Fuzzers for Stateful Systems: Survey and Research Directions

CRISTIAN DANIELE, SEYED BEHNAM ANDARZIAN, and ERIK POLL, Radboud University, The Netherlands

Fuzzing is a very effective testing methodology to find bugs. In a nutshell, a fuzzer sends many slightly malformed messages to the software under test, hoping for crashes or weird system behaviour. The methodology is relatively simple, although applications that keep internal states are challenging to fuzz. The research community has responded to this challenge by developing fuzzers tailored to stateful systems, but a clear understanding of the variety of strategies is still missing. In this paper, we present the first taxonomy of fuzzers for stateful systems and provide a systematic comparison and classification of these fuzzers.

CCS Concepts: • **Security and privacy → Software security engineering**.

Additional Key Words and Phrases: stateful fuzzing, state model, active learning

1 INTRODUCTION

With fuzzing (or fuzz testing) a system is fed a large collection of automatically generated inputs to find vulnerabilities, in particular memory corruption bugs. Fuzzing is a great way to improve software security: it can find lots of bugs with relatively little effort. The idea of fuzzing goes back to the late 1980s [42] but interest in fuzzing exploded in the 2000s. Major game changers here have been the advent of *white-box fuzzing* using symbolic (or more precisely, concolic) execution in the SAGE fuzzer [22] and the advent of *grey-box fuzzing*, also known as *evolutionary fuzzing*, pioneered in the fuzzer AFL [67], where code execution paths are monitored to guide the generation of inputs. Both approaches avoid the need for the user to provide an explicit grammar describing the input format.

Traditionally, most fuzzers target *stateless* systems, where the system under test takes a single input, say a JPEG image; the fuzzer then tries many possible inputs, including many malformed ones. This survey focuses on the fuzzing of *stateful systems*. The motivation that drove us to write this paper is obvious. In the last few years, the scientific community has shown great interest in fuzzing. This is demonstrated by the plethora of papers and tools on the topic and by the numerous papers and tools about stateful fuzzing. However, it is hard to understand the relations and differences between these tools. By a stateful system, we mean a system that takes a *sequence* of messages as input, producing outputs along the way, and where each input may result in an internal state change. Most protocols, including most network protocols, are stateful. So when people refer to 'network fuzzing' [25] or 'network protocol fuzzing' [24, 27], they are usually talking about the fuzzing of stateful systems. Obviously, fuzzing stateful systems is harder than fuzzing stateless systems, as the internal state changes increase the state space that we try to explore: it may be hard for a fuzzer to reach 'deeper' states. Indeed, fuzzing stateful systems is listed as one of the challenges in fuzzing by Boehme et al. [12].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2023 Association for Computing Machinery.
Manuscript submitted to ACM

The existing survey papers about fuzzing [40, 69, 71], despite giving a good overview of fuzzers in general, do not investigate the issues of fuzzing stateful systems in any depth, even though these surveys do include some fuzzers for stateful systems.

To the best of our knowledge, this paper provides the first comprehensive literature review about stateful fuzzing. It:

(1) provides a precise taxonomy of components and categories of stateful fuzzers;
(2) raises questions about the approaches these fuzzers take;
(3) discusses their differences, their pros and cons;
(4) sheds light on challenges and future directions.

In this survey, we included all the papers presented by the existing survey papers in the field, and all the recent *generic* stateful fuzzers. Therefore, we excluded all the fuzzers tailored to a specific protocol (like [1], [33], [29], [68], [48], [36]) that would require major modification to work on other protocols.

The outline of this paper is as follows. Section 2 defines some basic concepts and terminology for discussing stateful systems. Section 3 presents the traditional classification for fuzzers used in the literature and discusses some of the differences between fuzzing stateless and stateful systems. Section 4 presents our taxonomy for fuzzers for stateful systems, classifies existing fuzzers and compares the various approaches.

2 CONCEPTS AND TERMINOLOGY

By a stateful system we mean a system that takes a sequence of messages as input, potentially producing outputs along the way, and where each input may result in an internal state change. To avoid confusion, we reserve the term *message* or *input message* for the individual input that the System Under Test (SUT) consumes at each step and the term *trace* for a *sequence* of such messages that make up the entire input.

We use the term *response* for the output that the SUT produces along the way. In the case of a synchronous protocol, there is usually one response after each input message. In this case, the state machine describing the input-output behaviour will be a Mealy machine.

The input language of a stateful system consists of two levels[1]: 1) the language of the individual messages, which we will refer to as the *message format*, and 2) the language of traces, built on top of that. A description or specification of such an input language will usually come in two parts, one for each of the levels: for example, a context-free grammar for the message format and a finite state machine describing sequences of these messages. We will call the latter the *state model* or, if it is described as a state machine, the *protocol state machine*.

The state model usually involves a notion of *message type* where different types trigger different transitions. So then the set of message types is the input alphabet of any protocol state machine. This alphabet will typically abstract away from payloads inside messages. In some protocols, for example, the message type is determined by an instruction byte in a header. The output messages of the state models are often distinguished between *non-error* responses and *error* responses. A more fine-grained distinction of error responses could be based on the *error code*. Fuzzers that try to infer the protocol state machine (using active or passive learning, as discussed in Section 4) may require the user to provide abstraction functions to map concrete messages to message types. There can be two abstraction functions, one for the input messages and one for the responses. For example, the abstraction function for the inputs could map inputs to instruction bytes, and the abstraction function for outputs could map outputs to error codes.

[1]For text-based protocols, as opposed to binary protocols, there may even be a third level, namely the character set or character encoding used, but none of the fuzzers studied use that.

In some protocols, the format of input messages and responses is very similar. For fuzzing it is then a good strategy to also include responses as inputs as this may trigger unexpected behaviour: client and server are likely to share a part of the codebase and one may then accidentally process messages only intended for the other.[2]

We use the term *protocol state* to refer to the abstract state of an SUT that determines how future input messages are handled. The SUT will have a concrete program state, which is related to this protocol state, but which usually carries much more detail.

The term 'state' can quickly become overloaded: not only the SUT has a state, even if it implements a stateless protocol, but the fuzzer itself also has a state. We use the term *stateful fuzzing* to refer to the fuzzing of stateful systems, but we avoid the term 'stateful fuzzer' as even a fuzzer for stateless systems will have an internal state.

There are basically two ways for an SUT to record the protocol state: 1) it can keep track of the protocol state using program variables that record state information or 2) the state can be more implicitly recorded by the program point where the execution is (and possibly the call stack). Of course, these two ways can be combined. For fuzzers that use a grey- or white-box approach (discussed in more detail later), the difference can be important: white-box approaches that observe the values of program variables will work better for 1) than for 2), whereas grey-box approaches that observe execution paths will work better for 2) than for 1).

Stateless vs stateful systems. There is not always a clear border between stateless systems and stateful systems. For example, if a system has a memory leak, then it is (unintentionally) stateful since we can imagine every new memory allocation as a transition from one state to another. Its behaviour will then appear to be stateless for a very long time, namely until it runs out of memory and crashes or misbehaves.

More generally, we can think of any stateful system that takes a sequence of messages as input as a stateless system which takes that whole trace as a single input. Conversely, we can view a stateless system that takes a single complex value as input as a stateful system that processes a sequence of smaller inputs, say bits or bytes. For instance, a stateless program that processes JPEGs, which will always process the same JPEG in the same way, can be viewed as a *stateful* program that takes a sequence of bytes as input, and which will process the same byte in different ways depending on where in the JPEG image it occurs. However, a fundamental difference between a stateful system and a stateless system viewed as a stateful one in this way is that the former will typically provide some observable output after processing each input. Another difference is knowing that inputs are made up of smaller messages can help in making useful mutations, by swapping the order of messages or repeating messages.

Some stateless systems can process sequences of inputs like stateful systems do, but then the idea is that previous inputs do not have any effect on how subsequent inputs are handled. Some fuzzers use this possibility to avoid the overhead of having to restart the SUT between inputs. This is called *persistent fuzzing*. This does then involve a sequence of inputs, but it is the polar opposite of stateful fuzzing: the goal is not to explore the statefulness of the SUT, but rather it presupposes that there is no statefulness.

3 BACKGROUND

This section discusses existing classifications of fuzzers used in the literature, as this provides a starting point for our classification of stateful fuzzing, and makes some initial observations about how and why fuzzing stateful systems is different.

[2]CVE-2018-10933 is an interesting example of a bug of this kind in Libssh: if the message that is normally sent *from* the server to the client to confirm that the client has successfully authenticated is sent *to* the server, a malicious client could skip the whole authentication phase.

There are some good survey papers about fuzzing, but none pay much attention to issues specific to stateful fuzzing. The survey by Zhu et al. [71] classifies close to 40 fuzzers. Only two of these, namely AFLNet [50] and de Ruiter et al. [16] specifically target stateful systems. The more extensive survey by Manes et al. [40] categorises over 60 fuzzers. Thirteen of these are fuzzers for 'network protocols' so presumably these are fuzzers geared to fuzzing stateful systems; all of these 13 fuzzers are black-box fuzzers. One section in this survey (Section 5.1.2) discusses the statefulness of the SUT: here it discusses the inference of state machine models.

