FitM: Binary-Only Coverage-Guided Fuzzing for Stateful Network Protocols

Dominik Maier* TU Berlin dmaier@sect.tu-berlin.de
Otto Bittner* TU Berlin cobittner@posteo.net
Julian Beier TU Berlin j.beier@tu-berlin.de
Marc Munier TU Berlin m.munier@campus.tu-berlin.de

***Abstract*—Common network protocol fuzzers use complex grammars for fuzzing clients and servers with a (semi-)correct input for the server. In contrast, feedback-guided fuzzers learn their way through the target and discover valid input on their own. However, their random mutations frequently destroy all stateful progress when they clobber necessary early communication packets. Deeper into the communication, it gets increasingly unlikely for a coverage-guided fuzzer like AFL++ to explore later stages in client-server communications. Even combinations of both approaches require considerable manual effort for seed and grammar generation, even though sound input sources for servers already exist: their respective clients. In this paper, we present FitM, the *Fuzzer in the Middle*, a coverage-guided fuzzer for complex client-server interactions. To overcome issues of the State-of-the-Art, FitM emulates the network layer between client and host, fuzzing both server and client at the same time. Once FitM reaches a new step in a protocol, it uses *CRIU's* userspace snapshots to checkpoint client and server to continue fuzzing this step in the protocol directly. The combination of domain knowledge gathered from the proper peer, with coverage-guided snapshot fuzzing, allows FitM to explore the target extensively. At the same time, FitM reruns earlier snapshots in a probabilistic manner, effectively fuzzing the state space. We show that FitM can reach greater depth than previous tools by comparing found basic blocks, the number of client-server interactions, and execution speed. Based on AFL++'s *qemuafl*, FitM is an effective and low-effort binary-only fuzzer for network protocols, that uncovered overflows in the *GNU Inetutils* FTP client with minimum effort.**

***Index Terms*—snapshot, stateful, protocol, fuzzing**

I. Introduction

Fuzzing is one of the main methods to find memory corruption bugs. Even though fuzzing has been around for years, and despite many clients and servers being written in memory unsafe low-level languages, too few clients and servers are fuzzed in-depth. Fuzzing network-facing services still requires a large amount of manual effort. Instead of fuzzing client and server, as a whole, the security researcher has to hook up a fuzzer to fuzz individual parser functions manually. Clients and servers for elaborate protocols are usu-ally stateful and complex, rendering simple coverage-guided fuzzing ineffective. The typical approach to fuzz complex network services is to write a grammar by hand and pass input over slow sockets. We show that we can get around writing grammars and manual harnesses, as we already have the domain knowledge as part of the client-server communication. With FitM, the *Fuzzer in the Middle*, we present a qemuafl-based [1] coverage-guided fuzzer that fuzzes client and server at the same time. FitM-qemu uses network layer emulation, able to handle synchronous and asynchronous network interactions. The networking syscall hooks pass input through shared maps instead of sockets, increasing the speed many-fold. For each potential new state, meaning a new *recv→send→recv* transition, we create persistent snapshots using CRIU [2]. Persistent snapshots allow us to stop and continue fuzzing of target states at any time and even replay snapshots with *strace*, or other debugging features enabled. Additionally, we implement waypoints and smart state creation — based on new socket outputs and unique AFL maps — and fuzz each state transition independently. FitM is the first tool to use persistent snapshots of userspace processes. In comparison to hypervisor-based solutions [3]–[5] userspace fuzzing works without the need for additional introspection into the host system, is simpler, and can potentially reach faster speeds, without the need for expensive hypervisor interactions [6].

More important than the snapshotting itself, however, is our answer to: when to snapshot, how to schedule, and how to automatically explore stateful protocols. In this paper, we explain how we tackle these challenges. Our contributions are as follows:

- We design, develop and open-source FitM, a novel coverage-guided *Fuzzer in the Middle* for binary-only targets.
- We propose a scheme of state snapshots and stateful fuzzing.
- We extend qemuafl through a fuzzer-aware network emulation layer, taking away the need for kernel interactions. It emulates both synchronous and asynchronous sockets.
- We evaluate FitM against AFLNet, a State-of-the-Art coverage-guided fuzzer for network protocols, and show promising results.
- We find a previously unknown buffer-overflow in GNU Inetutils' FTP client by fuzzing the FTP Server and running it on other networking servers and clients.

Both Authors contributed equally to this work

Workshop on Binary Analysis Research (BAR) 2022
24 April 2022, San Diego, CA, USA
ISBN 1-891562-76-2
https://dx.doi.org/10.14722/bar.2022.23008
www.ndss-symposium.org

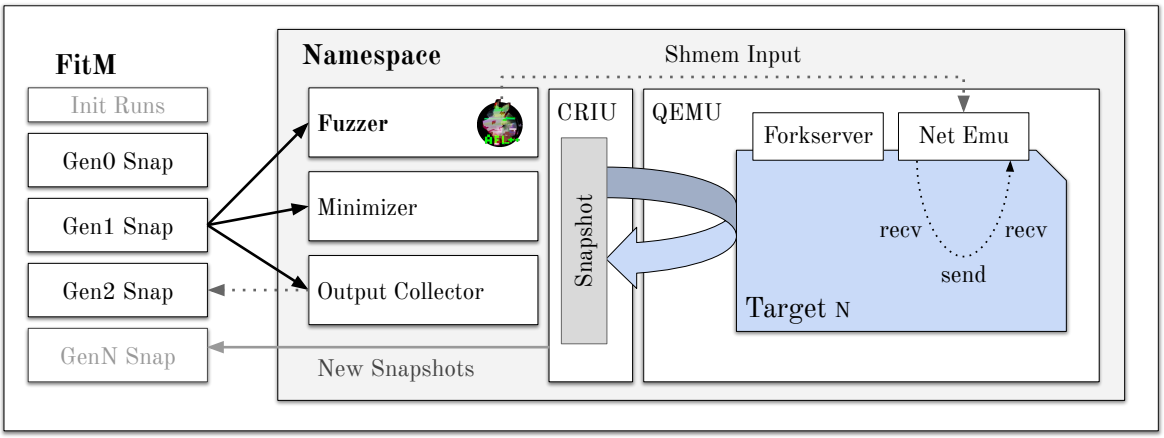

Fig. 1: High-level overview of FitM's structure. FitM takes one CRIU snapshot for each new interaction. The snapshot can spawn inside an individual namespace, and be fuzzed there. Afterwards, the minimizer and output collectors prepare the next generation snapshots.

