after each individual test case. This is already an excellent base to fuzz complex targets that communicate via network interfaces: the snapshot ensures that all state of network connections inside the VM is correctly reset between test cases. Even common, complex patterns such as forking a new process for each incoming connection, writing incoming data to a file system, or even a database in another process, are correctly handled.

However, several distinct challenges remain: first, creating new network connections inside of the VM is still a slow process, usually involving dozens of context switches. This severely limits the fuzzing throughput. Second, many network protocols tend to be fundamentally slow: many messages need to be exchanged to reach "interesting" states. Often, a complex handshake has to be performed before data interacting with the actual application logic is exchanged. Last, the message formats are often complex and a precise understanding of each field or value involved is hard to obtain. As such, the purely generative approach used in Nyx is hard to use for network fuzzing. Writing a specification that is precise enough to model all corner cases is cumbersome and would often take significant manual effort. On the other hand, it is usually easy to obtain some traces of the communication with the target. To address these issues and to make fuzzing networking servers efficient and effective, we introduce the following techniques.

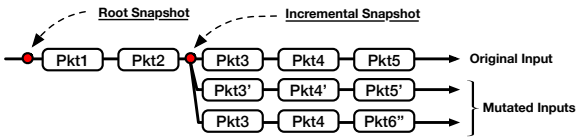

Figure 4. Using incremental snapshots to run mutated tests while skipping the common prefix consisting of packets one to three.

3.3 High Performance Networking

To ensure high performance networking and handle the complexity of shared memory based IPC mechanisms, we emulate significant fractions of the relevant functionality. To this end, we implement a library that adds a variety of hooks into existing `libc` networking functionality. This library is injected into the target (e.g., via `LD_PRELOAD` or by compiling it directly into the target) and intercepts all relevant calls. During startup of the target program, the hooks track various relevant interactions with the OS: we track which file descriptors are part of the external attack surface and various metadata associated with them. When the fuzzing starts, it runs the Nyx bytecode VM to generate data reaching the target process on each hooked function call. Functions such as `read()` or `recv()` on target file descriptors consume data from packets encoded in the test case. More complex APIs such as `epoll()` are emulated to indicate which file descriptor (fd) is ready to receive data (e.g., which fds are receiving packets next according to the bytecode). This library also ensures that packets are consumed correctly across multiple processes. Synchronizing the state of the bytecode stream is relevant if multiple processes are sharing file descriptors/-sockets. For example, forking network servers will usually inherit a recently opened socket from the main process. Similarly, complex IPC protocols often contain the ability to share new file descriptors across existing socket connections.

By emulating network functionality, we gain the following advantages: first, we can precisely identify which data is attacker-controlled and inject our own data at the right places. Stemming from the same features, we can automatically infer the right place to create the initial snapshot. Nyx-Net automatically places the first snapshot after starting the process and directly before the first byte of input data is passed to the target. Second, and maybe even more importantly, we can often run a whole test case without hitting the slow operating system paths handling real network data. We can faithfully emulate network behaviors such as packets being received in discrete chunks. While TCP is a stream based protocol, at a first glance it often looks like TCP reads "the same packets" that were written by a single call to `send()`. While fundamentally broken, a frightening amount of servers assume that a single call to `recv()` will never return data from more than one "packet" (e.g., corresponding `send()` call in the client). The same ability is also needed to properly emulate UDP connections, where packet boundaries are indeed semantic information.

3.4 Fuzzing with Incremental Snapshots

After ensuring network traffic is emulated with high precision and performance, we still face the challenge that in many cases, protocols contain long sequences of messages (and hence complex states). For example, Firefox's IPC traffic consists of hundreds and thousands of packets containing many kilobytes of relevant data. Even with fast network emulations, the number of such test cases that can be executed per second is strictly limited by the time needed to parse and consume these longs sequences. As an example, assume that we have a stream of 120 packets that we want to fuzz. Further, assume that we are currently fuzzing only the last 20 packets. Each test case will execute the same initial 100 packets over and over again. To overcome this hurdle for fuzzing performance, we add the ability to use incremental snapshots: Nyx starts each new execution from the root snapshot that represents a clean state. Nyx-Net adds the ability to quickly create and remove secondary snapshots after executing fragments of the input. This can be used to shave off common prefixes from the execution by taking an incremental snapshot after the common prefix was executed.

For example, Nyx-Net starts from the root snapshot and executes the first 100 packets of a 120 packet sequence. It then take a secondary "incremental" snapshot that represents the VM state after executing the first 100 packets. Now, Nyx-Net runs a handful (tens to hundreds) of test cases that mutate only the last 20 packets. After each test case, we reset the VM state to the incremental snapshot taken after already processing the initial 100 packets. This process is visualized both in Figure 1 and Figure 4. This saves the time required to handle those packets. As soon as Nyx-Net wants to schedule another input, the incremental snapshot is discarded, and Nyx-Net returns to the root snapshot for the next input.

In some cases (especially slow targets) we have seen test throughput increase by more than 10x. We took great care to ensure that the process of creating this secondary snapshot takes very little time. While this obviously introduced some engineering effort, it allows for great simplicity in other aspects of our design: we only ever keep one additional snapshot around. Creating incremental snapshots is so cheap that storing them would waste space and time. By recreating incremental snapshots on demand, we also avoid more complex structures such as trees of incremental snapshots building on top of each other [56]. Each time a new input is scheduled for fuzzing, we randomly decide whether to use incremental snapshots for this input (depending on the inputs performance and number of packets). Then, we pick a random packet in the input and create a snapshot after sending the given packet. Finally, we fuzz the remaining packets for a number of times before discarding the snapshot. In our experiments, we have seen that even for short state sequences reusing the snapshot as little as 50 times yields significant performance increases.

Snapshot Scheduling. A snapshot placement policy determines how the fuzzer selects the point at which it takes an incremental snapshot. To place snapshots, the fuzzer introduces a special "snapshot" opcode that can be injected at arbitrary positions in the input bytecode. The VM then creates an incremental snapshot when this opcode is executed. Nyx-Net utilizes three strategies for snapshot scheduling.

1) **Nyx-Net-none** As a baseline, running Nyx-Net without incremental snapshots is equivalent to a policy that always selects the root snapshot.

Taking snapshots towards the end of the input sequence allows the fuzzer to better exploit its incremental snapshot capabilities; in some cases though, taking snapshots earlier enables the fuzzer to backtrack and possibly explore branching paths in the sequence. To this end, we implemented two snapshot placement policies to explore this trade-off. The parameters were empirically determined via small-scale studies and for sequences smaller than four packets, both policies select the root snapshot.