The only attempt at classifying fuzzers for stateful systems that we are aware of is given in the paper by Yu et al. about SGPFuzzer [66]: Table 1 in this paper lists twelve other fuzzers for stateful systems (or "protocol fuzzers" in the terminology used in the paper), namely AutoFuzz [24], AspFuzz, SECFuzz [60], Sulley [3], BooFuzz [49], Peach [4], SNOOZE [8], PULSAR [21], tlsfuzzer [16], DTLS-fuzzer [19], ICSFuzz [61], and NLP-fuzzer [64]. The authors identify four challenges based on the shortcomings of these 12 fuzzers and then design SGPFuzzer to address these. Unfortunately, the definitions of the features used for the comparison are left implicit and the comparison fails to point out some very fundamental differences between tools. For instance, the man-in-the-middle nature of AutoFuzz and SecFuzz comes with an important inherent limitation, namely that the order of messages cannot be fuzzed (as we discuss in Section 4.6).

3.1 Existing classifications of fuzzers

The standard classification of fuzzers in the literature (e.g. [12, 40, 71]) distinguishes *black-box*, *grey-box* and *white-box fuzzers*, where the black-box fuzzers are sub-divided into *grammar-based* fuzzers and *mutation-based fuzzers*. Even though this classification is fairly standard, the terminology varies between papers, and there are combinations of approaches that do not neatly fit into one of these categories. We discuss this classification in more detail below.

Black-box fuzzers. As the name suggests, black-box fuzzers only observe the SUT from the outside. To stand any change of producing interesting inputs, black-box fuzzers require some knowledge of the input format. One approach here, taken by *generation-based* aka *grammar-based* fuzzers, is that the fuzzer is given knowledge of the input format in the form of grammar or model. A downside of such fuzzers is the effort required to produce such a model or grammar. Another approach, taken by *mutation-based fuzzers*, is that the fuzzer is supplied with a set of sample inputs which are then mutated in the hope of discovering bugs.

Most grammar-based fuzzers allow users to supply grammar for an arbitrary input format (or protocol) they want to fuzz [23], but there are also grammar-based fuzzers which have a specific input format hard-coded in them, such as the KiF fuzzer [2] for the SIP protocol. There are also commercial fuzzers for fuzzing specific protocols, for example, Codenomicon's DEFENSICS fuzzer [5] (since acquired by Synopsys), which grew out of the PROTOS [30] project at the University of Oulu in Finland started in 1999.

Some fuzzers combine the generation-based and mutation-based approaches. A grammar-based fuzzer should not only produce grammatically correct inputs but also malformed ones; this either has to involve some form of mutation or the grammar has to be too 'loose' to begin with. Conversely, a mutation-based fuzzer can be given some knowledge about the input format, for instance by providing a list of keywords or byte values with a special meaning, which the fuzzer can use in mutations in the hope of generating interesting mutations.

[3] https://github.com/OpenRCE/sulley
[4] https://wiki.mozilla.org/Security/Fuzzing/Peach
[5] https://www.codenomicon.com/defensics/

White-box fuzzers. White-box fuzzers require access to the (source or binary) program code and analyse that code in order to provide interesting inputs. With access to the code, it is possible to see which branches there are to then construct inputs that trigger particular branches. Typically white-box fuzzers use symbolic or concolic execution to construct interesting test cases. Microsoft's SAGE fuzzer [22] is the best-known example of this class.

Grey-box fuzzers. Grey-box fuzzers occupy the middle ground and can observe some aspects of the SUT as it executes and use this feedback to steer the fuzzer. This is also called *evolutionary fuzzing* as the inputs will gradually evolve into more interesting mutations. Grey-box fuzzers can be considered as a special kind of mutational fuzzers because the evolution always involves mutation. Grey-box fuzzers are sometimes called smart mutational fuzzers; the black-box mutational fuzzers that lack a feedback mechanism to guide the evolution are then called dumb mutational fuzzers.

Grey-box fuzzers often require some instrumentation of the code or running the code in some emulator. The approach has been popularised by the fuzzer AFL, which observes the code execution path – or, more precisely, the branches are taken in the execution – to see if some input mutation results in new execution paths. Grey-box fuzzers that observe the execution path in this way are also called *coverage-guided greybox fuzzers (CGF)*. This approach has proved to be very successful, providing much better coverage than 'dumb' mutational fuzzers but without the work of having to provide a grammar.

All fuzzers that involve mutation – dumb mutational fuzzers, evolutionary fuzzers, but also grammar-based fuzzers that use mutation – can be parameterised by *mutation primitives*, for instance random bit flips, repeating sub-sequences of the input, or inserting specific bytes, characters, or keywords.

Alternative classifications. Instead of classifying fuzzers into white-, grey- and black-box, an orthogonal classification is to consider the kind of applications targeted and the kind of input this involves [40]: e.g. some fuzzers are geared towards fuzzing file formats, others to network traffic, and others still to web applications or OS kernels. Fuzzers for web applications are often called 'scanners'. There is a relation between this classification and statefulness: applications that take a file as input are usually not stateful, whereas applications that implement a network protocol usually are.

3.2 White-, grey-, and black-box fuzzing for stateful systems

For stateful systems some basic observations about the classification into white-, grey-, and black-box can be made:

- The terms 'grey-box fuzzing' and 'evolutionary fuzzing' are often used as synonyms, but for stateful systems, they are not: for a stateful system the evolution of inputs can also be steered by the outputs of the SUT, which is then evolutionary but black-box. This is a key difference between a stateful and a stateless system: the response that the SUT produces is an observation that the fuzzer can make without any instrumentation of the code.
- For grammar-based fuzzers it does not really matter if the SUT is stateful or not: the grammar describing the system can describe both the message format and the protocol state machine.
- For dumb mutational black-box fuzzers it also does not matter that much if the SUT is stateful or not. Of course, it helps if the fuzzer is aware of the fact that inputs are traces of messages so that it can try swapping the order of messages, removing messages or repeating messages as interesting mutations. The same goes for any grammar-based fuzzer, which should also try re-ordering, repeating or dropping messages as interesting corruptions of the grammar.
- The techniques used in grey-box and white-box fuzzing to observe program execution may shed some light on the state that the SUT is in. However, as discussed in Section 2, there are different ways in which the SUT can

record protocol state: the state can be recorded in some program variables, it can be recorded in the program point that the SUT is in (and possibly the call stack), or a combination of these. The way in which the SUT does this can make a difference in how well some grey- or white-box technique can observe the protocol state.
The statefulness of the SUT may complicate observation for a grey- or white-box fuzzer. For white-box fuzzers that rely on symbolic or concolic execution input, the statefulness of the SUT is obviously a serious complication: a symbolic execution of a program handling a single input can already be tricky, and the execution for a sequence of symbolic inputs will be an order of magnitude harder. For example, if the SUT implements some loop to handle incoming messages then that loop would have to be unwound.

3.3 Bug detection for stateful systems

In addition to a mechanism to generate inputs, a fuzzer also requires some mechanism to observe the SUT to detect if an input triggered a bug. Typically fuzzers look for memory corruption bugs that crash the SUT using sanitisers such as ASAN (AddressSanitizer) [55], MSAN (MemorySanitizer) [58], or older less efficient sanitisers such as Valgrind [46]. When fuzzing programs written in memory-safe languages, e.g. when fuzzing Java programs with Kelinci [32], instead of looking for memory corruption bugs we can look for uncaught runtime exceptions; even if these bugs cannot be exploited in the way memory corruption can, they can still lead to Denial of Service problems.

A type of bug that is specific to stateful systems is deviations from the correct state behaviour: if a system is expected to behave in a specific way, for instance by only responding to certain inputs after authentication, then a deviation from this behaviour may be a security issue. For security protocols such as TLS, any deviations from the expected state behaviour are highly suspicious: security protocols are very fragile and even small deviations may break the security guarantees the protocol aims to provide. Unlike the detection of memory corruption bugs, this cannot be totally automated: it either requires a specification of expected behaviour (or, conversely, of unwanted behaviour), for instance with a state machine or in temporal logic, or it requires some manual analysis of the state behaviour inferred by the fuzzer.

4 FUZZERS FOR STATEFUL SYSTEMS

With the explosion of papers about fuzzers and new fuzzing tools, it is hard to get a comprehensive view of all the tools that exist (see e.g. the fuzzing survey website[6] or the Welcome to Awesome Fuzzing GitHub page[7]).

We relied on the existing surveys ([40, 69, 71]) to identify 33 stateful fuzzers and define seven categories of fuzzers for stateful systems. To the best of our knowledge, these categories cover the entire range of stateful fuzzers since every fuzzer we came across in the literature falls into one of these categories.

Table 1 summarises the main characteristics of each category. Some categories can be considered as a combination or sub-category of other categories, as illustrated in Fig. 1. Before we discuss each category in more detail in the sections below, we first discuss common ingredients involved in some of them:

- Some fuzzers require sample traces as inputs, either a few traces to act as seeds to further mutation, or many traces so that grammar can be inferred or machine model can be trained.
- Many fuzzers involve some form of grammar. This can be a grammar describing just for the message format, a grammar describing just the protocol state machine, or both. Some fuzzers require such grammars as inputs, but others can provide grammars that are inferred during the fuzzing as output.

