No Peer, no Cry: Network Application Fuzzing via Fault Injection

Nils Bars
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
nils.bars@cispa.de

Moritz Schloegel
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
moritz.schloegel@cispa.de

Nico Schiller
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
nico.schiller@cispa.de

Lukas Bernhard
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
lukas.bernhard@cispa.de

Thorsten Holz
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
holz@cispa.de

Abstract

Network-facing applications are commonly exposed to all kinds of attacks, especially when connected to the internet. As a result, web servers like Nginx or client applications such as curl make every effort to secure and harden their code to rule out memory safety violations. One would expect this to include regular fuzz testing, as fuzzing has proven to be one of the most successful approaches to uncovering bugs in software. Yet, surprisingly little research has focused on fuzzing network applications. When studying the underlying reasons, we find that the interactive nature of communication, its statefulness, and the protection of exchanged messages (e.g., via encryption or cryptographic signatures) render typical fuzzers ineffective. Attempts to replay recorded messages or modify them on the fly only work for specific targets and often lead to early termination of communication.

In this paper, we discuss these challenges in detail, highlighting how the focus of existing work on protocol state space promises little relief. We propose a fundamentally different approach that relies on *fault injection* rather than modifying messages. Effectively, we force one of the communication peers into a weird state where its output no longer matches the expectations of the target peer, potentially uncovering bugs. Importantly, this *weird peer* can still properly encrypt/sign the protocol message, overcoming a fundamental challenge of current fuzzers. In effect, we leave the communication system intact but introduce small corruptions. Since we can turn either the server or the client into the weird peer, our approach is the first that can effectively test client-side network applications. In an extensive evaluation of 16 targets, we show that our prototype Fuzztruction-Net significantly outperforms other fuzzers in terms of coverage and bugs found. Overall, Fuzztruction-Net uncovered 23 new bugs in well-tested software, such as the web servers Nginx and Apache HTTPd and the OpenSSH client.

CCS Concepts

• **Security and privacy → Systems security**; **Network security**; **Software and application security**.

Keywords

fuzzing, software testing

ACM Reference Format:
Nils Bars, Moritz Schloegel, Nico Schiller, Lukas Bernhard, and Thorsten Holz. 2024. No Peer, no Cry: Network Application Fuzzing via Fault Injection. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3658644.3690274

1 Introduction

The internet has defined many aspects of modern life, including the unhindered exchange of information, real-time communication across the globe, and the ability to conduct business entirely online. The very same connectivity that provides numerous benefits also introduces risks as internet traffic passes through multiple, potentially untrusted nodes. In effect, remote attackers can potentially exploit any bug or software fault to compromise systems, which emphasizes the importance of network-facing applications as a critical security boundary.

Even a single memory corruption error in the web server or network client can have disastrous consequences, which demands thorough testing of these applications. To this end, we first need to understand the attack surface: Network applications receive and process messages that use a multitude of protocols to exchange data, such as HTTP, SSH, SIP, or SMB. These protocols define the rules and structures for communication and ensure that different devices can interpret and exchange data coherently. Thus, automated testing must account for this structure, testing inputs that are valid to some degree, such that the application's parser does not immediately discard them but exercises deeper functionality. Cryptographic techniques are widely used to secure communication, making these testing efforts more difficult. Protocols like TLS (Transport Layer Security) provide secure communication over networks that may not be inherently secure. The transition to HTTP/3 underscores the importance of built-in encryption by incorporating the QUIC transport layer, which mandates TLS 1.3. In addition to encryption, compression is used in many network protocols to reduce overhead and maintain efficiency, and checksums are commonly employed to ensure a reliable data transfer. All these concepts complicate automated security testing, as these security measures make it difficult to generate valid messages.

Interestingly, the field of fuzzing, otherwise renowned for effective bug finding [19], shows a noticeable gap in testing network-facing applications. Despite the advent of AFL [43] sparking a renaissance of fuzzing methods with hundreds of new fuzzers proposed [2, 6, 14, 15, 29, 42, 45], most of them are limited to a specific subset of software. Most benchmarks [20, 21, 30], academic papers in general [34], and public industry initiatives such as OSS-Fuzz [19] focus predominantly on comparably simple C/C++ Linux programs that consume byte-oriented input. In contrast, network applications are significantly more complex but have attracted relatively little attention in fuzzing research [10].

Broadly speaking, existing work can be split into two types of approaches. First, *replay-based* fuzzers [3, 31, 33] record valid communication and use saved messages as seeds. By replaying (and mutating) these messages, they do not require extensive protocol knowledge. However, this approach is limited, mainly because most protocols use ephemeral values, such as session identifiers that are dynamically created during runtime. These values prevent replaying old messages and lead to *low input stability*, i.e., executing the same input more than once will exercise entirely different code parts. To avoid this problem, a second type of approach [28] uses a *Man-in-the-Middle (MitM)* technique to modify messages in transit rather than replaying old ones. This way, session identifiers are implicitly adapted. However, this causes a state divergence between one peer sending the unmodified message and the other receiving a modified one. This leads to *desynchronization*, where peers disagree on the current protocol state. Additionally, ubiquitous integrity checks or encryption prevent modification of messages in transit, as traditional fuzzer mutations immediately invalidate the integrity. The receiving peer then discards such invalidated messages before any interesting functionality is exercised. These challenges limit the effectiveness of existing work for many network applications, confining them to a small subset of applications that feature no integrity protection or ephemeral values. Instead of addressing these challenges, we find that state-of-the-art fuzzers focus on *protocol state coverage*. This metric measures the extent to which the fuzzer has explored the various states defined by the protocol's state machine as an additional feedback mechanism [31, 33, 44]. While this may be beneficial to navigate the intricacies of the protocol's logic, it implicitly assumes that the fuzzer can successfully exchange multiple messages without desynchronization and mutate messages without invalidating them—to the best of our knowledge, this is not the case for any state-of-the-art network application fuzzer.

In this paper, we present the design and implementation of Fuzztruction-Net, a powerful and flexible approach to network application fuzzing. Instead of replacing any peer or the communication channel, replaying observed messages, or modifying messages as MitM, our method uses one of the communication endpoints (i.e., either client or server) to generate protocol messages. This approach enables testing protocols that rely on continuous exchange and stateful interactions. Ephemeral values or encrypted messages pose no problem, as our mutations reside within the communication system. More precisely, Fuzztruction-Net uses high-level mutation strategies based on injecting faults, a method previously explored in other contexts [4, 23, 27, 36], into the endpoint that does not represent the system under test. These strategies include manipulating the application's control flow, such as flipping branching decisions or changing function call destinations to explore different execution paths and protocol states. This allows our approach to mutate the generation process of messages (rather than generated messages) while still preserving their semantics, effectively enabling us to explore deeper application logic. A compelling aspect of Fuzztruction-Net is its ability to handle common obstacles such as cryptographic operations, compression algorithms, and checksums, as faults can be injected before protocol messages are wrapped by these complex primitives. Moreover, our approach enables the fuzzing of both client and server components by switching the endpoint targeted for fault injection, a significant advantage over traditional methods focusing exclusively on server-side fuzzing.

We implemented a prototype of Fuzztruction-Net and found that it significantly outperforms existing state-of-the-art fuzzers, finding on average 16% more coverage and uncovering 3x more bugs than the second-best fuzzer, SGFuzz. Also, Fuzztruction-Net finds bugs in Nginx, the OpenSSH client, and apache2 that previous fuzzing attempts failed to uncover despite existing fuzzing harnesses and OSS-Fuzz integration.

Contributions. We make the following main contributions:

- We systematically analyze the shortcomings of state-of-the-art network application fuzzers, finding that they address neither desynchronization nor input stability.
- To address these challenges, we present Fuzztruction-Net, our approach that uses fault injection to turn one communication partner into a weird peer that can be used for testing. Fuzztruction-Net is the first network application fuzzer that can test both servers *and* clients, compared to existing methods that are limited to servers only.
- We found 23 remotely triggerable bugs in popular network applications, such as the Nginx web server, the OpenSSH client, and the popular QUIC library ngtcp2 used by curl.

We release our prototype at https://github.com/fuzztruction/fuzztruction-net upon acceptance and intend to participate in the artifact evaluation process.

2 Background & Challenges

Before diving into our approach towards fuzzing network applications, we briefly discuss how network applications operate, enumerate the challenges they pose for fuzzing, and discuss how state-of-the-art approaches attempt to tackle the problem.

2.1 Network Applications

When referring to network applications, we consider a scenario where multiple applications, called peers, exchange data. Typically, one of these peers takes the role of a *server*, responsible for serving the requests of one or multiple *clients*. The server continuously listens on a specified IP address and port, waiting for the clients to initiate a connection. To process multiple requests in parallel, the server-side software is either multithreaded or uses an event-driven design based on synchronous I/O multiplexing as provided by the `select` syscall and alike. Sometimes, as for WebRTC-based video calls, no peer takes a distinct role; instead, clients communicate

directly with each other. This work focuses on the more prevalent client-server architecture.

Regardless of the setup, (interactive) communication takes place across a network. To this end, a transport protocol based on the IP protocol is usually used to exchange messages. The most common protocols are TCP and UDP, which significantly differ in reliability, logical connection type, and data flow properties. Notably, some applications employ their own transport protocol, which can be based on existing ones or entirely independent of commonly used protocols. One example of such protocols is QUIC, which is based on UDP and is intended to replace TCP with better performance properties and native support for encryption and authentication while providing the same reliability as TCP.

Generally speaking, network applications significantly contrast traditional fuzzing targets, where we execute only a single application (e.g., a binary executable). Such targets usually do not interact with other applications, and their execution is—nondeterminism aside—mainly determined by the given input.

2.2 Fuzzing Challenges

The differences between network applications and typical targets lead to three key challenges that must be addressed by any fuzzing approach to effectively and efficiently test network applications.

C1. Session state To facilitate *meaningful* communication, the client and the server maintain state throughout their communication. Crucially, some parts of the state are *ephemeral*, such as session identifiers. This has a significant impact on the fuzzing process. An input that achieved high coverage in the first run may fail in the second run *unless* the temporary identifier is updated to the new one. The server may send and expect to receive these identifiers at runtime, challenging the fuzzer's ability to predict them in advance. One famous example of such behavior is HTTP digest authentication [37], where a server-chosen nonce protects communication against replay attacks. These dynamically chosen ephemeral values lead to the problem of *low input stability*, as inputs cannot be replayed. At the same time, failure to manage the session state across roundtrips leads to *desynchronization*, where the peers disagree on the current protocol state.

