A Survey of Network Protocol Fuzzing: Model, Techniques and Directions

Shihao Jiang[b], Yu Zhang[a], Junqiang Li[a], Hongfang Yu[a], Long Luo[a], Gang Sun[a,*]

[a]*School of Information and Communication Engineering, University of Electronic Science and Technology of China, Chengdu 611731, China*
[b]*School of Computer Science and Engineering, University of Electronic Science and Technology of China, Chengdu 611731, China*

Abstract

As one of the most successful and effective software testing techniques in recent years, fuzz testing has uncovered numerous bugs and vulnerabilities in modern software, including network protocol software. In contrast to other fuzzing targets, network protocol software exhibits its distinct characteristics and challenges, introducing a plethora of research questions that need to be addressed in the design and implementation of network protocol fuzzers. While some research work has evaluated and systematized the knowledge of general fuzzing techniques at a high level, there is a lack of similar analysis and summarization for fuzzing research specific to network protocols. This paper offers a comprehensive exposition of network protocol software's fuzzing-related features and conducts a systematic review of some representative advancements in network protocol fuzzing since its inception. We summarize state-of-the-art strategies and solutions in various aspects, propose a unified protocol fuzzing process model, and introduce the techniques involved in each stage of the model. At the same time, this paper also summarizes the promising research directions in the landscape of protocol fuzzing to foster exploration within the community for more efficient and intelligent modern network protocol fuzzing techniques.

Keywords: Fuzzing, Network protocol, Fuzzing process model, Systematic review

1. Introduction

Network protocol is a set of specifications that governs communication between devices over a computer network. It defines the format, sequence, and error handling of messages exchanged on the network and is critical to enabling mutual communication between devices in network infrastructures (Hermann et al., 1995). Based on these standardized specifications, network protocol software is implemented, which can complete complex functions such as connection establishment, data analysis, and error detection during computer network communication and data exchange. During execution, the network protocol software will expose its network interface and handle normal, malformed, or even malicious communication traffic according to the rules specified by the protocol (Luo et al., 2023). Unfortunately, due to the diversity of requirements, network protocol software is often complex and difficult to implement completely correctly. Moreover, a significant portion of network protocol software is implemented based on C/C++, which is known to be memory-unsafe. Consequently, widespread vulnerabilities have posed serious security issues to the network protocol software stack. For instance, the infamous Heartbleed (CVE-2014-0160, 2014) security vulnerability affecting thousands of servers around the world is caused by the incorrect handling of length parameter boundary checks in OpenSSL (open source software that implements the SSL protocol). Research indicates that the prevalence of security vulnerabilities in network protocol software continues to exhibit rapid proliferation (Natella, 2022), thereby intensifying the demand within the security community for advanced techniques in the efficient analysis of protocol software security.

Fuzz testing (Fuzzing) is one of the most efficient techniques for software security analysis, gaining widespread recognition in both academia and the industry (Serebryany, 2017; Zhu et al., 2022). Presently, Fuzzing has evolved to proficiently detect real-world software vulnerabilities among a diverse array of targets, including file processing software (Shi et al., 2023; You et al., 2019), network protocol software (Luo et al., 2023; Pham et al., 2020), Internet of Things (IoT) firmware (Chen et al., 2018b; Feng et al., 2021a), operating system kernels (Kim et al., 2020; Corina et al., 2017; Bulekov et al., 2023), deep learning frameworks (Wei et al., 2022; Deng et al., 2022), libraries (Zhang et al., 2021; Babić et al., 2019), and hypervisors (Liu et al., 2023; Pan et al., 2021). Fuzzing's core concept is generating a large number of test cases as input to the program under test (PUT) through automatic or semi-automatic means, all the while employing monitors to scrutinize the PUT's state for anomalies. Essentially, fuzzing aspires to automatically explore those particular inputs within the vast input space of the PUT that could potentially trigger vulnerabilities (Oehlert, 2005). As fuzzing techniques advance, an increasing number of researchers are incorporating the characteristics of the PUT into the design of intelligent and efficient fuzzing strategies (e.g., coverage-based greybox fuzzing), aiming to alleviate the considerable time and computational resource costs associated with

---

Table 1

List of contributions by survey papers. A ✓ denotes that the paper has a detailed summary or discussion on the topic, and a ✓̷ denotes that the paper has a high-level discussion of the topic, but ignores the characteristics of the network protocol to varying degrees.

Papers	Fuzzing fundamentals	Protocol fuzzing terminology	Fuzzer categorization	Unique challenges in protocol fuzzing	Protocol fuzzing process model	Review of key fuzzers
(Liang et al., 2018)	✓	✓̷	✓	✗	✓̷	✓
(Manès et al., 2019)	✓	✓̷	✓	✗	✓̷	✓
(Zhu et al., 2022)	✓	✓̷	✗	✓̷	✓̷	✓
(Munea et al., 2016)	✓̷	✗	✓̷	✗	✗	✓̷
(Daniele et al., 2023)	✓	✓̷	✓	✓̷	✓̷	✓
This survey	✓	✓	✓	✓	✓	✓

entirely random testing.

Since the inception of fuzzing, network protocols have consistently remained a mainstream testing target. Distinguished from other PUTs, network protocols possess unique characteristics, such as statefulness and highly structured input (Hess et al., 1992). These characteristics pose unique challenges to fuzzing. Traditional fuzzing techniques do not identify states and typically remain at a shallow level within the stateful protocol software, unable to find deep stateful vulnerabilities (Chen et al., 2018a). Furthermore, due to the highly structured nature of network protocol inputs, traditional test case generation methods can disrupt their structural integrity, resulting in a multitude of invalid inputs and reduced testing efficiency (further details will be discussed in Section 2.1). To tackle the aforementioned challenges, researchers have introduced various advanced protocol fuzzers. Unfortunately, while there are currently many papers reviewing and summarizing fuzzing techniques, few research specifically dedicated to systematizing the domain of network protocol fuzz testing. Furthermore, prior work has summarized general process models or algorithm-based descriptions of fuzzing, but these highly abstract process models overlook the uniqueness of network protocols as testing targets. We believe it is imperative to systematically review and consolidate the most cutting-edge network protocol fuzzing techniques.

In this paper, we comprehensively review high-quality literature and significant fuzzers in the field of network protocol fuzzing. We delve into the primary challenges that fuzzing encounters when applied to network protocol software and propose a generic process model in Section 2. Then we follow the four stages in the model: protocol syntax acquisition and modeling (Section 3), test case generation (Section 4), test execution and monitoring (Section 5), and feedback information acquisition and utilization (Section 6) to analyze as well as summarize the focal points of state-of-the-art research efforts. Furthermore, we put forth promising future research directions to stimulate the continued advancement of protocol fuzzing (Section 7).

Related Works. There are many survey and review papers on fuzzing as this field of research flourishes. However, to the best of our knowledge, there is a notable scarcity of systematic reviews specifically dedicated to network protocol software fuzzing techniques within the scope of our literature selection.

(Liang et al., 2018) reviews a range of state-of-the-art fuzzing techniques, summarized the process of general fuzz testing, and further investigated and classified several widely used fuzzing tools. (Manès et al., 2019) aims to unify the fuzzing field from a high-level perspective. They introduce a series of rigorously defined fuzzing taxonomies and comprehensively describe the process of general fuzzing using algorithms. (Zhu et al., 2022) systematically analyzes the three gaps faced by modern fuzzing techniques from a unique perspective. It consolidates research progress in the field of fuzz testing into solutions addressing these three gaps: reducing the input space, building fuzzing theory, and achieving automatic execution. (Munea et al., 2016) initial attempt at surveying the field of protocol fuzzing, providing a summary of fuzzing terminology, and reviewing representative protocol fuzzers. However, it lacks a comprehensive overview and modeling of the protocol fuzzing process. Additionally, it lacks specificity in summarizing taxonomies related to protocol testing scenarios. (Daniele et al., 2023) provides a detailed classification and separate summaries of fuzzers targeting stateful systems. It covers a wide range of state-of-the-art fuzzers from recent years. Table 1 summarizes the contributions of these existing works.

Contributions. The absence of focus on network protocol fuzzing in these surveys has motivated us to engage in a comprehensive and in-depth discussion and summary in this paper. In summary, the contributions of our research paper are as follows:

- We analyze four unique challenges posed by network protocol software for the implementation of fuzzing. It's noteworthy that these challenges are highly specific to the characteristics of the protocol software itself, differing from the general challenges discussed at a higher level in existing surveys. We combine mathematical definitions with examples of protocol software to describe the characteristics of protocols and their corresponding challenges.

- We observe distinctions in the design and fuzzing strategies between fuzzers targeting protocol software and general fuzzers. We propose a new classification criterion for protocol fuzzers and categorize representative protocol fuzzers based on our criteria.

- Furthermore, we present a generic process model for protocol fuzz testing. In contrast to the unified fuzzing process summarized in existing work, our process model fully

reflects the unique criteria for designing fuzzers for protocol software and distills numerous solutions proposed in recent years to address protocol-specific challenges. Moreover, we delve into each execution stage of the process model, summarizing the latest fuzzing strategies tailored for specific stages while providing both a high-level process model architecture and low-level process model details.

- Lastly, we discuss future promising research directions in the hope of inspiring new advancements in the field of protocol fuzzing.

2. Overview

The foundational concept of fuzzing can be traced back to 1990 when Miller et al. subjected Unix programs to random input testing. Approximately 24% tested programs exhibited anomalous crashes when feeding a stream of randomly generated characters, leading to the terminology "fuzz" for this method. The fundamental idea behind fuzzing is to employ an extensive array of random "fuzzy inputs" to encompass the input space of the target program, thereby identifying defects and vulnerabilities (Miller et al., 1990).

This chapter introduces in detail the particularities and challenges of network protocol fuzzing compared with traditional software under test. Additionally, we abstract the network protocol fuzzing process, presenting a unified process model encompassing four phases: protocol syntax acquisition and modeling, test case generation, test execution and monitoring, and feedback information acquisition and utilization. Lastly, we introduce the classification of traditional fuzzers, proposing a categorization method more tailored to protocol fuzzers.

2.1. Distinctive attributes and challenges inherent in network protocol fuzzing

Unlike other fuzzing targets, network protocol software has its unique characteristics, leading to conventional fuzzers ineffective or inefficient when applied to protocol software (Zuo et al., 2022). We summarize its distinctiveness as reliance on network links, statefulness, the highly structured nature of inputs, and non-uniformity.

Reliance On Network Links. The input for network protocol software is the network traffic, resulting in a tightly coupled relationship between the operation and implementation of protocol software and the interfaces of network functionalities. In more detail, the network traffic processed by network protocols arrives as a sequential sequence over time, with data packets constituting the smallest units within this traffic. Hence, when testing network protocols, unlike the input for traditional software (such as simple command-line parameters or file inputs), a fuzzer needs to construct a data packet and transmit it to the protocol software through a network interface. The dependency on network links necessitates that the protocol fuzzer possesses the capability to utilize standard network interfaces (such as TCP/IP or UDP/IP) for sending and receiving network packets. Most contemporary protocol fuzzers are equipped with the

capability to send and receive network packets. It is noteworthy that recent works have employed advanced approaches to overcome these constraints, as elucidated in detail in section 5.2.
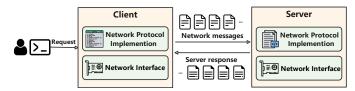

Figure 1: Protocol Software Interaction Process.