II. Background

Fuzzing, or fuzz testing, is a form of automated software testing. A fuzzer generates a large number of inputs it uses to run the target binary with. Its goal is to find inputs that crash the target, or otherwise trigger unwanted behavior.

A. Pitfalls of Coverage-Guided Fuzzing

Coverage-guided fuzzing is a common form of fuzzing that tracks the effect of an input on the target. Coverage-guided fuzzers like AFL [7] have had a large impact on the security landscape. Even though previous work allows them to fuzz network services directly, they rarely reach a state spanning multiple exchanged packets. Given a handshake, even if the fuzzer reaches a state where the receiver accepts the fuzzer's input and continues to parse the second packet, the fuzzer is still just as likely to mutate bytes in the initial package as in the second. Each mutation in the first packet *a)* is no longer interesting for the fuzzer, and *b)* has the potential to ultimately destroy the progress to packet two if it corrupts the first packet. Even State-of-the-Art solutions, such as *AFLSmart* [8], *Nautilus* [9], *Superion* [10] and *AFLNet* [11], still need user-specified grammar and inputs. Few automated solutions with network support exist that try to extract flows from network pcap [12] or even learn the grammar [13]. Mapped to Super Mario, Aschermann et al. improve coverage guided fuzzer performance by giving the fuzzer additional feedback on waypoints reached [14].

In the recent past, we have seen more and more fuzzing research proposing the use of snapshots for fuzzing, such as the oeuvre of Falk [5], [15], as well as Nyx by Schumilo et al. [3], and Agamotto by Song et al. [4]. Snapshots can be used to keep states of a program, which will be helpful for network protocol fuzzing. Most proposed snapshot engines work on a hypervisor level. Since these engines require expensive hypercalls and handle a full operating system in addition to the target, userspace snapshotting can reach far greater speeds [6].

B. QEMU and qemuafl

To make FitM work for binary-only fuzzing, we built on top of *qemuafl*. Using qemu-user, qemuafl lifts and recompiles each basic block it encounters during execution. As this happens only once, as long as it is in the cache, it is fast enough for fuzzing. When the basic block is emitted, qemuafl adds AFL++ compatible coverage instrumentation that the fuzzer uses as feedback for future mutations [1]. While it works for binaries of the same architecture, the recompilation even allows us to rehost a binary to a different architecture [16]–[18]. As we have full control of the target, we can replace any syscall, add unsupported calls, hook them, and alter behavior. In comparison to hooking at the library layer through a preloaded shared object, like in preeny [19] or networkemu [20], hooking at the syscall level also works for non-well-behaved targets, as long as they do not specifically fingerprint their environment.

III. Challenges

While working on FitM we faced several challenges that we tackled using different approaches. In this section we outline these challenges and explain how we addressed them in the subsequent section IV.

A. State Explosion

One challenge with fuzzing targets FitM is intended for is state explosion. Each output of one generation gen_i is input to one new snapshot in generation gen_{i+1}. As each snapshot in gen_{i+1} on average produces more than one output, the number of new states produced per generation is proportional to the number of states within that same generation. This exponential growth is a problem both from a practical point of view, as memory resources on any host are usually limited, but also from a logical point of view. Because the number of new snapshots grows faster than they can be processed, i.e., fuzzed, FitM would never arrive in the later stages of the communication. However, these later stages are critical to uncovering deep bugs within the protocol implementations.

In order to deal with state explosion we take multiple measures: *(1)* minimizing inputs from prior generations and the current corpus with `afl-cmin`, *(2)* picking random snapshots for the current loop and *(3)* deduplicating outputs using string similarity.

B. Dead Ends

There may be situations where client or server end up in a state where they constantly repeat a setup, teardown or error message. Typically the fuzzer is not able to recover from such a loop. Trace comparison for each new snapshot would catch cases where the new state behaves precisely the same as the old one. However, in most cases some data processing dependent on input length, randomness, or similar happens within the target, thus resulting in a different coverage map and consequently in a new state. To address this challenge we probabilistically abort some generations.

C. Desynchronisation Between Generations

Each generation should produce outputs that transition to its next state when used as input for the following generation. For example, in a basic FTP communication, the server will begin with a banner message, signaling successful connection establishment, the client will continue with sending the user for login, for which the server then requests the password, and so on. If the server does not receive the user name, for whatever reason, it will not transition to a state where it asks for a password. Consequently, the client will never receive the password request it would expect and instead may receive the welcome banner again. Client and server are now out of sync. This becomes problematic because the client is now missing a generation. While the server transition from generation 1 to generation 3 by receiving the username, the client is stuck in generation 2. Generation 3 does not have any relevant inputs at this point. To reduce the effect of desynchronization we use the output of multiple generations to feed a new generation.

IV. Design

We now present how we addressed the aforementioned challenges.

The general goal of FitM is to build a fuzzer that can autonomously discover paths deep within the state machine of a given network protocol and thus the client/server that implements the protocol. The two main tools to achieve this are: (a) fuzzing two binaries in turns and (b) snapshotting.

FitM is written in Rust and works natively on unpatched Linux systems. We use persistent snapshots through CRIU and a normal AFL-style forkserver for fuzzing. In future versions of FitM, we aim to replace forks by qemuafl's persistent mode, increasing the speed further. FitM automates harnessing by proxying all emulated network interactions like a Machine-in-the-Middle (MitM). FitM offers a comprehensive network emulation layer, directly intercepting client and server, and places socket inputs into the target via shared maps, without the need for slow kernel interaction. This produces very stable and reproducible results with persistent snapshots. While FitM is intended to be used in a client-server manner, using a custom client, or even an echo client, also makes it an effective means to fuzz a single peer. In the following, we drill down on each component depicted in Fig. 1. Most servers only respond to inputs abiding by strict rules and reject the inputs otherwise. To reach depth, correct input is necessary during each stage of the communication. Therefore we collect the response of the client and the server during each stage. They are known to abide by the rules. For every request to the server, we save the response, and for each response, we save the following request, see Fig. 2. The blue lines indicate test case flow; black lines are the snapshots. Dotted elements and lines address other generations. Whether the client or server begins the communication can be configured depending on the selected target. Every collected output is then an input for the upcoming stage and seeds the fuzzing of that stage.

Once the fuzzer finds multiple inputs for subsequent states, simply rerunning the inputs in order may lead to different behavior. Even if the exchanged messages are identical, randomness in the target hinders reproducibility and further progression. So, instead of rerunning, FitM creates a new snapshot whenever the target expects new input, i.e., at the *recv* syscall.

