



## **BLEEM: Packet Sequence Oriented Fuzzing for Protocol Implementations**

Zhengxiong Luo, Junze Yu, Feilong Zuo, Jianzhong Liu, and Yu Jiang,  
*Tsinghua University*; Ting Chen, *University of Electronic Science and  
Technology of China*; Abhik Roychoudhury, *National University  
of Singapore*; Jiaguang Sun, *Tsinghua University*

<https://www.usenix.org/conference/usenixsecurity23/presentation/luo-zhengxiong>


**This paper is included in the Proceedings of the  
32nd USENIX Security Symposium.**

**August 9–11, 2023 • Anaheim, CA, USA**

978-1-939133-37-3

**Open access to the Proceedings of the  
32nd USENIX Security Symposium  
is sponsored by USENIX.**

# BLEEM: Packet Sequence Oriented Fuzzing for Protocol Implementations

Zhengxiong Luo<sup>1</sup>, Junze Yu<sup>1</sup>, Feilong Zuo<sup>1</sup>, Jianzhong Liu<sup>1</sup>, Yu Jiang<sup>1</sup>, , Ting Chen<sup>2</sup>,  
Abhik Roychoudhury<sup>3</sup>, and Jianguang Sun<sup>1</sup>

<sup>1</sup>*KLISS, BNRist, School of Software, Tsinghua University*

<sup>2</sup>*University of Electronic Science and Technology of China*

<sup>3</sup>*National University of Singapore*

## Abstract

Protocol implementations are essential components in network infrastructures. Flaws hidden in the implementations can easily render devices vulnerable to adversaries. Therefore, guaranteeing their correctness is important. However, commonly used vulnerability detection techniques, such as fuzz testing, face increasing challenges in testing these implementations due to ineffective feedback mechanisms and insufficient protocol state-space exploration techniques.

This paper presents BLEEM, a *packet-sequence-oriented* black-box fuzzer for vulnerability detection of protocol implementations. Instead of focusing on individual packet generation, BLEEM generates packets on a sequence level. It provides an effective feedback mechanism by analyzing the system output sequence noninvasively, supports guided fuzzing by resorting to state-space tracking that encompasses all parties timely, and utilizes interactive traffic information to generate protocol-logic-aware packet sequences. We evaluate BLEEM on 15 widely-used implementations of well-known protocols (e.g., TLS and QUIC). Results show that, compared to the state-of-the-art protocol fuzzers such as Peach, BLEEM achieves substantially higher branch coverage (up to 174.93% improvement) within 24 hours. Furthermore, BLEEM exposed 15 security-critical vulnerabilities in prominent protocol implementations, with 10 CVEs assigned.


## 1 Introduction

Protocol implementations form the foundations of network infrastructures. Since they are usually exposed directly to the network, protocol implementations need to handle any malformed or malicious traffic from potential attackers correctly. Any flaws undetected in the implementations (0-days) can easily render devices vulnerable to adversaries. For instance, the infamous Heartbleed [15] vulnerability discovered in OpenSSL can cause sensitive data exposure. Many techniques have been proposed to increase the security of these

protocol implementations, such as formal analysis [40] and model checking [21]. Though they are somewhat effective in some regards, these techniques usually require significant prior knowledge to perform.

Fuzzing, an automated software testing technique, has emerged as one of the most effective techniques for detecting vulnerabilities in real-world software. At a high level, given a target protocol program, a fuzzer works by continuously generating packets and sending them to the program while observing for potential anomalies. While traditional protocol fuzzers have been widely adopted and detected many vulnerabilities, they still suffer from the following problems.

First, blind packet generation is ineffective and resource-consumption-intensive. Traditional fuzzers like Peach [18] generate packets based on a predefined test model without program feedback. Hence, fuzzers are unaware of whether the generated input triggered new program states. These valuable historical attempts cannot be recognized and utilized for further optimization, lowering their overall efficiency. Some recent approaches tackle this problem by applying an evolutionary process: (i) to recognize a successful attempt, they resort to code coverage [48, 58] or status codes extracted from server responses [45]; and (ii) to explore further based on these successful attempts, they directly retain the inputs involved and perform variations on them, based on the promise that these inputs have reproducibility behaviors when passed to the same target program. However, such a method does not scale across different protocol implementations: (i) First, these feedback mechanisms either require instrumenting the source code or binaries, which may be infeasible when the protocol implementation is black-box, or are tightly coupled with a specific protocol format, which is inappropriate when testing a diverse range of protocol implementations. (ii) Second, some protocols' specific validation rules can render existing evolutionary procedures futile. For example, some common protocols, such as TLS [3], DTLS [4], and SSH [39], employ random nonces during the handshake to prevent "replay attacks". In this scenario, the valuable inputs retained in the previous exploration can no longer replay the interesting behav-

 Yu Jiang is the correspondence author.

iors it once triggered, making the input generation attempts based on these inputs less likely to produce valuable results.

Second, since protocol implementations are stateful systems, the input space of the under-test program is tightly regulated by its state. Effectively traversing the massive state space and covering the versatile communication regulations (i.e., various transitions) requires carefully-crafted packet sequences. However, constructing such packet sequences is non-trivial because it involves complex protocol logic, e.g., in what order to transmit the packets and how to construct them to guarantee the format and parameter correctness. Some existing approaches, such as Peach [18], resort to a user-defined protocol model and generate packet sequences by strictly following the actions depicted in that model. While the protocol logic involved in the model can be covered effectively, logic beyond the confines of the model cannot be exercised. Other approaches, such as AFLNet [45], generate packet sequences by performing variants on existing packets. Due to unawareness of the protocol format, these approaches remain limited in providing valid inputs for protocol implementations that process highly-structured packets.

In this paper, we present BLEEM, a *packet-sequence-oriented* black-box fuzzer, to address the problems above. BLEEM employs an on-the-fly noninvasive feedback collection mechanism and conducts dynamic state-space tracking of all parties (i.e., both the client and server) from the system-under-test to guide the packet sequence generation.

First, we introduce a noninvasive on-the-fly feedback mechanism by analyzing the system's output. The insight is that the outputs of protocol implementations (i.e., packet sequences) can abstract the inner protocol state while also being typically structured. Hence, we collect the output sequence of the target system at runtime, analyze the semantics conveyed in the output, and leverage it to indicate the internal system state transitions from the perspective of considering the protocol parties as a whole system.

Second, we leverage heuristics to guide the packet sequence generation based on the feedback. We devise the *System State Tracking Graph* (SSTG), which is dynamically constructed at runtime and allows BLEEM to chart the explored state space. Navigated by the SSTG, we facilitate a comprehensive traversal to increase the exposure of different protocol behaviors and explore the unknown state space by providing diverse inputs for different states. To this end, we extend the SSTG with richer information to enable it to provide guidance for reaching desired states and introduce mutation operators working on different levels, including the packet and sequence levels, to support customized packet generation on different states. Meanwhile, we leverage information extracted from the interactive traffic of the protocol to generate packet sequences. We observe that since the protocol logic is implemented within the server and client, we can generate protocol-logic-aware packet sequences based on the observed traffic exchanged between the two parties, thus preserving parameter dependen-

cies and avoiding producing meaningless packets.

We evaluated the performance of BLEEM on 15 widely-used implementations of well-known protocols, including QUIC, TLS, and DNS. We compared BLEEM against five state-of-the-art protocol fuzzers, including Peach [18], BooFuzz [33], AFLNet [45], SGFuzz [8], and Snipuzz [20]. The experimental results demonstrate that BLEEM outperforms prior works in branch coverage by 28.5%, 48.9%, 35.7%, 23.4%, and 40.3% on average, respectively, over 24 hours. We also show that the proposed fuzzing strategy can effectively help BLEEM explore state space. BLEEM exposed 15 new vulnerabilities in previously well-tested real-world protocol implementations, and Peach, BooFuzz, AFLNet, SGFuzz, and Snipuzz can only expose 8, 5, 6, 7, and 5 of them, respectively. Most of these vulnerabilities are security-critical, where 10 CVEs were assigned due to their severe security influences. In summary, our main contributions are as follows:

- We develop a versatile technique to collect feedback by analyzing the system's output sequence.
- We design the SSTG that identifies the observed state space to navigate the fuzzing exploration direction and propose a protocol-logic-aware packet sequence generation method based on real-world prior information.
- We implement BLEEM and evaluate it on widely-used protocol implementations. The results demonstrate that BLEEM outperforms the state-of-the-art and has detected many security-critical vulnerabilities.

## 2 Protocol Fuzzing

Traditional protocol fuzzers focus mainly on testing server-side implementations. They act as clients, constantly generating packets and sending them to the servers. Based on how the packets are produced, these fuzzers can be roughly classified into two categories: mutation-based and generation-based.

### 2.1 Mutation-based Fuzzers

These fuzzers generate new inputs by randomly mutating existing inputs selected from a corpus [20, 37, 43, 45, 48, 58]. They require no prior knowledge of protocol specifications and message formats, thus, are easy to deploy. These fuzzers are good at testing stateless programs (e.g., file processing application) where no internal state is maintained. To fuzz the protocol implementations, developers use a workaround, where writing test harnesses that implement the unit testing of specific server state or simply treating the input sent to the server as a concatenation of messages to implement system testing. For better performance, AFLNet [45] borrows the mainstream design that augments the traditional fuzzers with a feedback loop [58], enabling the fuzzers to track the execution information exercised by each input and retain the inputs that contribute to new program behaviors for further utilization.