[6] https://fuzzing-survey.org
[7] https://github.com/secfigo/Awesome-Fuzzing

- Many fuzzers use some form of learning to infer information about the message format, the protocol state machine, or both. Evolution can be regarded as a form of learning because it produces and uses new knowledge about the input format, even though this knowledge is (usually) not expressed in the form of a regular expression, state machine, or context-fee grammar.
  Evolution is a form of *active learning* because it involves interaction with the SUT, where the next input we try can depend on the outcome of previous tests. Some fuzzers use forms of *passive learning* instead of (or in addition to) such active learning. By this we mean approaches where information about the input format is inferred *after* a set of traces has been collected, so without interactively trying new experiments.
  There is a long line of research into algorithms for inferring formal language descriptions, either actively or passively, which includes research into *regular inference* and *grammatical inference* that focus specifically on inference of regular expressions and context-free grammar respectively. Research in this field is presented at the bi-annual International Conference on Grammatical Inference (ICGI) and there are entire textbooks on the subject (e.g. [15]). For active learning of a protocol state machine, an algorithm that can be used is L* [3] or one of its improvements, e.g. the TTT algorithm used in LearnLib [28] or AALpy [44]. For the passive learning of protocol state machines, some fuzzers use ad-hoc solutions even though there are generic tools for passive learning of state machines [62]. For instance, the fuzzer by Hsu et al.[26] uses an algorithm called partial finite state automaton reduction. An important limitation of some learning algorithms, notably L* and its improvements, is that they cannot deal with the non-deterministic behaviour of the SUT, as it will cause the algorithm to diverge.
  A very different form of (passive) learning used by some fuzzers is *machine learning*. This does not produce knowledge in a nice concrete format like a regular expression, finite state machine, or context-free grammar. Also, it typically requires more samples. Still, possible advantages are that there are many existing machine learning approaches that can be used and that these may cope more easily with the non-deterministic behaviour of the SUT.

Below we first give a general description of the seven categories of fuzzers. In the subsequent sections, we discuss each category in more detail:

(1) ***Grammar-based fuzzers*** These fuzzers use a grammar – provided by the user – to craft slightly malformed messages able to reach deep states. Any grammar-based fuzzer can be used to fuzz stateful systems without any special adaptions. The grammar that is supplied to the fuzzer will have to describe the two levels of the input language, with some rules of the grammar describing the message format and some rules describing the protocol state machine. Apart from that, no change in the fuzzer itself is needed, except that course, swapping, dropping and repeating messages are useful – if not essential – mutation strategies for the fuzzer to include. But for a stateless SUT where the format of the inputs is quite complex it can also be useful to include swapping, dropping and repeating sub-components of inputs as mutation strategies.

(2) ***Grammar-learner fuzzers*** Whereas the grammar-based fuzzers require the users to provide a grammar, these fuzzers are able to extract a grammar through passive [43] or active learning tools [23]. They can be considered as the sequential composition of two tools: a *grammar extractor* that infers the grammar from a set of sample traces (using so-called passive grammatical inference) and a *grammar-based fuzzer* that then does the actual fuzzing using the inferred grammar.

As for the grammar-based fuzzers, for the grammar-learner fuzzers the statefulness of the SUT does not make any fundamental difference: it only means that the grammar will have two levels. So grammar-learner fuzzers can be applied to stateless as well as stateful SUTs.

(3) ***Evolutionary fuzzers*** These fuzzers basically take the same approach as stateless evolutionary fuzzers such as AFL: they take some sample traces as initial input and mutate these using a feedback system to steer the mutation process. Of course, evolutionary fuzzers for stateful systems should be aware that an input trace is a sequence of messages and should include swapping, omitting or repeating these messages as mutation strategies.

A difference between stateful and stateless systems when it comes to evolutionary approaches of fuzzing is that the responses that a stateful SUT provides after individual messages can be used in the feedback to guide the evolution, as mentioned before in Section 3.1.

(4) ***Evolutionary grammar-based fuzzers*** These fuzzers use both a grammar provided to the user to generate (correct, protocol-compliant) traces and an evolution mechanism to mutate these traces. We can think of them as evolutionary fuzzers that use a grammar instead of a set of sample input traces to provide the initial traces that will be mutated. We can also think of them as grammar-based fuzzers that include a feedback mechanism to steer the evolution of mutations. So in Fig. 1 they are the intersection of the evolutionary fuzzers and the grammar-based fuzzers.

(5) ***Evolutionary grammar-learner fuzzers*** This is the most complex category of fuzzers. These tools all use some form of grammar to describe the protocol state machine; one also uses a grammar to describe the message format. They involve two feedback mechanisms to steer two forms of evolution: (i) one for the mutation of individual messages, in the style of conventional evolutionary fuzzers like AFL, and (ii) another for the mutation of sequences, which then infers a protocol state machine. The second form of evolution is based on the response that the SUT provides as feedback, so it is black-box.

The final two categories of fuzzers are very different from the five above:

(6) ***Man-in-the-Middle fuzzers***: These fuzzers sit in the middle between the SUT and a program interacting with it and modify messages going to the SUT, as illustrated in Fig. 7. Responses coming back from the SUT are left untouched.

The Man-in-the-Middle fuzzers can take a dumb mutational approach to modify the messages, but they may leverage a protocol specification (automatically inferred or given as input) to modify messages.

(7) ***Machine learning fuzzers*** These fuzzers use a Machine Learning (ML) model trained on a large set of input traces. The model outputs slightly different — hopefully malicious — mutated traces. Machine learning methods used by these fuzzers include Seq2seq [59] and Seq-GAN [65].

The Machine learning fuzzers are similar to the grammar learner fuzzers in that they require a set of sample traces as input that is then used to infer a model of the input format which is then the basis for the fuzzing. The key difference is that for the grammar learner fuzzers, this model is a grammar, whereas for these machine learning fuzzers the model is an ML model.

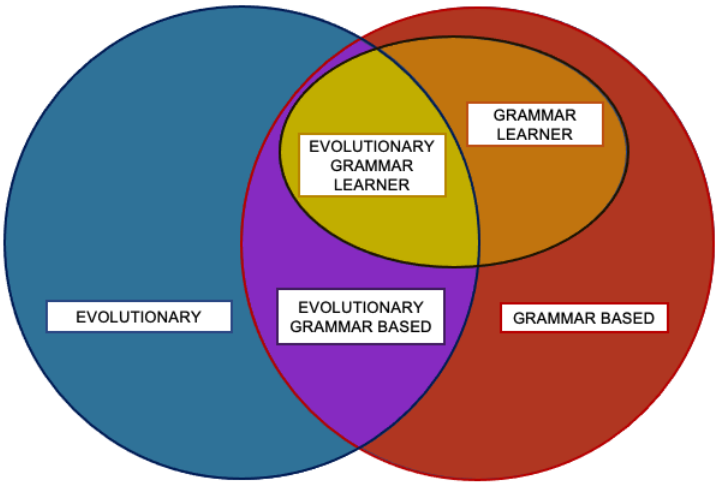


Fig. 1. We identified seven categories of fuzzers but only five of them involve a grammar, an evolutionary feedback mechanism, or both.



Fig. 2. Grammar-based fuzzers use a grammar provided by the user to craft slightly malformed messages to send to the SUT.

4.1 Grammar-based fuzzers

Table 2 lists the grammar-based fuzzers. As shown in Fig. 2, these fuzzers use a grammar provided by the user. In the case of a stateful SUT, this grammar should describe the syntax of the messages and the protocol state machine. For some fuzzers, e.g. Peach[11] and SNOOZE [8], this grammar is supplied in some XML format.

The obvious downside of these fuzzers is that they require an accurate grammar. Producing one can be a lot of work and it can be challenging and error-prone. Not all errors will matter – or matter equally: if the grammar is a bit too 'loose' this is not much of a problem, but if the grammar omits interesting parts of the language it may be, as this would mean that the fuzzer will not explore that part of the language. Ideally the documentation of the SUT, or the specification of the protocol it implements, simply provides a formal grammar that can be used. However, this will often not be the case: documentation or specifications may be unclear or incomplete. That the SUT is stateful does not make a difference here, Still, in earlier research [51] we found that documentation is more likely to include a clear (but informal) specification for the message format than for the protocol state machine. The protocol state machine is often – very poorly – specified in prose scattered throughout specification documents.

[11]Here we mean the community edition, available at https://gitlab.com/gitlab-org/security-products, which lacks some features of Peach Fuzzer Professional.

Some grammar-based fuzzers, e.g. SNOOZE [8] and Sulley, come with grammars for some standard protocols so that for these the hard work to produce a grammar has already been done for the user, but for other protocols, the user still has to do it themselves.