C2. Integrity checks and encryption To protect messages, server and client use various integrity checks, including encodings, checksums, and encryption. Modifying these messages is virtually impossible for typical fuzzer mutations, as local changes invalidate the message integrity. Consequently, the recipient discards the message during parsing without executing more interesting code parts. Naively, testing can record live communication and replay these messages. However, modern cryptographic algorithms require the use of ephemeral values to protect against codebook attacks, mandating to solve challenge C1.

C3. Network Application Setup Assuming a fuzzer can maintain state across iterations and has found a way to deal with any integrity checks or even encryption, it must still be tailored towards the system under test (SUT). In particular, this includes aspects such as when to send fuzzing input—as a server may need to perform initialization before accepting connections—or when to terminate the fuzzing session—as a server may wait for additional client messages rather than terminating itself, contrasting file processing applications, which usually exit after having consumed all input.

2.3 Modern Network Application Fuzzing

Several approaches have been proposed to fuzz test network applications, and we will summarize the state-of-the-art work next.

AflNet [33]. AflNet is the first grey box fuzzer for network protocol implementations based on AFL. It focuses on integrating protocol state feedback into AFL's coverage-guided fuzzing approach. Central to AflNet's approach is the isolation of *protocol data units* (PDUs), i.e., messages as defined by the underlying protocol. Using these isolated messages as building blocks, AflNet navigates the protocol state space by permuting the order of messages or by mutating their content. To infer the already covered protocol state space, AflNet uses manually crafted parsers to assign an identifier to received messages. In essence, the order of received messages and their assigned identifier determines the path exercised in the inferred state machine. Since AflNet is based on AFL, it inherits its limitations regarding ephemeral values (C1) and any kind of integrity protection (C2). Also, a human expert is required to create a parser for the target protocol to infer the protocol state from messages.

SGFuzz [3]. SGFuzz is an in-process fuzzing tool based on LibFuzzer and uses a library provided by Honggfuzz [18] to enable the fuzzing of network applications. The core design of SGFuzz assumes that the current state of the protocol is reflected in the runtime values of `enum` variables. To monitor these variables, the source code of the target application is preprocessed. This involves wrapping the assignment expressions of enum-typed variables (i.e., `enum_t v = <enum_variant>`), enabling observation of the assigned values. A *state transition tree* is constructed from the values observed during fuzzing. If new transitions in the protocol state are discovered, SGFuzz can use this information to determine if a new protocol state has been encountered. Like AflNet, SGFuzz entirely relies on the fuzzing logic provided by its baseline, i.e., LibFuzzer, which features bit-level mutations incapable of overcoming more complex fuzzing roadblocks (C1, C2). Furthermore, the Honggfuzz library used for network fuzzing does not support UDP, limiting SGFuzz' applicability to specific protocol implementations.

StateAFL [31]. Based on AflNet, StateAFL uses a different approach than its baseline to tackle the problem of protocol state space exploration. To avoid manual effort, StateAFL uses the content of memory allocations to infer the current protocol state. By intercepting APIs related to networking, such as those creating sockets or those reading or writing data to them, StateAFL infers possible points in time at which it checks if the content of tracked memory objects indicates that a new state has been covered. This approach comes with the risk of introducing errors into the target under test, e.g., if the memory tracking is unsound or incomplete, the instrumentation may access a memory object that is not allocated. Since StateAFL is based on AflNet, it suffers from the same limitations regarding more complex targets that rely on session state (C1) or cryptography (C2).

Bleem [28]. The recently proposed approach called Bleem does *not* replace the client with a fuzzer—contrasting previous approaches—but instead operates as MitM. It connects to both communication parties and forwards messages between them. This setup uses protocol parsers to deduce the protocol's current state, which helps construct a graph representing the system's state. Using this graph, Bleem generates new packet sequences or mutates the transmitted data. While this MitM approach addresses the input stability problem of C1 and allows Bleem, for example, to perform a TLS handshake by simply forwarding messages without modifications, this may still cause desynchronization, as the sent and received messages differ, causing different states in each peer. Also, Bleem lacks the ability to mutate encrypted or integrity-protected data (C2) and requires hand-crafted protocol parsers to infer protocol state.

All state-of-the-art fuzzers using a replay-based approach fail to address C1 and C2. Only Bleem, with its MitM-based approach, overcomes C1 partly but still does not account for integrity protection. All fuzzers tailor their approach towards network application fuzzing, addressing C3, and use different forms of protocol state as feedback for guiding the fuzzer. However, without solving C1 and C2, their approaches are severely limited and require patching of targets (e.g., to remove integrity protection) for any meaningful exploration.

3 Design

To effectively and efficiently test network applications, we need a design that maintains session state across exchanged messages (C1), accounts for integrity protection mechanisms (C2), and is tailored towards the needs of network applications (C3).

3.1 Idea & Rationale

A fuzzer intending to address these challenges needs to adapt its machinery to take previously exchanged messages into account. The complexity and diversity of network applications make this problem hard to solve in the general case: Without re-implementing all protocols, a fuzzer's heuristics will always fall short in some regard. We argue that replacing one peer with a fuzzer, as done by AflNet, SGFuzz, or StateAFL, is not only unnecessary but potentially limiting successful testing. The fuzzer taking the peer's place has no notion of state to preserve across exchanged messages, nor does it know the underlying protocol or state machine. Thus, discarding the original peer with all its domain knowledge potentially degrades the system's quality regarding communication breadth and depth. Furthermore, we argue that placing the fuzzer as MitM, as done by Bleem, has similar shortcomings. In a sense, this approach replaces the communication channel with a fuzzer-controlled one. While leveraging the domain knowledge inherent in a peer, any communication protected by integrity checks or encryption is out of reach, limiting the applicability of this approach to specific network protocols. Even if messages are not protected, any modifications made by the fuzzer will only be known to the recipient and not the sender. This leads to *peer desynchronization*, as one peer's internal state diverges from the other's.

In this work, we propose a method to introduce a fuzzer into this communication system while replacing *neither* any peer *nor* the communication channel. This way, we can preserve the dynamic nature of the communication between peers and maintain synchronicity of the protocol state on both sides. Naturally, these requirements leave us with little room for where to place our fuzzer. Especially the second constraint, i.e., preventing desynchronization, mandates including the fuzzer in one peer itself, albeit without replacing it entirely. Fundamentally, we rely on *fault injection* [4, 23, 27] to achieve our goal. Rather than mutating specific input bytes, we mutate the peer itself, causing its output to produce unexpected or—at times—incorrect messages. More specifically, we propose injecting faults into the peer's state-modifying operations, e.g., we could inject a fault into the selection of an encryption algorithm, provoking an untypical choice the other peer may not expect. At the same time, we need to avoid injecting faults into the encryption algorithm itself, as this would invalidate the produced output, which is unlikely to pass the other peer's decryption stage. In essence, our faults should neither invalidate the communication nor cause an unsynchronized protocol state transition, leading to the peers' disagreeing on the current state and, thus, terminating communication. If we can achieve these design goals, our fuzzer can play off one peer against the other. We stress that this enables us to fuzz one peer *or* the other, i.e., we could target the server, as all existing work does, or the client, which existing tools largely neglect.

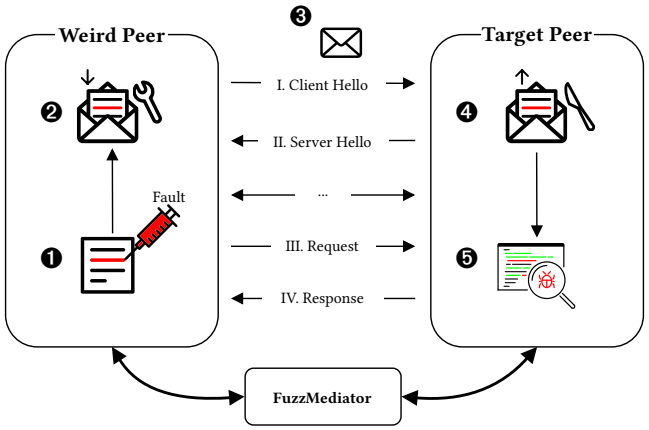

Figure 1: High-level overview of Fuzztruction-Net. Notably, the role of weird and target peer is independent of which peer is acting as server or client.

3.2 Overview

Figure 1 shows an overview of our proposed design and sketches the exemplary interaction of two peers. For the sake of this example, we assume our goal is fuzzing the server, i.e., the server is the *target peer*, while we inject faults into the client, making it the *weird peer* (in reference to *weird machine* [5]). We stress that the inverse setup is possible as well. Beyond these two communicating peers, we use an external component called *FuzzMediator* that manages the fuzzing loop by observing and controlling the other components of the system. The fuzzing loop starts with ❶ injecting some fault in the *weird peer* that leads to unexpected behavior, e.g., during the preparation of a protocol message. Ideally, the fault corrupts the message slightly, but the subsequent logic still processes it normally,

e.g., encrypting it as expected by the communication protocol (❷). Then, ❸ the client initiates communication with the server, e.g., by sending a Client Hello message (I.). Upon receiving the message, the server decodes it (❹) before further processing, potentially triggering a bug (❺) due to the corruption. If no bug leads to an abnormal termination, the communication (and thus the current fuzzing iteration) continues as the server responds to the received message. The weird peer then processes this response (II.). At this point, we can inject another fault to further alter the expected message flow or content of the following message (III.). Once the message is sent, the server will again respond (IV.), continuing this communication loop until one of the peers terminates (normally or abnormally). This concludes one fuzzing iteration, after which we collect coverage information of the target peer. Collected code coverage is reported to the *FuzzMediator*, which can be used as feedback to decide if one (or more) injected fault(s) lead to interesting behavior and schedule the next fuzzing iteration.