Also, it integrates state awareness by analyzing the status codes in server responses. Nevertheless, lacking the format specification, these fuzzers can quickly encounter barriers because the packets generated from blind mutation can be easily dropped by the protocol program that processes highly-structured packets. Furthermore, these feedback mechanisms do not apply to black-box fuzzing, and their evolutionary methods can be futile by the special checks (discussed in §1).

## 2.2 Generation-based Fuzzers.

These fuzzers generate the packets based on manually constructed protocol models, including data and state models [6, 18, 33, 53]. The state model is typically given in the form of a graph and specifies the valid message sequencing in the interaction with the server. The data model depicts the formats (e.g., field types, field sizes, and valid value ranges) of accepted messages for the corresponding states. The user needs to know the protocol interaction logic to provide this test model by analyzing the source code or reading the protocol specification. In the following, we take the QUIC handshake as an example and give the corresponding test model for Peach [18] initialization.

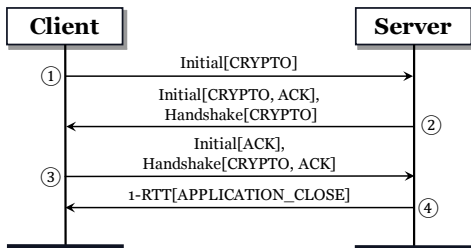


Figure 1: An example dialog between the QUIC server and client

Figure 1 depicts a sample dialog between a client-server pair compliant with QUIC specification (as per the RFC 9000 [32]). The QUIC client starts by sending an `Initial[CRYPTO]` packet (①, `Initial[CRYPTO]` denotes that the packet is of type `Initial` and contains a `CRYPTO` frame, which represents the packet on an abstract level by omitting some detailed fields). Then the server responds with `Initial[CRYPTO, ACK]` and `Handshake[CRYPTO]` packets, carrying the essential information for handshake establishment (②). The client then sends `Initial[ACK]` and `Handshake[CRYPTO, ACK]` packets, indicating the completion of the handshake (③). For simplification, we assume the server is set to terminate the connection via `1-RTT[APPLICATION_CLOSE]` after handshake (④).

Figure 2 depicts the QUIC handshake model (①-③ in Figure 1) in the format of Peach Pit [54], the XML configuration file for the protocol fuzzer Peach [18]. Lines 16-27 show the data model of packet `Initial[CRYPTO]`, which is sent at the first *Action* (Lines 3-5) of the *State* `HANDSHAKE` (Lines 2-12). Specifically, this *State* specifies how Peach interacts with the QUIC server to test the handshake phase: first, send the

`Initial[CRYPTO]` packet (Line 4); then, wait for responses from the server and check whether the responses conform to the format of the given data model `Initial[CRYPTO, ACK]` and `Handshake[CRYPTO]` (Line 7); if so, enter the third *Action* and send `Initial[ACK]` and `Handshake[CRYPTO, ACK]` packets (Line 10, the data models of other kinds of packets, e.g., `Initial[CRYPTO, ACK]`, have been omitted in Figure 2 due to limited space).

```

1 <StateModel name="QUIC" initialState="HANDSHAKE">
2 <State name="HANDSHAKE">
3 <Action type="output">
4 <DataModel ref="Initial[CRYPTO]"/>
5 </Action>
6 <Action type="input">
7 <DataModel ref="Initial[CRYPTO, ACK]+Handshake[CRYPTO]"/>
8 </Action>
9 <Action type="output">
10 <DataModel ref="Initial[ACK]+Handshake[CRYPTO, ACK]"/>
11 </Action>
12 </State>
13 ...
14 </StateModel>
15
16 <DataModel name="Initial[CRYPTO]">
17 <Block name="Header">
18 <Number name="info" size="8" value="cd" valueType="hex"/>
19 <Number name="version" size="32" value="faceb002"
20 <Number name="DCID_length" size="8">
21 <Relation type="size" of="DCID"/>
22 </Number>
23 <String name="DCID" nullTerminated="false"/>
24 ...
25 </Block>
26 <Block name="CRYPTO"> ... </Block>
27 </DataModel>
  
```

Figure 2: A Simplified QUIC Handshake Model as Peach Pit.

The generation-based fuzzers have successfully exposed plenty of vulnerabilities in real-world protocol implementations. However, their effectiveness is heavily limited by the quality of the user-provided protocol model, whose construction usually requires significant expert efforts to read the protocol specification or the source code. First, such specifications are prone to misinterpretation because they are typically expressed in a natural language, thus coming with inherent ambiguities. Second, the protocol generally features a massive state space, making it infeasible to construct a complete model that describes all the protocol behaviors. For example, Figure 2 only depicts the handshake logic of QUIC and does not cover many other logics, such as flow control. Unfortunately, these fuzzers conduct testing by strictly following the actions depicted in the protocol model (see Figure 2, Lines 1-14), and they do not update the state model at runtime. Therefore, the new interesting program behaviors beyond those introduced by the deterministic model cannot be recognized by these fuzzers and therefore cannot be utilized for further exploration. For example, say Peach, in the configuration of the Pit in Figure 2, accidentally triggers a new program behavior via some interesting `Initial[CRYPTO]` packet, and the server responds with another packet not defined in the state model instead of the desired `Initial[CRYPTO, ACK]` packet. Peach then discards it and continues to perform the deterministic testing behaviors instead of exploring more interesting behaviors based on this discovery. Third, some illegal be-

haviors in the protocol implementation may be introduced by implementation errors. These behaviors violate the standards and thus are not included in the protocol specification. Hence, these scenarios cannot be covered under the model constructed based on the protocol specification.

### 2.3 Challenges

In order to design an effective and efficient fuzzing method that can scale across diverse protocol implementations, we need to overcome two challenges.

1) *Lack of a scalable feedback mechanism.* Although the feedback-driven approach has shown effectiveness in optimizing fuzzing [58], the versatile applications of protocol implementations stress the scalability of the traditional feedback mechanisms. Nowadays, protocol implementations have been widely applied in industrial programmable logic controller (PLC) devices and Internet of Things (IoT) devices. Instrumenting the device’s firmware is challenging and even infeasible when the implementation is only accessible in a black-box fashion, which necessitates a noninvasive solution to obtain feedback for the protocol fuzzers.

2) *Highly complex protocol logic.* Protocols feature highly complex logic designed to guarantee correctness and reliability under diverse situations, making the implementations highly complex and stateful interactive targets. Therefore, in-depth implementation fuzzing requires carefully constructing the packet sequence by considering the protocol logic, including the packet format, the parameter dependency between packets, and the packet interaction logic. Specifically, the packet format identifies the standard packet structure, the packet interaction logic identifies the transmission order of different types of packets, and the parameter dependency specifies the value dependencies of parameters exchanged between packets. For example, in Figure 1, the `Initial[CRYPTO]` packet at ① and the `Initial[ACK]` packet at ② should carry the same value for the parameter `DCID`, which is a connection identification field. Otherwise, the handshake would fail, making the following test actions meaningless.

## 3 System Overview

Figure 3 illustrates the overview of BLEEM: given the *System Under Test* (SUT) of the target protocol implementation, BLEEM is initialized with the *Test Initialization* and outputs the *Bug Report* via packet-sequence-oriented fuzzing.

**System Under Test (SUT).** The SUT is the protocol implementation we intend to analyze. Such protocols usually adhere to the client/server model. Given a protocol implementation, traditional fuzzers mainly target one protocol side and perform tests by acting as another. For example, to test the server side, the traditional protocol fuzzer acts as a client and continuously sends generated packets. In this scenario, effective server fuzzing requires expert knowledge of the protocol

logic, i.e., in what order packets are transmitted for different server states and how these packets are built to guarantee parameter correctness. Instead of manually implementing such logic from scratch, as is the case with existing black-box techniques (e.g., Peach Pit in Figure 2), we argue that the client already encodes the desired protocol logic and can therefore be utilized to provide information for packet generation. Specifically, BLEEM requires a pair of client and server that can communicate with each other and considers this pair as a whole system to analyze. Meanwhile, unlike existing coverage-guided works requiring access to the source code [45] or the binary [48] of the implementation, BLEEM can be conducted in a black-box manner. All we assume is that BLEEM can interact with the parties of SUT.