Sec.	Category	Input required	Generate state model	Human interaction	Pros	Cons
4.1	Grammar-based	Grammar	No	No	- Able to reach deep states	- Cannot use coverage feedback - Require a grammar
4.2	Grammar-learner	Sample traces	Yes	Yes / No	- Infer the grammar	- Cannot use coverage feedback - Traces need to be comprehensive
4.3	Evolutionary	Sample traces	No	Yes / No	- Use coverage feedback	- Cannot use a grammar - Need to recompile the code
4.4	Evolutionary grammar-based	Grammar	No	No	- Use coverage feedback - Use grammar	- Need to recompile the code - Require a grammar
4.5	Evolutionary grammar-learner	Sample traces	Yes	No	- Use coverage feedback - Infer the grammar	- Need to recompile the code - Traces need to be comprehensive
4.6	Man-in-the-Middle	Live traffic	No	Yes / No	/	- Cannot use coverage feedback
4.7	Machine learning	Many sample traces	No	No	- Easy to use	- Cannot use coverage feedback - Traces need to be comprehensive - Cannot change the order of the messages

Table 1. The seven categories of fuzzers with their main characteristics. Human interaction refers to manual code or grammar annotation. Pros and cons are general for the category, but specific fuzzers might have different pros/cons.

Fuzzer	Based on	Mutates	Open source
Peach [8]	-	- Message	✓
SNOOZE [8]	-	- Message	✗
PROTOS[30]	-	- Message	✗
Sulley [9]	-	- Message	✓
BooFuzz [49]	Sulley	- Message	✓
Fuzzowski [10]	BooFuzz [49]	- Message - Trace	✓
AspFuzz [34]	-	- Message - Trace	✓

Table 2. Grammar-based fuzzers



Fig. 3. Grammar learner fuzzers use a grammar automatically inferred by a grammar extractor to craft slightly malformed messages to send to the SUT.

4.2 Grammar learner fuzzers

Table 3 presents the grammar learner fuzzers. These fuzzers require some sample traces or a set of messages to infer the state model, and operate in two phases:

(1) *grammar learning* phase: they use a passive or active *grammar learner* to infer the grammar as much as possible similar to the real one;
(2) *fuzzing* phase: they do the actual fuzzing using that inferred grammar, just like a grammar-based fuzzer would do. In principle, it is possible to use any of the grammar-based fuzzers discussed in the previous section in this phase.

The fuzzers proposed by Doupé et al. [17], LearnLib [3] and StateInspector [41] only implement the *grammar learning* phase. The tools themselves do not mutate individual messages but only try combinations of a fixed set of input messages to infer the state machine.

The fuzzers in Table 3 not only uses different algorithms for grammar learning but also tries to infer different aspects of the input format:

- PULSAR [21] infers both the message format and a protocol state machine, passively, from observed traffic. The learning techniques it uses are the ones developed earlier for the PRISMA fuzzer [35]. These can also infer rules for dependencies between messages, such as increasing sequence numbers.

Fuzzer	Learns	Based on	Input needed	Open source
PULSAR [21]	- State model - Message fields	- Passive learning (using PRISMA [21])	- Traces	✓
Hsu et al.[26]	- State model	- Passive learning (using partial finite state automaton reduction [26])	- Message field specification - Traces	✗
GLADE$^+$ [9]	- Message fields	- Active learning (using GLADE [9])	- Set of messages	✓
Doupé et al. et al.[17]	- State model	- Active learning (using a state-aware crawling approach[17])	- Set of messages	✓
LearnLib[3]	- State model	- Active learning (using L* [3], TTT [28], ADT [20], Discrimination Tree [31])	- Set of messages	✓
StateIspector [41]	- State model	- Active learning (using a memory monitoring approach [41])	- Set of messages	✓

Table 3. Grammar learner fuzzers

As the authors note, the approach relies on the completeness of the set of observed network traces and will be unable to model protocol paths not included in this traffic.

- The fuzzer proposed by Hsu et al. [26] cannot infer the message format: it only infers a protocol state machine. The tool requires that the message format is known; in fact, it needs an (un)parser for the message format to be supplied. The protocol state machine is then inferred from observed traffic – i.e. using passive learning – using a new algorithm they introduce. Once this state machine is inferred, the SUT can be fuzzed. At this point, a collection of mutation primitives is used, including mutations to mutate individual messages and mutations to reorder messages in the input trace.
  The first phase of this fuzzer, i.e. inferring a protocol state machine given a known message format, is very similar to what tools like LearnLib [52] do. However, it uses passive learning, whereas LearnLib uses active learning with a variant of L*. Hsu et al. report that they also tried active learning for this initial phase, using a variant of L*, as they also did in earlier work [56], but abandoned that approach because of 1) the difficulty in constructing concrete messages that active learning requires and 2) it being inefficient and not learning an accurate model.
- GLADE [9] infers context-free grammars which can infer both the message format and the protocol state machine. This inference uses active learning, so it does involve some fuzzing of the SUT. However, GLADE has been extended to be used as a front-end for a grammar-based fuzzer. In Table 3 we refer to this extension as GLADE$^+$ to avoid confusion.
  The algorithm used by GLADE$^+$ is shown to have better results than the active learning algorithms L* [3] and RNPI [47] for the case studies tried out by the makers [9]. The results of GLADE$^+$ are also compared with AFL for some of these case studies. However, the case studies are not typical stateful protocols but include interpreters for Python, Ruby and JavaScript. As AFL is best at fuzzing binary formats, it is maybe not that surprising that GLADE$^+$ beats AFL here.
- The tool proposed by Doupé et al. cannot infer the message format. The fuzzer uses an ad-hoc algorithm developed for the tool: it is a fuzzer for web applications, so the response of the SUT is a web page, and the tool analyses these web pages for similarity in an attempt to crawl the entire website.
- LearnLib infers the state model by observing the output of the SUT, in a black box fashion. It can use one of several algorithms for inferring the state model: ADT [20], Discrimination Tree [31], L* [3] and TTT [28], an improvement of L*.
- StateInspactor, unlike LearnLib, uses a grey box approach to monitor the content of the memory and then infer the state model. The approach is divided into two phases: first, the tool analyses snapshots of the SUT to identify the memory locations that define the protocol state; secondly, it infers the state model by examining the memory content after different inputs.

4.3 Evolutionary fuzzers

Table 4 presents the evolutionary fuzzers. As shown in Fig. 4, these use feedback to guide the mutation of inputs. This feedback can use different types of observation, namely the five options listed below or a combination:

F1 *Response.* Some fuzzers use the response of the SUT. This is the only type of observation that can be done in a black-box fashion.

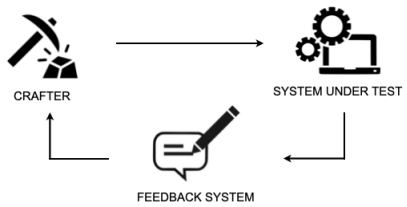


Fig. 4. Evolutionary fuzzers use feedback from the SUT to steer the generation of the messages to send to the SUT.

F2 *Coverage*. Some fuzzers observe branch coverage in the style of AFL, i.e. using a bitmap to observe branches taken during execution. This is a greybox approach that either requires re-compilation to instrument the code or running code in some emulator, just like AFL does.

F3 *Branches*: Some fuzzers observe branch coverage not by observing all branches as AFL does, but by observing specific branches that are manually marked as interesting to the user. This is a white-box approach and requires manual annotation of code by the user.

F4 *Variables*: Some fuzzers observe the value of specific program variables. This is a white-box approach and requires manual annotation of code by the user. The idea is that the program variables observed record information about the protocol state.

F5 *Memory*: Instead of observing specific individual program variables, one fuzzer observes memory segments: it takes snapshots of memory areas to see if inputs affect these. The idea is that changes in the memory signal change the protocol state. The only fuzzer using this, StateAFL, is not an evolutionary fuzzer but one of the more complicated evolutionary grammar learners, so it is discussed in Section 4.5.

Fuzzer	Feedback system	Based on	Input needed	Open source
nyx-net [54]	- Coverage	- AFL	- Target binary - Protocol specification - Seed inputs (optional)	✓
FitM fuzzer [39]	- Coverage	- AFL	- Client binary - Server binary - Seed inputs	✓
SNPSfuzzer [37]	- Coverage	- AFL	- Target binary - Seed inputs	✓
Chen et al. [14]	- Coverage - Branches	- AFL - Manual code annotation	- Target binary - Seed inputs	✗
SGFuzz [7]	- Coverage - Variables	- AFL - Automatic code annotation	- Target binary - Seed inputs	✓
IJON [5]	- Coverage - Variables	- AFL - Manual code annotation	- Target source code	✓

Table 4. Evolutionary fuzzers

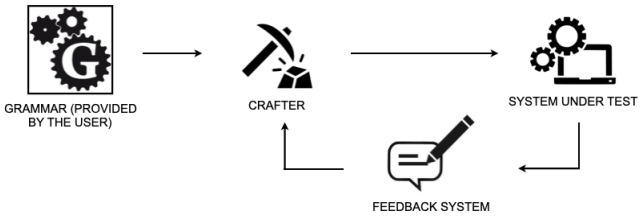


Fig. 5. Evolutionary grammar-based fuzzers use both feedback from the SUT and a grammar provided by the user to craft slightly malformed messages to send to the SUT.

All the evolutionary fuzzers in Table 4 are based on AFL, so all of them at least observe branch coverage in the style of ALF (i.e. F2, *Coverage*) to steer the evolution, but some tools use an additional feedback mechanism on top of this.