3.3 Fault Injection

Now that we have outlined our design goals and presented an overview, we focus on the *fault injection*, the centerpiece that turns one of the peers into the *weird peer*. This step is the key ingredient towards introducing some error into the regular communication flow that then, in turn, uncovers a bug in the *target peer*. In principle, our goal is to corrupt message content to some minor extent without modifying its later processing, such as calculating a checksum or encrypting it. This may uncover incorrect or missing validation in the other peer without the message being easily discarded due to violating the integrity protection mechanisms. In our design, this can be achieved easily *if* we know what part of the program to mutate and in which way. However, we lack information on both *where* and *how* to mutate the program. To address either, we rely on the inherent randomness in the fuzzing process.

Fault Types. Before focusing on the *where*, let us discuss the faults we can inject by mutating the weird peer. In particular, we target the following instructions to inject different fault types:

- Load/stores: modify *values* loaded from/stored to memory.
- Switch case: change the `case` statement executed.
- Conditions: invert conditional moves or branches.
- Calls: change the `call` destination or skip a call.

These faults operate at two levels: the *value* level, where we can modify data values, and the *control-flow* level, where we can invert a conditional branch or even change the called functions. The former helps corrupt data values, while the latter may execute unexpected code, leading to *weird* data processing. For fine-grained control over the particular fault, each location receives a fuzzer-generated input stream, which we can use to inject a different fault each time this location is executed. The length of this stream must account for the *fault type*, e.g., inverting a condition only requires one bit as opposed to modifying a 64-bit store, which requires eight bytes, and it needs to account for the number of times the location is executed. During each execution, we extract the necessary bytes from the input stream to XOR with the loaded/stored value, influence a condition's outcome, or affect which function is called.

Fault Locations. Now, as the instructions at which we can inject a fault occur thousands of times in network applications, the core question is *where* to introduce our faults. Fundamentally, we can address this by (1) trying to obtain more information using static analysis or similar approaches or (2) randomly introducing faults, pruning ineffective ones, and focusing on high throughput. Sticking to the fuzzing spirit, where high performance has been shown to trump informed but costly analyses [2], we focus on the second option.

Consequently, we mark all the instructions mentioned above during instrumentation at compile-time, leaving us with many potential targets for fault injection. To improve our odds of injecting faults at "good" locations, our fuzzing campaign features a dedicated *queue initialization*, which we describe subsequently when presenting the details of our fuzzing loop.

Despite our efforts, some fault locations will inevitably be less effective due to the random nature of the injection process. For example, faults could invalidate ephemeral value handling and encryption or lead to protocol state desynchronization. In such cases, the communication between the weird and target peer cannot proceed as desired, resulting in an early termination of the process. This early exit automatically leads to less coverage being exercised, so the fuzzer quickly avoids scheduling these faults again. Consequently, few computational resources are spent on these less effective faults. The same principle applies to *redundant* fault locations. By leveraging coverage to filter out ineffective locations, we can utilize the weird peer's capabilities in handling ephemeral values and managing state throughout the communication without needing to identify or model these capabilities explicitly.

3.4 Fuzzing Loop

Our fuzzing loop follows roughly those of regular fuzzing methods. The major difference is that one queue entry consists of a list of tuples `[(loc, stream), ...]`. `loc` denotes the fault injection location at which we apply the byte `stream` to mutate the value, condition, or call destination.

Calibration. As typical AFL-based fuzzers do, we run a calibration phase for each new fuzzing queue entry. In our case, we measure how often each fault injection location is hit, allowing us to calculate how much fuzzing input is needed during the execution of this queue entry. For example, a branch consumes 1 bit of fuzzing input to decide if it should be flipped. Thus, we need one bit of input each time one specific branch is executed.

Queue Initialization. Initially, our queue is empty. Due to the high number of fault locations, we initialize the queue as follows: We test a few mutations for each location and observe the target peer. If any of the mutations lead to new coverage, we create a new queue entry consisting of the tuple `[(loc, stream)]`. On the other hand, if the mutations repeatedly caused the weird peer to crash, we skip this location. This way, we automatically bias the fuzzer towards the initially successful fault locations, as the fuzzer's scheduler primarily picks entries from the queue.

Fuzzing Iteration & Mutations. From the fuzzer's perspective, one fuzzing loop consists of the following steps: (1) Pick one queue

entry, (2) mutate this queue entry, (3) execute the peers, and (4) collect exercised coverage. To mutate queue entries, we can:

- Mutate the `stream`: We mutate the applied `stream` using typical fuzzer mutations, such as bitflips or havoc. This, for example, changes the value loaded at the fault location or leads to the inversion of the branch condition.
- Splice queue entries: We can append a tuple from a second queue entry to the list of the current one. This effectively leads to combining two (or more) different faults.
- Extend queue entry: We can also attempt to append a newly generated tuple to the current queue entry, i.e., we try to find another interesting fault by selecting an unused fault location and generating a `stream` using the same mutation as in the first step.

3.5 Network Application Specifics

Given the nature of network communication, our design must address its unique needs and requirements, as discussed next.

Temporal Input Channel. The server must be running to ensure that the client can perform any meaningful action. Therefore, ensuring the client and server have the correct startup order is crucial. While for traditional fuzzing, it is sufficient to pass the fuzzing input via a file that is consumed as soon as the target is ready, the process is slightly more involved for networked targets. One common technique to determine whether the server is ready to process input sent by a client is polling, e.g., as used by AflNet. This is facilitated by consecutively trying to establish a connection to the server's port until it is successful or some timeout is hit. This comes at the cost of wasting CPU cycles and cannot deal with "connectionless" protocols such as UDP, where no handshake occurs that would indicate whether the target port is active.

We rely on the server to tell us when it is ready to accept connections to avoid these shortcomings. We achieve this by hooking two functions: `listen` and `bind`. When one of these functions is called, the server can notify the *FuzzMediator* that it is ready to accept client requests. This requires no active waiting and works for all protocols.

Termination Point. Server applications typically run indefinitely while consecutively serving clients. Thus, when fuzzing a server, the state is possibly carried over between multiple executions, i.e., client connections. This shares the same downsides as in-process fuzzing (e.g., the observed behavior is not necessarily reproducible). Therefore, server applications are typically patched such that they terminate after handling a single client connection. This approach works reliably for connection-oriented protocols since the fuzzing iteration is terminated when the server closes the client connection and the client closes the connection, which the server can observe due to its connection-oriented nature. Unfortunately, such patching mechanisms do not work well for targets based on connectionless protocols since the server may wait for a client that already exited. Since the connection between client and server has no state, the server cannot detect such a situation except by using timeouts as an indicator. Similar to other approaches [31, 33], we tackle this problem by terminating the target via a signal after each fuzzing iteration if the target peer has not terminated earlier.

4 Implementation

We implement a prototype of Fuzztruction-Net based on the fault injection framework offered by Fuzztruction [4]. In particular, we inherit its use of LLVM's stack maps and patch points to dynamically mutate `store` and `load` operations in a program. We have extended the framework to suit our needs and made several key modifications, which we discuss in the following. We implemented about 7,900 lines of Rust and 800 lines of C/C++ code to build Fuzztruction-Net on top of Fuzztruction.

Fault Injection Mechanism. Despite being conceptually similar in terms of injecting faults, we had to revamp Fuzztruction's way of mutating LLVM's live values, which is used to apply faults at desired locations. The existing framework uses a so-called stack map to track values it wants to mutate. At runtime, it would then identify the register holding this value and—using a JIT compiler—create a function to mutate this value. While effective for modifying "direct" values that are loaded or stored, this approach does not work for *conditions*, which are managed via the `EFLAGS` register: The compiler assumes that the function created by the JIT compiler clobbers the register and will evaluate the condition only *after* our JIT-ed function call, making it impossible to modify the condition. Therefore, we implemented a completely new fault injection strategy and replaced the existing one: (1) We spill values that we want to mutate onto the stack, (2) we track the spill locations in a stack map, (3) and if mutating a condition, we spill the condition onto the stack before using a JIT-compiled function to mutate the value directly on the stack. (4) After the function mutated the value, we read it back from the stack and use it, e.g., in the `cmp` instruction.

Function Call Injection. We also incorporate a new method of injecting faults that allows us to call another function from the entry of an executed function. To achieve this, for each function, we generate a function pointer table during instrumentation that tracks functions with the same signature (i.e., same argument number and types). Additionally, we insert a patch point at function entry points that—when selected as a fault injection site—consumes one byte of fuzzing input that serves as an index into the function pointer table and redirects the control flow to this function instead.

Network Applications Support. The Fuzztruction framework is by design incompatible with network targets because it expects its targets to be executed once and sequentially. We reworked the framework to support the concurrent and repetitive execution of targets, mimicking the interactive nature of network communication. Furthermore, we added support for network namespaces, which allow each application to have a distinct set of network interfaces. This allows server applications to run on the same port in parallel, a necessary precondition for multiple parallel fuzzing runs.

Weird Peer. We use our LLVM pass and its runtime with the abovementioned changes to instrument the weird peer. In addition, we implement logic to detect when the server is ready for client connections. For this, we add support for hooking the functions `listen`, `bind`, and `connect`. Hooking these functions allows us to determine when a server is ready to accept connections and whether a client tried to connect to the server. The latter becomes handy if a mutation causes a weird client to crash before being

able to connect to the server. If a mutation leads to a client crash before connecting, we discard that fuzzing run, as the crashed client cannot impact the server's operation.

Target Peer. For the target peer, we leverage AFL++ in version v4.08 for its LLVM-based coverage instrumentation. Beyond that, we use the same hooking mechanism, modified LLVM pass and runtime as for the weird peer. This is necessary as the server can be either the weird peer or the target peer.

5 Evaluation

We evaluate how our implementation FUZZTRUCTION-NET performs compared to other network application fuzzers and whether it is capable of uncovering new bugs in real-world targets. To this end, we conduct three different experiments. (1) We benchmark our approach regarding code coverage and compare it to state-of-the-art network fuzzers. (2) Based on the coverage benchmark data, we evaluate the bug-finding capabilities of each individual fuzzer by analyzing all bugs that have been found during the coverage experiment. (3) Then we study the capabilities of FUZZTRUCTION-NET by testing additional widely deployed software projects, including clients that no other network fuzzer is designed to fuzz.

5.1 Setup

Before discussing our experiments and their outcome in-depth, we outline our setup and present targets we use throughout our evaluation. Overall, we follow the recommendations by Klees et al. [25].