2) **Nyx-Net-balanced** On inputs with more than four packets, the balanced policy chooses the root snapshot in 4% of the cases. Otherwise it selects a random index in the whole (50%), or only in the second half (50%).
3) **Nyx-Net-aggressive** This policy cycles all available indices for snapshots. The first time an input is scheduled, it creates the snapshot at the end of the input. Each time no new inputs have been found by fuzzing this snapshot for 50 iterations, we place the snapshot one packet earlier. When Nyx-Net-aggressive reaches the smallest index, it starts again from the end of the input.

3.5 Generating Complex Inputs

As mentioned earlier, Nyx allows to express interactive protocols as input languages for fuzz targets by specifying opcodes for each possible interaction with the target. This features makes it easy to adapt Nyx to network fuzzing. Unfortunately, the tool does not support to load existing network traces as seed files. This poses a serious restriction for network fuzzing. For Nyx, the data exchanged with a hypervisor or emulated hardware is usually structurally rather simple (individual pointers, bitfields, etc.). However, the data passed between clients and servers is often deeply nested and precisely modelling all aspects in Nyx's description mechanism is difficult. To ease the burden on the user, we introduce a way to convert specific network connections taken from a network dump into Nyx bytecode inputs. This converter consists of a python library that uses a wireshark dissector for the network dumps. Via metaprogramming, this library creates Python functions for each opcode. Using the resulting python script, the user can parse PCAP files into raw bytecode streams usable by the fuzzer. This allows allows us to create specs that are practically identical to the algorithm used by AFLNet. While these specs are very primitive (no understanding of the data beyond package boundaries), it is straight forward to create them for formats understood by Wireshark without changes to the fuzzer (unlike in AFLNet, where deep modifications to the fuzzing engine are needed). While manually creating precise specs is often helpful for fuzzing, adding the ability to also work with such primitive specs allows Nyx-Net to be used in more use-cases and also allows us to compare against AFLNet in an apple-to-apple scenario where we are using the same amount of information as AFLNet.

4 Implementation Details

To evaluate the performance of incremental snapshots and selective emulation, we implemented a prototype of Nyx-Net. In the following section, we describe the implementation details of our fuzzer. We begin by describing the intricacies of emulating the network APIs used by real-world software, then we present the challenges and solutions of taking incremental snapshots, and lastly, we discuss turning Nyx's format specifications into a format that can be used to load complex existing network dumps as seeds.

4.1 Network Emulation

To speed up network targets for fuzzing purposes and to inject our own fuzzing data, we emulate most network APIs. To be able to intercept calls to network APIs, we use an `LD_PRELOAD` interceptor for common `libc` functions. Obviously, we intercept common networking APIs such as `accept()`, `recv()`, etc. to track network sockets and the data sent to each socket. However, we also emulate related APIs such as the `select/poll/epoll` interfaces to ensure compliant behavior. We also hook many APIs that operate on file descriptors in general, such as `dup()` and `close()` to keep track of aliasing file descriptors that are related to the targeted network connection. For example, the `dup` family of operations is commonly used to pass file descriptors to child processes. Overall, our code hooks a total of 30 `libc` functions and consists of roughly 2,000 lines of C code.

4.2 Creating Incremental Snapshots

Nyx only maintains a single root snapshot. Resetting the whole VM to this snapshot is very cheap: on small targets, Nyx is able to reset the VM about 12,000 times per second—about as fast as forking a similarly complex process once. While it is cheap to reset to the root snapshot, *creating* a root snapshot is expensive because it requires to copy the whole physical memory (often many gigabytes of data).

Nyx-Net extends Nyx's capabilities by introducing a second level snapshot that is much cheaper to create. This snapshot can be used to increase the performance on slow targets by skipping a whole prefix of each test case. Nyx-Net makes taking an incremental snapshot about as cheap as resetting

```
b = Builder(s)
con = b.connection()
b.packet(con, "HTTP/1.1 200 OK")
b.packet(con, "Content-Type: text/html")
```

Listing 2. A manually created seed file for the multi-connection specification from Listing 1.

the snapshot once. As a consequence, we do not have to maintain complex data structures to store a set of snapshots like Agamotto. Instead, we simply recreate the snapshot for the current test case whenever needed.

To obtain such a performance, we make use of similar facilities as resetting the original root snapshot. More specifically, we use Nyx-Net's ability to cheaply report the set of dirtied pages since the root snapshot was taken. The incremental snapshot can use this information to obtain a copy of all relevant memory. We also store another copy of QEMU's device state. To speed up resetting the VM to the incremental snapshot, we maintain a complete second mirror image of the VM's physical memory (see Figure 3). However, to avoid creating an expensive copy of all the physical memory, we simply remap the existing root snapshot to a second location as Copy-On-Write pages. This way, the incremental snapshot itself looks like a complete root snapshot without incurring anywhere near the full memory cost. As a consequence, we do not need to check whether to reset pages from the original root snapshot or the incremental one during the VM reset. To create the incremental snapshot, the pages that were dirtied by the execution since the root snapshot are overwritten with the content of the VM's physical memory. The Copy-On-Write mapping ensures that the original root snapshot remains unchanged.

Before creating another incremental snapshot, these pages are overwritten with the content of the root snapshot. Note that this means we accumulate real copies of pages already present in the root snapshot. In most cases, the executions affect the same memory. In these cases, reusing the existing copies avoids more expensive changes to the page tables. However, in the worst case, this could lead to storing two identical copies of the root snapshot, causing twice the memory usage. To avoid this, we mirror back the physical memory used in the incremental snapshot to a clean copy of the original root memory every 2,000 snapshots created.

To handle write accesses to emulated disks, Nyx-Net introduces a second caching layer to store dirtied sectors representing incremental snapshots. Like Nyx, we use a hashmap lookup to find sectors in the snapshot, otherwise we fall back to Nyx's root snapshot.

4.3 Using Incremental Snapshots

One of the core features of Nyx is to allow giving specifications for interactive targets. Each possible interaction is implemented as a small opcode that takes a set of arguments. When the fuzzer emits the opcode, the "agent" component performs whatever actions are requested. The opcode can also produce another set of values that may be used as arguments for future opcodes. Even though we are not using most of the features available in Nyx (e.g., affine types or even arguments/return values), this model is fundamentally a good fit for network fuzzing: the fuzzer is aware of the time dimension of each interaction. That is, the fuzzer knows about individual packets being sent and most importantly knows that packets that were not sent yet have also not affected the program state at all. This is crucial for incremental snapshots: we introduce a special "snapshot" opcode that the fuzzer injects at arbitrary positions in the input stream. When the agent encounters this packet, it requests a snapshot to be taken by a specific hypercall. Afterwards, the fuzzer continues fuzzing starting from the next packet only.