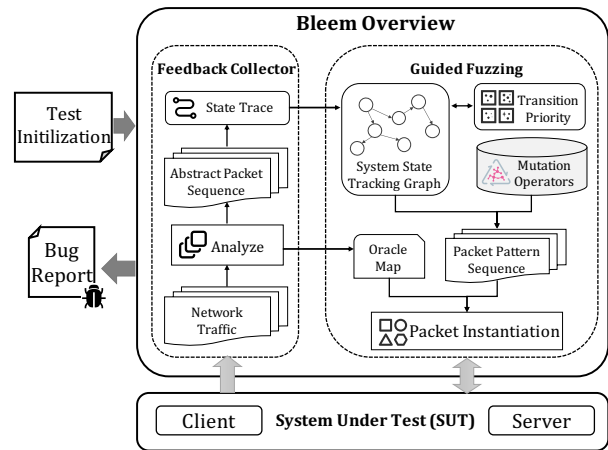


Figure 3: BLEEM Overview. The Feedback Collector monitors the SUT for feedback extraction, based on which the Guided Fuzzing module tracks the state space and applies heuristics to generate packet sequences by leveraging protocol-aware mutation operators and the interactive traffic with the SUT.

In most cases, providing such a client-server pair requires low effort. Protocol implementations usually provide users with client and server utilities, which can be used directly to compose the SUT. When one side of the pair is missing, we can resort to any other valid implementation of the same protocol since the protocols are generally standardized. In the worst case, when off-the-shelf utilities are unavailable on either side, the user needs to provide such a pair since a test entry is a prerequisite for fuzzing, in line with other existing fuzzers [9, 31, 34, 59, 60].

**Test Initialization.** Unlike existing black-box techniques that require manually constructed protocol models, BLEEM starts fuzzing based on the dialog captured by executing the SUT, thus is easier to set up. Since testing the protocol implementation by directly launching the SUT is improbable to reveal flaws, as such compliant cases should have been covered in pre-release tests, we need to generate different variants as inputs for the SUT parties. To this end, instead of letting the two parties communicate normally, BLEEM interacts directly with both the server and the client, which enables intercept-

ing their exchanged traffic and introducing adversely crafted packets for testing. Therefore, for test initialization, we need to provide BLEEM with the server’s service address (e.g., the socket address) and configure BLEEM with a different service address for the incoming connection from the client. In this way, BLEEM can collect the output packets sent by the SUT parties. These packets are then utilized for SUT inner state analysis and serve as a basis for packet generation. BLEEM leverages *Scapy*’s [50] parsing capability to better analyze the intercepted packets. For unsupported protocols, we can extend *Scapy* by identifying the protocol format (i.e., data model), which is more accessible than traditional generation-based fuzzers that require both the data model and the state model. Meanwhile, we can simplify this step by resorting to protocol reverse engineering, as discussed in §8.

**Workflow.** As shown in Figure 3, BLEEM consists of two components: the feedback collector and the guided fuzzing module. In each iteration, the noninvasive feedback collector captures the network traffic during the SUT execution until it finishes. After that, it analyzes the traffic, extracts message semantics to construct an *abstract packet sequence*, and translates the *abstract packet sequence* to a state trace, which incorporates both the client and server and serves as the SUT feedback to be passed to the guided fuzzing module.

The guided fuzzing module then merges the state trace to a *System State Tracking Graph* (SSTG), refines the SSTG’s traverse probability distribution, and applies a guided fuzzing strategy to select mutation operators and generate *packet pattern sequence* to be instantiated. Specifically, we devise the SSTG to describe the all-inclusive state-space of the SUT. BLEEM initializes the SSTG with the initial session embedded in the SUT and dynamically updates it with new findings, i.e., new states or transitions, discovered in the feedback. Then BLEEM instantiates the *packet pattern sequence* based on the packets exchanged between the SUT parties. It simultaneously interacts with both the client and server to intercept the exchanged packets, which are typically syntactically and semantically correct. Thus, the packet sequences generated based on them have a high probability of exercising deep protocol logic. We dive into the details in the following sections.

## 4 Feedback Collector

To augment BLEEM’s capability of obtaining feedback when fuzzing the targets of black-box nature, we propose to retrieve system feedback based on system output instead of invasive execution inspection. Unlike ordinary application programs, the protocol implementations are tailored for entity communication over networks. Hence, the program output, i.e., packets, can be adopted as the indicator of its inner state. Taking the QUIC session in Figure 1 as an example, when the client sends an `Initial[CRYPTO]` packet to the server, we can infer that the client may stay in the initial state of the connection. When the client sends `Initial[ACK]` and

`Handshake[CRYPTO,ACK]` packets, we can infer that it has completed the handshake and is ready to enter the data transmission phase. Furthermore, the packets’ structure is typically standardized and protocol-compliant, thus facilitating analysis. Based on this insight, BLEEM works as shown in Figure 4.

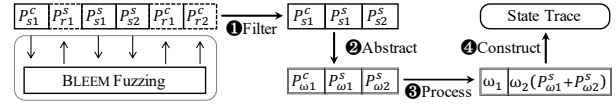


Figure 4: Feedback collector workflow

**Filtering the Packet Sequence.** In Figure 4,  $P_{s1}^c$  denotes the 1st packet sent by the client, while  $P_{r1}^s$  denotes the 1st packet received by the server. Given the network traffic captured during fuzzing, the feedback collector picks out the packets associated with the SUT state to assemble a packet sequence  $\mathcal{S}$ , where the relative order of these packets remains the same as in the original traffic (①). In detail, we only consider the packets sent by the SUT parties (e.g.,  $P_{s1}^c$ ) and discard their received packets (e.g.,  $P_{r1}^s$ ), which BLEEM sends for fuzzing purposes and thus cannot reflect the internal state.

**Abstracting the Packet Sequence and Processing.** To indicate the SUT states, using the concrete packets directly can create confusion because some fields, e.g., the data field, own low association with the system state, and their possible values are sometimes infinite. Take the `Initial[CRYPTO]` packet in Figure 2 as an example. The value of the `DCID` field used to identify a connection (Line 23) is randomly generated at the start of the connection [32]. Considering this field can make the state space too big or infinite. Therefore, we need to abstract away some details and focus on the crucial semantic information carried by the packets.

We abstract the packets in  $\mathcal{S}$  one by one to obtain an elementary *abstract packet sequence*  $\pi$  (②). For each packet  $P \in \mathcal{S}$ , the feedback collector employs *Scapy* to parse it and gets the dissection  $P_d$ . The dissection result is a hierarchically organized list of fields, where each field is represented as a type-value pair. Based on  $P_d$ , the module constructs the corresponding packet abstraction  $P_\omega$  by retaining these fields of type *enumeration*, which is a general field type defined in *Scapy* to describe the fields whose valid values are taken from a fixed small set. Through our investigation of over 50 supported protocols in *Scapy*, we found that the different values of the *enumeration* field typically represent different types of packets or frames (some protocols’ packet payload consists of a sequence of complete frames, like QUIC). In this way, BLEEM abstracts the first flight of the QUIC handshake (① in Figure 1, Lines 16-27 in Figure 2) as `Initial[CRYPTO]` by retaining the packet type `Initial` and frame type `CRYPTO` and their hierarchical relationships. We also provide a detailed example in Figure 11 in Appendix D. Moreover, BLEEM caches an intermediate map between each *abstract packet*  $P_\omega$  and its corresponding recent concrete packet  $P$  in a data structure called Oracle Map, which is further explored in §5.4.

Then, to facilitate the state trace construction, we further



process the *abstract packet sequence*  $\pi$  by concatenating these adjacent *abstract packets*  $P_{\omega}$ s sent from the same source into one *abstract packet*  $\omega$  (④). For example, in Figure 1, the two packets at ② can be abstracted and concatenated into `Initial[CRYPTO,ACK]+Handshake[CRYPTO]`. Therefore, in the final *abstract packet sequence*  $\pi$ , the adjacent two *abstract packets* are sent by different protocol parties.

**Constructing the State Trace.** For a given *abstract packet sequence*  $\pi: \{\omega_1, \omega_2, \dots, \omega_n\}$ , each *abstract packet*  $\omega_i$  can only indicate the temporal state of the corresponding SUT party  $p_i$  instead of the whole SUT. Because there may be multiple paths that can reach this temporal state for the party  $p_i$ , but their corresponding temporal SUT states at the point  $\omega_i$  vary. For example, assume that the client sends either packet  $a$  or packet  $b$  ( $a$  and  $b$  are different types) to the server at a certain point, and both packets are unexpected to the server. In this case, the server would respond to either with the same error message  $c$ . When the server sends out packet  $c$ , the whole SUT state should be different under these two different paths. Hence, we treat the protocol fuzzing as a bipartite SUT and model its state by introducing some prior information. However, considering a complete history can cause state space explosion. We have found that considering the most item in the bi-directional communication is enough to strike a practical balance in indicating the temporal state of the whole SUT while minimizing the overall state tracking space.

**Definition 1 SUT State.** A *SUT State* incorporates both parties of the SUT and can be represented as an ordered pair of objects in the form of  $\langle Obj_1 | Obj_2 \rangle$ . Each object is taken from the set  $\{T(\omega) | T \in \{C, S\}, \omega \in \Omega\}$ , where  $C$  represents the client while  $S$  represents the server, and  $\Omega$  is an *abstract packet* alphabet. Meanwhile, the order of these two objects is significant:  $\langle T_1(\omega_1) | T_2(\omega_2) \rangle$  represents that the party  $T_1$  is going to respond  $\omega_1$  (the prior), with the precondition that the party  $T_2$  has sent  $\omega_2$  (the posterior). Note that  $\omega_2$  only records the most recent *abstract packet* delivered by  $T_2$ .