Based on these characteristics, the communication topology between the protocol fuzzer and the target closely resembles the server-client structure commonly found in network applications. During the interaction process of protocol software, the client receives user commands or receive a request from the upper level software stack, constructs corresponding data packets based on the protocol implementation, and transmits them to the server through the network link. After receiving client requests transmitted via the network interface, the server undertakes the corresponding operations and sends a response, as illustrated in Figure 1. When conducting fuzzing on server-side protocol software, the fuzzer assumes the role of the client, constructing test cases through a series of components such as protocol syntax models, fuzzing strategies, and packet generators. Subsequently, the fuzzer, acting as the client, transmits the test cases to the server in the form of network packets, as depicted in Figure 2. Similarly, when fuzzing client-side protocol software, the fuzzer takes on the role of the server.
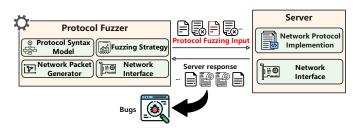

Figure 2: Protocol Fuzzing Process (for Server Side).

Statefulness. Network protocol software is typically stateful. The protocol software performs corresponding operations by identifying different states during the communication process, thereby realizing complex interactions. Specifically, the state of a network protocol defines a set $\mathbf{S} = \{I, P, O\}$, where I denotes the valid input space, O denotes the output space, and P denotes the processing logic for inputs in the current state. Stateful protocols are in a deterministic state at any given moment throughout the communication process and change state when a specific message is received. Furthermore, the same input may result in different program behaviors when occurring in different software states. A commonly employed method for describing states and their transition logic is the finite state machine.

Taking the protocol illustrated in Figure 3 as an example, from a state perspective, the protocol can be abstracted as a

collection of states S and the associated state transition logic T, where $S = \{S_0, S_1, ..., S_n\}$ and $T = \{C_{01}, C_{12}..., C_{xy}\}$. S_i denotes the different protocol states, and C_{xy} denotes the program inputs that lead to a transition from state S_x to state S_y. For example, the protocol program shown in the figure is initially in the S_0 state and enters the S_1 state after receiving packet C_{01}
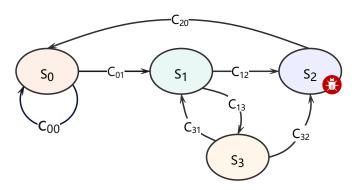

Figure 3: State Machine.

The statefulness of the protocol directly influences the design strategy of the fuzzer. It dictates that the input interacting with the protocol must be a sequence of messages with a strict chronological order (i.e., an input unit comprising multiple messages). Stateless fuzzers, in a single fuzzing iteration, provide only a single message to the protocol. This limitation confines the testing to the initial state of the protocol, resulting in highly inefficient testing.

To fuzz the vast state space of network protocol software, fuzzers need to construct message sequences meticulously. It begins by transmitting relevant prefix messages to transition the protocol software to the target state. Subsequently, randomly crafted data packets are sent to fuzz the target state. Figure 4 illustrates this process, with the server reaching state S_2 guided by the prefix message sequence. In this state, the server receives the message P, successfully triggering a stateful bug.
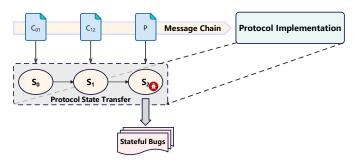

Figure 4: Stateful Bugs.

Highly Structured Input. Network protocol messages are highly structured. Messages can be partitioned into bit or byte fields with strict grammar constraints, each with a precisely defined type and range of valid values. If fundamental grammar constraints are violated, such as the absence of the expected fixed bytes in a checksum field, the protocol software

will discard the packet and terminate the connection. This outcome renders fuzzers incapable of exploring deep-seated program vulnerabilities. Therefore, fuzzers that are sensitive to input structure often have better performance in protocol fuzzing. For instance, random bit-level mutations, as employed in AFL (Zalewski, 2017), severely disrupts the message structure of a protocol, generating a plethora of invalid test inputs (Luo et al., 2020).

Non-uniformity. There are various types of network protocols, and different protocols lack uniformity in message syntax and state machines. This non-uniformity constrains the generality of protocol fuzzers, as distinct protocols may have entirely different message syntaxes and internal state machines. Existing solutions typically require additional processing to ensure coverage of protocols that can be fuzzed. For instance, the popular protocol fuzzer Peach (Eddington, 2004) establishes generality through labor-intensive efforts, relying on manually extracted protocol specifications. AFLNet (Pham et al., 2020) is capable of fuzzing general protocol programs, requiring users to manually add codes, extract response codes from the protocol's reply messages, and write corresponding response message processing scripts.

2.2. *Unified process model for network protocol fuzzing*

Faced with the aforementioned challenges, researchers in the security community have proposed a series of fuzzing methods. This section outlines our modeling of a unified process model for various network protocol fuzzing approaches. As depicted in Figure 5, the general process of protocol fuzzing comprises four stages: protocol syntax acquisition and modeling, test case generation, test execution and monitoring, and feedback information acquisition and utilization. Each stage in the model encapsulates key principles of protocol fuzzing techniques.

1) Protocol Syntax Acquisition and Modeling. This stage is a critical step before the formal initiation of fuzzing, highlighting the highly structured message syntax and stateful characteristics of the protocol. As indicated by I in Figure 5, this stage constructs strict constraints on the protocol input space through information sources such as network traffic, protocol specification documents, or protocol source code. We abstract these constraints as **protocol message syntax** and **protocol state syntax**. This stage is intricately connected with other components of modern protocol fuzzers and forms the foundation supporting the entire protocol fuzzing process. Importantly, this stage can run in parallel with the main fuzzing process, dynamically updating as the fuzzing progresses. Various approaches to syntax acquisition and modeling are detailed in section 3.

2) Test Case Generation. This stage focuses on generating specific inputs for the PUT. As depicted by II in Figure 5, based on the acquired protocol syntax model and input generation strategy, this stage determines the specific test cases to be provided to the PUT in the fuzzing iteration. Different
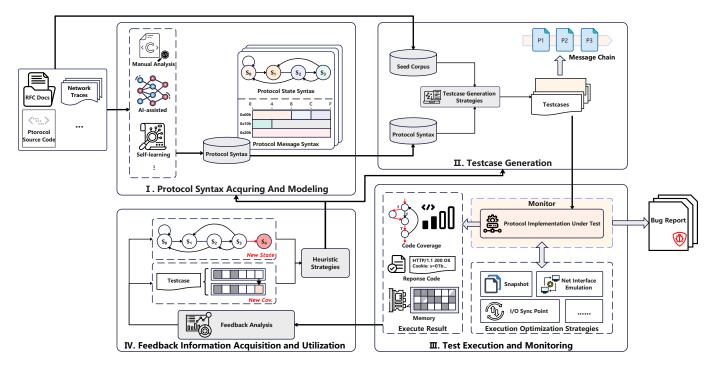
Figure 5: Unified Process Model for Network Protocol Fuzzing.

generation strategies include syntax-based strategies, mutation-based strategies, and the Fuzzer-In-The-Middle (FITM) strategy, which will be discussed in detail in section 4.

3) Test Execution and Monitor. This stage involves injecting and executing the generated test cases and monitoring the protocol program's behavior during execution. As depicted by III in Figure 5, due to the vast state space of the protocol, testing deep protocol states requires sending multiple prefix states, incurring significant time overhead. Additionally, the asynchronous nature of network communication brings various time delays, making network sockets significantly slower than traditional file reading. This poses a considerable performance overhead to the network protocol fuzzers and has become a focal point for many research efforts (Natella, 2022; Schumilo et al., 2022; Li et al., 2022; Andronidis, 2022). Techniques such as virtual machine-level or process-level snapshots, network interface emulation, and I/O synchronization points have elevated the efficiency and throughput of protocol fuzzing to a new scale. These optimizations will be discussed in detail in section 5.

4) Feedback Information Acquisition and Utilization. This stage includes the acquisition of feedback information and the utilization of feedback information to guide fuzzing. As depicted by IV in Figure 5, advanced smart fuzzers are often capable of obtaining feedback information generated during the execution of the target to track the effects of inputs and dynamically guide the entire fuzzing process. The widespread adoption and considerable success of coverage-based greybox fuzzing (CGF) have had a profound impact on fuzzing techniques, demonstrating the effectiveness of coverage information feedback (Natella,

2022; Schumilo et al., 2022; Zalewski, 2017). In section 6, we summarize the types of feedback information available in the field of protocol fuzzing, including coverage, response codes, state variables, memory, and methods for leveraging this feedback information when fuzzing network protocols.

We argue that the current state-of-the-art protocol fuzzing techniques involve innovation and improvement across the four aforementioned components. This article, based on this foundation, categorizes and compares representative protocol fuzzing techniques, highlighting their advancements in different stages of the unified process model, as shown in Table 2. In many cases, improvements in these four stages are orthogonal. For instance, while proposing more efficient test case generation techniques, further optimization of the execution speed of network protocol fuzzing can be considered.

2.3. Categorization of Protocol Fuzzers

The commonly used classification method for fuzzers is based on the amount of valuable information that can be captured during the execution process (Li et al., 2018; Yun et al., 2022; Zhu et al., 2022; Manès et al., 2019). Following this approach, fuzzers are categorized into three types: blackbox, whitebox, and greybox. The following provides detailed explanations of these three types of fuzzers:

Blackbox fuzzer. Blackbox fuzzers only consider the input and output of the target program, treating it as a completely closed black box and conducting fuzzing without knowledge of the internal implementation details (Godefroid, 2007). It explores vulnerabilities in the program by randomly mutating given initial seeds (e.g., file formats) or generating inputs based

Table 2

Summary of representative protocol fuzzers and their methodology. This paper distinguishes the key methodological contributions of different fuzzers according to the four execution stages of the general protocol fuzzing process.

general: This fuzzer can test generic protocols rather than being tailored to a specific one. **SIP:** Session Initiation Protocol. **FTP:** File Transfer Protocol. **WEBAPP:** Web Application. **SEC:** Security protocols. **TLS:** Transport Layer Security Protocol. **ICS:** Industrial Control System Protocol. **IoTD:** Internet of Thing Devices. **DTLS:** Datagram Transport Layer Security. **PAV:** Protocols in Autonomous Vehicles.

RT Info.: Runtime Information. ●: Blackbox Fuzzer. ◐: Greybox Fuzzer.

M: Message Syntax, representing that the fuzzer needs to manually obtain or be able to self-learn the protocol message format. **S:** State Syntax, representing that the fuzzer needs to manually obtain or be able to self-learn the protocol state machine and state transition. **ΔM:** Learn new protocol syntax information during the fuzzing process.