A. Corpus Minimization

afl-cmin is a known tool included in AFL++ that allows the user to minimize a given corpus by collecting the coverage map for a process on each input and only retaining the subset of inputs that still produce the initial coverage. Generally, each input for a state s_i from gen_i corresponds with one output of s_i (there may be cases where an input does not produce output). As long as this relation exists, less input means less output, which leads to fewer new states. Therefore it is desirable to have the minimum number of inputs before producing the outputs of the current generation since that reduces the number of new snapshots while keeping all discovered paths in the current snapshot.

B. Random Select

Even though afl-cmin helps in reducing the number of states, they still grow exponentially, since afl-cmin can not eliminate the fact that each state has more than one child state on average. Only a random, constant size subset of states is used for each generation during processing to further reduce the growth's impact. A constant size is chosen so that every generation takes the same time to complete, and no generations are favored over others. This also keeps AFL's original spirit of using simple and fast but possibly incorrect solutions over complex, but hopefully correct, solutions. More refined state selection algorithms are an ongoing research topic, as recently discussed by Liu et al. [21], and are out of scope for this paper.

C. String Distance Filtering

Another phenomenon that produces a lot of duplicate states are duplicate outputs. While the target may take different paths during execution, it might end up with the same error

messages for many inputs. A first idea could be to eliminate all exact duplicates in the output before starting snapshot creation. However, this will fail due to ever-changing components like timestamps or sequence numbers. In order to still identify these "almost"-duplicates, we define a minimal string distance between outputs that always have to be met. If any output is too close to any other output, it will be excluded from the *out_corpus*. The used string distance is the Jaro-Winkler distance [22].

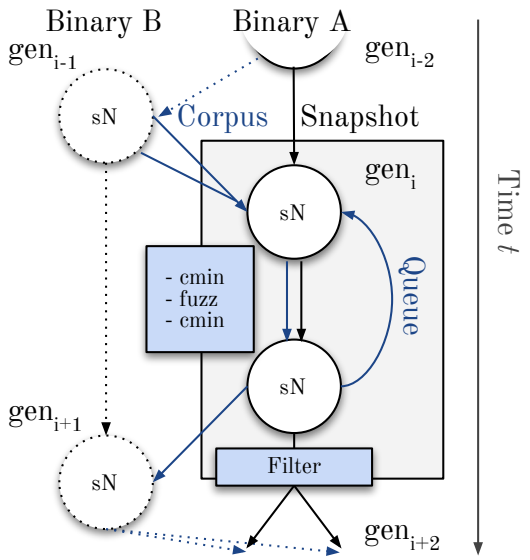

Fig. 2: Information flow in one generation of FitM. The previous outputs pose as inputs to this fuzzing run, the current outputs, filtered, feed into the next generation.

D. Escaping Error Loops

We introduce a random restart into the `run` loop to escape the potential error loops described in subsection III-B. During the `run` loop, two conditions will make FitM restart fuzzing from generation one. The first one is if no snapshots for upcoming generations exist, i.e., all test cases reached an exit before reading further bytes. On top of that, we added a second probabilistic restart, with a low probability of 2 in 100 generations. A constant factor decides whether it is time to restart or not. Just like with the random selects to battle state explosion, we follow AFL's simplicity doctrine here.

E. Resynchronization

Subsection III-C described the problem of generations going out of sync. To circumvent this situation, we decided to take the output of multiple generations as input for generation i. By default the output of generation $i-1$ would become the input for generation i. To resync generations we also take the outputs of generation $i-3$ and $i+1$, if present. Since afl-cmin is called on the inputs, this does not introduce new duplicate states as afl-cmin will eliminate all inputs that do not create new coverage.

V. Implementation

In the following, we discuss implementation related details of FitM.

A. Algorithm

To give a more detailed explanation of how FitM works, we outline the two main algorithms governing FitM's decision-making in this section. Algorithm 1 outlines how generations in their entirety are processed, while algorithm 2 highlights one single generation.

1) Main Loop: The first conceptual step is to run an `init_run` function on both client and server. This function will produce the initial states for each target, state S_1 and S_2. Additionally, it will record the output for the target that sends first in the targeted protocol, corpus C_0. There are three situations within which the loop will not process the current generation:

1) FitM skips a generation at random occasions, with a 7% chance for every generation, to emphasize deeper states in a probabilistic manner.
2) If the current generation is empty, a restart from the first generation is triggered. Running from beginning may find and explore new code paths from an earlier generation.
3) Thirdly, a run over all generations is aborted if another random threshold of 2% for every run is surpassed.

The thresholds are best-effort after non-exhaustive trials, and we see room for improvement here. These parameters shift the focus of fuzzing between generations. The optimal choice of these parameters is nontrivial [21] and may vary, depending on the choice of target and the desired type of fuzzing. Increasing the choice of the generation skip probability shifts the focus onto longer connections, which can help better leverage the stateful fuzzing approach, yet sacrifices some exhaustiveness for the initial states. The choice of the abort threshold has the opposite effect, shifting the focus onto earlier states, with the added benefit of possibly dropping dead states, i.e., states from which no meaningful new state transitions can be discovered. If none of these three checks succeed `process` is called, triggering algorithm 2. For the following step, the subsequent corpus C_{gen+1} is returned. The snapshots S_{gen+2} then get created for the loop step after the next one.

2) Process Stage: The second algorithm used in FitM, the `process` function, goes through the current generation and processes a subset of the included states. This subset is of constant size and randomly chosen. Each snapshot in this subset S'_{gen} then needs to be fuzzed, and its outputs need to be collected. Before fuzzing the input corpus C_{gen} (the outputs of the previous generation) are minimized using afl-cmin. By fuzzing this minimized corpus, the fuzzer produces a new queue which in turn will be minimized by afl-cmin to become C'_{gen}. The second afl-cmin happens to reduce the number of potential outputs C_{gen+1} that will be created by supplying the minimized corpus C'_{gen} to `create_outputs`. This constitutes the first return value of `process`. To generate

Algorithm 1 Main Loop

```
1: function FITM_LOOP
2:     gen = 1
3:     S_1 ← init_run(peer_1)
4:     C_0, S_2 ← init_run(peer_2)
5:     while True do
6:         if random_genskip() then
7:             gen ← gen + 1
8:             continue
9:         if snapshots_gen = ∅ then
10:            gen ← 1                          ▷ Restart
11:            continue
12:        if random_abort() then
13:            gen ← 1                          ▷ Restart
14:            continue
15:        C_gen+1, S_gen+2 ← process(S_gen)
16:        gen ← gen + 1
```

new snapshots S_{gen+2}, each input from the minimized corpus C'_{gen} is taken and checked to actually produce new traces and outputs (lines 9 and 11 in Algorithm 2). If that is the case, it will be used to advance snapshot `snap` to the next generation.