In this way, we can infer the executed *SUT States* by analyzing each pair of adjacent *abstract packets*  $\omega_i$  and  $\omega_{i+1}$  (their sources are opposite as guaranteed above) in  $\pi$ . Then, adding transitions between two adjacent *SUT States* forms a state trace (④). We represent the involved *SUT states* in Figure 1 using an *abstract packet* alphabet  $\Omega$  with 5 symbols:

$$\begin{aligned} \Omega = \{ & \emptyset: \text{NoPacket}, a: \text{Initial}[\text{CRYPTO}], \\ & b: \text{Initial}[\text{CRYPTO,ACK}] + \text{Handshake}[\text{CRYPTO}], \\ & c: \text{Initial}[\text{ACK}] + \text{Handshake}[\text{CRYPTO,ACK}], \\ & d: 1\text{-RTT}[\text{APPLICATION\_CLOSE}] \}, \end{aligned} \quad (1)$$

The *SUT State* at point ① can be represented as  $\langle C(a) | S(\emptyset) \rangle$ , indicating that the server has sent no packet while the client is going to respond  $a$  (`Initial[CRYPTO]`). Similarly, the *SUT State* at point ② can be represented as  $\langle S(b) | C(a) \rangle$ . In this way, the state trace constructed for this session would be:

$$\langle C(a) | S(\emptyset) \rangle \rightarrow \langle S(b) | C(a) \rangle \rightarrow \langle C(c) | S(b) \rangle \rightarrow \langle S(d) | C(c) \rangle \quad (2)$$

## 5 Guided Fuzzing

Empowered by the feedback collector, BLEEM can track the state transitions of the SUT and determine whether the SUT reaches a new, previously untraversed state region. We devise the SSTG to represent the explored all-inclusive system state-space formally. Taking the perspective of the SSTG, we introduce protocol-aware mutation operators to support diverse packet generation for different states. We also extend the SSTG with richer information to embrace the ability to provide a guideline for reaching the desired state. We design a guided sequence generation strategy to effectively explore the state space and leverage the packet instantiation sub-module to provide high-quality packet sequences.

### 5.1 Mutation Operators

We introduce protocol-aware mutation operators that work on different levels, including the packet and sequence levels.

**Packet-Level Mutation Operator.** This mutation operator (denoted as  $\sigma_P$ ) operates on the fields in the packet. For a given packet, it randomly selects several fields and performs corresponding mutation operations according to the field type. *Scapy* has identified five general field types, including *NumberField*, *StringField*, *ListField*, *EnumerationField*, and *LengthField*. Thus, based on their features, we devise mutation operators for them. For example, the *NumberField* mutation operator performs random addition or subtraction operations to the original value while considering the valid value range. We provide further details in Appendix A.1.

**Sequence-Level Mutation Operator.** This mutation operator (denoted as  $\sigma_S$ ) operates on the packets in the sequence. BLEEM offers the following two kinds of operators:

1. *Packet duplication.* Given a packet sequence, this operator selects an arbitrary packet  $P_i$  and, after it, introduces several duplicate copies of  $P_i$  (denoted as  $P_i^*$ ). Figure 5(a) gives an example. The original sequence is  $[P_1, P_2, P_3, P_4 \dots]$ . If selecting  $P_2$  and duplicating it twice, the sequence after mutation would be  $[P_1, P_2, P_2^*, P_2^*, P_3, P_4 \dots]$ . Note that the subsequent packets in the sequence may change after duplication, and we introduce the symbol “ $F$ ” to represent the packets lying after the manipulated packets. In this case, the packet following the intentionally introduced  $P_2^*$ s may no longer be  $P_3$  in real traffic. We use  $F_5$  to represent the fifth packet in the mutated sequence.
2. *Packet disordering.* Given a packet sequence, this operator selects an arbitrary packet  $P_i$  (the  $i$ -th one in the original order) and changes its order to be delivered. Similarly, Figure 5(b) gives an example, the fourth packet  $P_4$  in the sequence  $[P_1, P_2, P_3, P_4 \dots]$  is selected to be delivered after  $P_1$ , namely deleting  $P_2$  and  $P_3$  and yielding the mutated sequence  $[P_1, P_4, F_3, F_4 \dots]$ .

These two sequence-level mutation operators are designed to stress test anomalies under uncertain network environments that may introduce dynamic packet delays or losses, espe-

cially for the protocols over UDP. Specifically, the *packet disordering* operator enables the discovery of anomalies triggered by the abnormal order of packets carrying commands. For instance, the File Transfer Protocol (FTP) applies a set of commands to comprise the control information flowing from the user-FTP to the server-FTP process [47]. Consider a simplified legal command sequence from the user-FTP: [USER test, PASS test, STOU test.txt, ..., DELE test.txt], which means that the user-FTP first logs in to the server-FTP with username test and password test, then uploads a file test.txt, performs some actions, and finally deletes it. Due to possible malicious actions induced by adversaries, the server-FTP implementation should correctly handle bad command sequences, e.g., deleting the PASS test in the sequence to attempt an authentication bypass. The packet sequence given in this example is one-way for ease of understanding. In practice, BLEEM operates over a two-way packet sequence, i.e., packets exchanged by the two parties.

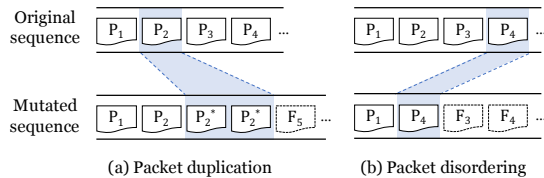


Figure 5: Sequence-level mutation operators

**Formalization.** To facilitate the representation, we can also regard the sequence-level mutation operator as operating on the involved packet, in line with the packet-level mutation operator. For example, the *packet duplication* in Figure 5(a) can be considered as operating on  $P_2$ . Similarly, the *packet disordering* shown in Figure 5(b) can be considered as operating on  $P_2$ , i.e., substituting  $P_2$  with cached  $P_4$  in the Oracle Map. Based on this premise, we introduce a symbol  $\oplus$  to associate the packet  $P$  with the adopted mutation operator  $\sigma$ . In this way,  $P \oplus \sigma$  can be used to represent the packet(s) generated by mutating  $P$  using  $\sigma$  (we use “packet(s)” because the *packet duplication* mutation operator can generate multiple packets). Hence,  $P \oplus \sigma$  can be considered a *packet pattern*, which can be employed to create new packet(s). Additionally, we introduce  $\sigma_\circ$  to indicate performing no mutation operation. Due to the inherent randomness of any mutation operator  $\sigma$  ( $\sigma \neq \sigma_\circ$ ), the set of the packet(s) that can be generated by the *packet pattern* is typically infinite. As a further generalization, the involved packet  $P$  can also be given on an abstract level. In this case, it is necessary to instantiate  $P$  before applying the mutation operators.

## 5.2 System State Tracking Graph

We devise the SSTG to identify the explored state space of the SUT and extend it with richer information to mitigate the reproducibility problem mentioned above.

**Labeling the State Transitions.** First, for the state trace

obtained from the feedback collector, BLEEM labels the triggering condition of each transition. Instead of retaining the concrete packets directly, as existing approaches do, BLEEM records how to generate these packets using the corresponding *abstract packet*  $\omega$  and the mutation operator  $\sigma$  selected to operate on it, i.e., *packet pattern*  $\omega \oplus \sigma$ . The *abstract packet*  $\omega$  can be extracted from the source *SUT State* as BLEEM generates packets based on the packet received from the SUT parties. The corresponding mutation operator  $\sigma$  can be obtained from BLEEM’s execution record. Meanwhile, we do not track the details of the mutation operators to avoid the state space explosion problem.

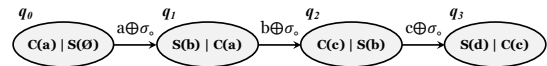


Figure 6: The initial SSTG for the QUIC session in Figure 1

For example, we label the state trace in Formula 2, which is equivalent to BLEEM directly forwarding the received packets without mutation (i.e.,  $\sigma_\circ$ ). Figure 6 shows the result. In detail, the SUT starts at the state  $q_0$ , where the client is going to output packet  $a$ . After receiving  $a$ , BLEEM generates packets as input for the SUT by mutating  $a$  using  $\sigma_\circ$ , namely  $a \oplus \sigma_\circ$  (actually,  $a \oplus \sigma_\circ$  is delivered to the server at the micro level, which is achieved by the packet instantiation sub-module in §5.4). After reading the input  $a \oplus \sigma_\circ$ , the SUT transitions to the state  $q_1$ . The remaining states are traversed similarly.

**Merging the State Trace.** Then, we use the SSTG to track the explored state space by continually merging the state trace.