Mut-based: Mutation-based Protocol Fuzzer. **Gene-based:** Generation-based Protocol Fuzzer. **FITM:** Fuzzer-in-the-middle.

SNAP: Snapshot. **FuncS:** Network Function Simulation. **N:** No execution optimization. **MemFS:** In-memory Filesystem. **SynP:** I/O Synchronisation Points.

Cov: Code Coverage. **S-Cov:** Protocol state coverage. **RC:** Response Code. **Var:** Branch and Variable. **Mem:** Memory. **FC:** Function Code (i.e., A key type of protocol message field in the ICS protocol). **N:** No feedback information utilization.

Protocol Fuzzer	Target	RT Info.	1.Protocol Syntax		2.Testcase Generation			3.Exec-Opt	4.Feedback
			Manual	Self-learning	**Mut**-based	**Gene**-based	FITM		
PROTOS(Kaksonen et al., 2001)	general	●	M			✓		N	N
Peach(Eddington, 2004)	general	●	M+S			✓		N	N
SNOOZE(Banks et al., 2006)	general	●	M+S			✓		N	N
Sulley(Amini, 2010)	general	●	M			✓		N	N
Kif(Abdelnur et al., 2007)	SIP	●	M	S		✓		N	N
LZFuzz(Bratus et al., 2008)	general	●		M			✓	N	N
AutoFuzz(Gorbunov, 2010)	FTP	●		M+S			✓	N	N
AspFuzz(Kitagawa et al., 2010)	general	●	M			✓		N	N
KFuzz(Duchene et al., 2014)	WEBAPP	●		M		✓		N	N
SECFuzz(Tsankov et al., 2012)	SEC	●	M				✓	N	N
BooFuzz(Pereyda, 2015)	general	●	M+S			✓		N	N
Pulsar(Gascon et al., 2015)	general	●	M+S			✓		N	S-Cov
TLSfuzzer(Ruiter and Joeri, 2015)	TLS	●	M	S		✓		N	RC
AFLNet(Pham et al., 2020)	general	●+◐		S	✓			N	Cov+RC
GANFuzz(Hu et al., 2018)	ICS	●		M		✓		N	N
IoTFuzzer(Chen et al., 2018b)	IoTD	●		M	✓			N	N
Polar(Luo et al., 2019)	ICS	◐		M	✓			N	Cov+FC
SeqFuzzer(Zhao et al., 2019)	ICS	●		S+M		✓		N	N
Peach*(Luo et al., 2020)	ICS	◐	M	ΔM		✓		N	Cov
DTLFfuzzer(Fiterau et al., 2020)	DTLS	●	M	S		✓		N	N
Diane(Redini et al., 2021)	IoTD	●		M	✓			N	N
SGFuzz(Ba et al., 2022)	general	◐		S	✓			N	Cov+S-Cov
PAVFuzz(Zuo et al., 2021)	PAV	◐	M	ΔM		✓		N	Cov
Snipuzz(Feng et al., 2021b)	IoTD	●		M	✓			N	N
Nyx-Net(Schumilo et al., 2022)	general	◐		S	✓			SNAP+FuncS	Cov
StateAFL(Natella, 2022)	general	◐		S	✓			N	Cov+Mem
SNPSFuzzer(Li et al., 2022)	general	◐		S	✓			SNAP	Cov+RC
SNAPFuzz(Andronidis, 2022)	general	◐		S	✓			FuncS+MemFS	N
FitM(Maier et al., 2022)	general	◐		S			✓	SNAP+FuncS	Cov+S-Cov
BLEEM(Luo et al., 2023)	general	●		M+S	✓			N	S-Cov
NSFuzz(Qin et al., 2023)	general	◐		S	✓			SynP	Cov+RC

on provided input specifications. The effectiveness of a black-box fuzzer during the fuzzing process relies crucially on the quality of the initial seeds or the precision of the input specifications. High-quality seeds and input specifications can explore more program paths in a given time and are more likely to trigger program vulnerabilities. It is worth noting that, unlike conventional software programs, blackbox fuzzing in the network protocol domain can leverage feedback from the target protocol software, such as response codes in reply messages (Pham et al., 2020).

Whitebox fuzzer. Whitebox fuzzers explore the internal structure and branches of a program by accessing its source code or binary files. Based on the code structure, they model the program's control flow, data flow, and other program information (Godefroid et al., 2008; Bounimova et al., 2013). Utilizing constraint solvers, whitebox fuzzers construct inputs that cover specific branches and analyze whether the program exhibits abnormalities. Compared to blackbox fuzzing, whitebox fuzzing achieves higher code coverage, generates higher-quality test cases, and finds more profound program vulnerabilities. However, in certain specialized scenarios, such as industrial control systems where many protocols are proprietary and their specifications and program source code are inaccessible, the applicability of whitebox fuzzing is significantly limited.

Greybox fuzzer. Greybox fuzzers lie between blackbox and whitebox fuzzers, having access to some portion of the system's source code and internal information. They combine the advantages of both blackbox and whitebox approaches, avoiding the "blindness" of blackbox testing while not requiring access to all information as in whitebox testing (Fioraldi et al., 2020; Wang et al., 2019). The concept is to provide inputs to the target software, collect valuable feedback during its runtime (e.g., branch coverage), and use this gathered feedback to guide the generation of more effective test cases. Greybox fuzzers can dynamically adjust inputs during the fuzz testing process, enhancing testing efficiency. While greybox fuzzing incurs lower resource overhead compared to whitebox testing, it cannot entirely replace whitebox fuzz testing. In software testing scenarios that require comprehensive testing of internal implementations, greybox fuzzers may have blind spots. Similarly, though more efficient than blackbox fuzzers, greybox fuzzers might be inadequate in certain situations (e.g., industrial control protocols) where access to internal program information is not possible.

Interaction between network protocol software is achieved through protocol messages. Unlike traditional software, in a blackbox scenario, response messages contained within network packets can also serve as feedback to guide fuzz testing (Luo et al., 2023). This distinction makes network protocol fuzzers not entirely conform to the classification characteristics of traditional fuzzers. In practice, the specification for each type of protocol defines the structure of protocol messages and the logic of state transitions. Once the protocol specification is available, a fuzzer can effectively construct request messages based on that specification to trigger different proto-col states, enabling more profound fuzzing of network protocol software. However, in certain scenarios, such as private protocols, where the protocol specifications are not publicly available, fuzzers need additional efforts to infer information about the protocol. In light of the characteristics of network protocols mentioned above and based on whether providing the protocol specification is required, we categorize existing network protocol fuzzers:

Specification-dependent protocol fuzzers. We observe that many existing protocol fuzzers require testers to provide a specific protocol specification. Subsequently, these fuzzers generate or mutate new test cases based on this specification. Peach (Eddington, 2004), for example, needs access to the specification of the protocol under test and uses XML language to write corresponding configuration files. These files include message formats, protocol behaviors, protocol states, and transition relationships between states. Similarly, Peach* and PAVFuzz parse XML-formatted files, converting them into protocol state machine models to generate well-formed messages. Peach* (Luo et al., 2020), an extension of the Peach framework, introduces coverage-guided packet disassembly and generation. It saves packets that trigger new path coverage, breaks them down through the disassembly process into individual fragments, and uses these fragments to construct higher-quality packets for a new round of fuzz testing. PAVFuzz (Zuo et al., 2021) models the structure of packets based on the protocol specification and introduces a state-sensitive mutation strategy to enhance the fuzz testing efficiency of Peach. Since users provide information about the tested protocol's specifications, these fuzzers are typically more efficient during testing and more likely to uncover deeper states and vulnerabilities in the tested protocol. However, in recent years, with the application of private protocols, especially in the industrial control protocol domain, where protocol specifications are not publicly available, specification-dependent protocol fuzzers cannot achieve ideal testing results in such scenarios.

Specification-free protocol fuzzers. Due to the specificity of network protocols, especially private protocols used in industrial control systems, many protocol specifications are unavailable. For such protocols, testers can only infer the program's state and the required format of input message sequences at runtime by capturing network traffic or utilizing feedback from the program's execution (such as response codes). Typically, traffic transmitted between protocol software is done through unique combinations of source IP, destination IP, and ports. PULSAR (Gascon et al., 2015) integrates the concepts of fuzzing with automatic protocol reverse engineering and simulation techniques. It captures, identifies, and extracts all traffic transmitted between protocol software, ultimately inferring message formats and the state machine model of the protocol through methods like clustering. PULSAR generates test cases based on this model, simulating the flow of traffic in subsequent fuzz testing processes to discover deep vulnerabilities in protocol software. Inspired by man-in-the-middle attacks (see Section 4.3), AutoFuzz (Gorbunov, 2010) learns protocol implementations

by constructing a finite state machine (FSM). The FSM automatically captures the communication flow between the client and server. Additionally, AutoFuzz employs bioinformatics algorithms to determine fields within individual protocol messages, categorizing them as fixed or variable. It intelligently modifies communication between the client and server for fuzz testing under the guidance of the FSM. Even without access to the protocol specification, specification-free protocol fuzzers can model the syntax of the testing protocol proficiently using techniques such as protocol reverse engineering and man-in-the-middle attacks. In recent years, this approach has gained significant traction, especially in private protocols.

3. Protocol Syntax Acquisition and Modelling

Protocol syntax acquisition and modeling are crucial stages that many modern network protocol fuzzers need to perform before they can formally enter fuzzing. Compared to traditional fuzzing objects such as file handlers (Fabrice Bellard, 2000; Cristy, 1990), protocol software has more complex constraints on the effective input space. We abstract the numerous constraints on protocol inputs at a high level into two syntaxes: protocol message syntax and protocol state syntax. The purpose of this stage is precisely to complete the inference and acquisition of the above two syntaxes to provide key information for subsequent syntax-based fuzzing. Effective protocol syntax acquisition can greatly reduce the gap between the input space and the effective input space, taking the efficiency of fuzzing to a new dimension (Zhu et al., 2022). It is worth noting that some fuzzers can reason during the fuzzing process in conjunction with program feedback and dynamically update the protocol syntax model (Schumilo et al., 2022; Somorovsky, 2016a; Fiteră̆u-Broştcan et al., 2022; Pham et al., 2020).

3.1. Terminologies

3.1.1. Network Protocol Syntax

Protocol syntax is part of network protocols and is designed to describe the structure, format, encoding method, and some other information about the data to transmit it securely and correctly to the receiver. Protocol syntax can be divided into two types: message syntax and state syntax. Message syntax is used to transmit data and metadata in communication, such as the content and size of the file being transferred. State syntax is used to describe the state of the system during communication. In stateful network protocols, the two together determine the effective input space and the desired output space at any point in time throughout the runtime cycle of the protocol software.