Algorithm 2 Process Stage

```
1: function PROCESS(C_gen, S_gen)
2:     C_gen+1, S_gen+2 ← [ ]
3:     for snap in S'_gen ⊂ S_gen do
4:         C_gen ← snap.cmin(C_gen)
5:         C_gen' ← snap.fuzz(Cgen)
6:         C_gen' ← snap.cmin(C_gen')
7:         C_gen+1 ← snap.create_outputs(C_gen')
8:         for inp in C_gen' do
9:             if snap.trace(inp).known? then
10:                continue
11:            if snap.get_output(inp).known? then
12:                continue
13:            S_gen+2.add(snap.next(inp))
14:
15:    return (C_gen+1, S_gen+2)
```

B. QEMU Syscalls

By default, the overhead of reading data through the operating system's network stack would be detrimental to fuzzing performance. In order to overcome this challenge, we decided to patch QEMU's syscall translation layer. All hooks for network emulation were created, step by step, by re-running real-world targets in FitM-QEMU with `strace` enabled. We then modify the syscall layer that handles the respective network-related syscall. We introduced `FITM_FD`, a special pseudo-file descriptor, over which all `AF_INET` (IP protocol) socket operations are handled, or even multiplexed in case the target, like FTP, uses multiple sockets to communicate. The patches applied to the respective syscalls are explained in Table I.

Syscall	FitM-QEMU HOOK
`accept(4)`	*Always* return static `FITM_FD`
`bind`	Succeed when called on `FITM_FD`.
`clone/fork`	We support the flavor of `clone` as used by `pthread_create` — it behaves normally until the first successful `FITM_FD` `accept`, `write`, or `send`, afterwards either always runs the child function, or returns 0 to continue in the parent (configurable)
`connect`	Succeed when called on `FITM_FD`
`dup(2)`	Return `FITM_FD` for `FITM_FD`, else forward
	After `FITM_FD` is registered via `EPOLL_ADD`, always return `EPOLLIN`/`OUT` Event on `EPOLL_WAIT`
`fnctl`	If `FITM_FD`, return 0 for `SETFL`, return `O_SYNC | O_RDWR` for `GETFL`
`epoll` `(p)poll` `(p)select`	Asynchronous primitives for monitoring multiple FDs. Always return as if `FITM_FD` had activity (`FITM_FD` has further Input/Outputs were drained)
`exit`	Redirect our reserved exit code 42 to 43, optionally immediately `_exit` for misbehaved targets
`read` `recv` `recvfrom` `recvmsg`	If `FITM_FD`, either return input from client or, if `is_sent`, special snapshot handling. See paragraph V-B2
`send(to)` `sendmsg` `write`	If `FITM_FD`, write output if output run, or nothing.
`socket`	If `AF_INET`, return static `FITM_FD` `999`,

TABLE I: Essential syscall hooks placed in FitM's *aflqemu* fork to make binary-only network peers fuzzable.

If a patch in Table I has specific conditions to be triggered, the syscall will adhere to QEMU 's original translation if the condition is not met, for example, all socket operations for Unix domain sockets still work, to allow the target to start up orderly. Some of the patches are immediately necessary to handle network and socket operations: `socket`, `bind`, `connect`, `accept`, `getpeername`, `getsockname`, `recv`, `send`. Patches to `read` and `write` are necessary as some targets might choose to use `read` or `write` instead of `recv` and `send`, refer to Sect V-B2. `select`, `poll` and other asynchronous primitives need to be modified to inform the target of state changes of the fuzzed FD. Skipping this modification leads, in most cases, to the target waiting indefinitely for events to arrive. More importantly, `select` also makes it necessary to chose `FITM_FD` to be another pointer than `1337`, since `select` is not able to wait on FDs higher than `FD_SETSIZE`, which is `1024`. Patching `fork` and `clone` is necessary so only the child is fuzzed, see Sect. V-B3. A change to `exit` allows us to communicate the correct creation of a snapshot from the code calling the CRIU server to FitM via exit code 42, and make sure the target can never trigger this exit code. We furthermore hook `fcntl`, `ioctl` and other syscalls retrieving or modifying the internal state of the socket. These hooks are necessary, as the target might want to change the socket's operation mode, e.g., set the

socket to non-blocking mode, or receive information about the `FITM_FD` pseudo-FD. A call like this on `FITM_FD` will break as the FD does not point to a valid socket. Likewise, `fstat` returns information about the faux socket, if needed. The `dup` hook is necessary because the target is not allowed to change the pseudo-FD it uses to read and write from and to the socket. In other words: once the target receives a `FITM_FD` from QEMU it should never be able to move it to another FD again. We patch `close` and `shutdown` as opening and closing of the `FITM_FD` needs to be done by the syscall translation layer.

For misbehaving targets we added the option to `exit(0)` immediately from `close`, and empty `recv`s, as the client would wait on *stdin* for the the user to establish a new connection. This means we do not fuzz teardown code, and miss stack overflows in calling functions, but we do not trigger unwanted hangs

In summary, these modifications allow for two things: *(a)* supply input to the target with files instead of sockets. And *(b)* create snapshots during specific events, namely receiving data from sockets during `recv` and `read` calls.

1) Target Input: In order to effectively get input into the target, we want to intercept all instances in which the target tries to communicate over a TCP/IP socket. Instead of opening and using the socket, we want the target to use fuzzer input.This should be transparent to the target as one of FitM's purposes is removing the necessity of manual target preparation. The target should still work regardless of where the socket FD points. The target receives fuzzing input in all places where it reads from `FITM_FD`, were socket communication with the other party would have happened.

2) Receive Functions: The receive function `recv` and `read` are the core components of our network emulation layer. If a receive function is called with the `FITM_FD`, previously returned by either `accept` or `socket`, several things happen.

Firstly, we increase the counter in a fixed location of AFL's fuzzer map each time we reach a send or receive function, similar to the **waypoints** of Ijon [14]. Further, if the target already sent network data in the meantime (or it is the initial run), we check if we are running in one of the four modes that can be switched to between snapshot restores:

- *fuzzing mode*: Return data from AFL++'s shared map input if available, else return 0 or exit.
- *replay mode*: Read next packet from sequentially numbered files in a folder. Always return the remaining contents of the last opened file, or open the next file if necessary and it still exists.
- *timewarp mode*: Create a CRIU snapshot for future fuzz runs.
- *output mode*: Open a file at a supplied path to write output to. Technically, this can also be done in either of the three previous modes. It is done in an extra run to save time during fuzzing.