**Definition 2** *System State Tracking Graph (SSTG)*. A SSTG is a variant of nondeterministic finite automaton (NFA), and can be represented formally by a 5-tuple  $(Q, q_0, \Omega, \Sigma, \Delta)$ , such that:  $Q$  is a finite set of the *SUT State*,  $q_0$  is the initial (or start) state,  $\Omega$  is an *abstract packet* alphabet,  $\Sigma$  is a set of mutation operators, and  $\Delta$  defines a transition function:  $Q \times \{\Omega \oplus \Sigma\} \rightarrow \mathcal{P}(Q)$ , where  $\mathcal{P}(Q)$  denotes the power set of  $Q$ . Due to the inherent randomness of any mutation operator  $\sigma \in \{\Sigma \setminus \sigma_\circ\}$ , for an *abstract packet*  $\omega \in \Omega$ , the set of packet(s) that can be generated using *packet pattern*  $\omega \oplus \sigma$  is infinite. Hence, the SSTG is nondeterministic.

BLEEM initializes the SSTG using the state trace defined in the initial dialog. For example, if BLEEM conducts fuzzing based on the dialog in Figure 1, Figure 6 gives the initial SSTG over the alphabet  $\Sigma = \{\sigma_\circ\}$  and  $\Omega$  given in Formula 1.

BLEEM interacts with the SUT parties during the fuzzing and generates diverse packets for the SUT parties instead of forwarding the received packets directly (which is equivalent to establishing the initial dialog). Although this may bring on different state traces, these state traces have the same start state. Because BLEEM conducts fuzzing based on the interactive traffic, the launch of each iteration is driven by the first action of the SUT. For example, as for the fuzzing based on the dialog in Figure 1, the client would always send the Initial[CRYPTO] packet to BLEEM at the beginning of each iteration. The difference lies in the follow-up actions



of BLEEM, e.g., how BLEEM mutates the received packets. Therefore, in order to merge the state trace into the implemented SSTG, BLEEM starts from SSTG's start state, which is also the start state of the state trace to be merged, merges their shared nodes or transitions, and updates the SSTG with new states or state transitions discovered in the state trace.

Furthermore, BLEEM implements a power schedule that dynamically refines the priority of exercised transitions during the merging. For a transition  $\tau$  with exercised times  $T_\tau$ , its priority is calculated by  $h(T_\tau)$ , where  $h(x)$  is a decreasing function. That is to say, the newly added transition during merging will be assigned the highest priority.

The SSTG captures the exercised state space of the protocol implementation and provides an efficient way for the state space traversing: given a target state  $q_i$  to be reached, the transition labels on any trace from  $q_0$  to  $q_i$  form a *packet pattern sequence*, which provides a guideline for the SUT input generation.

### 5.3 Guided Sequence Generation

Navigated by the SSTG, we devise heuristics to efficiently explore unknown state space and facilitate comprehensive traversal. This module outputs the *packet pattern sequence* and passes it to the packet instantiation sub-module as a guideline for packet sequence generation.

---

#### Algorithm 1: Guided packet sequence generation

---

**Input:** SSTG: *System State Tracking Graph*

**Input:**  $P_T$ : transition priority

**Output:**  $\varphi$ : *packet pattern sequence*

```

1  $S \leftarrow \text{TRAVERSE}(\text{SSTG})$ 
2 if  $S$  is not empty then
3    $q \leftarrow \text{SELECTONE}(S)$ 
4    $\varphi \leftarrow \text{ENTERANDAPPEND}(\text{SSTG}, q)$ 
5 else
6    $\varphi \leftarrow \emptyset$ 
7    $q \leftarrow \text{GETINITIALSTATE}(\text{SSTG})$ 
8   while  $\text{HASSUCCESSOR}(q)$  do
9      $\tau \leftarrow \text{SELECTTRANSITION}(\text{SSTG}, P_T, q)$ 
10     $\varphi \leftarrow \varphi \cup \text{GETPACKETPATTERN}(\tau)$ 
11     $q \leftarrow \text{TRANSITION}(\text{SSTG}, q, \tau)$ 
12 return  $\varphi$ 

```

---

Algorithm 1 provides an overview of the process: first, we attempt to stress test each SSTG state with diverse inputs by utilizing the proposed mutation operators (Lines 1-4). Specifically, for a *SUT State*  $\langle C(\alpha) | S(\beta) \rangle$ , although the state output  $\alpha$  is “deterministic” (the corresponding concrete packets of each execution may vary in some details) such that BLEEM receives the same packet type  $\alpha$  as the mutation basis, we can select a different mutation operator  $\sigma$  to construct a different *packet pattern*  $\alpha \oplus \sigma$  as test inputs. To this end, we first check whether there is a state  $q$  that can be further exercised by other kinds of *packet patterns* that have not been

applied (Line 1). If so, we construct a *packet pattern sequence*  $\varphi$  that can reach  $q$  and then append the desired *packet pattern* after  $\varphi$  (Line 4). For example, for the SSTG in Figure 6 with  $\Sigma' = \{\sigma_o, \sigma_P, \sigma_S\}$ , if stressing  $q_1$  with a new *packet pattern* that combines  $q_1$ 's output  $b$  with an unused mutation operator  $\sigma_P$ , i.e.,  $b \oplus \sigma_P$ , then we can construct such a *packet pattern sequence*:  $[a \oplus \sigma_o, b \oplus \sigma_P]$ . Second, after all the *SUT states* have been exercised by all kinds of *packet patterns* that are available, we attempt to facilitate comprehensive traversal of the implemented SSTG by steering towards low-density regions. We start from the initial state (Line 7) and then run in a loop until reaching an end state (Line 8): in each step, we select the one with the highest priority among the available transitions of the corresponding state (Line 9), record the corresponding *packet pattern* labeled on it (Line 10), and take this transition (Line 11). It is worth noting that even if we have once applied a *packet pattern*  $\mathcal{P}$  to stress a state  $S$  and failed to discover new behaviors, it is also worthwhile trying  $\mathcal{P}$  on  $S$  later because, as mentioned above, the set of the packet(s) that  $\mathcal{P}$  can produce is typically infinite.

### 5.4 Packet Instantiation

For a given *packet pattern sequence*  $\varphi: [\omega_1 \oplus \sigma_1, \omega_2 \oplus \sigma_2, \dots, \omega_n \oplus \sigma_n]$ , the packet instantiation sub-module generates a concrete packet sequence conforming to  $\varphi$  as the SUT input. Since the packets  $\omega_i$ s in  $\varphi$  are given on the abstract level, we need to instantiate them while maximally guaranteeing their syntactic and semantic correctness. To this end, this module utilizes the protocol logic encoded in the SUT parties. It runs as a proxy and simultaneously interacts with both the client and server, thereby leveraging their exchanged packets and introducing crafted packets.

The adjacent *packet patterns* in  $\varphi$  are prepared for different SUT parties, as guaranteed by the SSTG. The packet instantiation sub-module instantiates them one by one. For example, for a *packet pattern*  $\omega_i \oplus \sigma_i$  to be instantiated as server input, the packet instantiation sub-module intercepts a packet  $P$  (if there are multiple packets, concatenate them into one) from the client and checks whether  $P$  conforms to  $\omega_i$ 's structure: (i) if so, it directly takes  $P$  as the instantiation of  $\omega_i$  and then performs the corresponding mutation operator  $\sigma_i$  on  $P$  to generate test packet; (ii) if not, indicating that the realistic inner state transition of the SUT differs from the one designed by  $\varphi$ , there are two possible situations: (a) some new states or state transitions are discovered; and (b) due to SSTG's inherent non-deterministic, some transitions were not taken as expected, causing a different state trace. In these situations, to strike a balance between comprehensive traversal of the implemented SSTG and efficient exploration of unknown state space, BLEEM randomly selects the following two strategies to generate the concrete packets: (1) neglect  $\omega_i$  and perform  $\sigma_i$  on  $P$  directly; and (2) resort to the Oracle Map for the packet conforming to  $\omega_i$  and perform  $\sigma_i$  on it.

## 6 Evaluation

In this section, we implement and evaluate BLEEM to answer the following three research questions:

- RQ1** Is BLEEM more effective and efficient than traditional protocol fuzzers? (§6.3)
- RQ2** Is BLEEM effective in exposing unknown vulnerabilities in real-world protocol implementations? (§6.4)
- RQ3** Does the guided packet sequence generation strategy contribute to the effectiveness of BLEEM? (§6.5)

### 6.1 Implementation

We implement a prototype of BLEEM in Python 3 and use a modular approach to facilitate further extensions.

**Feedback Collector.** We implement the analysis of collected packets based on the packet parsing capability provided by *Scapy* [50], a packet manipulation library that supports more than 140 common protocols. Given the parse result with detailed field type and value along with the structure information (e.g., the hierarchical relationship of a packet and its contained frames), we abstract each packet by retaining the *enumeration* field values while maintaining the packet structure. Then we construct the *abstract packet sequence* by concatenating adjacent *abstract packets* with the same source. Based on this process, we implement the *state trace construction* following the construction rule in §4. Furthermore, to demonstrate the scalability of BLEEM, we also selected two protocols, i.e., QUIC and SSH, that have not been supported by *Scapy* for the experiment, as shown in §6.2. One of the authors completed coarse-grained *Scapy* extensions for these two protocols in about six hours.