Hardware Environment. To ensure fair experiments, we used the same hardware configuration for all our experiments: an Intel Xeon Gold 5320 CPU @ 2.20GHz (52 physical cores), 256 GB of RAM, and SSD memory as backing storage. To model realistic conditions, we spawned 13 fuzzer worker processes per fuzzer and pinned these workers to 13 (physical) cores. This allowed us to perform four experiments in parallel.

Fuzzers. We compare FUZZTRUCTION-NET against three state-of-the-art fuzzers: AFLNET [33], SGFUZZ [3], and STATEAFL [31]. We considered evaluating against other existing works like BLEEM [28], but their code is not publicly available, and the authors could not share the code when we contacted them. Other protocol fuzzers, such as BOOFUZZ [24] or PEACH [12], are available, but they offer only a limited range of protocol specifications by default, which is insufficient for testing more complex or diverse targets. Additionally, related work has found these tools to perform worse compared to fuzzers we included on our evaluation targets [28, 33]. In summary, we configured the tested fuzzers as follows:

(1) **AFLNET [33]** (`62d63a5`): We enabled its state-aware mode (`-E`) and the region-level mutation operators (`-R`) by default. In addition, we added a `NOP` protocol that does not provide any state feedback during fuzzing but allows for testing targets that are *not* supported by AFLNET. We used this protocol for the targets `mosquitto`^{S}, `nginx`^{S}, and `samba`^{S}.
(2) **SGFUZZ [3]** (`00dbbd7`): To use the distinct values of `enum` variables, SGFUZZ needs to preprocess the target's source code with a Python script that instruments assignments of `enum` variables. Unfortunately, this script crashes for some source files, aborting instrumentation. We identified and removed offending files from preprocessing.
(3) **STATEAFL [31]** (`d923e22`): As this tool is based on AFLNET (but uses memory allocations to deduce protocol state instead of handcrafted protocol parsers), we used it with the same flags as AFLNET. STATEAFL's instrumentation sometimes misses `free` operations, compromising their memory allocation tracking. As the fuzzer continues to access these already freed allocations to determine the protocol state, segmentation faults often occur (resulting in false positive crashes). Additionally, STATEAFL does not support certain functions, such as vectorized read and write operations, and initially refused to start if no network-related operations were detected, such as data written or read from a socket. To address these issues, we added support for multiple `libc` functions, including `readv` and `writev`. Despite these improvements, we encountered persistent issues with memory tracking for some targets, thereby limiting the overall robustness of STATEAFL on these targets.

Target Applications. For the comparative evaluation and the real-world bug-finding experiment, we selected 16 targets from eleven software projects in total. We also selected a second application acting as weird peer for each target. In all cases, we used a test client and server shipped by the software project, such that no manual work was needed to derive the weird peer. In the following, we indicate whether we refer to the client or server of a project by using superscripts, i.e., `gnutls`^{C} and `gnutls`^{S} refer to the client and server of the `gnutls` software suite, respectively.

When selecting suitable targets, we aimed to compile a set of applications that represent realistic targets for fuzzing network applications. While recommended, benchmarks [32] do not support our non-traditional way of fuzzing, making it impossible for us to use them. We selected targets according to the following criteria:

(1) Comparability with related work: The set of evaluated applications should intersect with the ones used by other network application fuzzers.
(2) Active projects: The selected targets should be actively maintained and relevant in practice.
(3) Optimally, the targets are thoroughly tested and have been fuzzed before. This allows us to quantify our new approach's impact rather than attributing findings to the fact that a project is tested for the first time.

To meet the first criterion, we selected a subset of the applications tested as part of the evaluation of the other fuzzers. From the evaluations of AFLNET and SGFUZZ, we selected `live555`^{S}, `dcmtk`^{S}, and `openssl`^{S}. Since both fuzzers found over 90% of their bugs in `live555`^{S} and `dcmtk`^{S}, we expected these targets to be excellent candidates for our evaluation.

To ensure we meet criteria (2) and (3), we selected the remaining targets based on whether they are shipped with fuzzing harnesses or are part of OSS-Fuzz [19] and if they seem to be actively maintained and do not have stale bugs in their respective bug trackers. Table 1 lists all targets we selected, outlining whether these targets have fuzzing harnesses as part of their repository and if they are integrated into OSS-FUZZ. Out of the eleven applications, only two are not integrated into OSS-FUZZ, `dcmtk` and `live555`. Interestingly, both have been the primary source of new bugs found by AFLNET

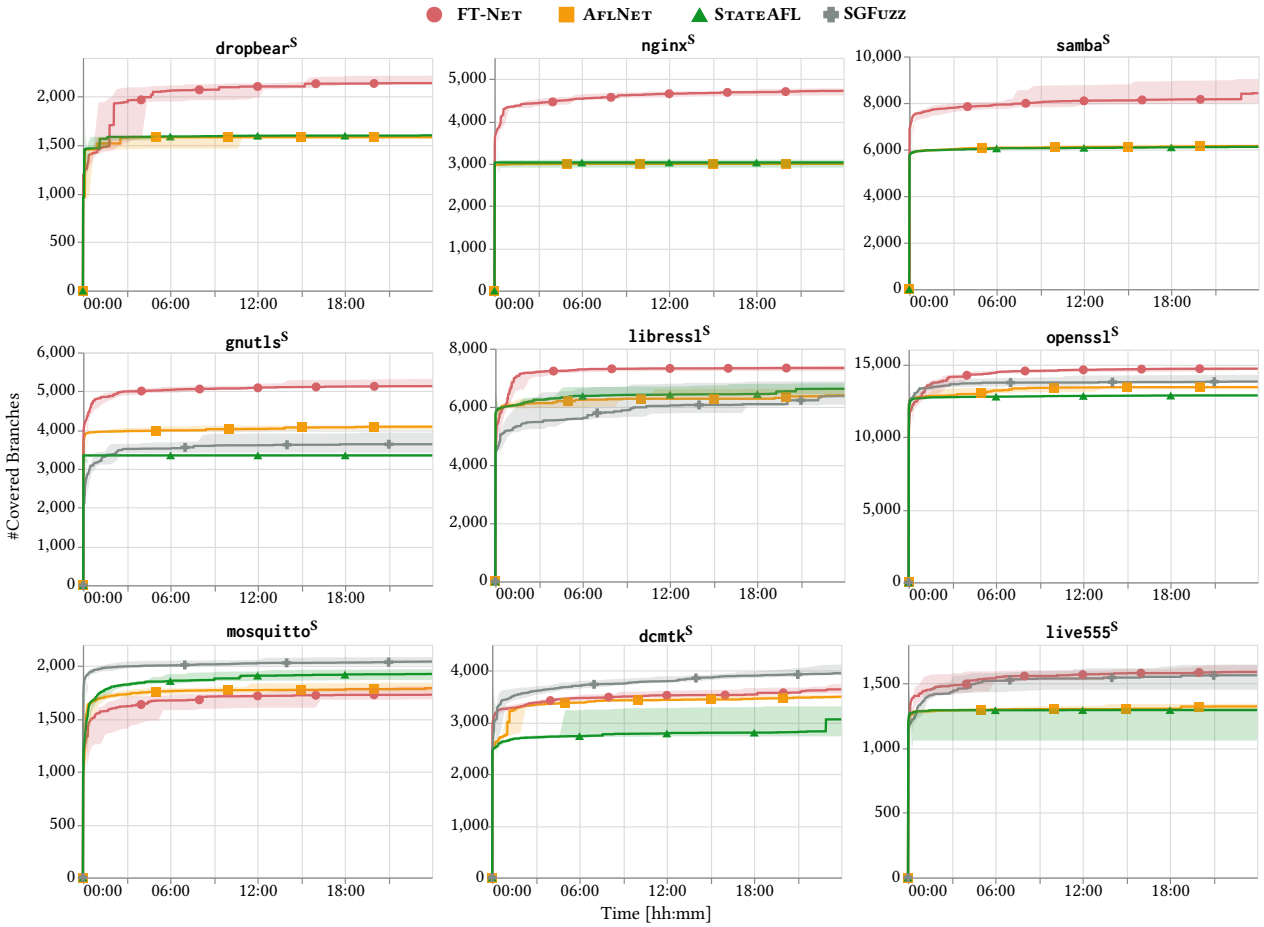

Figure 2: The coverage (in `llvm-cov` branches) various fuzzers achieve over ten 24h runs on different targets. We display the median run and 66% intervals.

and SGFuzz. Table 5 in the Appendix lists the exact configuration of each target.

Target Configuration. We compiled all targets using the compiler pass provided by the respective fuzzer. Since most targets have already been extensively tested, we deployed ASAN for each fuzzer on all fuzzing targets to detect bugs that generally would not cause a crash. Where applicable, we disabled custom allocators to avoid bugs being shadowed by ASAN's inability to determine when a memory chunk was freed.

Since all other network application fuzzers rely on replay-based techniques, i.e., traffic of a real client/server communication must be recorded and used as seed, their fuzzing efforts are impeded by sources of randomness such as session identifiers or ephemeral tokens. Notably, this limitation is shared by all publicly available network fuzzers and is not a result of our fuzzer selection. To address this shortcoming, we attempted to disable sources of randomness in all fuzzing targets. This includes the use of static session IDs as well as seeding random number generators with constant seeds. Furthermore, we removed functions typically used to reseed the PRNGs during runtime. However, it is important to note that it is impossible to encompass *all* sources of randomness simply due to the complexity of the tested targets. Furthermore, we only patched the target applications themselves and the libraries shipped alongside them. We did not patch libraries that are pulled from the system as external dependencies, since this would have been infeasible to accomplish. We provide the patches as part of our artifact at https://github.com/fuzztruction/fuzztruction-net-experiments. We stress that patching out randomness is required by the other tested tools, not Fuzztruction-Net.

For Fuzztruction-Net, we compiled the targets using a custom version of AFL++ v4.08c, as detailed in Section 4. Furthermore, we enabled laf-intel [26] via setting `AFL_LLVM_LAF_ALL`. We compiled the weird peer applications using our compiler pass, allowing us to mutate the application during execution dynamically. Note that we disabled randomness as for the other approaches to ensure a fair evaluation, even though our approach does not necessarily require this step.