4.4 Creating Seed Files

Since most hypercalls or MMIO accesses in emulated devices follow reasonably simple patterns, Nyx only supported specifying the interaction fully. No support for loading seed inputs exists in Nyx and the fuzzer needs to find all sequences of interactions on its own. This is not viable for network based fuzzing: protocols tend to be much more complex and it becomes prohibitively expensive to model them down to the last byte in Nyx's specification format. On the other hand, dumping network traffic is easy. As such, loading seed inputs adds tremendous value to fuzzing campaigns. To enable using PCAPs as seed inputs, we extended Nyx's specification engine with a Python library that allows to create inputs directly from Python code. The library consumes a specification and dynamically creates all function for each node. Each function logs the arguments and returns tracking objects that know which function call returned them. Later, calls that use those tracking objects as input can track where the values they use, were created. This way, the script builds a graph of function calls as well as their arguments and return values. Finally, when calling `build()`, the graph is serialized into the flat bytecode that Nyx uses. An example seed file for the specification shown in Listing 1 can be seen in Listing 2.

We use this library in combination with `pyshark` to turn PCAP network dumps into seed files. To fragment TCP streams into logical packets, we use the same logic that AFLNet uses. While this is some protocol-specific code, the dissectors are usually very simple. For example, one of the more common packet boundary dissector uses the `CRLF` newline sequence to split the data stream into logical packets.

4.5 Compile-Time Coverage

Nyx supports only Intel PT to obtain coverage feedback. Yet, if available, compile-time instrumentation as introduced by AFL can be faster and more robust. To ease the use of Nyx-Net on platforms that do not support Intel PT and to improve performance on open-source targets, we enable compatibility

with AFL's compile time instrumentations. The shared memory that contains the coverage bitmap is optionally exposed to the agent. The agent can then redirect AFL's coverage data to the shared memory that is used by QEMU.

5 Evaluation

To evaluate the consequences of our design choices, we compare the prototype implementation of Nyx-Net both against baseline performance and other state-of-the-art network fuzzing tools. As we will see, Nyx-Net outperforms the state-of-the-art in network service fuzzing on almost all targets. On the ProFuzzBench benchmark for network fuzzing, Nyx-Net uncovers more coverage (usually between 10% and 70%). Additionally, it usually reaches the same coverage between 10x to 100x, sometimes even 1000x faster. In fact, on around half of the targets, Nyx-Net finds more coverage in the first five minutes than AFLNet in 24 hours. Additionally, Nyx-Net managed to find bugs in two targets of ProFuzzBench that no other fuzzers is able to uncover. Lastly, we see that even for simple (in the case of AFL++ with libpreeny) to moderately complex (in the case of AFLNet) targets, the approaches used by existing methods begin to fail in practice. In contrast, Nyx-Net is not only able to handle all targets in the ProFuzzBench suite, it even works for significantly more complex targets such as Firefox's IPC.

5.1 Evaluation Setup

All experiments were performed on Intel Xeon Gold 6230 CPUs. Each machine had 52 physical cores and 192GB of memory as well as an SSD. When running experiments in parallel, each one was pinned to it's own physical core. We also disabled hyper-threading do reduce variance in performance. In experiments on ProFuzzBench, we used Nyx-Net's ability to use AFL's compile-time instrumentation. This way, we can use the same target binary across all fuzzers. Coverage experiments where repeated ten times and checked for statistical significance as recommended by Klees et al. [33]. We compared against AFLNet and AFLNwe in the most recent commits supported by ProFuzzBench (0f51f9e and 6ba3a25). Likewise, we used a recent release of AFL++ (2dac4e7).

libpreeny contains two approaches used to turn network servers into targets suitable for fuzzing with AFL++. The simple one only replaces sockets with `stdin` by hooking `accept()`. As this approach is unable to handle most real-world targets, libpreeny also ships a more complex desockifying emulator. We found that it was able to handle more of the ProFuzzBench targets. Hence, we chose to use the second, better performing emulation layer (`desock.c`).

5.2 ProFuzzBench

In the first experiment, we compare Nyx-Net against AFLNet and AFL++ in combination with libpreeny's socket emulation layer. Each individual fuzzing campaign was ran for 24h. We use the public ProFuzzBench test suite. It contains a total of 13 different network services for various types of protocols (from FTP file transfer over VoIP to media streaming). Notably, ProFuzzBench is published and maintained by the authors of AFLNet, and is used to showcase AFLNet's strength in fuzzing stateful network targets. We used the coverage measurement and reporting features that are part of ProFuzzBench. A full set of final coverage results for all fuzzers is presented in Table 2 and coverage over time is also shown in Figure 5. Note that on some targets AFL++ with libpreeny is unable to even start the service. On most other targets, it makes some initial progress, but as coverage is only measured in five minute intervals, most or all coverage was found within the first five minutes and hence it seems that no coverage is found at all. Additionally, AFLNwe significantly under-performs compared to AFLNet. We therefore excluded both from Figure 5. This results demonstrate how libpreeny is far less powerful than our emulation layer. Similarly, AFLNet and AFLNet-no-state perform almost identical, and we excluded AFLNet-no-state from the plots. Overall, Nyx-Net is outperforming AFLNet on all but two targets that show no statistically significant difference. We also investigated each tool's ability to find crashes in the targets contained in ProFuzzBench. AFLNet, AFLNet-no-state and AFLNwe all find crashes in the exact same four targets. Similarly, Nyx-Net was able to crash the same four targets. Additionally, Nyx-Net also was able to crash two additional targets. A full list of the crashes uncovered can be found in Table 1.

Table 1. Crashes found by each fuzzer in ProFuzzBench. We excluded OOM crashes that were only due to the very narrow limits introduced by the docker setup of ProFuzzBench. On dcmtk, Nyx-Net only finds crashes reliably if Asan is enabled (✓). This is due to the fact that in contrast to AFLNet, Nyx-Net does not build up memory corruption state until it crashes. With Asan, the crash is found within the first 10 seconds. Without Asan, Nyx-Net is able to find the bug in some runs, but not others depending on the initial memory layout. On pure-ftpd, AFLNet-no-state managed to trigger an OOM that was due to an internal limit and not the ProFuzzBench limit (*). The targets that AFL++ +libpreeny was unable to run are marked with n/a. We excluded targets where no fuzzer found anything of interest.