Regarding F3: the fuzzer by Chen et al. allows the user to mark some specific branches in the code. The idea is that taking these marked branches is an indication of the SUT moving to a different protocol state. Given that the AFL instrumentation already observes branch coverage, it is somewhat surprising that additional observation of selected branches improves the performance of the fuzzer. The fuzzer not only observes if these branches are taken in execution, but when this happens it effectively starts a new AFL session for this specific state (i.e. using a new bitmap for recording branches and creating a new queue of messages to mutate). So whereas AFL and all the other AFL-like evolutionary fuzzers in Table 4 maintain a single bitmap to record which branches have been taken, the fuzzer by Chen et al. has one such bitmap for each of the marked branches. This allows it to learn different strategies for generating test cases for different protocol states. Intuitively this makes sense: messages in different stages of a protocol may have different formats, so learning different mutation strategies, each tailored to a specific protocol state, can improve the fuzzing.

Regarding F4: IJON [5] observes specific program variables during the fuzzing. The user has to mark these in the source code. The idea is that the user marks variables that record information about the protocol state. SGFuzz is an improvement of this: instead of the user having to annotate code to specify which program variables record interesting state information, the fuzzer automatically infers which program variables have an enumeration type, and it assumes that all these program variables record state information.

As discussed in Section 2, there are different ways in which the SUT can record its protocol state. If the protocol state is recorded in program variables, approach F4 of IJON and SGFuzz can be expected to work well. If the program point is used in the protocol state, approach F3 as used by Chen et al. might work better.

All evolutionary fuzzers require initial seeds as input traces to start fuzzing. The choice of these initial seeds can influence the performance. Some fuzzers provide some automation to create initial seeds: for instance, Nyx-net [54] provides functionality (in the form of a Python library) to generate seed messages from PCAP network dumps. The creators of IJON [5] note that in some cases IJON's feedback mechanism works so well that manually picking good seeds is no longer necessary to obtain good coverage. In some experiments, they could simply use a single uninformative seed containing only the character 'a' [5] .

4.4 Evolutionary grammar-based fuzzers

Table 5 presents the evolutionary grammar-based fuzzers. These fuzzers combine the *grammar-based* and *evolutionary* approaches, as shown in Fig. 5: they require a grammar as the starting point to generate messages but they also include some feedback to observe the effects of inputs in an effort to fuzz more intelligently.

RESTler [6] is an open-source fuzzer by Microsoft for fuzzing REST APIs. It uses a grammar in the form of an OpenAPI[12] specification (as can be produced by Swagger tools) to generate messages but then observes response combinations of messages that always lead to the same error. The fact that RESTful API typically come with grammar in the form of an OpenAPI spec is a big win: it means we can use a grammar-based approach but avoid the downside of having to produce a grammar.

SPFuzz [57] and EPF [25] observe branch coverage in the style of AFL to get information about coverage (i.e. F2)), whereas RESTler only uses the response of the SUT (i.e. F1).

RESTler does not require any initial seeds provided by the user, as you would expect of a fuzzer that has a grammar that can be used to generate inputs. However, SPFuzz and EPF do require the user to provide initial seeds. For EPF these are provided in the form of a PCAP file.

SPFuzz does not use some standard specification formats like OpenAPI, but it has its own format to describe the protocol grammar and dependencies.

These dependencies, like the ones between requests and responses [6], or the ones between the length field, the content of the message or the data types [25, 57], significantly influence the quality of the inputs generated by the fuzzer.

Fuzzer	**Feedback system**	**Based on**	**Inputs needed**	**Open source**
RESTler [6]	- Response	-	- State model specification - Target source file	✓
SPFuzz [57]	- Coverage	- AFL	- Protocol specification - Target source code - Initial seeds	✓
EPF [25]	- Coverage	- AFL - Fuzzowski	- Protocol specification - Target source code - PCAP files (as initial seeds)	✓

Table 5. Evolutionary grammar-based fuzzers

Fuzzer	**Learns**	**Feedback (i)**	**Feedback (ii)**	**Based on**	**Inputs needed**	**Open source**
AFLNet [50]	- State model	- Coverage	- Response	- AFL	- Target binary - Sample traces	✓
FFUZZ [11]	- State model	- Coverage	- Response	- AFLNet	- Target binary - Sample traces	✓
StateAFL [45]	- State model	- Coverage	- Memory	- AFLNet	- Target binary - Sample traces	✓
SGPFuzzer [66]	- State model - Message fields	- Coverage	- Response	- AFL	- Target binary - PCAP file	✓
BLEEM [38]	- State model	- Response	- Response	- Packet-oriented black-box fuzzer[38]	- PCAP file	✓

Table 6. Evolutionary grammar learner fuzzers

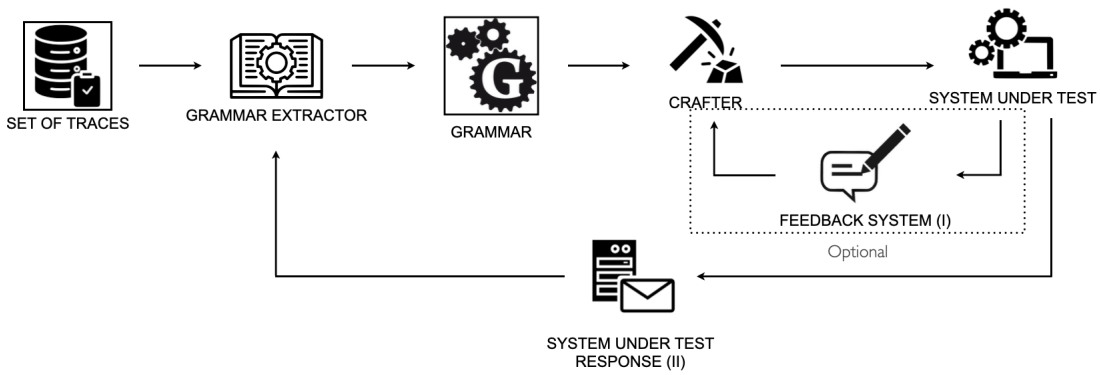


Fig. 6. Evolutionary grammar learner fuzzers use both feedback from the SUT and a grammar inferred by a grammar extractor to craft slightly malformed messages to send to the SUT.

4.5 Evolutionary grammar learner fuzzers

Table 6 presents the evolutionary grammar-learner fuzzers. This is the most complex category of fuzzers. These fuzzers involve a grammar, which only describes (an approximation of) the protocol state machine. They use two forms of evolution, illustrated by the two feedback loops in Figure 6:

(i) *Message evolution*: like for the evolutionary fuzzers discussed in Section 4.3, feedback from the system is used to mutate traces, using one or several of the five types of observation discussed there.

(ii) *State machine evolution*: here feedback from the system is used to improve an approximation of the protocol state machine. This comes down to a form of active-state machine learning.
For all tools except StateAFL, the feedback used here is the response from the SUT (i.e. F1) or some information extracted from that response; for example, for AFLNet it is the response code in the response, for EPF it is just information about whether the connection was dropped. StateAFL observes whether the content of long-lived memory areas has changed (i.e. F5).

Fuzzer	Limitations	Uses	Input needed	Open source
AutoFuzz [24]	- Cannot fuzz the message order	Passive learning [10] [26]	Live traffic	✓
Black-Box Live Protocol Fuzzing [53]	- Cannot fuzz the message order - User needs to specify the fields to fuzz	N/A	Live traffic	✓
SecFuzz [60]	- Limited fuzzing of message order	N/A	Live traffic	✓

Table 7. Man-in-the-middle fuzzers

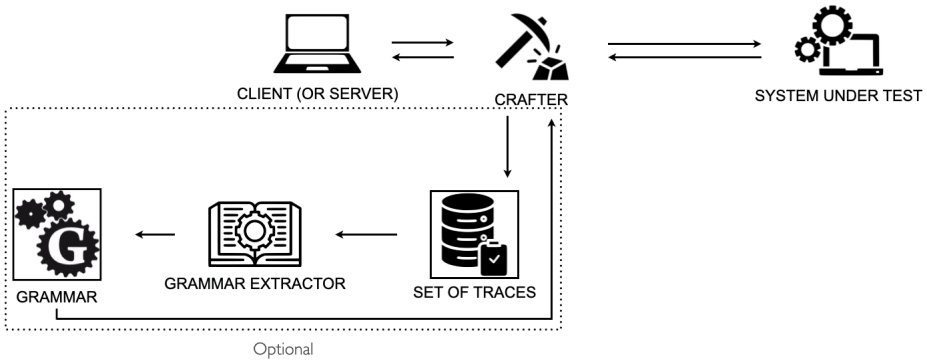


Fig. 7. Man-in-the-middle fuzzers plug between the client and the server to intercept and modify the live traffic.



Fig. 8. Machine learning fuzzers use ML models to produce slightly malformed messages to send to the SUT.

4.6 Man-in-the-middle fuzzers

Table 7 presents the man-in-the-middle-fuzzers. As shown in Fig. 7, these fuzzers sit between the SUT and another application that interacts with the SUT to intercept the communication and modify the communication going to the SUT. If the SUT is a server then this other application will be a client.

A fundamental limitation of these fuzzers is that they are only able to modify the order of the messages in a limited way. In fact, AutoFuzz [24] and Black-Box Live Protocol Fuzzing [53] do not modify the order of messages at all, so the exploration of the protocol state machine will be very limited. SecFuzz [60] does fuzz the order of the messages, but only a little bit, namely by inserting well-formed messages at random positions in the input trace (i.e. in the sequence of messages sent by the other application).