All receives to all potential sockets are multiplexed through `FITM_FD`, so each receive is called on this pseudo-FD sequentially. This works as no multiprocessing exists after the socket opens. Even if the client would send on multiple sockets in a different order each time, we handle desynchronization (as described in subsection III-C) as we pass input for multiple stages to each run when calling `process`.

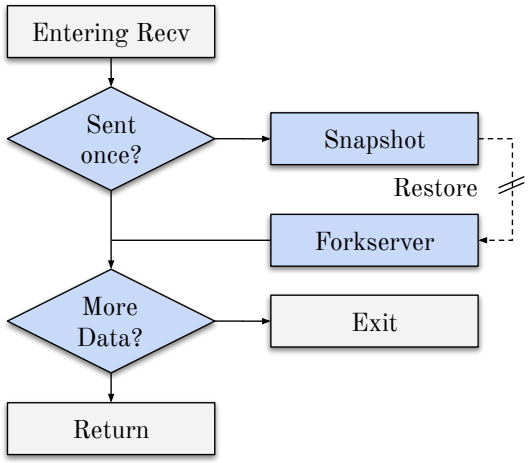

Fig. 3: Hooked *recv* function of FitM. It will take a new snapshot after each send, and can be restored at this point later in time.

In *timewarp mode*, the `read` and `recv` syscalls trigger an remote Procedure Call (RPC) from the target to another process, the CRIU server. This CRIU server will then snapshot the target that sent the call. The logic to decide when to trigger a snapshot is visualized in Fig. 3.

In order to only create new snapshots when the target actually expects new input, a snapshot is only triggered after the target has sent data at least once. The assumption is that the basic program loop of a client-server architecture program will consist of "recv input" $\rightarrow$ "process input" $\rightarrow$ "send output". If a target does not fulfill this assumption, FitM will not be able to fuzz it effectively.

To decide whether or not to create a snapshot, we introduce a second constraint next to the "sent"-constraint: snapshots are only created if the target runs in timewarp mode. If timewarp mode is not active, the target will exit at the point where it would otherwise issue a snapshot request to the CRIU server. This is done to increase AFL++ 's speed. The fuzzer does not need to go beyond the point where the next recv() happens since it found an input that arrives at that point. Consequently, we can create a snapshot after fuzzing is done and start from there in the next generation.

While these two constraints are always active, the user can activate the third one if a given target requires it. In the case of the live555 media streaming server, the server will send and `recv` an initial packet to determine its own IP address. See Listing 1 for a simplified version of the code. These **unrelated sockets** would let FitM go out of sync from the very beginning, as the client never answers it fully. Logically this is part of another protocol that happens before the actual *Real Time Streaming Protocol (RTSP)* interaction that should be fuzzed with this target. Because the server communicates

with other parties than the target client, we want to skip this protocol prelude. For this, a configurable number of `recv` syscalls can be skipped before the first snapshot happens. The skipped `recv` syscalls will return 0.

```
// Get our address by sending a (0-TTL) multicast packet,
// receiving it, and looking at the source address used.
// (This is kinda bogus, but it provides the best guarantee
// that other nodes will think our address is the same as we do.)
// ...
  testAddr.s_addr = our_inet_addr("228.67.43.91"); // arbitrary
  Port testPort(15947); // ditto

  sock = setupDatagramSocket(env, testPort);
  // ...
  if (!writeSocket(env, sock, testAddr, testPort.num(), 0,
        testString, testStringLength)) break;
  // ...
  int result = select(numFds, &rd_set, NULL, NULL, &timeout);
  // ...
  int bytesRead = readSocket(env, sock,
       readBuffer, sizeof readBuffer,
       fromAddr);
  // ...
```

Listing 1: Pre-Run Socket Interaction in Live555

3) Clone, Fork, and Async: Servers usually want to accept connections from multiple clients. A widespread way is to spawn a thread for each client or a threadpool for all clients. As our actual fuzzer uses a normal afl forkserver, a forking target poses a challenge, especially if it forks again during fuzzing. However, disabling all forks and always following child or parent can break targets that run external programs as subprocesses, e.g., a shell script during startup. For FitM, we chose a middle ground. We disable all forks and clones *after* we started operating on `FITM_FD`. That way, the process can still spawn as usual. After the first IP socket interaction occurs, our patches kick in. From here on, as described in Table I, we always follow the child or always follow the parent. This way, we can fuzz the threaded server of LightFTP, see Listing 2, even though it has multiple forks and sockets, one for each client and an additional one for each data connection. The alternative way to monitor multiple FDs usually revolves around monitoring events about status changes thereof. For asynchronous networking, the Linux kernel provides a range of APIs. We implemented a modified version of the various APIs into the network emulator, with each of them reporting the `FITM_FD` as always having new activity.

4) Snapshot Configuration: Whenever running a snapshot, the FitM sets up various parameters, as discussed in Sect. V-B2. One snapshot will always run in multiple modes: fuzzing, timewarp, and output mode, so after snapshot creation, we need to be able to alter the target's configuration. We use environment variables within the syscall translation layer for this. The environment variables are part of the snapshot and will be restored every time. Hence we had to wrap calls to `getenv` so that a file called `envfile` is checked for the searched variable instead of the actual process environment. We read the env file before spawning the forkserver to stay out of the hot path. Only if that file is not present, the original `getenv` function gets called.

5) Restore from QEMU's perspective: In order to restore a process, some startup steps have to be taken inside QEMU. The previously introduced `getenv` wrapper is used to get parameters into the process. Snapshot restoration is then concerned with reading the `envfile` to set any needed variables. Afterward, FitM-QEMU starts the AFL forkserver and maps shared map for communication with AFL.

```
clientsocket = accept(ftpsocket,
                (struct sockaddr *)&laddr, &asz);
if (clientsocket != INVALID_SOCKET) {
  rv = -1;
  for (i=0; i<g_cfg.MaxUsers; i++) {
    if ( scb[i] == INVALID_SOCKET ) {

      scb[i] = clientsocket;
      rv = pthread_create(&th, NULL, (
     __ptr_thread_start_routine)
          &ftp_client_thread, &scb[i]);
      if ( rv != 0 )
        scb[i] = INVALID_SOCKET;

      break;
    }
  }
...
```

Listing 2: Common Threaded Server in LightFTP

C. CRIU

CRIU, *Checkpoint/Restore in Userspace* is an existing Linux tool that allows snapshotting process to disk and to restore them later.