	AFL-based			Nyx-Net		
Target	AFLNet	AFLNwe	AFL++	none	balanced	aggressive
dcmtk	✓	✓	n/a	(✓)	(✓)	✓
dnsmasq	✓	✓	✓	✓	✓	✓
exim	-	-	n/a	✓	✓	✓
live555	✓	✓	n/a	✓	✓	✓
proftpd	-	-	n/a	✓	✓	✓
pure-ftpd	*	-	n/a	-	-	-
tinydtls	✓	✓	n/a	✓	✓	✓

5.3 Incremental Snapshots

The targets that are part of ProFuzzBench are configured in a way that AFL and its derivatives such as AFLNet and AFLNwe perform reasonably well. To this end, very short

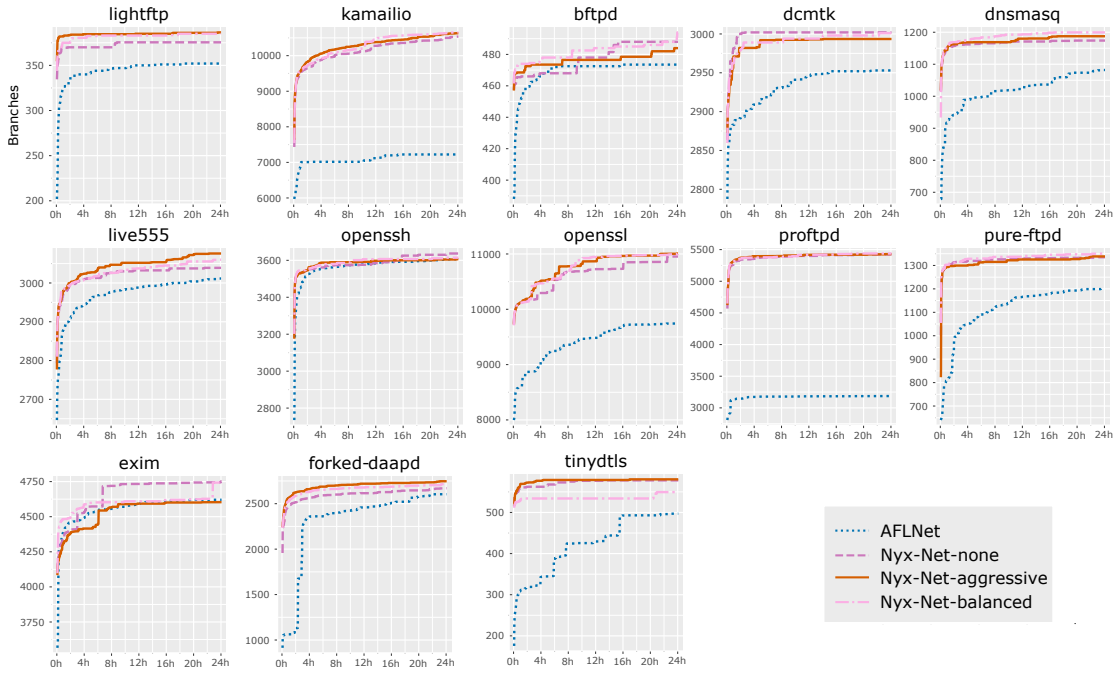

Figure 5. The median branch coverage across 10 experiments on all ProFuzzBench targets. Note that we use the original plotting tools provided by ProFuzzBench. This has two consequences: (i) the first measurement was taken after 10 seconds (sometimes hiding initial progress) and (ii) the y-axis is truncated to only show the coverage found after the seed files and hence does not start at 0.

Table 2. Median branch coverage found by various fuzzers across 10 runs of 24h each, compared to AFLNet. The column for AFLNet displays the number of branches. All other columns show the changes compared to AFLNet. Changes that are statistically significant ($\rho < 0.05$) according to a Mann-Whitney u-test are rendered bold.

	AFL-based				Nyx-Net		
	AFLNet	AFLNet-no-state	AFLNwe	AFL++	Nyx-Net	Nyx-Net-balanced	Nyx-Net-aggressive
bftpd	473.5	+1.3%	+1.6%	n/a	**+3.1%**	**+4.3%**	**+2.2%**
dcmtk	2953.0	**+0.7%**	**+1.3%**	n/a	**+1.7%**	**+1.6%**	**+1.4%**
dnsmasq	1082.0	−0.9%	**−10.5%**	**−37.2%**	**+8.5%**	**+10.9%**	**+9.8%**
exim	4620.5	**+1.5%**	**−18.0%**	n/a	+2.9%	+2.7%	−0.4%
forked-daapd	2604.5	+0.1%	**−23.4%**	**−46.7%**	**+2.5%**	**+4.5%**	**+5.5%**
kamailio	7222.5	−3.1%	**−29.9%**	n/a	**+45.9%**	**+47.5%**	**+47.2%**
lightftp	352.0	+0.4%	**−53.4%**	**−69.3%**	**+6.7%**	**+9.4%**	**+9.8%**
live555	3011.5	**+1.3%**	**+0.9%**	n/a	**+0.9%**	**+1.6%**	**+2.1%**
openssh	3609.0	+0.3%	**−2.0%**	**−49.7%**	+0.8%	+0.2%	−0.1%
openssl	9744.5	−0.2%	**−51.2%**	**−18.8%**	**+12.4%**	**+13.0%**	**+13.0%**
proftpd	3186.5	**−0.9%**	+0.4%	n/a	**+70.2%**	**+70.7%**	**+70.4%**
pure-ftpd	1201.5	+4.8%	+2.2%	n/a	**+11.1%**	**+12.4%**	**+11.4%**
tinydtls	497.0	+3.9%	**−38.8%**	n/a	**+16.3%**	+10.6%	**+16.8%**

seeds with only a handful (e.g., usually less than five) of packets where chosen by us. In such a scenario, high performance emulation and snapshot fuzzing make up most of the impact. While incremental snapshots still increase the throughput, they can not add their full potential. To evaluate the impact of incremental snapshot, we hence picked a more complex target with longer runs. Specifically, we demonstrate how incremental snapshots greatly increase the fuzzing throughput when fuzzing the game *Super Mario Bros.* also used to showcase other fuzzing tools [3].

Super Mario. We recreate the Super Mario experiment presented in Ijon and demonstrate how Nyx-Net's incremental snapshots lead to 10x-30x increases over Ijon in time to solve. On all levels, Ijon was the slowest fuzzer. Nyx-Net-None added a modest 4x average speedup (standard deviation 2.4x, min/max: 1x/9.4x). Nyx-Net-Balanced managed to achieve an 5.8x average speedup (standard deviation: 3x, min/max: 1.6x/12.7x), while Nyx-Net-Aggressive found solutions on average 11x faster (standard deviation: 6.8x,

Table 3. Test throughput of various AFL based fuzzers and Nyx-Net configurations. Each entry shows the average **executions per second ± standard deviation** across our ten 24h runs. Nyx-Net-none is Nyx-Net without incremental snapshots. "Aggressive" and "balanced" denote the two different strategies each. It can be seen that aggressively using incremental snapshots drastically gives the highest test throughput in all cases. However, the biggest gains come from the root snapshot avoiding initialization all together.