Table 1: Information on targets that are part of our evaluation. A complexity (Cmpl.) of easy indicates the protocol is amenable to byte-oriented mutations, while we use hard to indicate targets using checks and/or encryption to protect communication. We also survey whether the targets offer fuzzing harnesses (H) or are even included in OSS-Fuzz' testing (OF). The applications in the Weird Peer column are the ones used by FuzzTruction-Net as a peer for the target. The targets above the line were selected for our comparative evaluation.

Target	Protocol	Version	Weird Peer	Fuzzing Support		
				Cmpl.	H	OF
dropbear^S	SSH	9925b00	dropbear^C	hard	✓	✓
dcmtk^S	DICOM	1549d8c	dcmtk^C	easy	✘	✘
gnutls^S	TLS	e840a07	gnutls^C	hard	✓	✓
libressl^S	TLS	fbb21ed	libressl^C	hard	✓	✓
live555^S	RTP	2023.06.14	live555^C	med.	✘	✘
mosquitto^S	MQTT	3923526	mosquitto^C	easy	✘	✓
nginx^S	HTTP/3	6b1bb99	ngtcp2^C	hard	✓	✓
openssl^S	TLS	7b649c7	openssh^C	hard	✓	✓
samba^S	SMB	95474d8	samba^C	hard	✓	✓
apache^S	HTTP/2	a751ae5	curl^C	hard	✓	✓
pjsip^S	SIP	12d0468	pjsip^C	easy	✓	✓
curl^C	HTTP/3	de7b3e8	nginx^S	hard	✓	✓
dropbear^C	SSH	9925b00	dropbear^S	hard	✓	✓
libressl^C	TLS	fbb21ed	libressl^S	hard	✓	✓
ngtcp2^S	HTTP/3	f3f15b6	ngtcp2^C	hard	✓	✓
openssh^C	SSH	86bdd38	dropbear^S	hard	✓	✓

Seeds. Since FuzzTruction-Net uses a mutated client or server application to produce fuzzing inputs for the SUT, it does not require seed files but another peer. For the other fuzzers, we prepared seeds by recording the network traffic between each client/server pair (same pair as used by FuzzTruction-Net). Thus, each approach had the same initial conditions for its fuzzing campaign.

Coverage. To calculate coverage, we used a version of the target compiled with LLVM's Source-based Code Coverage, which enabled us to measure the coverage for each input processed by the target being tested. LLVM-Cov specifically uses the branches in the source code as the units for coverage measurement. This approach ensures that the coverage results are independent of the machine code produced by the compiler. As a result, these findings are consistent across various compiler versions and even among binaries that have undergone different instrumentation processes.

5.2 Coverage Experiments

While fuzzing primarily aims to identify bugs, coverage is a well-established and widely used proxy metric for comparing the efficiency and effectiveness of different fuzzers [7]. For our evaluation, we selected nine different targets (cf. first half of Table 1) and fuzzed them for 24 hours. Given the stochastic nature of fuzzing, we repeated each experiment ten times to ensure reliability [25].

We encountered challenges in deploying SGFuzz on three targets: dropbear^S, samba^S, and nginx^S. For nginx^S, the primary issue was the lack of UDP support in SGFuzz. The samba^S target refused to compile when instrumented with the SGFuzz instrumentation. The reason for SGFuzz failing on dropbear^S was the lack of support for targets using execve, which is mandatory if dropbear^S is not intended to be used in one-shot mode (i.e., it terminates after processing one connection), which is also not support by SGFuzz.

Table 2: Statistical evaluation of coverage results. We use a bootstrap-based algorithm [34] to determine significance and Vargha and Delaney's $\hat{A}_{12}$ test to measure effect size [41]. If the difference is significant, we print the effect size in bold.

Target	Best Competitor	$\hat{A}_{12}$ effect size
dropbear^S	StateAFL	**+L(1.00)**
dcmtk^S	SGFuzz	**-L(0.00)**
gnutls^S	AflNet	**+L(1.00)**
libressl^S	StateAFL	**+L(0.94)**
live555^S	SGFuzz	+S(0.63)
mosquitto^S	SGFuzz	**-L(0.00)**
nginx^S	StateAFL	**+L(1.00)**
openssl^S	SGFuzz	**+L(1.00)**
samba^S	AflNet	**+L(1.00)**

The results of the coverage experiment are shown in Figure 2. Overall, our prototype FuzzTruction-Net outperforms all other fuzzers in six out of nine cases. In one of the nine cases, namely for the target live555^S, our performance is on par with SGFuzz. However, FuzzTruction-Net underperforms SGFuzz for dcmtk^S, and both SGFuzz and StateAFL showed better performance for the mosquitto^S target.

Statistical evaluation. We statistically evaluate our results in line with recommended best practices [25]. To this end, we use a bootstrap-based two-sample t-test [34] and measure effect size via Vargha's and Delaney's $\hat{A}_{12}$ test [41], comparing FuzzTruction-Net to the best other tool (based on the coverage of the median run). As shown in Table 2, the effect size is large (>0.71 [41]) in all cases with the exception of live555^S, where we observe no statistically significant difference. In two cases, dcmtk^S and mosquitto^S, we see a negative effect size, as FuzzTruction-Net performed worse than SGFuzz. Overall, the statistical evaluation confirms our intuition that FuzzTruction-Net is better on six out of nine targets, worse on two, and tied with SGFuzz on live555^S.

Next, we analyze these results in more detail, first focusing on cases where FuzzTruction-Net succeeded before then taking a closer look at the targets where FuzzTruction-Net was not the fuzzer with the best performance. We reference the challenges outlined in Section 2.2.

Targets with good performance. For six targets, FuzzTruction-Net shows excellent performance characteristics, outperforming the best competitor by at least 6% (on openssl^S) but up to 56% (nginx^S), with an average improvement of 16% over all nine targets. When analyzing the coverage achieved by various fuzzers, it becomes clear that other fuzzers encounter significant obstacles with certain targets such as samba^S, nginx^S, and dropbear^S. The reason is that these targets use ephemeral values (challenge C1), such as session identifiers, or encryption (challenge C2).

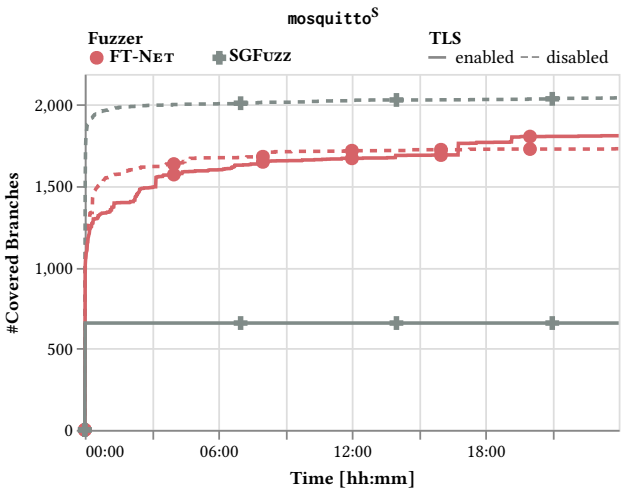

Figure 3: Coverage on `mosquitto`^{S} when enabling TLS support as typical for real-world use. We compute coverage computed only for the target, not the TLS library implementation. TLS stalls SGFuzz but not FuzzTruction-Net.

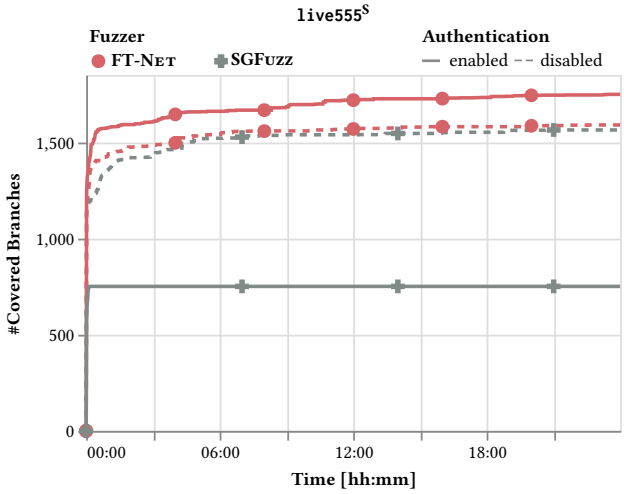

Figure 4: Coverage on `live555`^{S} with HTTP Digest Access Authentication [37] enabled. We still provide replay-based fuzzers with a recording of a successful authentication (HTTP plain text).

For the TLS libraries `gnutls`^{S}, `libressl`^{S}, and `openssl`^{S}, the other fuzzers successfully replayed the initial seed provided by us. This was only possible because we modified these targets' Pseudo Random Number Generators (PRNGs) to yield predictable outputs. However, the fuzzers still failed to initiate any new successful TLS handshakes beyond using the initial seed file. This failure occurs because any slight alteration in the seed file requires a recalculation of the message authentication codes used during the handshake (C1, C2), due to the protocol's inherent design. The situation is even more complicated for SGFuzz because of its in-process approach. Although the PRNGs produce predictable values, they are not reset by SGFuzz after processing a connection. Therefore, in subsequent fuzzing iterations, the random numbers used during the TLS handshake differ from those in the seed file (C1), leading to earlier rejection of generated messages. This limitation impedes SGFuzz's ability to explore as much error-handling code as the other fuzzers.

FuzzTruction-Net, in contrast, does not suffer from any of these limitations, leading to the observed outcome. Upon closer inspection, we found that some of the other fuzzers covered more error-handling code than FuzzTruction-Net, suggesting that using multiple fuzzers may yield a beneficial synergy.

Targets with bad performance. In the following, we analyze in detail the targets where FuzzTruction-Net underperformed our expectations.