For snapshotting we utilize CRIU's RPC protocol. It allows FitM targets to trigger snapshots of themselves during execution. During restoration, FitM leverages CRIU to change FDs within a restored process via `inherit-fd`. The target knows when to dump itself, inside a socket receive of FitM-QEMU. Within FitM's *qemuafl*-layer this is implemented as follows: When the target reaches a state that we want to snapshot, stdin/stdout and stderr are closed to be reattached upon restoration. FitM-QEMU issues a dump request to the CRIU server. The CRIU-server will freeze the target, serialize the process state and write the resulting images to disk. After the process has been restored, we reattach the manually closed stdout & stderr, return and run the fuzzing setup described next.

1) Restores: Subsection V-B4 introduced a way to load environment variables from the calling process into the restored process. AFL uses named pipes to communicate with the forkserver that spawns the fuzz targets. Typically AFL spawns the forkserver by itself and thus also knows the names of the pipes it uses to communicate with the forkserver. However, when AFL is used to target a restored process, the restored process needs to start the forkserver. The point in the execution where we saved the snapshot is where the network emulation layer spawns the forkserver and returns new input after restore.

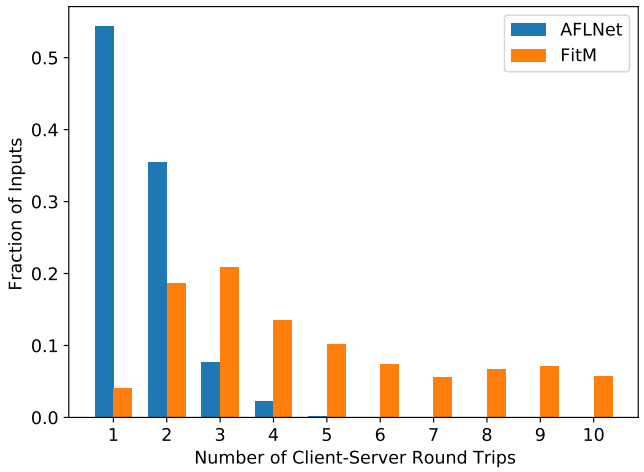

Fig. 4: Depth of fuzzer input interactions. FitM triggers far more client-server interactions.

Fuzzer	Traces[a]	BBs[b]	Hangs	Crashes	Depth	Total Execs
AFLNet	17	5880	22	0	5	424.701
FitM	146	6158	0	0[c]	10	113.683.192[d]

TABLE II: Fuzzing LightFTP for 15 hours in AFLNet & FitM

[a]We replayed each input back to back and printed each basic block translated by qemu during these runs. This number is the number of trace files with different basic block counts. There is still a chance that runs coincidentally reached the same BB count with different basic blocks, or they took a different path using the same basic blocks.

[b]Total amount of unique basic blocks across all inputs found by this fuzzer.

[c]While fuzzing the LightFTP server, FitM did uncover crashes in the connected ftp client.

[d]In contrast to AFLnet, each execution during fuzzing only covers a single state change, the speed for a complete trace. Therefore the overall speed would be up to **depth** times lower, for the case that we reach maximum **depth**. Note that AFLNet's speed was constantly below 10 execs/sec, while FitM stayed around 2k execs/sec on each run. On top, we only fuzzed the server for about half of the time, the client for the other half.

2) Namespaces: If a snapshotted Process Identifier (PID) clashes with an existing PID in the system, no snapshots can be restored as long as the offending process lives. We circumvent this situation by wrapping every invocation of `restore` in a newly initialized Linux namespace. On restore, FitM spawns a new mount and PID namespace and clones itself into those namespaces. After clone, the *proc* filesystem inherited by the caller process has to be reset not to include the information previously stored there, as it potentially include the offending PID. Afterward, we start the CRIU server in this PID namespace. Lastly the standard streams `stdin`, `stdout` and `stderr` are opened for the newly created namespace.

If the target has a very low PID it might conflict with one of the processes during namespace setup (e.g., bash, afl-fuzz, CRIU server). To be sure that this situation does not arise within the namespace we forcibly advance the next PID used by the process to be `(1 << 14) + rand(9000)` before the initial spawn of the client and server. Lastly, the target runs isolated and without any PID clashes.

While the usage of Unix namespaces increases FitM's kernel overhead, the resulting isolation of target processes comes with further benefits. Isolation into multiple different namespaces allows us to fuzz multiple different snapshots on the same machine, which allows us to scale fuzzing horizontally. Additionally, wrapping each fuzz-run in a new namespace simplifies the cleanup of the fuzzer since all processes of this namespace are automatically killed upon exit of its *init*-process.

VI. Evaluation

We decided to evaluate FitM by comparing it to AFLNet, which makes use of compile-time instrumentation and actual network sockets, regarding multiple metrics throughout a 15h fuzzing session. The sessions were run on a single core of an Intel i7-6850K each, supplied with a total of 128 GBs of RAM. Both used a ramdisk in their current working directory. The target we chose was LightFTP (v2.1), the server listed in AFLNet's example section. Furthermore, we supplied FitM with GNU's Inetutils (v1.9.4) FTP-client as its secondary target. We provide an overview of the key metrics in Table VI. Note that the basic block collection for AFLNet was done using a patched FitM-QEMU to gather comparable numbers. While our network emulation layer reached 2105 execs per second over 15 hours, not including FitM-internal snapshot reruns, AFLNet only reached around 8, even though it uses compile-time instrumentation. We do not attribute this to poorly designed algorithms but to the lack of network emulation. The use of TCP sockets for input delivery dampens throughput and scalability. Since one of the stated goals of FitM is the automatic exploration of stateful protocols, we discuss the amount of "long-lasting" connections and their prevalence in the entire corpus next.