	AFL-based				Nyx-Net		
Target	AFLNet	AFLNet-no-state	AFLNwe	AFL++	Nyx-Net-none	Nyx-Net-balanced	Nyx-Net-aggressive
bftpd	4.2 ± 1.9	3.1 ± 1.0	3.5 ± 1.8	-	670.3 ± 42.4	1027.2 ± 53.8	**1250.1** ± 88.7
dcmtk	33.8 ± 2.3	34.1 ± 3.2	38.4 ± 1.0	-	1716.7 ± 127.3	1673.9 ± 246.5	**1782.5** ± 96.3
dnsmasq	3.3 ± 1.6	3.6 ± 1.2	1.6 ± 2.0	4.5 ± 0.0	2732.7 ± 167.1	2583.3 ± 135.8	**2749.1** ± 142.2
exim	4.8 ± 2.2	4.4 ± 2.5	17.8 ± 7.9	69.3 ± 0.2	**312.9** ± 116.7	307.9 ± 92.1	299.9 ± 76.7
forked-daapd	0.4 ± 0.0	0.4 ± 0.0	0.5 ± 0.0	1.2 ± 0.0	13.0 ± 2.5	13.5 ± 1.7	**13.6** ± 2.2
kamailio	4.1 ± 0.4	4.3 ± 0.2	4.9 ± 0.0	-	274.8 ± 19.4	352.1 ± 26.5	**624.6** ± 56.6
lightftp	6.1 ± 1.5	5.6 ± 0.9	13.0 ± 6.6	14.4 ± 0.1	1557.1 ± 352.1	1760.3 ± 306.5	**2040.9** ± 264.0
live555	7.7 ± 2.6	8.8 ± 2.0	25.9 ± 8.8	-	63.1 ± 5.8	84.2 ± 11.5	**105.4** ± 7.7
openssh	23.6 ± 8.3	8.1 ± 3.7	29.3 ± 1.0	126.9 ± 3.6	136.4 ± 3.3	215.4 ± 3.3	**830.2** ± 33.2
openssl	0.3 ± 0.1	0.8 ± 1.6	16.0 ± 0.3	16.8 ± 0.6	454.0 ± 15.0	**467.1** ± 21.6	462.3 ± 17.0
proftpd	2.6 ± 1.2	1.7 ± 0.7	2.9 ± 1.0	-	332.7 ± 37.1	452.9 ± 73.0	**518.1** ± 149.8
pure-ftpd	6.3 ± 3.5	5.8 ± 1.3	5.3 ± 2.2	-	849.8 ± 56.9	1450.8 ± 61.5	**1806.1** ± 120.2
tinydtls	2.2 ± 0.5	2.2 ± 0.2	12.5 ± 0.3	-	1011.2 ± 358.0	942.3 ± 283.5	**1228.0** ± 315.7

min/max: 1.8, 29.8x). In fact, when fuzzing some of the simple levels on 52 cores in parallel, Nyx-Net is able to find a solution faster than a flawless player optimizing for speed (commonly known as "speedrun") is able to play the level even once. This unlikely feature is made possible by a combination of factors: most importantly, as can be seen in Figure 1, Nyx-Net's incremental snapshot allowed the fuzzer to focus only on the difficult part of the current execution by using incremental snapshots right in front of the difficult jump, leading to solve the level 10x – 20x faster than Ijon. Additionally, Ijon's experiment setup is skipping rendering and removes the framerate limit of 60 FPS. Lastly, we parallelize fuzzing to 52 cores. All these speedups together allow us to perform tens of thousands of test cases per second. As a consequence, Nyx-Net is able to solve the first level in less than the 26 seconds wall-clock time needed to speedrun the level at normal framerates.

The original Ijon paper mentioned that Ijon was occasionally able to use wall jumps to escape from pits. However, Nyx-Net actually was able to exploit this ability to much greater results: Nyx-Net is routinely able to solve a level (2-1) by exploiting a wall jump glitch. Ijon was unable to find this glitch and the authors of Ijon believed that level 2-1 might be impossible to solve. Nyx-Net seems to be able to trigger this glitch somewhat regularly (it was found in two out of three of our configurations).

Scalability. It is important to be able to scale to many cores for fuzzing purposes. Naively parallelizing the fuzzer like Agamotto or Nyx will consume prohibitive amounts of memory (e.g., many 100s or even 1,000s of GBs). We share the root snapshots between different instances. As a consequence, in our experiments, 80 instances of Nyx-Net require only about 2x the memory of a single instance.

Snapshot Overhead. To better understand the performance impact of incremental snapshots, we also perform detailed experiments evaluating the performance overhead introduced by our approach. To this end, we used three different policies of Nyx-Net for most experiments: *None* (only a root snapshot is used), *Balanced* (we are rather conservative about snapshots), and *Aggressive* (almost every execution is using snapshots, and we are mostly placing the snapshot close to the end of the input). This allows us to explore the impact of using incremental snapshots. Our experiments on ProFuzzBench (seen in Table 3) show that while snapshots are an additional cost, aggressive snapshot produces the highest execution throughput on all targets. Even the balanced strategy still usually increases throughput. While it also reduces throughput in some cases, the difference is usually smaller than the variance between the different runs. As mentioned before, ProFuzzBench mostly consists of short sequences of inputs. As discussed earlier, when using incremental snapshots on *Super Mario*, which has longer message sequences, more aggressive snapshots significantly improve the time to solve a target.

Agamotto. Lastly, we compare our implementation of incremental snapshots against Agamotto, another recent fuzzer that was developed to speed up syzkaller with incremental snapshots. We used both Agamotto's implementation and ours to create and restore incremental snapshots using the base VM image from our network experiments. We varied the number of dirtied pages and measured the time needed both for creating as well as resetting an incremental snapshot. Note that in contrast to the other experiments, this experiment was performed on a Intel Core i7-6700HQ CPU @ 2.60GHz with 32 GBs of RAM as the larger servers were blocked by more computationally expensive experiments.

All experiments were consecutively performed on a single CPU core (pinned on core 0). For creating snapshots, n pages were dirtied, a temporary snapshot was created, and n pages where dirtied again. Then, the old root snapshot was restored. This experiment was repeated 1,000 times each and the average times were measured. Note that the 500 MB VM was unable to dirty 10^5 pages as not enough memory could be allocated. The results are show in Figure 6. A few notable results can be observed: first of all, Nyx-Net is almost an order of magnitude faster than Agamotto in the relevant range of dirtied pages. As expected, usually creating/restoring snapshots on smaller VMs is slightly faster. Surprisingly, for both Agamotto and Nyx-Net, restoring large numbers of dirty pages on the 512MB VM is slower than on the 4GB VM. This is due to the fact that allocating a significant fraction of the whole available memory is much more work than allocating the same number of pages if there is plenty of memory. Also, for large numbers of dirty pages, Agamotto becomes marginally faster than Nyx-Net. This is due to the fact that when the number of dirty pages approaches the number of pages available, the size of our dirty stack approaches (and eventually even exceeds) the size of the bitmap used. It should be noted that in contrast to Nyx-Net, Agamotto maintains a tree of snapshots. After one gigabyte is used to store snapshots, Agamotto begins to discard old snapshots causing it to slow down. This state is usually quickly reached during fuzzing (as well as in this experiment). We also performed offline experiments where we aborted before Agamotto would start using its LRU policy to evict snapshots. This increases Agamotto's performance, as no cleanups are performed. However, the performance was still behind Nyx-Net's throughput for typical workloads.