Message Syntax. Message syntax is a regularised encoding method for describing the structure and syntax rules of valid messages in a protocol without being influenced by context, environment, or previous messages. In other words, message syntax provides a standardized format to ensure correct and reliable communication. Message syntax specifies the division of the different fields in a message, the order in which the fields are organized, and the dependencies between fields (e.g., the
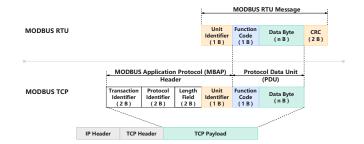

Figure 6: Modbus Protocol Specification.

contents of some fields are the length and checksum of other fields). Within each field are attributes such as data type, length, and value range. As shown in Figure 6, the Modbus protocol specification divides the message into fields such as transaction identifier, protocol identifier, length, and data fields, each with its own specified concrete value or valid value interval, length, and other attributes.

State Syntax. State syntax specifies the set of states of a network protocol and the logic of transitions between states, usually described by protocol state machines. As shown in Figure 7, state syntax is a detailed description of the set S of states and the set t of state transition logic. The client must send messages in a specific sequential order to interact correctly with the server.

3.1.2. Protocol Specification

Protocol specifications are usually provided in the form of an official document that includes a complete description of the message syntax and state syntax as well as some other relevant information such as protocol architecture, error handling, security considerations, and extensibility. A common category of protocol specification is RFC (Postel and Reynolds, 1997), e.g., the FTP protocol is defined by RFC 959, 3659, 2228, 2428, 5797.

3.1.3. Network Protocol Reverse Engineering

Network protocol reverses engineering (Cui et al., 2007) refers to the automated process of extracting protocol format, syntax, and semantics by monitoring and analyzing the inputs and outputs of the protocol software, the system behavior, and the instruction execution flow without relying on the protocol specification. It is characterized by its a priori-free nature, i.e., it does not require prior knowledge of the protocol specification.

3.2. Message Syntax Acquisition

Network protocol fuzzing is highly dependent on message syntax. For most syntax-based network protocol fuzzers, message syntax is essential (Eddington, 2004; Comparetti et al., 2009). This section systematically summarises the methods for acquiring protocol message syntax, including both manual acquisition and automatic learning. It is worth noting that some of the work in this section (e.g., automatic learning) is not a direct

improvement on fuzzing algorithms or techniques, but is more oriented towards protocol reverse aspects (Duchene et al., 2018; Narayan et al., 2015). For instance, techniques based on field segmentation, keyword recognition, and delimiter recognition. These techniques extract the protocol message format and then infer the semantics in context.

3.2.1. Manual Acquisition

Manual extraction of protocol syntax from official protocol documentation, source code, and captured network traffic is the most straightforward way. In most of the current mainstream protocol fuzzers, manual acquisition of protocol message syntax is required, e.g., Peach (Eddington, 2004). The premise of using Peach is that the user can write a Pit file using the XML language while being familiar with the target protocol syntax. As shown in Figure 7, the DataModel of the file is a description of the protocol syntax.

However, manual protocol syntax extraction is a tedious, time-consuming, and error-prone process. More seriously, 1) manual extraction methods are not scalable, and when confronted with a completely new protocol, all the work has to be completely restarted; 2) Currently, a large amount of protocol security analysis work involves objects that do not have publicly available specifications or description files, such as industrial control systems (ICS). It uses private protocols due to their specificity, which makes manual extraction of message syntax almost impossible to achieve.

3.2.2. Automatic Learning

Automatic learning of protocol specifications involves the use of techniques such as traffic analysis and deep learning. The fuzzers use machine learning techniques to analyze the response of the protocol to various input behaviors. Based on the information obtained, the fuzzer can automatically generate a formal specification of the protocol for further testing and validation. Some fuzzers use clustering or classification methods (Feng et al., 2021b) to gather behaviors generated by different inputs according to the procedure and identify patterns in the protocol behaviors. The fuzzer can use these patterns to generate a formal syntax specification for the target protocol, improving the accuracy and efficiency of fuzzing. Specifically, the following approaches exist:

Network Traffic Analysis. Network traffic analysis refers to building a corpus by capturing network traffic during the normal operation of a protocol and extracting from it different fields, delimiters, and other message syntax of the protocol by combining various heuristic algorithms. For instance, Discoverer (Cui et al., 2007) utilizes many idioms that are common in application layer protocols and extracts syntax information from them by clustering network traffic. Beddoe (Beddoe, 2004) utilizes bioinformatics algorithms to analyze consistent sequences samples by aligning similar message; Roleplayer (Cui et al., 2006) uses a byte-stream alignment algorithm to identify fields that can be mutated to abstract protocol specifications; IPspex (Sun et al., 2022) uses field semantics to locate the target message of an ICS protocol, and then uses a data-stream backtracking and

sequence-alignment algorithms to infer message syntax. Network traffic analyses are usually more effective in practice, but are often limited by the accuracy of the reasoning (Comparetti et al., 2009).

Procedural Behaviour Analysis. Program behavior analysis refers to the whitebox approach to analyze the specific behavior of a program when processing message data, thus obtaining a large amount of information about the protocol message format (Lin et al., 2008; Wondracek et al., 2008; Cui et al., 2008; Jero et al., 2019). Polyglot (Caballero et al., 2007) is a pioneer in the use of dynamic analysis techniques (e.g., taint analysis and symbolic execution), which successfully extracted the message syntax of protocols from records of program behavior by monitoring the actions of programs as they processed input. Polar (Luo et al., 2019) combines static analysis with byte-level dynamic taint analysis to extract protocol function code variables and their associated message syntax units. NS-Fuzz (Qin et al., 2023) also uses static analysis to identify and filter state variables in protocol source code and event loops during protocol execution. However, this approach cannot be used in blackbox scenarios where protocol binaries and source code are not accessible.

AI-assisted learning methods. In recent years, machine learning methods have contributed to several aspects of fuzzing. Among them, Natural Language Processing (NLP) techniques are widely used for protocol grammar analysis. Several studies have implemented automatic protocol message syntax learning systems based on this (Jero et al., 2019; Hu et al., 2018; Zhao et al., 2019).GANFuzz (Hu et al., 2018) proposes a specification-free testcase generation method for ICS protocol. It learns the protocol syntax by training a generative model in a generation-adversarial network to estimate the latent distribution function of protocol messages. However, these methods are expensive to train and the trained protocol specifications are greatly limited by the set of test cases. More importantly, when dealing with complex scenarios, NLP models often struggle to predict and explain program behavior.

Message Fragment Inference Method. Snipuzz (Feng et al., 2021b) proposed a novel approach: mutate the message to be sent byte by byte and collect the responses generated by the mutated message, combining the bytes that triggered the same response into a fragment based on the consistency of the obtained responses, with each fragment corresponding to a specific code execution path in the protocol. Although it may differ from the actual syntax rules, Snipuzz builds a hidden, functionally oriented special message syntax structure that fuzzers can use to efficiently constrain the input space. However, for some complex applications and protocols, this approach may not be able to determine the correct message structure and content. In addition, a large amount of redundant test cases are generated during the message field inference process, resulting in a waste of resources.
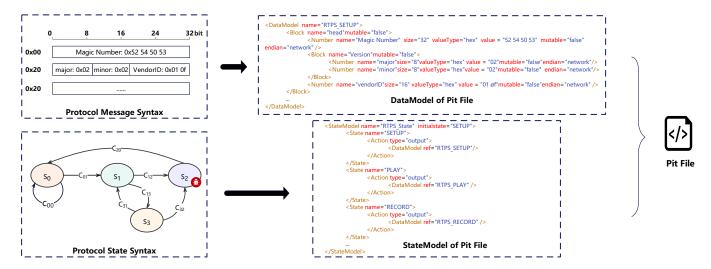
Figure 7: Peach Pit File.

3.3. State Syntax Acquisition

The input space of a protocol is constrained by message syntax and state syntax. State syntax focuses on describing the dynamic properties of the protocol software, which need to be obtained by analyzing the protocol behavior. The results of protocol behavior analysis are generally obtained by constructing a protocol state machine, which mainly includes the steps of protocol session segmentation, different types of message sequence construction, and state machine concise. State syntax-based guided fuzzers send a specific sequence of messages in the protocol state space to reach the target state of interest. Although part of the mutation-based fuzzers do not consider the message syntax format, they need to consider the protocol state syntax when fuzzing protocols. This section systematically summarises the three basic approaches currently used to infer protocol state syntax, i.e., manual learning, active inference, and passive inference.

3.3.1. Manual Learning

Manual learning of state syntax involves manually analyzing the state transitions of a network protocol and then writing syntax rules to describe these transitions. These syntax rules specify the inputs and outputs in network protocol state transitions, as well as changes in protocol state. This is a customized approach for a specific network protocol and requires an in-depth understanding of how the protocol works and the details of state transitions. The general idea is essentially the same as manually acquiring message syntax. The main difference between the two is that the data structure describing the state syntax is different from that describing the message syntax, and state syntax is usually described by constructing a finite state machine (FSM), as shown in the DataModel of Pit File in Figure 7. In the Pit file, the StateModel module is utilized to describe the protocol state and state transition logic from the fuzzer's perspective, as shown in the StateModel of Pit File in Figure 7. By learning the state syntax manually, testers can build more efficient fuzzing test cases to help discover vulnerabilities and security issues in network protocol implementations.

3.3.2. Active Inference

Active inference is a stage independent of the main process of fuzzing testing. In this phase, the learning machine actively generates a series of test messages and sends them to the PUT to obtain the corresponding outputs, while using model learning algorithms (e.g., Angluin's L* algorithm (Angluin, 1987), LearnLib (Raffelt et al., 2009b)) to infer and construct the complete state machine of the target protocol. Through this series of work (Ruiter and Joeri, 2015; Somorovsky, 2016b; Ferreira et al., 2021; Fiterău-Broştean et al., 2016; Raffelt et al., 2009a), fuzzers can obtain a protocol state syntax and perform stateful fuzzing on target protocol software based on that syntax. For instance, TLS-Attacker pioneered the use of protocol state fuzzing (or simple stateful fuzzing). Rather than looking for defects that traditional fuzzing is good at finding, such as buffer overflows, string error handling, etc., the purpose of stateful fuzzing is to search for state machine bugs. Stateful protocol fuzzers can utilize complex protocol state machines constructed by active learning to test for flaws in state transition logic that can be exploited to construct attacks such as authentication bypass (Fiterau et al., 2020). The active inference approach applies to protocols where the state space is not very complex.

3.3.3. Passive Inference

Different from active inference, passive inference does not require active messages to be sent to the test object to accomplish state learning before the fuzzing formally starts. Passive inference learns state transition logic through samples (Gascon et al., 2015; Gorbunov, 2010; Hsu et al., 2008). There are two main sources of samples: network traffic generated by the protocol software under normal behavior and test cases used during fuzzing performed in parallel. AFLNet (Pham et al., 2020) dynamically maintains the finite state machine (FSM) of the protocol by determining whether this test explores a new state of the protocol based on the response code feedback during the fuzzing. Prospex (Comparetti et al., 2009) acquires network traces of protocol runs and abstracts message formats to construct message clustering (message clustering). Based on this, it

constructs a state tree using a heuristic algorithm [49] to merge similar states and generate a minimum state machine. The disadvantage of passive inference protocol state syntax is that it is difficult to capture the full context of the protocol state, which makes it more challenging to generate inputs approximating the real situation in subsequent stages.