`mosquitto`^{S}. On this target, FuzzTruction-Net falls short compared to all other tools. When investigating this behavior, we found that the characteristics of the MQTT protocol are the root cause: MQTT handles a very limited set of message types, primarily `subscribe` and `unsubscribe` messages. The MQTT protocol is straightforward and lightweight to be used on embedded devices, making it relatively easy for fuzzers to generate syntactically correct messages purely by chance. We hypothesize that it is, hence, less important to generate "good" inputs by faulting the peer than to generate as many (random) inputs as quickly as possible. Further, our data shows that more than 55% of the inputs crafted by the fuzzers and sent to the `mosquitto`^{S} were shorter than 32 bytes, indicating the simplicity of the protocol format. Such compact input sequences are ideal targets for bit-level mutational fuzzers, which excel in enumerating through small variations of input data. If MQTT used more complex mechanisms, such as session IDs that require maintaining state across messages, this might have significantly hindered the effectiveness of such basic fuzzing techniques. Another interesting effect that we observed is that SGFuzz was the only fuzzer to cover code related to data persistence. The reason for that is that SGFuzz is the only tool using an in-memory fuzzing approach, which causes `mosquitto`^{S} to persist the server data after a number of connections have been processed. This does not happen for the other fuzzers, which terminate the target after each iteration.

However, depending on which features are enabled, our approach can be beneficial even for targets with such a simple protocol specification. To showcase this claim, we enabled TLS support for `mosquitto`^{S}, which is arguably a realistic scenario, and ran the fuzzers again. As shown in Figure 3 (with full data for all fuzzers in Figure 5 in the Appendix), traditional bit-level mutation fuzzers then struggle with this target. The coverage (measured *only* for the target, excluding the TLS library implementation itself) for FuzzTruction-Net remains unimpeded, while other fuzzers are less effective and can no longer fuzz the target thoroughly.

`dcmtk`^{S}. For `dcmtk`^{S}, SGFuzz outperforms FuzzTruction-Net. Similar to the situation with `mosquitto`^{S}, the subset of the DCMTK protocol used by the target lacks complex primitives that challenge

traditional fuzzers. The main function of dcmtk^S involves receiving an x-ray image along with accompanying metadata sent by the client. Analyzing the coverage differences of FuzzTruction-Net and SGFuzz, we found that the coverage advantage primarily stems from parsing functions that, for example, convert integer or float values into strings or vice versa. For fuzzing these kinds of functions, approaches with a high throughput rate, like SGFuzz, are inherently better suited and, therefore, have an advantage.

live555^S. The third case where FuzzTruction-Net did not perform better than all state-of-the-art tools, but was on par with SGFuzz, is live555^S. Notably, we patched the target (as recommended by the authors of SGFuzz [3]) such that it uses a constant value as session IDs, such that replayed messages are still accepted by the server. In terms of source code coverage, both fuzzers covered mostly the same code. However, one notable area that neither fuzzer addressed is the server's authentication functionality. This feature was intentionally disabled for our experiment. In a real-world application, similar to TLS in the mosquitto^S scenario, authentication is likely enabled. The authentication system used by live555^S incorporates a server-selected nonce as part of HTTP authentication. Despite the simplicity, this renders traditional fuzzers ineffective as evident from Figure 4 (data for all fuzzers in Figure 6 in the Appendix). With this authentication enabled, the other fuzzers did not manage to establish a single successful authentication, in contrast to FuzzTruction-Net, even though the provided seed contained the exact HTTP plain text packets recorded during a successful authentication.

5.3 Finding New Vulnerabilities

To measure the bug-finding capability of our approach, we conducted two experiments. First, we compared the number of unique bugs found by all fuzzers based on the deduplicated crashes found during the coverage evaluation. In the second experiment, we tested FuzzTruction-Net's ability to find bugs in an additional set of targets, outlined in the bottom half of Table 1. In particular, these targets include several clients that no other network application fuzzer can test. All selected targets were evaluated using the most recent versions available at the time of writing. Interestingly, most of the targets are integrated into OSS-Fuzz and provide, in addition, fuzzing harnesses for a variety of their interfaces. Therefore, it can be assumed that most bugs found during our evaluation eluded previous fuzzing efforts.

To map found crashes to underlying bugs, we used a two-staged schema for pre-filtering: (1) Crashes were bucketed according to the last five called functions of the ASAN stack traces that are generated when a crash is found. (2) The resulting buckets were manually triaged and mapped to unique bugs.

Comparative Evaluation. We present the results of our comparison with other fuzzers in Table 3. A total of 10 unique bugs were identified across all targets. Out of these, FuzzTruction-Net successfully detected 9 bugs, missing only a single bug in live555^S. In general, the effectiveness of each fuzzer in identifying bugs aligns with its coverage performance. Targets where FuzzTruction-Net excelled in coverage also yielded a higher number of detected bugs compared to other fuzzers. The only exception is mosquitto^S, where SGFuzz achieved significantly higher coverage than FuzzTruction-Net, yet none of the fuzzers found any bugs. StateAFL, on the other hand, identified only a single bug. This limited detection capability can be attributed to frequent crashes caused by its memory allocation tracking instrumentation, which often prevented the fuzzer from making meaningful progress.

Table 3: Unique bugs found by the different fuzzers. We derive this number by first bucketing found crashes via the last five functions on the call stack in the ASAN report and then deduplicating them manually.

Target	FuzzTruction-Net	AflNet	SGFuzz	StateAFL	#Unique Bugs
dropbear^S	0	0	0	0	0
dcmtk^S	2	2	2	1	2
gnutls^S	2	0	0	0	2
libressl^S	0	0	0	0	0
live555^S	2	0	1	0	3
mosquitto^S	0	0	0	0	0
nginx^S	3	0	0	0	3
openssl^S	0	0	0	0	0
samba^S	0	0	0	0	0
Sum	9	2	3	1	10

Real-World Applicability. To test the real-world applicability of FuzzTruction-Net, we fuzzed further targets, including many client applications that other network fuzzers cannot test. A list of reported bugs found by FuzzTruction-Net is shown in Table 4. Overall, FuzzTruction-Net found bugs in a variety of security-relevant targets, including the OpenSSH client and the webservers HTTPd and Nginx (where two CVEs have been assigned). The detected bugs cover the full spectrum of typical memory safety violations, ranging from nullptr dereferences to out-of-bounds memory accesses on the heap to use-after-frees.

Case Study: Nginx. One of the bugs FuzzTruction-Net found in Nginx allows for an 8-byte out-of-bounds write in the HTTP/3 dynamic table, a critical component used within the QPACK header compression mechanism of QUIC. The HTTP/3 dynamic table stores previously sent headers to optimize header compression over QUIC, thereby reducing redundancy and enhancing the efficiency of subsequent requests. FuzzTruction-Net triggered the bug via two steps: First, the weird peer ngtcp2^C adjusted the dynamic table size to a new capacity by sending a "Set Dynamic Table Capacity" message. Then, via a "Duplicate" message, it requested to duplicate an entry in the dynamic table. Crucially, the size requested initially could not accommodate the duplication, leading to an overflow. This overflow introduced the potential for memory corruption, posing a serious security risk in the management of HTTP/3 headers in Nginx. It is noteworthy that none of the other fuzzers found this bug, as their design did not allow them to access the key generated during the TLS handshake.

6 Discussion

In the following, we discuss potential limitations of our approach and explore directions for future research in the area of fault injection in the context of network fuzzing.

Access Requirement. Ideally, we have access to both peers so that we can observe the behavior of the client and the server and

Table 4: Overview of the 23 bugs we found in different targets. All bugs have been responsible disclosed. Weird Peer indicates the program in which we induced faults to generate the input uncovering the bug. The Status column links to the CVE or bug reports, unless a bug was reported via email. All CVEs have been fixed.

Target	Weird Peer	Status	Type	Description
`nginx`^{S}	`ngtcp2`^{C}	*fixed*	Heap OOB read	`ngx_quic_parse_transport_param` uses user-controlled length value
`nginx`^{S}	`ngtcp2`^{C}	CVE-2024-32760	Heap OOB write	Duplication of an entry in the HTTP3 dynamic table, while the table is too small
`nginx`^{S}	`ngtcp2`^{C}	CVE-2024-31079	Heap UAF	During an infinite recursion, the recurringly accessed memory pool is freed
`nginx`^{S}	`ngtcp2`^{C}	CVE-2024-34161	Info. leak.	QUIC packets can cause worker processes to leak previously freed memory
`nginx`^{S}	`ngtcp2`^{C}	CVE-2024-35200	Nullptr deref	Access to a variable holding the user-agent header value is unexpectedly null
`curl`^{C}	`nginx`^{S}	*fixed*	Nullptr deref	`ngtcp2_ksl_begin` in the `ngtcp2` library tries to accesses the head of a list that is nul
`curl`^{C}	`nginx`^{S}	*fixed*	Assertion	In the `nghttp3` library, an assertion checking if enough data is remaining is triggered
`apache`^{S}	`curl`^{C}	*reported*	Heap UAF	`apr_file_write` attempts to write error log after freeing its path
`apache`^{S}	`curl`^{C}	*reported*	Assertion	While adding a key-value pair into an APR table, an assertion is triggered
`openssh`^{C}	`dropbear`^{S}	*fixed*	Heap UAF	The asynchronous callback `verify_host_key` accesses the previously freed host key
`dropbear`^{C}	`dropbear`^{S}	*fixed* (#285)	Assertion	`signkey_type_from_signature` tries to cast an invalid integer to an `enum`
`gnutls`^{S}	`gnutls`^{C}	*fixed* (#1529)	Nullptr deref	During a retry-handshake, `_gnutls_cipher_auth` dereferences a null pointer
`gnutls`^{S}	`gnutls`^{C}	*fixed* (#1534)	Nullptr deref	`_gnutls_figure_common_ciphersuite` dereferences a null ptr if a PSK cipher is used
`libressl`^{C}	`libressl`^{S}	*fixed* (#1037)	Nullptr deref	Attempting to print key material in `ssl_print_tmp_key` after it has been freed
`ngtcp2`^{S}	`ngtcp2`^{C}	*fixed* (#1529)	Assertion	The `wolfssl` library returns a TLS session using the unauthenticated `CTR` AES mode, even though the `CCM` mode was negotiated
`ngtcp2`^{S}	`ngtcp2`^{C}	*fixed* (#7406)	Heap OOB write	In `wolfSSL_read_early_data`, header bytes are written to a insufficiently sized buffe
`pjsip`^{S}	`pjsip`^{C}	*fixed*	Heap OOB read	A timer object is deleted before it expires, causing an out-of-bound read
`pjsip`^{S}	`pjsip`^{C}	*fixed*	FP Exception	Proposing a clock rate of 0 for an audio stream causes a division by zero
`pjsip`^{S}	`pjsip`^{C}	*fixed*	Heap OOB write	Dumped source address of an SDP message is copied into too small buffer
`dcmtk`^{S}	`dcmtk`^{C}	CVE-2024-34508	Nullptr deref	`checkAndProcessSTORERequest` accesses the value of an element pointing to null
`dcmtk`^{S}	`dcmtk`^{C}	CVE-2024-34509	Nullptr deref	`DIMSE_parseCmdObject` attempts to access a string that points to null
`live555`^{S}	`live555`^{C}	*reported*	Heap UAF	`sendDataOverTCP` sends MP3 bytes after freeing the `RTSPClientConnection`
`live555`^{S}	`live555`^{C}	*reported*	Heap UAF	During execution of `handleHTTPCmd_TunnelingPOST`, the processed input is freed

instrument one of them to turn it into a weird peer. As such, our current setup cannot be used to test instant messenger clients such as WhatsApp or Apple's iMessage, given that we cannot instrument the server to turn it into a weird peer. A related challenge is that our prototype implementation requires access to the weird peer's source code so that we can instrument it, which may not be possible if the software is proprietary or closed-source.