Overall, Nyx-Net is significantly faster across the relevant part of the spectrum of snapshot sizes. This observation is related to multiple factors. First, Nyx-Net uses a simpler mechanism: while Agamotto constructs trees of prefixed snapshots, Nyx-Net only uses a single snapshot. Second, Nyx-Net is not iterating the whole bitmap that tracks dirty pages, while Agamotto has to walk the whole bitmap of all pages present in the physical memory of the VM. Last, Nyx-Net also uses faster emulated device resets, reducing the fixed cost of resetting devices.

5.4 Case Study: MySQL Client

After we evaluated Nyx-Net on various server components, we now present a high-level view of using Nyx-Net for testing clients. For this case study, we fuzzed MySQL's client software that is used to connect to and administrate MySQL databases. Running Nyx-Net requires five steps: (i) obtain the target binary, (ii) pick or create a protocol specification, (iii) obtain seed inputs (optionally), (iv) bundles all required data, and (v) finally run the fuzzer.

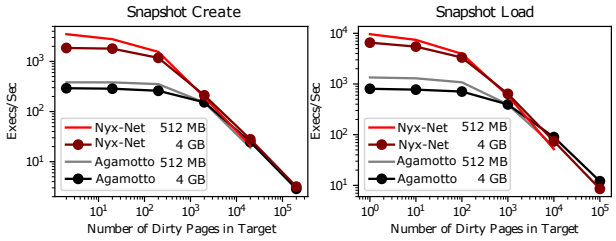

Figure 6. Measuring the throughput of creating/loading incremental snapshots with n dirty pages on VMs with 512MB and 4GB memory respectively.

1. In a first step, we obtain a binary of MySQL client to fuzz by compiling the software with AFL's compiler.
2. Next, we need to chose or create a format specification. As we do not want to spend the time to learn about this protocol, we simply pick the generic default specification that assumes raw packets.
3. To gather seed inputs in step three, we use Wireshark to obtain a set of PCAPs. As the capture was taken locally, TCP packets directly correspond to logical packets in the protocol. Hence, we use the generic script to split the PCAP into individual packets used as seed.
4. The fourth step is to bundle a share folder that contains all relevant data. We use the packer script that copies the target, all of its dependencies, and the seeds into the share folder. It also parses the specification and auto-generates the LD_PRELOAD library that is used as agent component during fuzzing.
5. In the last step, we run the fuzzer by passing the path to the share folder. The fuzzer automatically loads the VM image, which runs a script that downloads the share directory and runs the target.

Performing these steps yields an out-of-bound read on the current version of the client (as shipped by Ubuntu) after a few minutes of fuzzing on 52 cores.

5.5 Case Study: Lighttpd

We also used Nyx-Net on Lighttpd's development branch and found a memory corruption issue where a negative amount of memory could be allocated under specific circumstances. We reported the issue and the bug was fixed before being merged into master.

5.6 Case Study: Firefox

To demonstrate the versatility of Nyx-Net, we also fuzzed the IPC interface used by Firefox to separate high-risk, sandboxed content processes from the main process that contains all critical data. The fuzzing team at Firefox recently specifically asked for this kind of fuzzing in a public blog post [32]. Luckily, Nyx-Net matches their needs very closely. It should be noted that the IPC interface is much more complex than the usual network services. It combines many different kinds of communication patters, from sockets, over shared memory to custom actor implementations used by JavaScript code to

communicate between processes. Nonetheless, Nyx-Net is able to fuzz Firefox IPC with only some changes to the agent component (the LD_PRELOAD library). Specifically, Firefox uses dozens of processes and threads and approximately a hundred sockets—many of which are needed at the same time. We extended the agent to find the relevant sockets and to allow the agent to talk to multiple connections at the same time. While fuzzing Firefox, we found three bugs and the Firefox team found two additional security issues while evaluating and integrating Nyx-Net into their workflow. It should also be noted that we only fuzzed a very small subset of the available packets due to our limited understanding of Firefox's IPC mechanisms. After seeing the impact of Nyx-Net, the fuzzing team at Mozilla is currently planning to integrate Nyx-Net into their testing pipeline. We were awarded a $20.000 bug bounty for enabling fuzzing of the IPC interface of Firefox, mainly because our approach solves a long-standing problem at Mozilla.

5.7 Handling of New Vulnerabilities

We worked closely with a security engineer at Mozilla to understand and mitigate the security impact of the bugs found with Nyx-Net. While our three bugs where only null pointer dereferences (which are still regarded as high severity), the additional two bugs found by Mozilla were exploitable. During the evaluation, we also found one crash in MySQL's client that affects the version of MySQL currently shipped in Ubuntu. Lastly, when fuzzing Lighttpd, we also uncovered an integer underflow in malloc that was fixed before it was shipped. Additionally, Nyx-Net managed to find two new bugs in targets from the ProFuzzBench suite. While no other fuzzer in our evaluation found these bugs, it seems like these bugs have been fixed in the latest release.

6 Related Work

Following the publication of AFL [70], its impact soon caused a wave of additional research. Almost every design choice was investigated: AFL's input mutation algorithm where extended upon [2, 4, 17, 24, 43, 46] as was its ability to trigger and identify bugs [6, 6, 30, 41, 42, 62, 68]. To improve the strength of AFL's semi-random mutations, many researchers proposed to combine fuzzing with more elaborate program analysis techniques such as taint tracking [12, 49] and symbolic or concolic execution [20–23, 28, 38, 45, 57, 60, 69, 72]. Similarly, as the fast coverage guidance is one of the defining features of AFL, it was heavily scrutinized. Additional feedback mechanism where invented and tested for improvements [14, 19, 27, 35, 39, 61]. The last big component of AFL, after mutating and obtaining coverage feedback, is picking which input is fuzzed next. Like the other two components, input scheduling has been investigated thoroughly [3, 8–11, 50, 63]. For a more in-depth overview of recent developments in fuzzing, please refer to Manès et al.'s SoK paper [37].