4. Test Case Generation

This section will describe in detail methods for generating test cases. In the unified process model shown in Figure 5, the inputs to the module are the protocol syntax model and the generation strategy, and the outputs are the actual test cases executed by the protocol software (e.g., server, client). In network protocol fuzzing, the test cases are usually sequences of packets in chronological order, which are sent by the fuzzer to the network protocol software under test in a given order and complete execution. The quality of the test cases greatly determines whether or not a vulnerability or security flaw in the program under test will be triggered, so the approach and decisions made at this stage will directly impact the effectiveness of fuzzing.

The approach to generating effective test cases involves two key issues:

Q1:How do generate test cases that can pass protocol message syntax checking?

Q2:How do make the generated message sequences effective in exploring the complex state space of a protocol?

Depending on the test case generation method, fuzzers are usually classified into two categories: generation-based and mutation-based. Generation-based fuzzers use a provided syntax model to generate test cases. Mutation-based fuzzers mutate the seed file provided by the user using various mutation operators, which in turn generates large amounts of test cases. It is worth noting that, inspired by the idea of MITM attacks, in recent years there has been a category of fuzzers that intercept the communication traffic between the client and the server and modify it as a way to construct test cases. This category of fuzzers has shown unique advantages in the field of protocol fuzzing. We refer to such fuzzers as FITM (fuzzer-in-middle). The network topology of the FITM at runtime is different from the traditional case shown in Figure 2, as it acts as an intermediary to intercept the communication traffic between the client and the server, rather than acting as a server or a client as in the traditional fuzzer idea, as shown in Figure 9. The following is a detailed description of these three categories of fuzzers.

4.1. Generation-based Protocol Fuzzers

Generation-based protocol fuzzers utilize two categories of syntax models (i.e., message syntax and state syntax described in Section 3) to resolve the above two key issues separately. Corresponding to the two categories of syntax, the generation-based protocol fuzzers need to accomplish two dimensions of generation: the generation of individual messages and the combination of message sequences.

The earlier protocol fuzzers such as Peach (Eddington, 2004), PROTOS (Kaksonen et al., 2001), Sulley (Amini, 2010),

SPIKE (Aitel, 2002), and SNOOZE (Banks et al., 2006) are typical generation-based fuzzers. These fuzzers provide a standard interface in the form of a configuration file for the user to use, and the user needs to write a test configuration file for the target protocol based on the tool specification and protocol syntax. Configuration files usually contain some form of a data structure describing the protocol syntax, so the process of writing a configuration file is actually a modeling of the protocol syntax from the user's point of view.

Peach is a representative of generation-based fuzzers, which requires the user to write a configuration file, i.e., a Pit file, using the XML language, as shown in Figure 7. The Pit file consists of five parts: a generic configuration, a data model, a state model, an agent and a monitor, and a fuzzing configuration, where the data model and the state model are syntax modeling of the target protocol. Data models provide sub-elements at different levels of granularity, such as Number, Blob, or String, which are used to define the message syntax structure of a single message. Generally, a complete message is described by a data model. The state model describes the basic state machine logic required to test a protocol via the state and action sub-elements. Peach generates individual messages by selecting several variable data fields from the corresponding data model and uses a mutation operator to generate test cases by randomly mutating these fields; message sequences are generated by executing the local state machine logic defined in the state model. SNOOZE provides the user with a set of fuzzing primitives, and a fuzzing scenario summary document written using the fuzzing primitives defines the complete process of fuzzing. For example, the SnoozeMessage primitive is used to declare a message object, the setField primitive is used to specify the values of data fields in a message, and so on.

This initial generation-based fuzzer can efficiently generate inputs that conform to the protocol specification, guided by the syntax model. However, its dependence on the quality of the syntax model greatly limits the effectiveness of the input. Incomplete or even incorrect syntax models can lead fuzzers to an inefficient dilemma, while manually extracting complex protocol syntax specifications is usually a time-consuming and error-prone process. More importantly, such fuzzers lack the ability of protocol state space heuristic auto-exploration and can only execute exactly according to a pre-set finite state machine, e.g., Peach repeatedly iterates through the execution of all actions defined in the state model.

The current advanced protocol fuzzer improves on several aspects of the test case generation scheme. Different from the probabilistic selection decision of target mutation fields such as Peach, PAVFuzz (Zuo et al., 2021) argues that data elements in a data packet have different levels of importance. It calculates dynamic mutation weights for data elements and stores them in a relational table, performing state-sensitive mutations on data elements that are more likely to trigger system vulnerabilities. Peach* (Luo et al., 2020) utilizes coverage feedback to preserve valuable packets, decompose them into fragments, and construct higher-quality test cases in the next round of fuzzing. In addition, research on protocol syntax self-learning (as we describe in Section 3) can also contribute directly to the efficiency

improvement of generation-based fuzzers. Prospex (Comparetti et al., 2009) extends the work on acquiring message formats based on the analysis of program behavior and introduces techniques for identifying and clustering different types of messages. It is capable of automatically inferring state machines and supports the automatic transformation of the two types of syntax obtained from learning into Pit files, which contributes positively to the development and wide application of Peach.
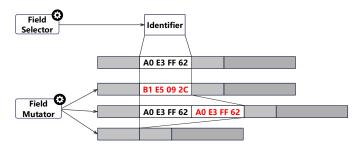

Figure 8: Mutation-Based Fuzzers: Principles of Mutator Operation.

4.2. Mutation-based Protocol Fuzzers

The mutation-based protocol fuzzer does not need to provide a syntax specification. It uses various mutation strategies (e.g., bitflip, arithmetic, havoc, and splice) on pre-provided seeds to generate test cases, As shown in Figure 8. Due to its simplicity and ease of implementation, this category of fuzzers is extensively used. Based on the traditional mutation-based fuzzer, AFL pioneered the introduction of coverage feedback, making coverage-based greybox fuzzer (CGF) one of the hottest research directions in the fuzzing field in recent years. However, the application of mutation-based fuzzers in the field of protocol fuzzing is greatly hampered by the specificity of network protocols compared to traditional software. Without syntax guidance, bit granularity mutation severely disrupts the message structure, making it difficult for test cases to pass the initial syntax check of the protocol. The results of an existing study (Li et al., 2021) showed that up to 90% of the test cases generated using the AFL mutation strategy were rejected by the protocol software due to syntax errors that failed the initial check.

To break this inherent limitation of traditional fuzzers, advanced mutation-based protocol fuzzers attempt to use protocol syntax to guide the mutation of test cases (Schumilo et al., 2022; Li et al., 2022; Andronidis, 2022; Pham et al., 2020; Zuo et al., 2022; Liang et al., 2018), i.e., the learned syntax specification is utilized to protect the structure of the inputs to be tested from being corrupted during the mutation process. This attempt also blurs the boundary between mutation and generation and is gradually becoming a tendency. AFLNet sequentially combines all messages sent by a client during a single interaction into a single seed, while using specific ASCII characters to divide the different messages. At the same time, it maintains and updates a finite state machine by extracting response codes. During the fuzzing process, AFLNet can not only judge the quality of test cases through state feedback but also continuously add newly discovered states to the existing state machine, thus completing the self-learning of protocol state syntax. StateAFL (Natella, 2022) similarly learns the state syntax of the protocol software continuously during the fuzzing process. It uses snapshot techniques to track memory allocation and network IO operations of the PUT and maps them to unique state identifiers. Since many protocols do not embed specific state description fields (e.g., state codes, etc.) in response messages, this state extraction method is more widely used than AFLNet. It is worthy of mentioning that IJON (Aschermann et al., 2020) argues that current fuzzers do not correctly explore the state of a program that exceeds code coverage, e.g., where program execution leads to the same code coverage, but with different state traces. The human analyzer is capable of greatly improving the efficiency and performance of the fuzzer by annotating those parts of the state space that should be explored more thoroughly (usually with only one or two additional lines of code). Therefore, based on AFL, IJON introduces a manual labeling mechanism to guide CGFs to explore the hard-to-cover parts of the state space more efficiently.

In addition to the need to match the protocol syntax to ensure the validity of the generated test cases, mutation-based fuzzing has several inherent disadvantages. For example, the quality of the seed file severely affects the effectiveness of mutation-based fuzzers. In the absence of high-quality, type-rich seed files, the fuzzer is much less likely to find deeper vulnerabilities, and it will take longer to discover them, ultimately leading to relatively low overall efficiency.
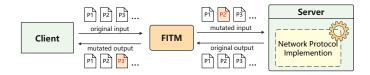

Figure 9: FITM (Fuzzer-in-the-Middle) Fuzzing Process.

4.3. FITM (fuzzer-in-middle)

Man-in-the-middle (MITM) attack is a method of network attack. An attacker intercepts an existing dialogue or data transmission between two parties by eavesdropping or masquerading as a legitimate participant and unknowingly accessing, tampering with, or manipulating information exchanged between the parties. The protocol fuzzer (man-in-the-middle fuzzer) designed based on this idea captures the network traffic between the server and the client and selectively tampers and replays it as shown in Figure 9. Such fuzzers do not need to do the complex, syntactically-compliant message construction themselves, and they also alleviate the dependence on state syntax. Notably, FITM can initiate the fuzzing process at any stage of the communication between the client and the server. For example, if a tester wishes to explore vulnerabilities in the protocol implementation at a later stage of the communication, FITM can do this without disrupting packets from earlier communications. Moreover, in the network topology of FITM, the server and the client are two peers, which makes it possible to perform fuzzing for both the client and the server at the same time (Tsankov et al., 2012).

In recent years, more and more protocol fuzzers have turned to using man-in-the-middle attacks to generate test cases, e.g., ProxyFuzz (Sklenar, 2011), AutoFuzz (Gorbunov, 2010), SEC-FUZZ (Tsankov et al., 2012), FITM (Maier et al., 2022), and so on. However, this approach requires specialized tools and techniques to intercept and modify the communication between protocol software, which can increase the complexity of the fuzzing process. Also, in special fuzzing situations, FITM requires as detailed a specification of protocol syntax as other generation-based fuzzing. For example, when performing fuzzing on security protocols where encryption mechanisms are present, the FITM needs to know the encryption and decryption algorithms to perform effective mutations on network traffic.

5. Test Execution and Monitor

Test execution denotes the process in which the PUT receives and executes test cases generated by the fuzzer. The monitor senses the trigger of a bug by monitoring the running state of the PUT. Like common fuzzers, most protocol fuzzers catch abnormal crashes of the program under test or monitor them in combination with memory error-checking tools such as Sanitizer provided by the compilation toolchain.

In fuzzers for common software, test execution is not a worthwhile concern as the execution process is usually very simple. For example, AFL (Zalewski, 2017) redirects the standard input of the program to be tested directly to the test case file or adds the file path of the test case in the command line arguments. In the protocol fuzzing scenario, the test execution process imposes a huge performance load on the protocol fuzzer, which seriously impacts the fuzzing efficiency.