Even though the overall setup is the same, the fuzzers use different techniques:

- Like the grammar learner fuzzers, AutoFuzz operates in two phases. Prior to the actual fuzzing, it starts with a passive learning phase to infer an approximation of the protocol state machine. For this AutoFuzz uses the same algorithm as Hsu et al. [26], i.e. partial finite state automaton reduction. So, as for Hsu et al. the user has to supply implementations of abstraction functions that map concrete messages to some abstract alphabet. During the fuzzing AutoFuzz then uses its knowledge of the protocol state machine to guide the input selection.
- Black-Box Live Protocol Fuzzing uses a function to generate the message field specification from a PCAP file, but the user is required to choose the fields of the messages to fuzz.
- SecFuzz is able to deal with the cryptography of the protocol. To do that, the client has to share with the fuzzer (through a log file) all the information necessary for the decryption.

[12] https://www.openapis.org

4.7 Machine learning fuzzers

Table 8 presents the machine learning fuzzers. As shown in Fig. 8, like the grammar learner and the evolutionary grammar learner fuzzers, these fuzzers require a set of traces that are used as a dataset to train the machine learning model. Once trained, the machine learning model is able to output traces that slightly differ from legitimate, correct traces. Likewise the man-in-the-middle fuzzers, the machine learning fuzzers observe protocol executions that follow the happy flow. This causes an unbalanced dataset in favour of the correct traces and the model's inability to output traces with messages in the wrong order.

Although these fuzzers use a machine learning model — trained on real protocol execution — to output traces to forward to the SUT, they employ different strategies. GANFuzz [27] uses a generative adversarial network (GAN) and an RNN (recursive neural network), while the fuzzer by Fan et al. [18] and SeqFuzzer [70] use seq2seq. We refer to the review by Wang et al. [63] for a more exhaustive explanation of fuzzing using machine learning.

5 GENERIC FUZZERS IMPROVEMENTS

Irrespective of the category of fuzzer, there are some generic improvements that several fuzzers include.

Pre-processing of raw network traffic. Many fuzzers take raw network traffic in the format of a PCAP file as input and provide some automated pre-processing of that input. Each tool implements it in its own way but it includes some common ingredients, such as chopping up the traces to extract the individual messages to then clustering similar messages or recognizing specific fields in the messages.

Using snapshots. One factor that makes fuzzing of stateful systems slow is that a fuzzer often needs to repeat a sequence of inputs to get the SUT in a particular state, to then start the actual fuzzing in that program state. To avoid the overhead, some fuzzers [37, 39, 54] use snapshots (aka checkpoints) to capture the program state of the SUT at a particular point in time, to then be able to quickly re-start fuzzing from that point on.[13]This can speed up fuzzing, as the initial trace to reach some specific state does not have to be repeated, but taking and re-starting snapshots also introduces overhead, so in the end it may not be faster. Depending on the execution platform there are different snapshotting techniques that can be used. For instance, FitM and SNPSFuzzer use CRIU's userspace snapshots and nyx-net uses hypervisor-based snapshots. For SNSP, different snapshotting technologies have been compared [37]: CRIU [14], DMTCP [4], BLCR [15], and OpenCZ [16].

[13]The same idea is behind the use of forking by AFL, where even for stateless SUTs it has been shown to improve performance.
[14]https://github.com/checkpoint-restore/criu
[15]https://github.com/angelos-se/blcr
[16]https://openvz.livejournal.com

Fuzzer	Based on	Input needed	Open source
GANFuzz [27]	seq-gan model	Traces	✗
Machine Learning for Black-box Fuzzing of Network Protocols [18]	seq2seq model	Traces	✗
SeqFuzzer [70]	seq2seq model	Traces	✗

Table 8. Machine learning fuzzers

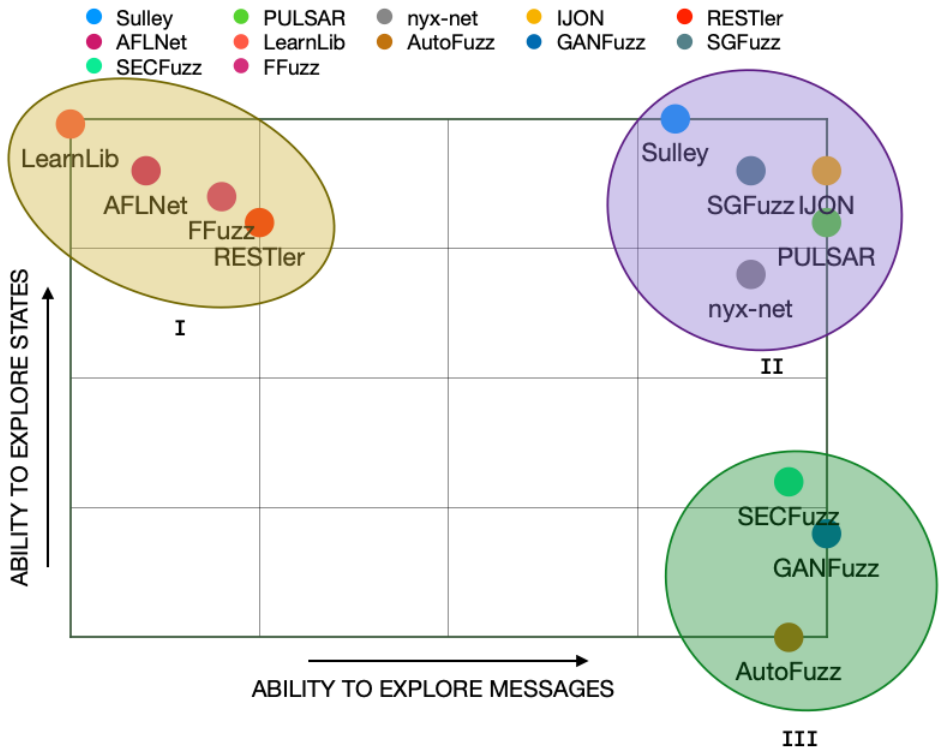


Fig. 9. Cluster of fuzzers

Mutation primitives and heuristics. Any fuzzer that uses some form of mutation (of individual messages or of traces) can use a variety of strategies and primitives to do this. For individual messages, typical mutations may include random bit-flipping, deleting some parts of a message or inserting some data. For traces, (as opposed to individual messages) interesting mutation primitives are of course removal, insertion, or repetition of messages.

The fuzzers we discussed come with a variety of primitives for all this. Some offer possibilities for the user to provide their own custom mutators. We have not gone into the details of this, as the focus was on understanding the overall approach. Some fuzzers, notably SNOOZE, PROTOS, SPFuzz, SGPFuzzer, and the fuzzer by Hsu et al. [26], provide more advanced heuristics and tricks for mutations than some of the others. For example, SNOOZE can provide mutations to try out SQL or command injection or use specific numbers to test boundary conditions. SPFuzz distinguishes different types of data inside messages (e.g. headers vs payloads) to then use different mutation strategies for specific types of data. In practice, it may of course make a big difference for a particular case study in which mutation primitives or heuristics are used.

6 LIMITATIONS OF EXISTING APPROACHES

An intrinsic problem with stateful fuzzing is the need to deal with two different levels of input language (as explained in Section 2). In contrast to the stateless fuzzers that 'only' need to fuzz one level of input language (i.e. the language

of the individual messages), stateful fuzzers need to fuzz the individual messages *and* the traces. This second level brings an additional layer of *computational complexity* since it is possible to send the same message to every state. Fuzzers try to cope with this problem by employing heuristics and approaches to improve the fuzzing speed (e.g. snapshots [37, 39, 54]), but often it is not enough and fuzzing campaigns require much time and significant computation power.

Moreover, dealing with complex state models also makes it challenging to reach deep states, adding an additional layer of *problem complexity*. State-of-the-art stateful fuzzers deal with the problem complexity by using grammar describing the state model of the SUT or by inferring the state model from observed traffic. Unfortunately, such a grammar can be inaccurate or incomplete and observed traffic will (nearly always) be incomplete.

7 CONCLUSIONS

It was challenging to disentangle the ways that various fuzzers for stateful systems work and arrive at the classification we presented. It seems like every fuzzer picks another set of building blocks, combines them in its own way, and then adds some ad-hoc heuristics and possibly performance optimisations. New fuzzers are typically evaluated on some case studies and then compared with some other fuzzers, but it is hard to draw broader conclusions that then go beyond a particular case study or a particular pair of fuzzers. This underlines the importance of initiatives such as ProFuzzBench [45] for bench-marking stateful fuzzing approaches. Benchmarking has also been pointed out as a challenge for fuzzers in general, not just for stateful fuzzing [12].

We have noted some apparent contradictory observations – although this may simply be because researchers looked at different case studies. For instance, Shu et al. [56] abandoned the use active learning of protocol state machine using L* (or its variants) because they found it too slow and inaccurate, while in other research this has proved to be very useful in finding security flaws [16].