A. State Depth

Figure 4 shows the proportion of inputs in the minimized queue in relation to the number of round-trips they would eventually reach. Even though AFLNet learns the state, according to our evaluation, they still get stuck in an early stage comparably often. We see this by the high proportion of inputs exiting within the first Server→Client→Server round-trip. This problem persists for all following messages, resulting in a quick decline of longer-lasting sessions, with the longest one ending in the fifth round-trip. Because FitM only briefly examines the base state and then explores multiple of its children, the exponential state-space growth leads to an increasing proportion of different inputs up to generation 3. However, our limit on the number of explored snapshots per generation starts to come into effect in generation 2 and leads FitM to discover roughly the same average number of snapshots in each following generation. If every snapshot resulted in the same number of new branches on average, the proportion of all following generations would stay roughly the same, up to the first reset. In reality, pruning and selecting related states to propagate into higher generations leaves fewer states to explore

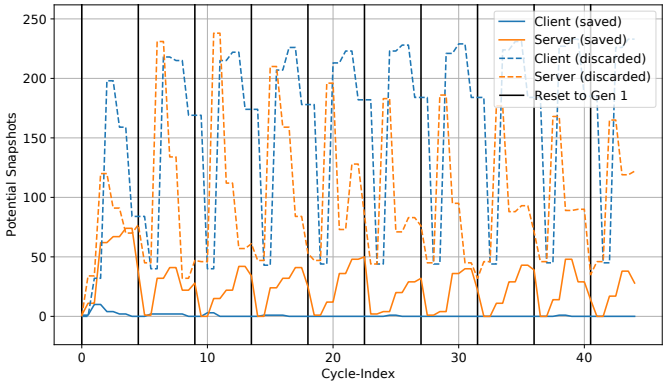

Fig. 5: Potential snapshots generated, and filtered, over time. FitM's filter discards unwanted potential snapshots by similar outputs and preexisting afl maps. After a while, hardly any new states need to be stored thanks to the filtering.

later than the earlier ones. The exact trend likely depends on target-specifics but provides a good indicator on which states to focus efforts. In this specific run, no reset condition got hit. The FitM experiment hit its timeout in the 21^{st} generation.

B. State Filtering

We continued the analysis of our fuzzer by examining the core problem of state-space growth. By examining Figure 5 it is discernible that the growth of new states is dependent on the state of the program and thus the number of round-trips (cycles) that ran previously. It is noticeable that there were many more unique snapshots taken in the server than the client, hinting at an overall more complex program with more available branches. Furthermore, the number of new states found (and discarded) is highly dependent on the generation, showing the effectiveness of state-aware, snapshot-based fuzzing, reaching deeper states, and exploring them. For long-running connections, the divergence of their states can be recognized by the reducing number of discarded snapshots, implying that later states accumulate increasingly unique paths. State filtering could be tuned more aggressively in the future, however looking at the ratio of filtered to saved stated in the client shows that almost all potential new states get rejected, indicating that the pruning is already effective. A possible solution could be an adaptive adjustment of the filtering constants or the inclusion of further information.

C. Further Targets

Apart from the statistical analysis, we also looked at several real-world targets. These included *Teamspeak*, an open-source SIP server called *Kamailio*, and *LightFTP*. With those targets, we were able to validate that our syscall patches work for more than our toy examples and test how easily we could bootstrap our fuzzer for a new target. We found crashes in the FTP client of Inetutils, which we used to interact with LightFTP without any custom patches.

VII. Future Work

Some areas we leave unexplored. For example, FitM does not fuzz different initial inputs to the client or the server's configuration. Fuzzing them would be beneficial in case of a strict client that disconnects after the first illegal server response. Alternating the client's request parameters would increase the possible state space by a large margin. Further research topics are listed in the following subsections.

A. Automatic Scaling

So far, we scale FitM by running multiple instances independently of each other, each target in their own namespace, exploring snapshots independently from each other. The part that can be synced between multicore instances is each generation's test case corpus. For further improvement, parallel fuzzing of different snapshots should be introduced, effectively sharing snapshots and states, allowing for horizontal scaling and quicker exploration through different generations. This mainly requires additional implementation work to extend the current isolation by namespacing and setting up dynamically *chrooted* environments to get around limitations imposed by CRIU.

B. Embedded Device and IoT Fuzzing

As many IoT devices run on Linux, one idea would be fuzzing the servers installed on such devices. Because FitM uses QEMU as a way to the instrument and execute the target binaries, little adaption is required for fuzzing binaries originating from IoT devices, such as routers. QEMU allows us to run binaries for other architectures which, in turn, enable FitM to effectively fuzz servers across all supported platforms. For this to work, the respective syscalls used by the binaries have to be emulated by QEMU correctly. A server may want to call an `ioctl` that needs do be stubbed out.

C. Instrumentation Improvements

As FitM is built upon *qemuafl*, many features shipped with AFL++ are already part of FitM-QEMU. For example, we make use of *CmpCov* [23] to break up multibyte compares into multiple single-byte compares and gather feedback as we reach correct values. For the future recent novelties in *qemuafl*, like replacing forks with a lightweight auto-snapshotting persistent loop, and the introduction of *QASan* sanitization [24] can be merged.

D. Multi-process Improvements and Race Condition Fuzzing

Currently, on `fork` and `clone` calls after the fuzz start, FitM-QEMU always either follows the child or the parent. However, in this way, we may lose other valid paths for specific targets and will never be able to uncover multiprocessing-related bugs, such as race conditions. Here the taken path could be chosen according to randomly generated fuzzer input and extended with a reproducible userspace scheduler, similarly to other race condition fuzzers [25].

E. IPC Connections

The concepts applied in FitM could be extended to allow fuzzing non-network services, even with multiple connected two parties in a FitM-like manner. By scheduling targets manually, this could be used to fuzz distributed systems or Inter Process Communication (IPC) in multi-process applications, with potential interest on targets like microservices and browsers.

F. Exploration of Protocol Reversing

In section VI we show, that FitM outperforms State-of-the-Art coverage-guided fuzzers on protocol depth. This ability can be explored further. While FitM already supports a *server-only* mode, in which most fuzzing time is spent on one target, with the other target not necessarily being a proper client, this functionality has not yet been appropriately evaluated. In test runs against *Live555*, we verified that the state snapshotting functionality of FitM finds paths even without a functioning peer. This gives us confidence that it could be leveraged to reverse engineer unknown protocols, especially when paired with additional binary introspection, such as automatic dictionaries, *Redqueen* [26] and methods used by Weizz [27].

VIII. Conclusion

FitM, the Fuzzer in the Middle, improves stateful fuzzing capabilities of coverage-guided protocol fuzzers with userspace snapshotting. For this, it uses existing protocol implementations of respective network peers. It fuzzes both, the client and the server, at the same time.