To make feedback fuzzing more applicable in various scenarios, the harnessing was improved. Fuzzers such as syzkaller or kAFL [26, 44, 54, 56, 59, 64] adapted AFL's fuzzing model to kernel fuzzing. Fuzzers like VDF and Nyx even target hypervisors [25, 52, 53]. Hypervisor-based fuzzing is also commonly used to fuzz firmware [13, 36, 51, 73] Snapshots were also used previously to speed up fuzzing. Besides Nyx, Falk proposed to use hypervisor-based snapshot fuzzing [16]. Similarly, snapshots were used to improve fuzzing of Android apps [15]. Agamotto even uses incremental snapshots to accelerate kernel-level fuzzing. Xu et al. proposed the use of specifically optimized fuzzing primitives to accelerate fuzzers, such as a replacement of the `fork()` syscall with a custom snapshot syscall [65]. CRIU provides in-userland checkpointing [1] which is widely used by OS-level virtualization such as vOpenVZ and docker and could potentially be used to improve fuzzing performance by utilizing its checkpoint capabilities to replace `fork()`. Additionally, snapshots were also used to systematically explore and uncover inconsistencies in distributed systems such as complex storage systems and distributed databases [66, 67].

7 Conclusion

In this paper, we present Nyx-Net, an approach to fuzz complex network services with high performance and fidelity. We believe that snapshot-based network fuzzing makes fuzzing significantly easier to use: the user does not need to ensure that there are no artifacts due to residual states from previous executions. At the same time, our approach can also clearly outperform state-of-the-art approaches based on sending data via real network interfaces—often by orders of magnitude. We also found Nyx-Net very easy to use: using Nyx's support for binary-only fuzzing, we can directly take targets from the Ubuntu repositories and fuzz test them. Nonetheless, our network emulation is still not 100% accurate in more complex scenarios (e.g., when multiple connection are needed at the same time). As such, we still needed to perform some changes to the agent when fuzzing Firefox's IPC. A more complete emulation would make Nyx-Net even easier to use, we leave this engineering challenge as future work.

Acknowledgements. This work was funded by the German Federal Ministry of Education and Research (BMBF grant 16KIS1523 KMU-Fuzz) and the German Research Foundation (DFG) under the German Excellence Strategy – EXC-2092 CaSa – 390781972.

References

[1] CRIU: Checkpoint/Rollback in User Space. https://www.criu.org. Accessed: March 16, 2022.

[2] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. Nautilus: Fishing for Deep Bugs with Grammars. In *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[3] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. Ijon: Exploring deep state spaces via fuzzing. In *IEEE Symposium on Security and Privacy*, 2020.

[4] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[5] Domagoj Babić, Stefan Bucur, Yaohui Chen, Franjo Ivančić, Tim King, Markus Kusano, Caroline Lemieux, László Szekeres, and Wei Wang. Fudge: fuzz driver generation at scale. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 2019.

[6] William Blair, Andrea Mambretti, Sajjad Arshad, Michael Weissbacher, William Robertson, Engin Kirda, and Manuel Egele. HotFuzz: Discovering Algorithmic Denial-of-Service Vulnerabilities Through Guided Micro-Fuzzing. In *Symposium on Network and Distributed System Security (NDSS)*, 2020.

[7] Marcel Böhme and Brandon Falk. Fuzzing: On the exponential cost of vulnerability discovery. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 2020.

[8] Marcel Böhme, Valentin JM Manès, and Sang Kil Cha. Boosting Fuzzer Efficiency: An Information Theoretic Perspective. In *Joint Meeting on Foundations of Software Engineering*, 2020.

[9] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[10] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[11] Sang Kil Cha, Maverick Woo, and David Brumley. Program-adaptive mutational fuzzing. In *IEEE Symposium on Security and Privacy*, 2015.

[12] Peng Chen and Hao Chen. Angora: Efficient Fuzzing by Principled Search. In *IEEE Symposium on Security and Privacy*, 2018.

[13] Abraham A Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, and Mathias Payer. HALucinator: Firmware Re-hosting Through Abstraction Layer Emulation. In *USENIX Security Symposium*, 2020.

[14] S. Dinesh S. Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization. In *IEEE Symposium on Security and Privacy*, 2020.

[15] Zhen Dong, Marcel Böhme, Lucia Cojocaru, and Abhik Roychoudhury. Time-travel Testing of Android Apps. In *International Conference on Software Engineering (ICSE)*, 2020.

[16] Brandon Falk. Chocolate Milk. https://github.com/gamozolabs/chocolate_milk. Accessed: March 16, 2022.

[17] Andrea Fioraldi, Daniele Cono D'Elia, and Emilio Coppa. WEIZZ: Automatic grey-box fuzzing for structured binary formats. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2020.

[18] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++: Combining incremental steps of fuzzing research. In *usenix-woot*, 2020.

[19] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. CollAFL: Path Sensitive Fuzzing. In *IEEE Symposium on Security and Privacy*, 2018.

[20] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. Grammar-based whitebox fuzzing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2008.

[21] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed Automated Random Testing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[22] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated whitebox fuzz testing. In *Symposium on Network and Distributed System Security (NDSS)*, 2008.

[23] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations. In *USENIX Security Symposium*, 2013.

[24] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines. In *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[25] Andrew Henderson, Heng Yin, Guang Jin, Hao Han, and Hongmei Deng. VDF: Targeted Evolutionary Fuzz Testing of Virtual Devices. In *Symposium on Recent Advances in Intrusion Detection (RAID)*, 2017.

[26] Jesse Hertz and Tim Newsham. Project Triforce: Run AFL on Everything! https://www.nccgroup.trust/us/about-us/newsroom-and-events/blog/2016/june/project-triforce-run-afl-on-everything/. Accessed: March 16, 2022.

[27] Chin-Chia Hsu, Che-Yu Wu, Hsu-Chun Hsiao, and Shih-Kun Huang. INSTRIM: Lightweight Instrumentation for Coverage-guided Fuzzing. In *Symposium on Network and Distributed System Security (NDSS), Workshop on Binary Analysis Research*, 2018.

[28] H. Huang, P. Yao, R. Wu, Q. Shi, and C. Zhang. Pangolin: Incremental hybrid fuzzing with polyhedral path abstraction. In *IEEE Symposium on Security and Privacy*, 2020.

[29] Kyriakos Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. Fuzzgen: Automatic fuzzer generation. In *USENIX Security Symposium*, 2020.

[30] Yuseok Jeon, Wookhyun Han, Nathan Burow, and Mathias Payer. FuZZan: Efficient Sanitizer Metadata Design for Fuzzing. In *USENIX Annual Technical Conference*, 2020.

[31] Jinho Jung, Stephen Tong, Hong Hu, Jungwon Lim, Yonghwi Jin, and Taesoo Kim. WINNIE: Fuzzing Windows Applications with Harness Synthesis and Fast Cloning. In *Symposium on Network and Distributed System Security (NDSS)*, 2021.