First, the protocol software transmits data through the network interface. Specifically, the network interface constructs and sends data according to the protocol, and also receives and interprets data received from the network to transmit it to the higher-level protocol stack. This also means that traditional methods of transmitting input via redirects or command-line arguments are not viable. In the most ideal case, the server process under test is located on the same physical machine as the client process, in which case it is still necessary to use the local loopback IP address as well as the port number, and then complete the input of the test cases over a standard network interface (e.g., TCP or UDP). Since network interfaces are much slower compared to file reads, this dependency results in additional time overhead. Second, complex, multi-threaded server processes generally have a higher startup time cost. Further, since there is no way to confirm that the server program after a new startup or receipt of a message enters a state where it can continue to receive messages, the fuzzer often needs to empirically set a regular waiting time, which introduces another portion of the time overhead. Finally, part of the protocols require extra effort to fully restore to the pre-test state, e.g., the FTP protocol may have changes to the target's filesystem during testing.

The above problems have attracted extensive attention from researchers. In recent years, some advanced works have keenly observed this and proposed some efficient techniques to increase the speed and throughput of protocol fuzzing to a new level of order of magnitude.

5.1. Snapshots

Snapshots are static copy files that store the state of the operating system or process in physical memory and various devices at a particular moment. Commonly, there are system snapshots, virtual machine snapshots, file system snapshots, etc. By taking and recovering snapshots, a specific target state can be completely preserved and recovered. Snapshots have a wide range of practical applications in various scenarios. Xu et al. (Xu et al., 2017) pioneered the use of snapshotting techniques in fuzzing, providing optimized fuzz primitives replacements for system calls such as fork, speeding up LibFuzzer by up to 736 times. After that, snapshot techniques have been extensively focused and applied (Schumilo et al., 2022; Li et al., 2022; Andronidis, 2022; Xu et al., 2017; Schumilo et al., 2021; Schumilo et al., 2017; Song et al., 2020; Giuffrida and van der Kouwe, 2022; Maier et al., 2022), and the application in the field of protocol fuzzing is also one of the successful practices.

In addition to improving the state restoration speed of the PUT in fuzzing, the snapshot technique enables the fuzzer to span the state space of the protocol at a minimal cost. Nyx-net (Schumilo et al., 2022) introduces kernel-state, VM-level incremental snapshots to ensure that all states of the PUT are reset before each test case is executed. SNPSFuzzer (Li et al., 2022) uses the user-state, process-level snapshot tool CRIU to save the context of each state to be fuzzed in the PUT and restores that snapshot directly when fuzzing a specific state is required. This saves a large amount of time wasted during test execution by sending frequent prefix messages to span the state space. FITM (Maier et al., 2022) also uses CRIU to independently fuzz each state transition by creating a snapshot of each protocol's new state discovered by the fuzzer.

5.2. Network Function Replacement or Emulation

Some research has focused on improving the underlying causes of inefficiency in protocol fuzzing by replacing network function API, file system API, etc. with more efficient interface functions or customized simulation methods. Nyx-net implements a virtual machine-based emulation of relevant network functions. It uses the hook to intercept network functions, APIs related to file descriptor operations (e.g., *accept(), recv(), dup(), close()*), and a customized emulation API to implement a high-performance and high-throughput conversion interface. Snap-Fuzz (Andronidis, 2022) is enabled to intercept system calls issued by the program under test, replacing slow standard Internet sockets with fast UNIX Domain sockets. In addition, it replaces the operating system's persistent filesystem API with an in-memory filesystem API to mitigate the time overhead caused by filesystem resets.FITM uses hooks instead of slow sockets and shared maps instead of network system calls.

NSFuzz (Qin et al., 2023) proposes a solution to set I/O synchronization points based on protocol event loops to accelerate test execution, based on insights into how code is implemented in the service processing phase of network protocol software. Specifically, the network protocol enters its main service processing stage after completing its initialization work and waits for message input from the client. Service processing is usually implemented as an iterative event loop, where each client input triggers a round of related message processing, and after completing the current processing, it returns to the event loop entry to wait for the next input from the client. Therefore, NSFuzz takes the entry of the event loop as a sign that the server can accept the next input, and sets the IO synchronization point at the loop entry to send the fuzzer a sign that it can continue. This design effectively mitigates the overhead associated with setting a fixed sending time window on the fuzzer side and optimizes the performance of test execution.

6. Feedback Information Acquisition and Utilization

Feedback refers to a category of available attributes contained in the output produced by a software system. The fuzzer extracts valid information from these attributes and makes dynamic adjustments to the subsequent fuzzing process, e.g., by making heuristically fuzzing decisions. Passive learning, as mentioned above, involves collecting state traces to infer information about the input format. Different from passive learning, feedback is a form of active learning that interacts with the software program to be tested during fuzzing. The fuzzer tracks and evaluates the effectiveness of existing test cases, using the results of historical test cases to make adjustments before attempting the next input. Without the introduction of feedback and heuristic strategies, fuzzing simply remains an inefficient blind random test.

Since the popularity of coverage-based greybox fuzzing (CGF) represented by AFL (Zalewski, 2017), most of the state-of-the-art fuzzers proposed in academia and industry have been intelligently guided to proceed with subsequent fuzzing tests based on the feedback of the results of previous tests. In this section, we systematically summarise the commonly used categories of feedback in existing protocol fuzzers (including response codes, coverage, branch, variables, and memory) and introduce the corresponding representative fuzzers.

6.1. Response Code

The response code is included in the message replied to by the server. After a client sends a request to a server, the server replies with a message containing a state response code, where the state response code is used to ensure that the client's request is acknowledged and to notify the client of the current server state. The state trace of the system can be inferred by observing the state response code or some information extracted from the response. For example, the exchange of information between the FTP client (c) and the LightFTP server (s) on the control channel is described in Figure 10. AFLNet (Pham et al., 2020)

uses the server's response code to track the state traces of the test case, which in turn guides the fuzzer to test valid regions in the state space. Bleem (Luo et al., 2023) collects output sequences from target clients and servers at runtime. It supports and guides fuzzing by analyzing the output sequences to infer the system state and dynamically update the state space of all protocol entities (i.e., clients and servers) of the system to be tested.

```
c: [SYN]
s: [SYN, ACK]
c: [ACK]
s: 220 Service ready for new user.
c: USER
s: 331 User name okay, need password.
c: PASS
s: 230 User logged in, proceed.
c: SYST
s: 215 NAME system type.
c: RSS
s: 211 Feature listing.
c: PWD
s: 257 Return to current path.
c: CWD
s: 250 Return to the modified directory.
c: STOR
s: 150 File status okay.
   226 Transfer complete.
c: QUIT
s: 221 Goodbye!
```

Figure 10: Message exchange between an FTP client (c) and lightFTP server (s) on the control channel.

6.2. Coverage

Coverage includes code coverage, function coverage, branch coverage, path coverage state, coverage, etc., which are generally applied to greybox fuzzers. The fuzzers perform instrumentation of the program under test and use bitmaps to record the coverage of paths, branches, etc. during test case execution. Many fuzzers such as stateAFL (Natella, 2022), AFLNet, and libfuzzer (Serebryany, 2015) use coverage feedback to guide fuzzing.

Anti-fuzzing techniques deceive the fuzzer by methods such as inserting pseudo-paths or adding delay and obfuscation code to the error handling code to slow down the dynamic analysis. This plays a significant preventive role against coverage-based fuzzers. In recent years, due to the widespread use of anti-fuzzing techniques, fuzzers that do not distinguish between different types of edge coverings have a harder time detecting vulnerabilities. Wang et al. (Wang et al., 2020) proposes to focus on edges associated with sensitive memory operations. It

evaluates and labels edges at three levels: function, loop, and basic block, and prioritizes inputs according to new security-sensitive overlays as a way to counter anti-fuzzing techniques and detect more vulnerabilities.

6.3. Branch Feedback and Variable Feedback

Branch feedback is similar in principle to branch coverage in coverage, but the main difference between the two is that branch feedback requires the user to manually annotate the code. The user manually marks branches that are more likely to trigger a vulnerability as being of interest by observing them. Based on the method of monitoring branch coverage in AFL, Chen et al. (Chen et al., 2019) improves the performance of the fuzzer by performing additional observations on those branches of interest that are manually labeled. There is a category of network protocol implementations that make use of enumeration-type program variables to record information about the state of the protocol. The idea of variable feedback is precisely to obtain information about the state of the protocol by observing this type of procedural variables and to update and maintain the protocol state machine to improve the effectiveness of fuzzing. IJON (Aschermann et al., 2020) argues that current fuzzers are unable to properly explore the state space of programs beyond code coverage, such as those test cases that result in the same code coverage, but different state trace. By annotating those parts of the state space that should be explored further (usually with a line or two of code), programmers can help fuzzers overcome several current obstacles associated with fuzzing complex applications. The core idea is to explore the behavior of a program more systematically using program variables that represent the internal state of the program. SGFuzz (Ba et al., 2022) uses pattern matching to identify enumerated type variables as possible state variables, using variable feedback to guide fuzzing and thus explore the program state space more efficiently.

In specifically, the specific application scenarios of branch feedback and variable feedback depend on whether the PUT records the protocol state of the program as it runs via program variables or program points. In this context, program variables refer to identifiers that are used to store data in programming. Variables can contain various types of data, such as numbers, strings, booleans, etc. Program point refers to a specific location or moment used to describe the execution of a program. It may refer to a particular line in the code, the location of a function call, the location where a particular state or event occurs, etc.

6.4. Memory Feedback

Memory signals refer to the electrical signals or combinations of signals that are used to control the operation of and memory access (e.g., RAM, ROM, etc.) in a computer system or electronic device. They control the reading, writing, and processing of data. Some fuzzers argue that when the memory signal changes, it causes the protocol state to change as well. The core idea of memory feedback is to observe whether the current system inputs have an impact on these memory regions by taking a snapshot of the memory regions and comparing the

relevant information. StateAFL (Natella, 2022) takes snapshots of memory regions at runtime and applies a hash algorithm to map the state in each memory to a unique protocol state identifier. Through the above operations, the fuzzer can infer the current protocol state of the target server and gradually construct a protocol state machine to guide the fuzzing.

7. Directions and Perspectives for future research

Network protocol fuzzing techniques have evolved at a high rate over the past 20 years or so, but some limitations still exist. To stimulate the design of more efficient and practical protocol fuzzers in the future, we summarise our viewpoints and thoughts on future directions.

7.1. High Expandability

Studies have shown (Schumilo et al., 2022; Böhme and Falk, 2020) that any protocol object that is fuzzed for the first time reveals a large amount of new security-related findings. We believe that while focusing on improvements to the fuzzing algorithms, it is also necessary to enhance the expandability of the protocol fuzzers for the test objects. Protocol fuzzers are more dependent on syntax input than fuzzers oriented to other test objects, which also leads to the fact that existing protocol fuzzers tend to be less easy to expand with new test objects. The Peach extension requires the user to provide the full Pit file for the new protocol object, and the AFLNet extension requires the user to write C code for the new protocol response extraction. Sections 3.2 and 3.3 have summarised part of the methods, but these methods have their limitations and it is difficult to deal with new protocols in a convenient way. We expect fuzzing techniques to be further combined with deep learning and model inference techniques in the future.