**Guided Fuzzing Module.** First, we implement the proposed mutation operators that work on two levels: (i) to support the *packet-level mutation*, we design and implement mutation strategy for the five general field types used in *Scapy*, as summarized in Appendix A.1; and (ii) we perform the *sequence-level mutation* based on the Oracle Map, which caches an intermediate map between each *abstract packet* and its corresponding recent concrete packet. These two mutation operators are applied in the packet instantiation sub-module. Second, following the rule in §5.2, we implement the runtime construction of the SSTG by continuously merging *abstract packet sequences* and refining the priority of transitions. Based on it, we implement the sequence generation strategy proposed in §5.3 for efficient state space traversal. Third, the *packet instantiation sub-module* interacts with the client and the server simultaneously and intercepts the exchanged packets. These packets usually contain several layers (e.g., network layer and transport layer). To focus on the target protocol's layer  $\ell$ , we implement the proxy that provides reliable communication underlying  $\ell$ . At present, we support layers in the network stack. We present the details in Appendix A.2.

**Crash Detection.** Besides the widely-used local process monitoring, we also implement several network monitors that can remotely detect system crashes to support black-box fuzzing. We provide the details in Appendix A.3.

### 6.2 Experiment Setup

**Subjects.** We measured the performance of BLEEM on two benchmarks. (i) *Open-source protocol implementations.* We selected several open-source implementations of widely used protocols, as shown in Figure 7 and Table 5 in Appendix B. The selected protocols are pretty diverse, ranging from security protocols to messaging protocols, transport protocols, and industrial control system protocols. These protocols' corresponding implementations are typical and widely used in practice. To facilitate a fair comparison, all the selected SUTs (cf. the fourth column of Table 5) come from off-the-shelf utilities. (ii) *Closed-source protocol implementations.* We also collected firmware of mainstream IoT manufacturers and used the contained protocol binaries as the closed-source targets.

**Compared Fuzzers and Manual Configurations.** Since BLEEM is a black-box fuzzer, we select three famous black-box protocol fuzzers widely used in academia and industry as baselines for black-box schemes, including Peach [18], BooFuzz [33], and Snipuzz [20]. Also, to demonstrate the effectiveness of BLEEM, we pick AFLNet [45] and SGFuzz [8], two state-of-the-art grey-box protocol fuzzers integrating coverage and state feedback, for comparison.

We use the corresponding server utilities of the selected SUTs to evaluate the compared protocol fuzzers (these fuzzers can only test one side of the protocol at a time, and they usually focus on the server side). To adapt these fuzzers to the selected subjects, we followed their tutorials [5, 10, 18, 30, 33] to prepare the required configurations: (i) For generation-based fuzzers BooFuzz and Peach, we offered the required test model for each implementation by modeling the corresponding session embedded in the dialog of the SUT. Specifically, in the test model, we depicted the format of the involved protocol messages (data model) and their sequencing over the session (state model). (ii) For mutation-based fuzzers Snipuzz, SGFuzz, and AFLNet, we prepared the initial seed corpus based on the messages captured by executing the SUT. We evaluated SGFuzz on partial TCP-based protocol implementations due to the limitation of the *netdriver* it bases on and the in-process requirement of its base fuzzer Libfuzzer [10]. We also extended AFLNet to recognize the protocols that have not been supported (e.g., QUIC and PPTP).

**Experiment Settings.** Since the fuzzing performance fluctuates to a certain degree due to the inherent randomness, we ran each fuzzing tool on each selected project with a 24-hour time budget and repeated each 24-hour experiment 10 times to establish statistical significance of results [35]. For fairness, each fuzzing campaign runs on a Docker container that is configured with 1 CPU core and 1G RAM.

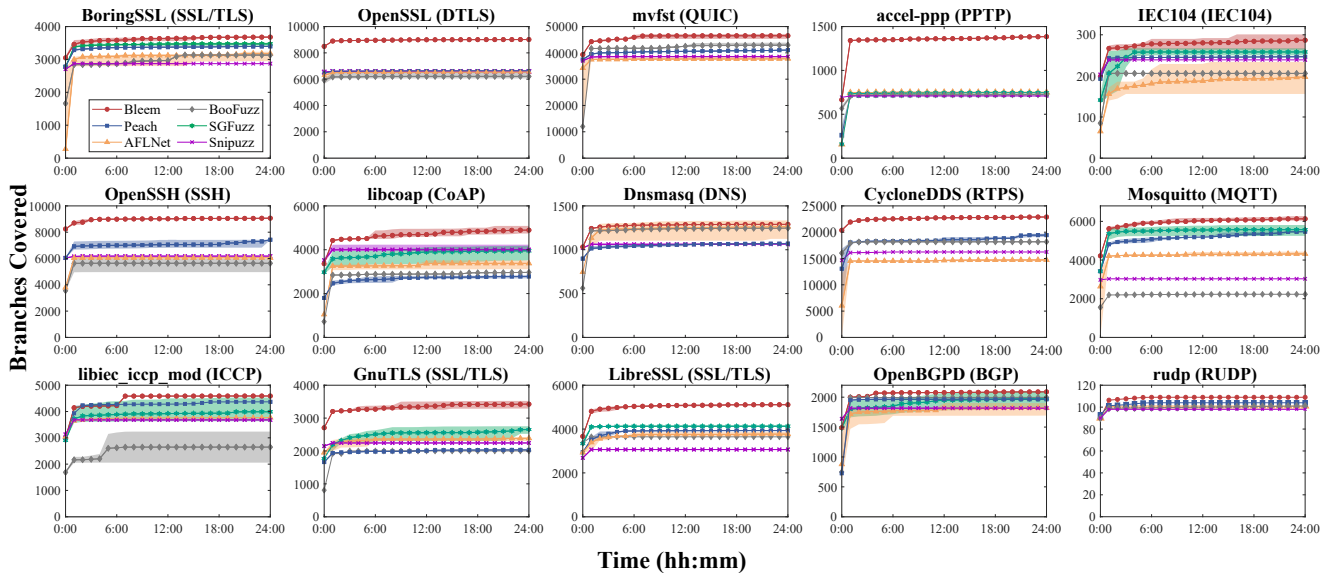


Figure 7: The number of unique branches covered (only on the server-side) by different fuzzers on each protocol implementation over ten 24-hour runs. The average number of discovered branches is displayed, alongside with minima and maxima over the individual runs.

### 6.3 Coverage Analysis

Since the compared fuzzing schemes are pretty diverse, we need a uniform metric for a fair comparison. Branch coverage is a commonly used metric to measure the effectiveness of fuzzers in software testing. Therefore, we use branch coverage as the metric for comparison and utilize LLVM’s SanitizerCoverage [36] to count the number of unique branches covered by each fuzzer on a target program.

Figure 7 shows the branches covered on the server side by different fuzzers. On average, BLEEM achieved 40.3%, 35.7%, 23.4%, 48.9%, and 28.5% higher branch coverage than Snipuzz, AFLNet, SGFuzz, BooFuzz, and Peach, respectively, within 24 hours. All results are statistically significant according to the Mann–Whitney U test, as recommended by Klees et al. [35]. In 64 out of 69 times, the *minimum* branches achieved by BLEEM exceed the *maximum* branches of the prior approaches. In other words, even the worst run of BLEEM performs better than the best run of other prior approaches, except for partial runs on Dnsmasq, IEC104, and OpenBGPD. The reason is that these protocols employ simple interactive logic and packet format. Hence, other fuzzers also did well on it. Even so, BLEEM also outperforms them on the average branches achieved. DTLS and SSH employ random nonces to provide replay protection [4, 39], as mentioned in §1. When adapted to their implementations, i.e., OpenSSL and OpenSSH, existing protocol fuzzers have difficulty in completing the handshake, thus covering fewer branches. In contrast, BLEEM can penetrate deeply into the protocol logic by utilizing the intercepted interactive traffic and the proposed SSTG. TLS also employs this mechanism, and the results of other fuzzers on BoringSSL are fine. The reason is that we enabled the fuzzer mode provided by BoringSSL [11] in the experi-

ments. This mode modifies the library to disable randomness and thus is more friendly to traditional fuzzers. Still, BLEEM covers more branches than prior techniques on BoringSSL.

**Answer to RQ1.** Overall, BLEEM can achieve higher coverage than existing protocol fuzzers, which means that BLEEM can test protocol implementations broadly and deeply.

### 6.4 Bug-Detection Capability

To measure the bug detection capability, we adapt BLEEM to fuzz real-world protocol implementations, including open-source and closed-source.

**Open-Source Targets.** We use the number of unique vulnerabilities reported by AddressSanitizer [51] and Undefined-BehaviorSanitizer [13] (a.k.a., ASan and UBSan) as the uniform metric. The reason is that the vulnerability detection methods of BLEEM and other fuzzers vary. For example, the classic black-box fuzzer Peach typically detects vulnerabilities by checking the liveness of under-test service via port probing. However, not all the vulnerabilities (e.g., some buffer-overflow vulnerabilities) will crash the program. Therefore, we utilize ASan and UBSan to enhance the target program and use the crashes identified by different fuzzers as the metric to represent their vulnerability detection ability. Furthermore, some Sanitizer-reported crashes may result from the same root cause. To eliminate duplicate entries, we utilize the stack traces in the Sanitizer report for bug deduplication and only consider unique vulnerabilities.