[32] Christoph Kerschbaumer and Christian Holler. Effectively Fuzzing the IPC Layer in Firefox. https://blog.mozilla.org/attack-and-defense/2021/01/27/effectively-fuzzing-the-ipc-layer-in-firefox/. Accessed: March 16, 2022.

[33] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating Fuzz Testing. In *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[34] Lafintel. Circumventing fuzzing roadblocks with compiler transformations. https://lafintel.wordpress.com/. March 16, 2022.

[35] Li, Yuekang and Chen, Bihuan and Chandramohan, Mahinthan and Lin, Shang-Wei and Liu, Yang and Tiu, Alwen. Steelix: Program-state Based Binary Fuzzing. In *Joint Meeting on Foundations of Software Engineering*, 2017.

[36] Dominik Maier, Lukas Seidel, and Shinjo Park. Basesafe: Baseband sanitized fuzzing through emulation. In *ACM Conference on Security and Privacy in Wireless and Mobile Networks*, 2020.

[37] Valentin Jean Marie Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. In *IEEE Transactions on Software Engineering*, 2019.

[38] David Molnar, Xue Cong Li, and David Wagner. Dynamic Test Generation to Find Integer Bugs in x86 Binary Linux Programs. In *USENIX Security Symposium*, 2009.

[39] Stefan Nagy and Matthew Hicks. Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing. In *IEEE Symposium on Security and Privacy*, 2019.

[40] Roberto Natella 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*, 2021.

[41] Manh-Dung Nguyen, Sébastien Bardin, Richard Bonichon, Roland Groz, and Matthieu Lemerre. Binary-level directed fuzzing for use-after-free vulnerabilities. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2020)*, pages 47–62, 2020.

[42] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. ParmeSan: Sanitizer-guided Greybox Fuzzing. In *USENIX Security Symposium*, 2020.

[43] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. Validity Fuzzing and Parametric Generators for Effective Random Testing. In *International Conference on Software Engineering (ICSE)*, 2019.

[44] Hui Peng and Mathias Payer. USBFuzz: A Framework for Fuzzing USB Drivers by Device Emulation. In *USENIX Security Symposium*, 2020.

[45] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-Fuzz: Fuzzing by Program Transformation. In *IEEE Symposium on Security and Privacy*, 2018.

[46] V. Pham, M. Bohme, A. E. Santosa, A. Caciulescu, and A. Roychoudhury. Smart greybox fuzzing. *IEEE Transactions on Software Engineering*, 47(09), 2021.

[47] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. AFLNET: A Greybox Fuzzer for Network Protocols. In *IEEE International Conference on Software Testing*, 2020.

[48] LLVM Project. libfuzzer. https://llvm.org/docs/LibFuzzer.html. March 16, 2022.

[49] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. VUzzer: Application-aware Evolutionary Fuzzing. In *Symposium on Network and Distributed System Security (NDSS)*, 2017.

[50] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan M Foote, David Warren, Gustavo Grieco, and David Brumley. Optimizing seed selection for fuzzing. In *USENIX Security Symposium*, 2014.

[51] Jan Ruge, Jiska Classen, Francesco Gringoli, and Matthias Hollick. Frankenstein: Advanced wireless fuzzing to exploit new bluetooth escalation targets. In *USENIX Security Symposium*, 2020.

[52] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. HYPER-CUBE: High-Dimensional Hypervisor Fuzzing. In *Symposium on Network and Distributed System Security (NDSS)*, 2020.

[53] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. Nyx: Greybox hypervisor fuzzing using fast snapshots and affine types. In *USENIX Security Symposium*, 2021.

[54] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *USENIX Security Symposium*, 2017.

[55] Yan Shoshitaishvili. Preeny. https://github.com/zardus/preeny. March 16, 2022.

[56] Dokyung Song, Felicitas Hetzelt, Jonghwan Kim, Brent Byunghoon Kang, Jean-Pierre Seifert, and Michael Franz. Agamotto: Accelerating kernel driver fuzzing with lightweight virtual machine checkpoints. In *USENIX Security Symposium*, 2020.

[57] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Symposium on Network and Distributed System Security (NDSS)*, 2016.

[58] Peach Tech. Peach. http://www.peachfuzzer.com/. Accessed: March 16, 2022.

[59] Dmitry Vyukov. syzkaller: Linux syscall fuzzer. https://github.com/google/syzkaller. Accessed: March 16, 2022.

[60] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *IEEE Symposium on Security and Privacy*, 2010.

[61] Yanhao Wang, Xiangkun Jia, Yuwei Liu, Kyle Zeng, Tiffany Bao, Dinghao Wu, and Purui Su. Not All Coverage Measurements Are Equal: Fuzzing by Coverage Accounting for Input Prioritization. In *Symposium on Network and Distributed System Security (NDSS)*, 2020.

[62] Cheng Wen, Haijun Wang, Yuekang Li, Shengchao Qin, Yang Liu, Zhiwu Xu, Hongxu Chen, Xiaofei Xie, Geguang Pu, and Ting Liu. Memlock: Memory usage guided fuzzing. In *International Conference on Software Engineering (ICSE)*, 2020.

[63] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. Scheduling black-box mutational fuzzing. In *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[64] Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo Kim. Krace: Data Race Fuzzing for Kernel File Systems. In *IEEE Symposium on Security and Privacy*, 2020.

[65] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Designing New Operating Primitives to Improve Fuzzing Performance. In *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[66] Maysam Yabandeh, Nikola Knezevic, Dejan Kostic, and Viktor Kuncak. Crystalball: Predicting and preventing inconsistencies in deployed distributed systems. In *The 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI'09)*, 2009.

[67] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. Modist: Transparent model checking of unmodified distributed systems. In *NSDI'09*, pages 213–228, 2009.

[68] Wei You, Xuwei Liu, Shiqing Ma, David Perry, Xiangyu Zhang, and Bin Liang. Slf: fuzzing without valid seed inputs. In *International Conference on Software Engineering (ICSE)*, 2019.

[69] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *USENIX Security Symposium*, 2018.

[70] Michał Zalewski. american fuzzy lop. http://lcamtuf.coredump.cx/afl/. Accessed: March 16, 2022.

[71] Z. Zhang, W. You, G. Tao, Y. Aafer, X. Liu, and X. Zhang. STOCHFUZZ: Sound and Cost-effective Fuzzing of Stripped Binaries by Incremental and Stochastic Rewriting. In *IEEE Symposium on Security and Privacy*, 2021.

[72] Lei Zhao, Yue Duan, Heng Yin, and Jifeng Xuan. Send Hardest Problems My Way: Probabilistic Path Prioritization for Hybrid Fuzzing. In *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[73] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. Firm-afl: High-throughput greybox fuzzing of iot firmware via augmented process emulation. In *USENIX Security Symposium*, 2019.