7.2. More Effective Feedback

The availability of feedback information has led to the evolution of fuzzing techniques from blindness to intelligence, and AFL's use of coverage has created the world of CGF. As described in the feedback section, there are many varieties of feedback available for protocol fuzzers, but none of them do as much to bring about a milestone increase in fuzzing effectiveness as coverage. In recent years, protocol security in ICS and IoT has attracted a lot of concern, but the acquisition of coverage in test scenarios for these protocols would be limited. Is it possible to present proprietary feedback for special protocol testing scenarios (e.g., real ICS devices, IoT devices)?

7.3. Complex Testing Scenarios

Existing protocol fuzzers tend to consider scenarios where only one server interacts with one client. In real-world scenarios, the server can connect to a large amount of clients at the same time, i.e., one to multiple or even multiple to multiple interaction scenarios are very common, will the complex testing scenarios with multiple entities introduce new types of protocol bugs?

7.4. Underlying Protocol Testing Capabilities

Existing protocol fuzzers focus more on application layer protocols and only a few research efforts have been made to analyse the security of TCP and UDP. Security analysis techniques for transport layer or lower layer protocols differ from upper layer protocols and have difficulties in technical implementation. Can the function replacement and emulation techniques mentioned in Section 5 be applied to test the underlying protocols?

8. Conclusions

Network protocols are more challenging compared to traditional program testing due to their highly structured inputs and enormous state spaces. Fuzzing has been widely used in the field of network protocols as one of the most effective and efficient methods to discover vulnerabilities in recent years. In this paper, we summarise a unified process model for network protocol fuzzing by analyzing a large amount of high-quality literature and fuzzers and propose four concerns for network protocol fuzzing research, including protocol syntax acquiring and modeling, testcase generation, test execution and monitoring, and feedback information acquisition and utilization. Based on this model, we systematically summarise the design decisions as well as the innovations of existing protocol fuzzing techniques in the four stages of the unified process model. Finally, we propose promising directions and perspectives for future research in the area of protocol fuzzing.

References

Abdelnur, H.J., State, R., Festor, O., 2007. Kif: a stateful sip fuzzer, in: Proceedings of the 1st international conference on Principles, systems and applications of IP telecommunications, pp. 47–56.

Aitel, D., 2002. The advantages of block-based protocol analysis for security testing. Immunity Inc., February 105, 106.

Amini, 2010. Sulley fuzzing framework.

Andronidis, Anastasios, C., 2022. Snapfuzz: high-throughput fuzzing of network applications, in: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 340–351.

Angluin, D., 1987. Learning regular sets from queries and counterexamples. Information and computation 75, 87–106.

Aschermann, C., Schumilo, S., Abbasi, A., Holz, T., 2020. Ijon: Exploring deep state spaces via fuzzing, in: 2020 IEEE Symposium on Security and Privacy (SP), IEEE. pp. 1597–1612.

Ba, J., Böhme, M., Mirzamomen, Z., Roychoudhury, A., 2022. Stateful greybox fuzzing, in: 31st USENIX Security Symposium (USENIX Security 22), pp. 3255–3272.

Babić, D., Bucur, S., Chen, Y., Ivančić, F., King, T., Kusano, M., Lemieux, C., Szekeres, L., Wang, W., 2019. Fudge: fuzz driver generation at scale, in: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 975–985.

Banks, G., Cova, M., Felmetsger, V., Almeroth, K., Kemmerer, R., Vigna, G., 2006. Snooze: toward a stateful network protocol fuzzer, in: ISC, Springer. pp. 343–358.

Beddoe, M.A., 2004. Network protocol analysis using bioinformatics algorithms. Toorcon 26, 1095–1098.

Böhme, M., Falk, B., 2020. Fuzzing: On the exponential cost of vulnerability discovery, in: Proceedings of the 28th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering, pp. 713–724.

Bounimova, E., Godefroid, P., Molnar, D., 2013. Billions and billions of constraints: Whitebox fuzz testing in production, in: 2013 35th International Conference on Software Engineering (ICSE), IEEE. pp. 122–131.

Bratus, S., Hansen, A., Shubina, A., 2008. Lzfuzz: a fast compression-based fuzzer for poorly documented protocols .

Bulekov, A., Das, B., Hajnoczi, S., Egele, M., 2023. No grammar, no problem: Towards fuzzing the linux kernel without system-call descriptions, in: Network and Distributed System Security (NDSS) Symposium.

Caballero, J., Yin, H., Liang, Z., Song, D., 2007. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis, in: Proceedings of the 14th ACM conference on Computer and communications security, pp. 317–329.

Chen, C., Cui, B., Ma, J., Wu, R., Guo, J., Liu, W., 2018a. A systematic review of fuzzing techniques. Computers & Security 75, 118–137.

Chen, J., Diao, W., Zhao, Q., Zuo, C., Lin, Z., Wang, X., Lau, W.C., Sun, M., Yang, R., Zhang, K., 2018b. Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing., in: NDSS.

Chen, Y., Lan, T., Venkataramani, G., 2019. Exploring effective fuzzing strategies to analyze communication protocols, in: Proceedings of the 3rd ACM Workshop on Forming an Ecosystem Around Software Transformation, pp. 17–23.

Comparetti, P.M., Wondracek, G., Kruegel, C., Kirda, E., 2009. Prospex: Protocol specification extraction, in: 2009 30th IEEE Symposium on Security and Privacy, IEEE. pp. 110–125.

Corina, J., Machiry, A., Salls, C., Shoshitaishvili, Y., Hao, S., Kruegel, C., Vigna, G., 2017. Difuze: Interface aware fuzzing for kernel drivers, in: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pp. 2123–2138.

Cristy, J., 1990. Imagemagick. URL: https://imagemagick.org/index.php.

Cui, W., Kannan, J., Wang, H.J., 2007. Discoverer: Automatic protocol reverse engineering from network traces., in: USENIX Security Symposium, pp. 1–14.

Cui, W., Paxson, V., Weaver, N., Katz, R.H., 2006. Protocol-independent adaptive replay of application dialog., in: NDSS.

Cui, W., Peinado, M., Chen, K., Wang, H.J., Irun-Briz, L., 2008. Tupni: Automatic reverse engineering of input formats, in: Proceedings of the 15th ACM conference on Computer and communications security, pp. 391–402.

CVE-2014-0160, 2014. Heartbleed - a vulnerability in openssl. URL: http://heartbleed.com.

Daniele, C., Andarzian, S.B., Poll, E., 2023. Fuzzers for stateful systems: Survey and research directions. arXiv preprint arXiv:2301.02490 .

Deng, Y., Yang, C., Wei, A., Zhang, L., 2022. Fuzzing deep-learning libraries via automated relational api inference, in: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 44–56.

Duchene, F., Rawat, S., Richier, J.L., Groz, R., 2014. Kameleonfuzz: evolutionary fuzzing for black-box xss detection, in: Proceedings of the 4th ACM conference on Data and application security and privacy, pp. 37–48.

Duchene, J., Le Guernic, C., Alata, E., Nicomette, V., Kaâniche, M., 2018. State of the art of network protocol reverse engineering tools. Journal of Computer Virology and Hacking Techniques 14, 53–68.

Eddington, M., 2004. Peach fuzzing platform. Peach Fuzzer 34, 32–43.

Fabrice Bellard, B.B., 2000. Ffmpeg. URL: https://ffmpeg.org/.

Feng, X., Sun, R., Zhu, X., Xue, M., Wen, S., Liu, D., Nepal, S., Xiang, Y., 2021a. Snipuzz: Black-box fuzzing of iot firmware via message snippet inference, in: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, pp. 337–350.

Feng, X., Sun, R., Zhu, X., Xue, M., Wen, S., Liu, D., Nepal, S., Xiang, Y., 2021b. Snipuzz: Black-box fuzzing of iot firmware via message snippet inference, in: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, pp. 337–350.

Ferreira, T., Brewton, H., D'Antoni, L., Silva, A., 2021. Prognosis: closed-box analysis of network protocol implementations, in: Proceedings of the 2021 ACM SIGCOMM 2021 Conference, pp. 762–774.

Fioraldi, A., D'Elia, D.C., Coppa, E., 2020. Weizz: Automatic grey-box fuzzing for structured binary formats, in: Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis, pp. 1–13.

Fiterau, P., Jonsson, B., Merget, R., De Ruiter, J., Sagonas, K., Somorovsky, J., 2020. Analysis of dtls implementations using protocol state fuzzing, in:

29th USENIX Security Symposium, Online, August 12–14, 2020, pp. 2523–2540.

Fiterău-Broştcan, P., Jonsson, B., Sagonas, K., Tåquist, F., 2022. Dtls-fuzzer: A dtls protocol state fuzzer, in: 2022 IEEE Conference on Software Testing, Verification and Validation (ICST), IEEE. pp. 456–458.

Fiterău-Broştean, P., Janssen, R., Vaandrager, F., 2016. Combining model learning and model checking to analyze tcp implementations, in: Computer Aided Verification: 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II 28, Springer. pp. 454–471.

Gascon, H., Wressnegger, C., Yamaguchi, F., Arp, D., Rieck, K., 2015. Pulsar: Stateful black-box fuzzing of proprietary network protocols, in: Security and Privacy in Communication Networks: 11th EAI International Conference, SecureComm 2015, Dallas, TX, USA, October 26-29, 2015, Proceedings 11, Springer. pp. 330–347.

Giuffrida, E.G.C., van der Kouwe, H.B.E., 2022. Snappy: Efficient fuzzing with adaptive and mutable snapshots .

Godefroid, P., 2007. Random testing for security: blackbox vs. whitebox fuzzing, in: Proceedings of the 2nd international workshop on Random testing: co-located with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), pp. 1–1.

Godefroid, P., Kiezun, A., Levin, M.Y., 2008. Grammar-based whitebox fuzzing, in: Proceedings of the 29th ACM SIGPLAN conference on programming language design and implementation, pp. 206–215.

Gorbunov, S., 2010. Autofuzz: Automated network protocol fuzzing framework. Ijcsns 10, 239.

Hermann, H., Johnson, R., Engel, R., AG, A.T., 1995. A framework for network protocol software, in: Proceedings OOPSLA '95, ACM SIGPLAN Notices.

Hess, D.K., Safford, D.R., Pooch, U.W., 1992. A unix network protocol security study: Network information service. ACM SIGCOMM Computer Communication Review 22, 24–28.

Hsu, Y., Shu, G., Lee, D., 2008. A model-based approach to security flaw detection of network protocol implementations, in: 2008 IEEE International Conference on Network Protocols, IEEE. pp. 114–123.

Hu, Z., Shi, J., Huang, Y., Xiong, J., Bu, X., 2018. Ganfuzz: a gan-based industrial network protocol fuzzing framework, in: Proceedings of the 15th ACM International Conference on Computing Frontiers, pp. 138–145.