BLEEM has detected 15 new vulnerabilities in several extensively used implementations of well-known protocols, with 10 CVE identifiers assigned after a coordinated disclosure. We also tried to reproduce these bugs using the other fuzzers based on the similar configuration construction method mentioned in §6.2. Table 1 summarizes the vulnerabilities exposed



by BLEEM and whether other fuzzers can find them. Specifically, Peach, BooFuzz, AFLNet, SGFuzz, and Snipuzz can only expose 8, 5, 6, 7, and 5 bugs, a strict subset of the bugs uncovered by BLEEM. These protocol implementations have been thoroughly tested, and some of them, e.g., GnuTLS [1] and LibreSSL [2], have even been incorporated into the OSS-Fuzz [27], which demonstrates the effectiveness of BLEEM in bug detection. Some of these bugs are hard to trigger, and we provide the bug details and a case study in Appendix C.

Table 1: Previously unknown vulnerabilities exposed by BLEEM and the statistics of the compared fuzzers

Subject	Type	AFLNet	Snipuzz	SGFuzz	BooFuzz	Peach	BLEEM	CVE ID
LibreSSL	Stack Buffer Overflow						●	CVE-2021-41581
GnuTLS	Null Pointer Dereference					●	●	CVE-2021-4209
BoringSSL	SIGPIPE						●	-
accel-ppp	Stack Buffer Overflow						●	CVE-2021-42870
accel-ppp	Stack Buffer Overflow						●	CVE-2021-42054
accel-ppp	Memory Leak	●	●	●	●	●	●	-
IEC104	Stack Buffer Overflow	●	●	●	●	●	●	CVE-2020-20486
IEC104	Segmentation Violation	●	●	●	●	●	●	CVE-2020-18731
rdup	Memory Leak	●	●	●	●	●	●	CVE-2020-20665
libiec_iccp_mod	Heap Buffer Overflow						●	CVE-2020-20490
libiec_iccp_mod	Heap Buffer Overflow						●	CVE-2020-20662
libiec_iccp_mod	Heap Buffer Overflow						●	CVE-2020-20663
OpenBGPD	Undefined Behavior	●	●	●	●	●	●	-
OpenBGPD	Undefined Behavior	●	●	●	●	●	●	-
mvfst	Heap Buffer Overflow						●	-
SUM		6	5	7	5	8	15	10 CVEs

**Closed-Source Targets.** We collected four firmware containing vulnerable protocol implementations, as disclosed by the CVE dataset [16], from different mainstream IoT manufacturers to evaluate the performance of the selected black-box fuzzers in discovering severe vulnerabilities. These CVEs seriously threaten various devices and are classified as CRITICAL by CVSS 3.x Severity and Metrics (see Table 4 in Appendix B). We compared BLEEM against selected black-box fuzzers and used the network-related monitors to detect crashes by checking the liveness of under-test services through port probing. We use the time to first crash as the metric to evaluate the bug-detection capability of these fuzzers. As shown in Table 2, BLEEM achieves the best CVE discovery performance compared to other fuzzers. BLEEM and Peach can find all of these CVEs, while BooFuzz and Snipuzz can find only 3 and 1, respectively. On average, BLEEM can find a crash at least 7.5 $\times$ , 13.3 $\times$ , and 87.1 $\times$  faster than Peach, BooFuzz, and Snipuzz, respectively, demonstrating BLEEM’s efficiency boost over the state-of-the-art.

Table 2: Average time to expose published CVEs

CVE ID	Protocol	Snipuzz	BooFuzz	Peach	BLEEM
CVE-2018-5767	HTTP	-	25min	34min	6min
CVE-2020-25067	UPnP	26min	36s	47s	33s
CVE-2019-14457	HTTP	-	-	652min	35min
CVE-2019-1663	HTTP	-	501min	307min	72min

**Answer to RQ2.** BLEEM is capable of finding unknown bugs effectively in real-world protocol implementations.

## 6.5 Effectiveness of Sequence Generation

To evaluate the effectiveness of the guided sequence generation (§5.3), we implemented BLEEMRand, a variant of BLEEM, in which we replaced it with random sequence selection and maintained the SSTG construction for comparison.

Table 3 shows the mean value of each metric across repetitions. The column “Paths” indicates the number of unique

state traces discovered during the SSTG construction, and the column “ $\overline{Len}$ .” indicates the average length of these paths. The column “Types” indicates the number of different types of *abstract packets* (after concatenation), which are the elements of the *SUT States*. The columns “Nodes” and “Trans.” indicate the state and state transition numbers of the SSTG, respectively. The “Branch Coverage” shows the achieved unique branches of the whole SUT, including the coverage achieved on both sides, which is different from §6.3 because both BLEEMRand and BLEEM can test the whole system. Note that the “Paths” and “ $\overline{Len}$ .” are not necessarily proportional to the complexity of the constructed SSTG, as different transition ways of existing nodes and transitions can trigger a new path while no new nodes or transitions will be found. The table shows that the overall unique paths are finite, indicating that our provision for the SSTG construction effectively avoids state space explosion. To further illustrate this, we also tried constructing a long initial sequence using the provided client utility when testing the Dnsmasq, libcoap, and Mosquitto. The results demonstrate that the path numbers achieved on these projects are also within an acceptable range.

Table 3: Statistics about the constructed SSTG and the unique branches achieved by BLEEM and BLEEMRand.

Subject	Fuzzer	SSTG Construction		SSTG Metrics			Branch Coverage
		Paths	$\overline{Len}$ .	Types	Nodes	Trans.	
BoringSSL	BLEEMRand	42	4.11	32	72	84	4293
	BLEEM	75	3.82	69	152	183	4549
OpenSSL	BLEEMRand	266	6.84	73	247	397	10512
	BLEEM	256	5.96	90	267	442	10614
mvfst	BLEEMRand	2352	7.83	492	1494	2806	53942
	BLEEM	10781	7.99	671	2779	6581	55575
accel-ppp	BLEEMRand	101	6.07	14	33	46	1384
	BLEEM	30	4.62	11	25	31	1385
IEC104	BLEEMRand	39	5.57	84	113	137	279
	BLEEM	49	5.86	88	149	164	321
OpenSSH	BLEEMRand	43	5.23	20	59	82	12444
	BLEEM	97	5.58	30	104	171	14579
libcoap	BLEEMRand	10013	85.08	325	1340	3400	8292
	BLEEM	10286	83.42	331	1427	4143	8530
Dnsmasq	BLEEMRand	2958	22.60	60	163	511	1271
	BLEEM	1783	14.62	58	155	413	1292
CycloneDDS	BLEEMRand	55	4.55	18	67	132	22912
	BLEEM	139	4.26	31	153	303	23710
Mosquitto	BLEEMRand	19522	15.67	215	1037	2815	9284
	BLEEM	20652	13.09	253	1085	3142	10285
libiec_iccp_mod	BLEEMRand	116	9.05	28	96	155	6059
	BLEEM	314	9.71	41	139	294	6265
GnuTLS	BLEEMRand	50	4.66	25	70	98	5057
	BLEEM	57	4.12	38	92	114	5222
LibreSSL	BLEEMRand	209	3.93	67	248	394	5473
	BLEEM	196	3.78	100	277	385	6157
OpenBGPD	BLEEMRand	222	17.73	35	92	131	2072
	BLEEM	253	17.21	43	111	169	2086
rdup	BLEEMRand	149	5.86	21	82	151	112
	BLEEM	154	5.70	30	109	185	115

From each row of Table 3, the complexity of our proposed SSTG is roughly in positive correlation to the packet types and the covered unique branches, indicating that the proposed SSTG can reflect the inner system execution status of the SUT in some degree. With the help of the guided sequence generation strategy, BLEEM achieves 5.7% more unique branches than BLEEMRand on average, and the improvement on the server is typically comparable to that on the client since they are mutually reinforcing. We also note that BLEEMRand performs better on Dnsmasq and accel-ppp in the complexity of the implemented SSTG. Through investigation, we found that the logic of the corresponding SUTs is relatively simple

compared with other subjects. The randomly generated long packet sequences of BLEEMRand can easily trigger more outputs of the SUT parties, resulting in a more complex SSTG. Nonetheless, BLEEMRand is hard to reach deep states in the protocol implementation without guided packet sequence generation, thus covering fewer branches than BLEEM.