It is not surprising that the performance of fuzzers may depend heavily on the case study. When fuzzing a stateful system there is a trade-off between a) trying out many variations of individual messages and b) trying out many different *sequences* of messages. The complexity of an application (and hence the likely problem spots) may occur more in the message format or in the protocol state machine; a corresponding strategy when fuzzing, focusing more on a) or on b), is then most likely to find bugs. Very broadly we can make a rough distinction into three classes of tools, illustrated in Fig. 9: I) fuzzers that are very good at aggressively exploring the protocol state machine but poor at trying out variations of messages; II) fuzzers that try to explore both the protocol state machine and the format of individual messages; and III) fuzzers that are good at trying out variations in messages but poor at exploring the protocol state machine.

There is some relation between this classification and the seven categories we have described. For instance, the man-in-the-middle fuzzers and the machine learning fuzzers are in class III, as they do not explore the protocol state machine and mainly (or even exclusively) stick to message sequences observed in real communications between client and server. The grammar-based fuzzers can deal quite well with both dimensions of fuzzing so are in class II. Evolutionary-based fuzzers that try to infer the protocol state machine (typically using the response of the SUT as a feedback mechanism) are good at exploring the protocol state space but may lack mutation primitives or observation mechanisms to aggressively explore the message formats. LearnLib is an extreme instance of class I as it *only* fuzzes the message order.

The exact positioning of tools in Figure 9 is not based on experimental data, but more informally based on the general characteristics of the tools so should be taken with caution. Also, for tools that require grammars as input or manual code annotation a lot will depend on the quality of these.

It may seem like fuzzers of type II are the best of both worlds, but given the rapid state space explosion when we fuzz both individual messages and sequences of messages this need not be the case: using a fuzzer of type I and a fuzzer of type III to explore different aspects may be more effective than using one fuzzer of type II that tries to do both. For fuzzing of non-stateful systems, it has already been demonstrated that using a combination of tools may be the optimal approach, especially if these tools can exchange information [13]; we expect that this will be even more so for stateful systems.

8 FUTURE WORK

By providing insight into the components used in various fuzzing approaches, our research suggests several interesting directions for future research. One is trying new combinations of approaches and components; for example, using LearnLib as a pre-processing phase to get a good initial approximation of the protocol state machine. Another is using the SUT response in the man-in-the-middle fuzzers to build a more accurate protocol state model. Moreover, some of the performance optimisations implemented by specific fuzzers (e.g. the use of snapshots) can be applied to a broader set of tools. Another improvement is a more systematic, empirical comparison: having identified that some tools use the same overall approach but different algorithms for some sub-components, allows a more systematic comparison between the algorithms themselves.

REFERENCES

[1] Humberto J Abdelnur, Radu State, and Olivier Festor. 2007. KiF: a stateful SIP fuzzer. In *Proceedings of the 1st international conference on Principles, systems and applications of IP telecommunications*. 47–56.

[2] Humberto J. Abdelnur, Radu State, and Olivier Festor. 2007. KiF: A Stateful SIP Fuzzer. In *Proceedings of the 1st International Conference on Principles, Systems and Applications of IP Telecommunications (IPTComm'07)*. ACM, 47–56. https://doi.org/10.1145/1326304.1326313

[3] Dana Angluin. 1987. Learning Regular Sets from Queries and Counterexamples. *Inf. Comput.* 75, 2 (1987), 87–106.

[4] Jason Ansel, Kapil Arya, and Gene Cooperman. 2009. DMTCP: Transparent checkpointing for cluster computations and the desktop. In *International Symposium on Parallel & Distributed Processing*. IEEE, 1–12.

[5] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. 2020. IJON: Exploring Deep State Spaces via Fuzzing. In *Security and Privacy (S&P'20)*. IEEE, 1597–1612. https://doi.org/10.1109/SP40000.2020.00117

[6] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. 2019. RESTler: Stateful REST API Fuzzing. In *International Conference on Software Engineering (ICSE'09)*. ACM, 748–758. https://doi.org/10.1109/ICSE.2019.00083

[7] Jinsheng Ba, Marcel Böhme, Zahra Mirzamomen, and Abhik Roychoudhury. 2022. Stateful Greybox Fuzzing. arXiv:2204.02545 [cs.CR] http://export.arxiv.org/abs/2204.02545v3

[8] Greg Banks, Marco Cova, Viktoria Felmetsger, Kevin Almeroth, Richard Kemmerer, and Giovanni Vigna. 2006. SNOOZE: toward a Stateful NetwOrk prOtocol fuzZEr. In *International conference on information security*. Springer, 343–358.

[9] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2017. Synthesizing program input grammars. *ACM SIGPLAN Notices* 52, 6 (2017), 95–110. https://doi.org/10.1145/3140587.3062349 Proceedings of PLDI'17.

[10] Marshall A Beddoe. 2004. Network protocol analysis using bioinformatics algorithms. *Toorcon* (2004).

[11] Wu Biao, Tang Chaojing, and Zhang Bin. 2021. FFUZZ: A Fast Fuzzing Test Method for Stateful Network Protocol Implementation. In *Conference on Computer Communication and Network Security (CCNS'21)*. IEEE, 75–79. https://doi.org/10.1109/CCNS53852.2021.00023

[12] Marcel Boehme, Cristian Cadar, and Abhik Roychoudhury. 2021. Fuzzing: Challenges and Reflections. *IEEE Software* 38, 3 (2021), 79–86.

[13] Yuanliang Chen, Yu Jiang, Fuchen Ma, Jie Liang, Mingzhe Wang, Chijin Zhou, Xun Jiao, and Zhuo Su. 2019. EnFuzz: Ensemble Fuzzing with Seed Synchronization among Diverse Fuzzers. In *USENIX Security Symposium*. USENIX Association.

[14] Yurong Chen, Tian lan, and Guru Venkataramani. 2019. Exploring Effective Fuzzing Strategies to Analyze Communication Protocols. In *Proceedings of the 3rd ACM Workshop on Forming an Ecosystem Around Software Transformation (FEAST'19)*. ACM, 17–23. https://doi.org/10.1145/3338502.3359762

[15] Colin de la Higuera. 2010. *Grammatical inference: learning automata and grammars*. Cambridge University Press.

[16] Joeri de Ruiter and Erik Poll. 2015. Protocol State Fuzzing of TLS Implementations. In *USENIX Security Symposium*. USENIX Association, 193–206.

[17] Adam Doupé, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. 2012. Enemy of the State: A State-Aware Black-Box Web Vulnerability Scanner. In *USENIX Security Symposium*. USENIX, 523–538.

[18] Rong Fan and Yaoyao Chang. 2017. Machine learning for black-box fuzzing of network protocols. In *International Conference on Information and Communications Security*. Springer, 621–632.

[19] Paul Fiterau-Brostean, Bengt Jonsson, Robert Merget, Joeri De Ruiter, Konstantinos Sagonas, and Juraj Somorovsky. 2020. Analysis of {DTLS} implementations using protocol state fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*. 2523–2540.

[20] Markus Theo Frohme. 2019. Active automata learning with adaptive distinguishing sequences. *arXiv preprint arXiv:1902.01139* (2019).

[21] Hugo Gascon, Christian Wressnegger, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. 2015. Pulsar: Stateful black-box fuzzing of proprietary network protocols. In *International Conference on Security and Privacy in Communication Systems*. Springer, 330–347. https://doi.org/10.1007/978-3-319-28865-9_18

[22] Patrice Godefroid, Michael Y. Levin, and David Molnar. 2008. Automated whitebox fuzz testing. In *Network and Distributed System Security (NDSS'08)*, Vol. 8. The Internet Society, 151–166.

[23] Rahul Gopinath, Björn Mathis, and Andreas Zeller. 2020. Mining input grammars from dynamic control flow. In *Proceedings of the 28th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 172–183.

[24] Serge Gorbunov and Arnold Rosenbloom. 2010. AutoFuzz: Automated network protocol fuzzing framework. *International Journal of Computer Science and Network Security (IJCSNS)* 10, 8 (2010), 239.

[25] René Helmke, Eugen Winter, and Michael Rademacher. 2021. EPF: An Evolutionary, Protocol-Aware, and Coverage-Guided Network Fuzzing Framework. In *International Conference on Privacy, Security and Trust (PST)*. IEE. https://doi.org/10.1109/PST52912.2021.9647801

[26] Yating Hsu, Guoqiang Shu, and David Lee. 2008. A model-based approach to security flaw detection of network protocol implementations. In *2008 IEEE International Conference on Network Protocols*. IEEE, 114–123. https://doi.org/10.1109/ICNP.2008.4697030

[27] Zhicheng Hu, Jianqi Shi, YanHong Huang, Jiawen Xiong, and Xiangxing Bu. 2018. GANFuzz: A GAN-Based Industrial Network Protocol Fuzzing Framework. In *International Conference on Computing Frontiers (CF'18)*. ACM, 138–145. https://doi.org/10.1145/3203217.3203241

[28] Malte Isberner, Falk Howar, and Bernhard Steffen. 2014. The TTT algorithm: a redundancy-free approach to active automata learning. In *International Conference on Runtime Verification*. Springer, 307–322.

[29] Zu-Ming Jiang, Jia-Ju Bai, and Zhendong Su. 2023. DynSQL: Stateful fuzzing for database management systems with complex and valid SQL query generation. In *Proceedings of USENIX Security Symposium (USENIX Security)*.