Timing-Related Vulnerabilities. Our instrumentation and the fuzzing setup (e.g., synchronization via the FuzzMediator) introduce some performance overhead. While this does not affect the testing process per se (i.e., we have successfully detected many types of memory safety violations), this overhead can affect the communication timing between the client and the server, potentially masking timing-related issues that Fuzztruction-Net might miss.

Differential and Cross-Implementation Testing. Besides detecting classic memory safety violations, our approach could also be well-suited for testing differences in behavior between different implementations of the same protocol. This aspect is particularly crucial for protocols requiring strict adherence to their specifications to maintain strong security guarantees. Furthermore, this approach could be particularly effective in uncovering implicit constraints encoded within the program code (e.g., because the specification is ambiguous). Protocols such as TLS, with implementations such as OpenSSL, GnuTLS, and LibreSSL, are particularly sensitive to deviations from their defined standards that could potentially introduce side-channel and other vulnerabilities. To implement differential testing, we could observe and compare the behavior of different implementations when interacting with a weird peer application that has been mutated using our approach. By applying cross-implementation testing, where various clients are paired with different servers, we could expose and address hidden assumptions, ensuring the protocol functions are robust and secure across diverse environments. This approach would allow us to systematically investigate and identify inconsistencies in protocol execution, revealing potential security weaknesses.

Weird Peer Configurations. Another interesting topic for future work is the comprehensive enumeration of different client and server configurations. At the moment, a particular fuzzing setup uses a fixed, user-defined configuration of the peers. While some protocol features can be unlocked by changing one of the peers, others require a manual effort (e.g., configuring different types of public keys or en-/disabling options could be beneficial when testing a TLS implementation to cover more code). Future work could also adapt Fuzztruction-Net to support multi-threaded weird peers. While supported in principle, scheduling introduced nondeterminism that needs to be addressed. That said, most clients—that usually serve as weird peer—are single-threaded.

7 Related Work

Our work combines techniques from network application fuzzing and fault injection. As we test network applications that all speak specific protocols, our work is also related to protocol testing.

Network Application Fuzzing. Closest to our work are network fuzzers, including SGFuzz [3], StateAFL [31], AflNet [33], Bleem [28], Peach [12], BooFuzz [24], Snipuzz [13], Snapfuzz [1], and Nyx-Net [35]. Beyond these tools, there is a Windows version of AflNet called NetAFL [38]. Fundamentally, all of these methods primarily use one of two strategies. First, *replay-based fuzzers* work by intercepting and replaying live network traffic, performing mutations to produce server-compatible messages that respect the underlying message format. Second, *MitM fuzzers* actively intercept and modify live (plaintext) messages without the need for pre-recorded communication. Our method differs from these paradigms by using fault injection, eliminating the dependence on pre-recorded traffic or in-transit message changes. This unique approach allows us to use either the client or the server for fuzzing by taking advantage of the peer's ability to synthesize messages on its own. This not only bypasses the complexity associated with understanding proprietary message structures or cryptographic roadblocks, but also improves the breadth and depth of our fuzzing capabilities by testing the target in a more dynamic and unpredictable way.

Fault Injection. Several fuzzers that use fault injection to test regular applications exist [4, 23, 27, 36]. For example, Fuzztruction [4] targets specific application pairs, such as in encryption or compression, to test the handling of data formats by generating and consuming data across applications. More recently, FuzzERR [36] strategically inserts faults into API calls to evaluate the robustness of error handling. These techniques have conceptual similarities with our approach. However, they are narrowly tailored to the interaction of data formats and cannot account for the complexity of network applications. In contrast, our tool is specifically designed to control and exploit the nuanced dynamics of network protocols, enabling effective testing of network applications beyond the scope of existing domain-specific fuzzers.

Protocol Testing. Given that all of our targets use specific protocols, our work relates to the area of *protocol testing* [16, 17, 39], which is essentially about verifying that protocols conform to their specifications, maintain interoperability, and are robust against various types of attacks [22]. For example, TLS-Attacker [39] is a flexible framework for evaluating the security features of TLS implementations based on a manual implementation of the TLS specification. Crucially, our goal is *not* to test the protocol itself, i.e., the specification, or to find implementation flaws in the sense of protocol deviations. Instead, we focus on injecting faults in a protocol-agnostic way to test the robustness of network applications. Fuzztruction-Net cannot detect specification errors or implementation deviations unless they manifest themselves in memory corruption.

State Machine Inference. Orthogonal research on automata learning [8–11, 33, 40] has focused on inferring the state machine of network protocols based on observed behaviors, a generally hard task. These methods treat the application as a black box oracle and infer the structure of the state machine by analyzing the application's responses to various inputs. In contrast, our approach is agnostic of the state machine, and we rely on the instrumentation of one of the communication peers. Fuzztruction-Net could be used together with existing state machine learning algorithms to infer the current state or state transitions more efficiently.

8 Conclusion

Despite being a critical security boundary, network applications have enjoyed little attention in terms of fuzz testing. In this work, we have analyzed challenges preventing fuzzers from effectively and efficiently doing so, then proposed a fault injection-based approach to overcome them. Injecting faults allows us to strike a balance between sending unexpected input that stresses the target and adhering to the input structure to exercise deeper functionality. Our evaluation shows that fault injection is a promising approach, outperforming all state-of-the-art fuzzers regarding coverage and found bugs. Notably, our approach is the only network fuzzer to test both clients and servers.

Acknowledgments

We thank our anonymous reviewers for their valuable feedback. For their feedback on an earlier draft, we thank Tobias Scharnowski and Addison Crump. We would also like to express our gratitude to the team maintaining DCMTK and the F5 security incident response and development team at NGINX, particularly Jordan Zebor. Their quick and outgoing response during our responsible disclosure process and their thoughtful feedback on how to test their products have significantly helped us.

This work was funded by the European Research Council (ERC) under the consolidator grant RS^3 (101045669) and the German Federal Ministry of Education and Research (BMBF) under the grant CPSec (16KIS1899).

References

[1] Anastasios Andronidis and Cristian Cadar. Snapfuzz: High-throughput fuzzing of network applications. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2022.

[2] 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.

[3] Jinsheng Ba, Marcel Böhme, Zahra Mirzamomen, and Abhik Roychoudhury. Stateful Greybox Fuzzing. In *USENIX Security Symposium*, 2022.

[4] Nils Bars, Moritz Schloegel, Tobias Scharnowski, Nico Schiller, and Thorsten Holz. Fuzztruction: Using Fault Injection-based Fuzzing to Leverage Implicit Domain Knowledge. In *USENIX Security Symposium*, 2023.

[5] Sergej Bratus, Michael E. Locasto, Meredith L. Patterson, Len Sassaman, and Anna Shubina. Exploit Programming: From Buffer Overflows to "Weird Machines"' and Theory of Computation. *Usenix; Login*, 2011.

[6] Marcel Böhme, Valentin J. M. Manès, and Sang Kil Cha. Boosting Fuzzer Efficiency: An Information Theoretic Perspective. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2020.

[7] Marcel Böhme, László Szekeres, and Jonathan Metzman. On the Reliability of Coverage-Based Fuzzer Benchmarking. In *ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2022.

[8] Merlin Chlosta, David Rupprecht, and Thorsten Holz. On the Challenges of Automata Reconstruction in LTE Networks. In *ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, 2021.

[9] Lesly-Ann Daniel, Erik Poll, and Joeri de Ruiter. Inferring OpenVPN State Machines Using Protocol State Fuzzing. In *IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, 2018.

[10] Cristian Daniele, Seyed Behnam Andarzian, and Erik Poll. Fuzzers for Stateful Systems: Survey and Research Directions. *ACM Computing Surveys (CSUR)*, 56(9), 2024.

[11] Joeri De Ruiter and Erik Poll. Protocol State Fuzzing of TLS Implementations. In *USENIX Security Symposium*, 2015.

[12] Michael Eddington. Peach Fuzzer: Discover Unknown Vulnerabilities. https://peachtech.gitlab.io/peach-fuzzer-community/, 2004.

[13] Xiaotao Feng, Ruoxi Sun, Xiaogang Zhu, Minhui Xue, Sheng Wen, Dongxi Liu, Surya Nepal, and Yang Xiang. Snipuzz: Black-box Fuzzing of IoT Firmware via Message Snippet Inference. In *ACM Conference on Computer and Communications Security (CCS)*, 2021.

[14] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++ : Combining Incremental Steps of Fuzzing Research. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2020.

[15] Andrea Fioraldi, Dominik Christian Maier, Dongjia Zhang, and Davide Balzarotti. LibAFL: A Framework to Build Modular and Reusable Fuzzers. In *ACM Conference on Computer and Communications Security (CCS)*, 2022.

[16] Paul Fiterau-Brostean, Bengt Jonsson, Robert Merget, Joeri de Ruiter, Konstantinos Sagonas, and Juraj Somorovsky. Analysis of DTLS Implementations Using Protocol State Fuzzing. In *USENIX Security Symposium*, 2020.