Jero, S., Pacheco, M.L., Goldwasser, D., Nita-Rotaru, C., 2019. Leveraging textual specifications for grammar-based fuzzing of network protocols, in: Proceedings of the AAAI Conference on Artificial Intelligence, pp. 9478–9483.

Kaksonen, R., Laakso, M., Takanen, A., 2001. Software security assessment through specification mutations and fault injection, in: Communications and Multimedia Security Issues of the New Century: IFIP TC6/TC11 Fifth Joint Working Conference on Communications and Multimedia Security (CMS'01) May 21–22, 2001, Darmstadt, Germany, Springer. pp. 173–183.

Kim, K., Jeong, D.R., Kim, C.H., Jang, Y., Shin, I., Lee, B., 2020. Hfl: Hybrid fuzzing on the linux kernel., in: NDSS.

Kitagawa, T., Hanaoka, M., Kono, K., 2010. Aspfuzz: A state-aware protocol fuzzer based on application-layer protocols, in: The IEEE symposium on Computers and Communications, IEEE. pp. 202–208.

Li, J., Li, S., Sun, G., Chen, T., Yu, H., 2022. Snpsfuzzer: A fast greybox fuzzer for stateful network protocols using snapshots. IEEE Transactions on Information Forensics and Security 17, 2673–2687.

Li, J., Zhao, B., Zhang, C., 2018. Fuzzing: a survey. Cybersecurity 1, 1–13.

Li, S., Li, J., Fu, J., Xue, M., Yu, H., Sun, G., 2021. Protocol fuzzing with specification guided message generation, in: 2021 International Conference on UK-China Emerging Technologies (UCET), IEEE. pp. 164–170.

Liang, H., Pei, X., Jia, X., Shen, W., Zhang, J., 2018. Fuzzing: State of the art. IEEE Transactions on Reliability 67, 1199–1218.

Lin, Z., Jiang, X., Xu, D., Zhang, X., 2008. Automatic protocol format reverse engineering through context-aware monitored execution., in: NDSS, pp. 1–15.

Liu, Q., Toffalini, F., Zhou, Y., Payer, M., 2023. Videzzo: Dependency-aware virtual device fuzzing, in: 2023 IEEE Symposium on Security and Privacy (SP), IEEE Computer Society. pp. 3228–3245.

Luo, Z., Yu, J., Zuo, F., Liu, J., Jiang, Y., Chen, T., Roychoudhury, A., Sun, J., 2023. Bleem: Packet sequence oriented fuzzing for protocol implementations .

Luo, Z., Zuo, F., Jiang, Y., Gao, J., Jiao, X., Sun, J., 2019. Polar: Function code aware fuzz testing of ics protocol. ACM Transactions on Embedded Computing Systems (TECS) 18, 1–22.

Luo, Z., Zuo, F., Shen, Y., Jiao, X., Chang, W., Jiang, Y., 2020. Ics protocol fuzzing: coverage guided packet crack and generation, in: 2020 57th ACM/IEEE Design Automation Conference (DAC), IEEE. pp. 1–6.

Maier, D., Bittner, O., Munier, M., Beier, J., 2022. Fitm: Binary-only coverage-guided fuzzing for stateful network protocols, in: Workshop on Binary Analysis Research (BAR).

Manès, V.J., Han, H., Han, C., Cha, S.K., Egele, M., Schwartz, E.J., Woo, M., 2019. The art, science, and engineering of fuzzing: A survey. IEEE Transactions on Software Engineering 47, 2312–2331.

Miller, B.P., Fredriksen, L., So, B., 1990. An empirical study of the reliability of unix utilities. Communications of the ACM 33, 32–44.

Munea, T.L., Lim, H., Shon, T., 2016. Network protocol fuzz testing for information systems and applications: a survey and taxonomy. Multimedia tools and applications 75, 14745–14757.

Narayan, J., Shukla, S.K., Clancy, T.C., 2015. A survey of automatic protocol reverse engineering tools. ACM Computing Surveys (CSUR) 48, 1–26.

Natella, R., 2022. Stateafl: Greybox fuzzing for stateful network servers. Empirical Software Engineering 27, 191.

Oehlert, P., 2005. Violating assumptions with fuzzing. IEEE Security & Privacy 3, 58–62.

Pan, G., Lin, X., Zhang, X., Jia, Y., Ji, S., Wu, C., Ying, X., Wang, J., Wu, Y., 2021. V-shuttle: Scalable and semantics-aware hypervisor virtual device fuzzing, in: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, pp. 2197–2213.

Pereyda, J., 2015. boofuzz: A fork and successor of the sulley fuzzing framework.

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

Postel, J., Reynolds, J., 1997. Instructions to RFC authors. Technical Report.

Qin, S., Hu, F., Ma, Z., Zhao, B., Yin, T., Zhang, C., 2023. Nsfuzz: Towards efficient and state-aware network service fuzzing. ACM Transactions on Software Engineering and Methodology .

Raffelt, H., Merten, M., Steffen, B., Margaria, T., 2009a. Dynamic testing via automata learning. International journal on software tools for technology transfer 11, 307–324.

Raffelt, H., Steffen, B., Berg, T., Margaria, T., 2009b. Learnlib: a framework for extrapolating behavioral models. International journal on software tools for technology transfer 11, 393–407.

Redini, N., Continella, A., Das, D., De Pasquale, G., Spahn, N., Machiry, A., Bianchi, A., Kruegel, C., Vigna, G., 2021. Diane: Identifying fuzzing triggers in apps to generate under-constrained inputs for iot devices, in: 2021 IEEE Symposium on Security and Privacy (SP), IEEE. pp. 484–500.

Ruiter, D., Joeri, 2015. Protocol state fuzzing of {TLS} implementations, in: 24th {USENIX} Security Symposium ({USENIX} Security 15), pp. 193–206.

Schumilo, S., Aschermann, C., Abbasi, A., Wörner, S., Holz, T., 2021. Nyx: Greybox hypervisor fuzzing using fast snapshots and affine types., in: USENIX Security Symposium, pp. 2597–2614.

Schumilo, S., Aschermann, C., Gawlik, R., Schinzel, S., Holz, T., 2017. kafl: Hardware-assisted feedback fuzzing for os kernels., in: USENIX Security Symposium, pp. 167–182.

Schumilo, S., Aschermann, C., Jemmett, A., Abbasi, A., Holz, T., 2022. Nyxnet: network fuzzing with incremental snapshots, in: Proceedings of the Seventeenth European Conference on Computer Systems, pp. 166–180.

Serebryany, K., 2015. libfuzzer–a library for coverage-guided fuzz testing. LLVM project .

Serebryany, K., 2017. Oss-fuzz-google's continuous fuzzing service for open source software, in: USENIX Security symposium, USENIX Association.

Shi, J., Wang, Z., Feng, Z., Lan, Y., Qin, S., You, W., Zou, W., Payer, M., Zhang, C., 2023. {AIFORE}: Smart fuzzing based on automatic input format reverse engineering, in: 32nd USENIX Security Symposium (USENIX Security 23), pp. 4967–4984.

Sklenar, P., 2011. The proxyfuzz project. URL: https://src.fedoraproject.org/rpms/proxyfuzz.

Somorovsky, J., 2016a. Systematic fuzzing and testing of tls libraries, in: Proceedings of the 2016 ACM SIGSAC conference on computer and communications security, pp. 1492–1504.

Somorovsky, J., 2016b. Systematic fuzzing and testing of tls libraries, in: Proceedings of the 2016 ACM SIGSAC conference on computer and communications security, pp. 1492–1504.

Song, D., Hetzelt, F., Kim, J., Kang, B.B., Seifert, J.P., Franz, M., 2020. Agamotto: Accelerating kernel driver fuzzing with lightweight virtual machine checkpoints, in: Proceedings of the 29th USENIX Conference on Security Symposium, pp. 2541–2557.

Sun, Y., Lv, S., You, J., Sun, Y., Chen, X., Zheng, Y., Sun, L., 2022. Ipspex: Enabling efficient fuzzing via specification extraction on ics protocol, in: Applied Cryptography and Network Security: 20th International Conference, ACNS 2022, Rome, Italy, June 20–23, 2022, Proceedings, Springer. pp. 356–375.

Tsankov, P., Dashti, M.T., Basin, D., 2012. Secfuzz: Fuzz-testing security protocols, in: 2012 7th International Workshop on Automation of Software Test (AST), IEEE. pp. 1–7.

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

Wang, Y., Jia, X., Liu, Y., Zeng, K., Bao, T., Wu, D., Su, P., 2020. Not all coverage measurements are equal: Fuzzing by coverage accounting for input prioritization., in: NDSS.

Wei, A., Deng, Y., Yang, C., Zhang, L., 2022. Free lunch for testing: Fuzzing deep-learning libraries from open source, in: Proceedings of the 44th International Conference on Software Engineering, pp. 995–1007.

Wondracek, G., Comparetti, P.M., Kruegel, C., Kirda, E., Anna, S.S.S., 2008. Automatic network protocol analysis., in: NDSS, Citeseer. pp. 1–14.

Xu, W., Kashyap, S., Min, C., Kim, T., 2017. Designing new operating primitives to improve fuzzing performance, in: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pp. 2313–2328.

You, W., Wang, X., Ma, S., Huang, J., Zhang, X., Wang, X., Liang, B., 2019. Profuzzer: On-the-fly input type probing for better zero-day vulnerability discovery, in: 2019 IEEE symposium on security and privacy (SP), IEEE. pp. 769–786.

Yun, J., Rustamov, F., Kim, J., Shin, Y., 2022. Fuzzing of embedded systems: A survey. ACM Computing Surveys 55, 1–33.

Zalewski, M., 2017. American fuzzy lop.

Zhang, C., Lin, X., Li, Y., Xue, Y., Xie, J., Chen, H., Ying, X., Wang, J., Liu, Y., 2021. {APICraft}: Fuzz driver generation for closed-source {SDK} libraries, in: 30th USENIX Security Symposium (USENIX Security 21), pp. 2811–2828.

Zhao, H., Li, Z., Wei, H., Shi, J., Huang, Y., 2019. Seqfuzzer: An industrial protocol fuzzing framework from a deep learning perspective, in: 2019 12th IEEE Conference on software testing, validation and verification (ICST), IEEE. pp. 59–67.

Zhu, X., Wen, S., Camtepe, S., Xiang, Y., 2022. Fuzzing: a survey for roadmap. ACM Computing Surveys (CSUR) 54, 1–36.

Zuo, F., Luo, Z., Yu, J., Chen, T., Xu, Z., Cui, A., Jiang, Y., 2022. Vulnerability detection of ics protocols via cross-state fuzzing. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 41, 4457–4468.

Zuo, F., Luo, Z., Yu, J., Liu, Z., Jiang, Y., 2021. Pavfuzz: State-sensitive fuzz testing of protocols in autonomous vehicles, in: 2021 58th ACM/IEEE Design Automation Conference (DAC), IEEE. pp. 823–828.