[30] Rauli Kaksonen, Marko Laakso, and Ari Takanen. 2001. Software security assessment through specification mutations and fault injection. In *Communications and Multimedia Security Issues of the New Century*. Springer, 173–183.

[31] Faisal Kamiran, Toon Calders, and Mykola Pechenizkiy. 2010. Discrimination aware decision tree learning. In *2010 IEEE international conference on data mining*. IEEE, 869–874.

[32] Rody Kersten, Kasper Luckow, and Corina S Păsăreanu. 2017. POSTER: AFL-based Fuzzing for Java with Kelinci. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2511–2513.

[33] Kyungtae Kim, Taegyu Kim, Ertza Warraich, Byoungyoung Lee, Kevin RB Butler, Antonio Bianchi, and Dave Jing Tian. 2022. Fuzzusb: Hybrid stateful fuzzing of usb gadget stacks. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2212–2229.

[34] Takahisa Kitagawa, Miyuki Hanaoka, and Kenji Kono. 2010. AspFuzz: A state-aware protocol fuzzer based on application-layer protocols. In *IEEE Symposium on Computers and Communications (ISCC)*. IEEE, 202–208. https://doi.org/10.1109/ISCC.2010.5546704

[35] Tammo Krueger, Hugo Gascon, Nicole Krämer, and Konrad Rieck. 2012. Learning stateful models for network honeypots. In *Workshop on Security and artificial intelligence (AISec'12)*. ACM, 37–48. https://doi.org/10.1145/2381896.2381904

[36] Ao Li, Rohan Padhye, and Vyas Sekar. 2022. Spider: A practical fuzzing framework to uncover stateful performance issues in sdn controllers. *arXiv preprint arXiv:2209.04026* (2022).

[37] Junqiang Li, Senyi Li, Gang Sun, Ting Chen, and Hongfang Yu. 2022. SNPSFuzzer: A Fast Greybox Fuzzer for Stateful Network Protocols using Snapshots. *arXiv preprint arXiv:2202.03643* (2022).

[38] Zhengxiong Luo, Junze Yu, Feilong Zuo, Jianzhong Liu, Yu Jiang, Ting Chen, Abhik Roychoudhury, and Jiaguang Sun. 2023. Bleem: Packet Sequence Oriented Fuzzing for Protocol Implementations. In *32nd USENIX Security Symposium (USENIX Security 23)*. 4481–4498.

[39] Dominik Maier, Otto Bittner, Marc Munier, and Julian Beier. 2022. FitM: Binary-Only Coverage-Guided Fuzzing for Stateful Network Protocols. In *Workshop on Binary Analysis Research (BAR)*. Internet Society.

[40] Valentin Jean Marie Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. 2021. The Art, Science, and Engineering of Fuzzing: A Survey. *Transactions on Software Engineering* 47, 11 (2021). https://doi.org/10.1109/TSE.2019.2946563

[41] Chris McMahon Stone, Sam L Thomas, Mathy Vanhoef, James Henderson, Nicolas Bailluet, and Tom Chothia. 2022. The closer you look, the more you learn: A grey-box approach to protocol state machine learning. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 2265–2278.

[42] Barton P. Miller, Louis Fredriksen, and Bryan So. 1990. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM* 33, 12 (1990), 32–44. https://doi.org/10.1145/96267.96279

[43] Kevin P Murphy et al. 1995. Passively learning finite automata. Citeseer.

[44] Edi Muškardin, Bernhard K Aichernig, Ingo Pill, Andrea Pferscher, and Martin Tappler. 2022. AALpy: an active automata learning library. *Innovations in Systems and Software Engineering* 18, 3 (2022), 417–426.

[45] Roberto Natella. 2022. StateAFL: Greybox fuzzing for stateful network servers. *Empirical Software Engineering* 27, 7 (2022), 191. https://doi.org/10.1007/s10664-022-10233-3

[46] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices* 42, 6 (2007), 89–100.

[47] José Oncina and Pedro Garcia. 1992. Identifying regular languages in polynomial time. In *Advances in structural and syntactic pattern recognition*. World Scientific, 99–108.

[48] Haram Park, Carlos Kayembe Nkuba, Seunghoon Woo, and Heejo Lee. 2022. L2Fuzz: Discovering Bluetooth L2CAP Vulnerabilities Using Stateful Fuzz Testing. In *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 343–354.

[49] Joshua Pereyda. 2019. boofuzz Documentation. THIS REFERENCE STILL NEEDS TO BE FIXED.

[50] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. 2020. AFLNET: A Greybox Fuzzer for Network Protocols. In *13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 460–465. https://doi.org/10.1109/ICST46399.2020.00062

[51] Erik Poll, Joeri de Ruiter, and Aleksy Schubert. 2015. Protocol state machines and session languages: specification, implementation, and security flaws. In *Symposium on Security and Privacy Workshops (SPW)*. IEEE, 125 – 133. https://doi.org/10.1109/SPW.2015.32

[52] Harald Raffelt, Bernhard Steffen, Therese Berg, and Tiziana Margaria. 2009. LearnLib: a framework for extrapolating behavioral models. *Int. J. Softw. Tools Technol. Transf.* 11 (2009), 393–407. Issue 5.

[53] Timo Ramsauer. 2021. Black-Box Live Protocol Fuzzing. *Target* 2 (2021), 1–2.

[54] Sergej Schumilo, Cornelius Aschermann, Andrea Jemmett, Ali Abbasi, and Thorsten Holz. 2021. Nyx-Net: Network Fuzzing with Incremental Snapshots. *arXiv preprint arXiv:2111.03013* (2021).

[55] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *Annual Technical Conference (ATC)*. USENIX, 309–318.

[56] Guoqiang Shu, Yating Hsu, and David Lee. 2008. Detecting Communication Protocol Security Flaws by Formal Fuzz Testing and Machine Learning. In *Formal Techniques for Networked and Distributed Systems (FORTE 2008)*. Springer, 299–304.

[57] Congxi Song, Bo Yu, Xu Zhou, and Qiang Yang. 2019. SPFuzz: A Hierarchical Scheduling Framework for Stateful Network Protocol Fuzzing. *IEEE Access* 7 (2019), 18490–18499. https://doi.org/10.1109/ACCESS.2019.2895025

[58] Evgeniy Stepanov and Konstantin Serebryany. 2015. MemorySanitizer: fast detector of uninitialized memory use in C++. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 46–55.

[59] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. *Advances in neural information processing systems* 27 (2014).

[60] Petar Tsankov, Mohammad Torabi Dashti, and David Basin. 2012. SECFUZZ: Fuzz-testing security protocols. In *International Workshop on Automation of Software Test (AST)*. IEEE, 1–7.

[61] Dimitrios Tychalas, Hadjer Benkraouda, and Michail Maniatakos. 2021. {ICSFuzz}: Manipulating {I/Os} and Repurposing Binary Code to Enable Instrumented Fuzzing in {ICS} Control Applications. In *30th USENIX Security Symposium (USENIX Security 21)*. 2847–2862.

[62] Sicco Verwer and Christian Hammerschmidt. 2022. FlexFringe: Modeling Software Behavior by Learning Probabilistic Automata. *arXiv preprint arXiv:2203.16331* (2022).

[63] Yan Wang, Peng Jia, Luping Liu, Cheng Huang, and Zhonglin Liu. 2020. A systematic review of fuzzing based on machine learning techniques. *PloS one* 15, 8 (2020).

[64] Zhuzhu Wang and Ying Wang. 2023. NLP-based Cross-Layer 5G Vulnerabilities Detection via Fuzzing Generated Run-Time Profiling. arXiv:2305.08226 [cs.CR]

[65] Lantao Yu, Weinan Zhang, Jun Wang, and Yong Yu. 2017. Seqgan: Sequence generative adversarial nets with policy gradient. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 31.

[66] Yingchao Yu, Zuoning Chen, Shuitao Gan, and Xiaofeng Wang. 2020. SGPFuzzer: A State-Driven Smart Graybox Protocol Fuzzer for Network Protocol Implementations. *IEEE Access* 8 (2020), 198668–198678.

[67] Michal Zalewski. 2014. American Fuzzy Lop (afl). https://lcamtuf.coredump.cx/afl

[68] Hangwei Zhang, Kai Lu, Xu Zhou, Qidi Yin, Pengfei Wang, and Tai Yue. 2021. SIoTFuzzer: fuzzing web interface in IoT firmware via stateful message generation. *Applied Sciences* 11, 7 (2021), 3120.

[69] Zhaowei Zhang, Hongzheng Zhang, Jinjing Zhao, and Yanfei Yin. 2023. A Survey on the Development of Network Protocol Fuzzing Techniques. *Electronics* 12, 13 (2023), 2904.

[70] Hui Zhao, Zhihui Li, Hansheng Wei, Jianqi Shi, and Yanhong Huang. 2019. SeqFuzzer: An Industrial Protocol Fuzzing Framework from a Deep Learning Perspective. In *IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 59–67. https://doi.org/10.1109/ICST.2019.00016

[71] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. 2022. Fuzzing: a survey for roadmap. *ACM Computing Surveys (CSUR)* 54, 11s (2022), 1–36.