[17] Paul Fiterau-Brostean, Bengt Jonsson, Robert Merget, Joeri De Ruiter, Konstantinos Sagonas, and Juraj Somorovsky. Analysis of DTLS Implementations Using Protocol State Fuzzing. In *USENIX Security Symposium*, 2020.

[18] Google. Honggfuzz.

[19] Google. OSS-Fuzz: Continuous Fuzzing for Open Source Software.

[20] Google. Fuzzer-Test-Suite, 2016.

[21] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. Magma: A Ground-Truth Fuzzing Benchmark. *ACM on Measurement and Analysis of Computing Systems (POMACS)*, 4(3):49:1–49:29, 2020.

[22] Robert M Hierons, Kirill Bogdanov, Jonathan P Bowen, Rance Cleaveland, John Derrick, Jeremy Dick, Marian Gheorghe, Mark Harman, Kalpesh Kapoor, Paul Krause, et al. Using Formal Specifications to Support Testing. *ACM Computing Surveys (CSUR)*, 41(2), 2009.

[23] Zu-Ming Jiang, Jia-Ju Bai, Kangjie Lu, and Shi-Min Hu. Fuzzing Error Handling Code using Context-Sensitive Software Fault Injection. In *USENIX Security Symposium*, 2020.

[24] Joshua Pereyda et al. boofuzz: Network Protocol Fuzzing for Humans. https://github.com/jtpereyda/boofuzz.

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

[26] lafintel. laf-intel - Circumventing Fuzzing Roadblocks with Compiler Transformations. https://lafintel.wordpress.com.

[27] Peiyu Liu, Shouling Ji, Xuhong Zhang, Qinming Dai, Kangjie Lu, Lirong Fu, Wenzhi Chen, Peng Cheng, Wenhai Wang, and Raheem Beyah. IFIZZ: Deep-State and Efficient Fault-Scenario Generation to Test IoT Firmware. In *ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2021.

[28] Zhengxiong Luo, Junze Yu, Feilong Zuo, Jianzhong Liu, Yu Jiang, Ting Chen, Abhik Roychoudhury, and Jiaguang Sun. Bleem: Packet Sequence Oriented Fuzzing for Protocol Implementations. In *USENIX Security Symposium*, 2023.

[29] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. MOPT: Optimized Mutation Scheduling for Fuzzers. In *USENIX Security Symposium*, 2019.

[30] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. FuzzBench: An Open Fuzzer Benchmarking Platform and Service. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2021.

[31] Roberto Natella. StateAFL: Greybox Fuzzing for Stateful Network Servers. *Empirical Software Engineering*, 27(7):191, 2022.

[32] Roberto Natella and Van-Thuan Pham. ProFuzzBench: A Benchmark for Stateful Protocol Fuzzing. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2021.

[33] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. AFLNet: A Greybox Fuzzer for Network Protocols. In *IEEE International Conference on Software Testing, Validation and Verification (ICST)*, 2020.

[34] Moritz Schloegel, Nils Bars, Nico Schiller, Lukas Bernhard, Tobias Scharnowski, Addison Crump, Arash Ale-Ebrahim, Nicolai Bissantz, Marius Muench, and Thorsten Holz. SoK: Prudent Evaluation Practices for Fuzzing. In *IEEE Symposium on Security and Privacy (S&P)*, 2024.

[35] Sergej Schumilo, Cornelius Aschermann, Andrea Jemmett, Ali Abbasi, and Thorsten Holz. Nyx-Net: Network Fuzzing with Incremental Snapshots. In *European Conference on Computer Systems (EuroSys)*, 2022.

[36] Shashank Sharma, Sai Ritvik Tanksalkar, Sourag Cherupattamoolayil, and Aravind Machiry. Fuzzing API Error Handling Behaviors using Coverage Guided Fault Injection. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2024.

[37] Rifaat Shekh-Yusef. HTTP Digest Access Authentication, 2015.

[38] M. Shudrak. NetAFL: WinAFL Patch. https://github.com/intelpt/winafl-intelpt, 2018.

[39] Juraj Somorovsky. Systematic Fuzzing and Testing of TLS Libraries. In *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[40] Bernhard Steffen, Falk Howar, and Maik Merten. *Introduction to Active Automata Learning from a Practical Perspective*. 2011.

[41] András Vargha and Harold D Delaney. A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000.

[42] 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.

[43] Michał Zalewski. American Fuzzy Lop. https://lcamtuf.coredump.cx/afl/, 2013.

[44] Bodong Zhao, Zheming Li, Shisong Qin, Zheyu Ma, Ming Yuan, Wenyu Zhu, Zhihong Tian, and Chao Zhang. StateFuzz: System Call-Based State-Aware Linux Driver Fuzzing. In *USENIX Security Symposium*, 2022.

[45] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. Fuzzing: A Survey for Roadmap. *ACM Computing Surveys (CSUR)*, 54(11s):1–36, 2022.

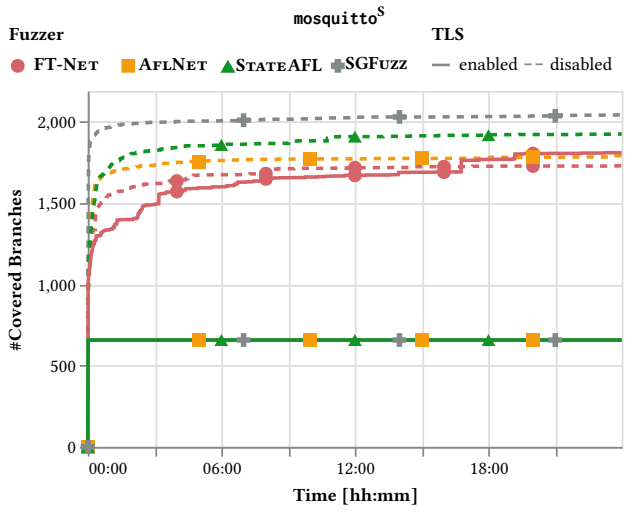

Figure 5: Coverage results for all fuzzers when enabling TLS support, as is often the case in practice. We compute coverage only for the target but not the TLS implementation.

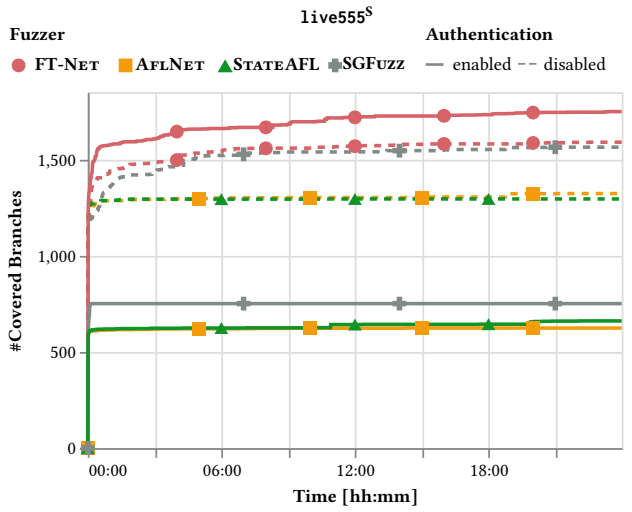

Figure 6: Coverage results when enabling HTTP Digest Access Authentication [37] for live555^S. We provided all fuzzers with a recording of one successful authentication (HTTP plain text).

Table 5: Target configurations used during the evaluation.

Target	Configuration
dropbear^S	`dropbear -p <ip> -a -F -r ed25519.key -E -B`
dropbear^C	`dbclient user@<ip> -yy -i ed25519.key pwd`
dcmtk^S	`dcmrecv --config-file storescp.cfg Default -d <port`
dcmtk^C	`dcmsend -aet YOU_AET -aec DCM4CHEE \` `-d <host> <port> image-00000.dcm`

gnutls^S	`gnutls-serv -a -d 1000 --earlydata \`
	`--x509certfile=cert.pem \`
	`--x509keyfile=key.pem -b -p <port>`

gnutls^C	`gnutls-cli <ip> --rehandshake --starttls \` `-d <port> -b`

libressl^S	`openssl s_server -key key.pem \`
	`-cert cert.pem -accept <port> \`
	`-www -naccept 1`

libressl^C	`openssl s_client -connect <ip> -status`
live555^S	`testOnDemandRTSPServer <port>`
live555^C	`testRTSPClient rtsp://<ip>/mp3AudioTest`
mosquitto^S	`mosquitto -p <port>`

mosquitto^C	`mosquitto_pub -h <ip> -p <port> -t TOPIC -m m \`
	`--will-topic wt --will-retain --will-payload wp`
	`--insecure --will-qos 1 --repeat 10 -r`

nginx^S	`nginx -c nginx.conf`
openssl^S	`openssl s_server -key key.pem -cert cert.pem \` `-accept 44330 -www -naccept 1`
openssl^C	`openssl s_client -connect localhost:44330 -reconnect`
samba^S	`smbd -s smb.conf -F -i`
samba^C	`smbclient -p <port> -L //<ip>`
apache^S	`httpd -X -f httpd.conf`
pjsip^S	`siprtp -p 5080 -r 4010 --auto-quit --call-report \` `-i <ip> <remote-ip>`
pjsip^C	`siprtp -p <port> -r 4000 -i <ip>`
curl^C	`curl -v --insecure --parallel <ip>`
ngtcp2^S	`wsslserver <ip> <port> server.key server.cert`
ngtcp2^C	`wsslclient --exit-on-all-streams-close <ip> <port>`
openssh^S	`sshd -f sshd_config -4 -D -d -r -p <port>`

openssh^C	`ssh user@<ip> -p<port> -o StrictHostKeyChecking=no \`
	`-i rsa.key -g -A -R 127.0.0.1:18888 \`
	`-R 127.0.0.1:18889 -vvvvv -X pwd`

A Target Configuration

To enable reproducibility, we list the exact commandline parameters for our targets in Table 5.

B Mosquitto and Live555

When enabling TLS support for mosquitto^S, only FuzzTruction-Net manages to effectively explore it. In Figure 3, we compared FuzzTruction-Net only to SGFuzz for readability reasons, and we provide the full data in Figure 5. We observe similar results for live555^S when using HTTP basic authentication, with the comparison of FuzzTruction-Net and SGFuzz in Figure 4 and the full data in Figure 6.