For the evaluation, we ran experiments against an existing State-of-the-Art protocol fuzzer. In comparison to prior work, we improved stateful exploration over time, and achieve a significantly higher execution speed per second. Within our use cases, the network emulator presented a reliable way to deliver input to a target with high throughput for synchronous and asynchronous workloads. With low effort, FitM found buffer overflows in GNU Inetutils that we reported to the maintainers. In summary, this work bootstraps a new direction of potential research in this area with possible improvements for faster snapshotting, more advanced metrics, and more general handling of syscalls for different target behaviors. Additionally, we hope FitM will help the community to discover and fix, bugs in network software in order to secure it further.

Availability

The source code for FitM has been open-sourced at
https://github.com/FGSect/FitM

References

[1] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++ : Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies, WOOT 2020, August 11, 2020*, Y. Yarom and S. Zennou, Eds. USENIX Association, 2020. [Online]. Available: https://www.usenix.org/conference/woot20/presentation/fioraldi

[2] checkpoint restore, "CRIU," Jan 2021, [Online; accessed 30. Jan. 2021]. [Online]. Available: https://github.com/checkpoint-restore/criu

[3] S. Schumilo, C. Aschermann, A. Abbasi, S. Wör-ner, and T. Holz, "Nyx: Greybox hypervisor fuzzing using fast snapshots and affine types," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 2597–2614. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/schumilo

[4] D. Song, F. Hetzelt, J. Kim, B. B. Kang, J.-P. Seifert, and M. Franz, "Agamotto: Accelerating kernel driver fuzzing with lightweight virtual machine checkpoints," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 2541–2557. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/song

[5] gamozolabs, "chocolate_milk," Jan 2021, [Online; accessed 29. Jan. 2021]. [Online]. Available: https://github.com/gamozolabs/chocolate_milk

[6] S. Schumilo, C. Aschermann, A. Jemmett, A. Abbasi, and T. Holz, "Nyx-net: Network fuzzing with incremental snapshots," 2021.

[7] M. Zalewski, "American Fuzzy Lop - Whitepaper," https://lcamtuf.coredump.cx/afl/technical_details.txt, 2016.

[8] V. Pham, M. Böhme, A. E. Santosa, A. R. Caciulescu, and A. Roychoudhury, "Smart greybox fuzzing," *IEEE Transactions on Software Engineering*, 2019.

[9] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A. Sadeghi, and D. Teuchert, "NAUTILUS: fishing for deep bugs with grammars," in *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/nautilus-fishing-for-deep-bugs-with-grammars/

[10] J. Wang, B. Chen, L. Wei, and Y. Liu, "Superion: Grammar-aware greybox fuzzing," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 724–735.

[11] V. T. Pham, M. Böhme, and A. Roychoudhury, "Aflnet: A greybox fuzzer for network protocols," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, 2020, pp. 460–465.

[12] V. Ulitzsch, D. Maier, and B. Shastry, "Follow the white rabbit: Simplifying fuzz testing using FuzzExMachina," 2018, BlackHat USA 2018. [Online]. Available: https://i.blackhat.com/us-18/Thu-August-9/us-18-Ulitzsch-Follow-The-White-Rabbit-Simplifying-Fuzz-Testing-Using-FuzzExMachina.pdf/

[13] H. Gascon, C. Wressnegger, F. Yamaguchi, D. Arp, and K. Rieck, "Pulsar: Stateful black-box fuzzing of proprietary network protocols," in *Security and Privacy in Communication Networks*, B. Thuraisingham, X. Wang, and V. Yegneswaran, Eds. Cham: Springer International Publishing, 2015, pp. 330–347.

[14] C. Aschermann, S. Schumilo, A. Abbasi, and T. Holz, "Ijon: Exploring deep state spaces via fuzzing," in *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 2020, pp. 1597–1612.

[15] "FuzzOS," Dec 2020, [Online; accessed 29. Jan. 2021]. [Online]. Available: https://gamozolabs.github.io/fuzzing/2020/12/06/fuzzos.html

[16] D. Maier, L. Seidel, and S. Park, "BaseSAFE: baseband sanitized fuzzing through emulation," in *WiSec '20: 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks, Linz, Austria, July 8-10, 2020*, R. Mayrhofer and M. Roland, Eds. ACM, 2020, pp. 122–132. [Online]. Available: https://doi.org/10.1145/3395351.3399360

[17] J. Ruge, J. Classen, F. Gringoli, and M. Hollick, "Frankenstein: Advanced wireless fuzzing to exploit new bluetooth escalation targets," in *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, S. Capkun and F. Roesner, Eds. USENIX Association, 2020, pp. 19–36. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/ruge

[18] E. Gustafson, M. Muench, C. Spensky, N. Redini, A. Machiry, Y. Fratantonio, D. Balzarotti, A. Francillon, Y. R. Choe, C. Kruegel, and G. Vigna, "Toward the analysis of embedded firmware through automated re-hosting," in *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. Chaoyang District, Beijing: USENIX Association, Sep. 2019, pp. 135–150. [Online]. Available: https://www.usenix.org/conference/raid2019/presentation/gustafson

[19] Y. Shoshitaishvili, "preeny," Jan 2021, [Online; accessed 30. Jan. 2021]. [Online]. Available: https://github.com/zardus/preeny

[20] guidovranken, "network-emulator," Jan 2021, [Online; accessed 30. Jan. 2021]. [Online]. Available: https://github.com/guidovranken/network-emulator

[21] D. Liu, V.-T. Pham, G. Ernst, T. Murray, and B. I. P. Rubinstein, "State selection algorithms and their impact on the performance of stateful network protocol fuzzing," 2021.

[22] W. W. Cohen, P. Ravikumar, S. E. Fienberg *et al.*, "A comparison of string distance metrics for name-matching tasks." in *IIWeb*, vol. 3, 2003, pp. 73–78.

[23] laf intel, "Circumventing Fuzzing Roadblocks with Compiler Transformations," Aug. 2016, [Online; accessed 11. Jan. 2021]. [Online]. Available: https://lafintel.wordpress.com

[24] A. Fioraldi, D. C. D'Elia, and L. Querzoni, "Fuzzing binaries for memory safety errors with qasan," in *IEEE Secure Development, SecDev 2020, Atlanta, GA, USA, September 28-30, 2020*. IEEE, 2020, pp. 23–30. [Online]. Available: https://doi.org/10.1109/SecDev45635.2020.00019

[25] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin, "Razzer: Finding kernel race bugs through fuzzing," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 754–768.

[26] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "Redqueen: Fuzzing with input-to-state correspondence." in *NDSS*, vol. 19, 2019, pp. 1–15.

[27] A. Fioraldi, D. C. D'Elia, and E. Coppa, "Weizz: Automatic grey-box fuzzing for structured binary formats," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 1–13.