**Case Study.** To intuitively illustrate how BLEEM implements guided fuzzing and its effectiveness, we use the session discovered during fuzzing *mvfst* as a case study. By executing the selected SUT of *mvfst*, BLEEM constructed an initial SSTG in Figure 6 as the basis. Directed by Algorithm 1, BLEEM tried to stress  $q_1$  with the *packet pattern*  $b \oplus \sigma_P$ , yielding the *packet pattern sequence*  $[a \oplus \sigma_o, b \oplus \sigma_P]$ . Then BLEEM instantiated it and triggered the session in Figure 8 (in reality the client and server interact with BLEEM’s packet instantiation sub-module, we omit it to facilitate the understanding). The mutated Initial[CRYPTO, ACK] (denoted as Initial[CRYPTO, ACK]\* ) transmitted at ② triggered retransmission of the client (③, this Initial[CRYPTO] differs from ① at the concrete level ). Through our investigation of the implementation logic, after reading the second Initial[CRYPTO], the server immediately responds ACK for ③ (Initial[ACK]) instead of packing ACK and CRYPTO frames into one Initial packet like ②. Then, it invokes *CloningScheduler*, a packet scheduler designed to clone exiting packets that are still outstanding, to derive the following Initial[CRYPTO] and Handshake[CRYPTO]. In this way, the testing procedure covered the logic of *CloningScheduler*, where a known problem has been exposed [19].

As a result, exercising  $q_1$  using the new *packet pattern*  $b \oplus \sigma_P$  achieved a new state trace. Empowered by the feedback collector, BLEEM can monitor this automatically and update it on the initial SSTG with an extended alphabet:

$$\tilde{\Omega} = \{e: \text{Initial}[\text{ACK}] + \text{Initial}[\text{CRYPTO}] + \text{Handshake}[\text{CRYPTO}], \\ f: \text{Handshake}[\text{CRYPTO}, \text{ACK}]\},$$

yielding the overall SSTG shown in Figure 9.

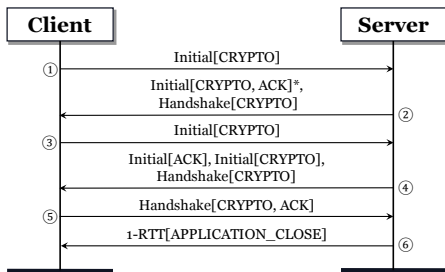


Figure 8: The *mvfst* handshake flow if the packet at ② is mutated (Initial[CRYPTO, ACK]\* ).

Unfortunately, triggering such logic is non-trivial for other fuzzers because they need to craft the two Initial[CRYPTO] packets carefully: (i) The two packets need to be syntactically correct, which can be easily guaranteed by generation-based fuzzers like Peach but is hard for mutation-based fuzzers like AFLNet. (ii) They need to correctly set some parameters in the packets to ensure semantic correctness. For example,

these two packets should own the same DCID field to identify the same connection, and the PN field value  $x$  of the former packet and the value  $y$  of the latter should satisfy  $y = x + 2$ . If these two packets are semantically incorrect, the server will reject them without a response. In the 24-hour experiments across 10 repetitions on *mvfst*, we found that all the compared fuzzers failed to trigger this behavior. Instead, BLEEM generates packets by resorting to the packets provided by the protocol parties. Therefore, it can easily guarantee the above conditions and trigger this behavior ( $q_4 - q_7$ ) at an early stage. Meanwhile, BLEEM discovered 410 more successor states on average based on  $q_4 - q_7$ . This state region is hard to trigger for other fuzzers, but discovering this region effectively contributes to BLEEM’s branch coverage.

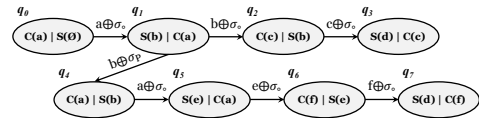


Figure 9: The *System State Tracking Graph* (SSTG) after introducing the new *packet pattern*  $(b \oplus \sigma_P)$  on  $q_1$ .

**Answer to RQ3.** The guided sequence generation strategy of BLEEM is able to increase the exposure of different protocol behaviors, thus contributing to fuzzing effectiveness.

## 7 Related Work

**Protocol Fuzzing.** Fuzzing has been widely adopted to test protocol implementations [23–25, 38, 61–63]. Existing works focus on individual packet generation and lose sight of contextual correctness between packets in a sequence. Peach [18] and BooFuzz [33] select a state trace in the user-defined state model each time and separately generate packets for these states. *Scapy* fuzzing API [49] generates individual packets based on given values, which is equivalent to a packet-level mutation operator. Although these approaches provide syntactically valid packets that prevent early parse error, they are incapable of handling the parameter dependency and generating correct values for the context-sensitive dynamic fields (e.g., handshake configuration of SSL). Meanwhile, they are pretty random and do not use feedback to guide fuzzing. Instead, BLEEM generates packets at a sequence level. It provides a sequence-based feedback mechanism to navigate protocol-logic exploration and a sequence-level manipulation to discover anomalies under out-of-order or duplicated packets. Meanwhile, it accounts for the contextual information by leveraging the intercepted packets between the SUT parties.

Some recent works introduce state awareness for protocol fuzzing. SGFuzz [8] relies on a programmatic intuition that the state variables used in protocol implementations encode fine-grained program processing actions and often appear in enum-type variables. It recognizes these variables, injects instrumentation to monitor their assignment, and uses their different values to identify different server program states.

In comparison, without instrumentation, BLEEM leverages the enum in the packet fields to identify packet type and constructs the states by combining the packet types with bidirectional communication information. We also provide an example in Appendix D. StateAFL [43] adopts fine-grained compile-time instrumentation to obtain runtime information for protocol state inference. Nyx-Net [48] solves the reproducibility problem mentioned in §1 by ensuring noise-free fuzzing through a snapshot-based approach. These approaches require the source code or the binary of the protocol implementation and thus do not scale to the black-box nature. Instead, BLEEM applies a noninvasive feedback mechanism and models state transitions across packet exchanges, thus can be scaled to diverse protocols even in the black-box setting.

Some works utilize the server response to optimize fuzzing. AFLNet [45] leverages the status code in the server response as state feedback and uses it to guide packet mutation. Snipuzz [20] infers the grammatical role of each message byte by analyzing the server response using a hierarchical clustering strategy. This strategy works well for IoT protocols whose responses are usually textual and organized in a common format such as JSON. These approaches work well but are tightly coupled with specific protocol formats. In contrast, BLEEM analyzes the semantics conveyed in the output to obtain system feedback and can be applied to both textual and binary protocols. Meanwhile, it analyzes both client and server output and thus can obtain more information.

**State Machine Inference.** The most closely related works employ learning algorithms to infer protocol state machines, and there are two different technologies.

*Active-inference-based* approaches [17, 21–23, 52] actively generate packet sequences to query a protocol implementation and infer a state machine using model learning algorithms, such as Angluin’s  $L^*$  algorithm [7]. Applying this algorithm usually requires tailoring a mapper to translate between the abstract alphabets of the model and the concrete packets of the implementation, which is not reusable for different protocols and implementations. Instead, BLEEM abstracts packets by automatic semantic extraction and instantiates packets using the interactive traffic.

*Passive-inference-based* approaches [14, 26, 28, 29] infer a state machine by analyzing a corpus of packet sequences sampled on the network. Pulsar [26] analyzes the sampled network traces and infers a generative model for message format and protocol states. AutoFuzz [28] analyzes the sampled traffic to infer the server’s finite state machine (FSM) and conducts server fuzzing based on this stationary FSM. These approaches perform fuzzing based on the inferred model. Therefore, their fuzzing effectiveness relies on the completeness of the captured network traffic. Instead, BLEEM constructs the initial SSTG based on the SUT dialog and gradually enriches it at runtime with the packets generated during guided fuzzing, which builds a closed-loop of vulnerability detection.

Most of all, we do not infer the protocol’s state machine but

derive the SSTG structure from the observed network traffic to identify the state space and guide fuzzing. Meanwhile, the SSTG encompasses all protocol parties, unlike the traditional state machine that depicts only one protocol party.

## 8 Discussion

Despite BLEEM’s positive results, we briefly discuss limitations and avenues of further improvement.

First, the feedback collector analyzes the output packets based on *Scapy*’s parsing capability. However, some protocols, especially those proprietary protocols, are not supported by *Scapy*. Since BLEEM can capture the network traffic and gain additional traffic beyond the initial session through fuzzing, we can recognize unsupported protocols by resorting to traffic-based protocol reverse engineering [12, 57].

Second, the current representation of the SSTG cannot always guarantee reproducibility due to its inherent non-deterministic and coarse-grained transition labeling. We can solve this by transforming the SSTG into a deterministic finite automata using typical algorithms [41, 55] and recording fine-grained mutation information, such as the detailed subclass and parameters of the mutation operator.

Third, BLEEM now supports crash detection and memory-related bugs with ASan/UBSan. BLEEM can also detect semantic bugs if provided with corresponding oracles. For example, if provided with the packet-exchange constraints, BLEEM can detect non-compliance with protocol specification by analyzing the SUT’s state trace. The cause of CVE-2021-40523 [42] is that the server may fail to send `WILL/WONT` response for `WILL` commands, which violates the property restricted in RFC 854 [46]. To detect this bug, we can provide such a packet-exchange constraint for BLEEM: a `WILL` packet from the client should be followed by a `WILL/WONT` packet from the server under normal conditions. With the help of the *abstract packets*, BLEEM can focus on the packet types, thus facilitating analysis. Then, it can detect the bug by checking, for the *SUT State* that matches  $\langle S(*) \mid C(\text{WILL}) \rangle$  (“\*” is a wildcard), whether its outgoing edge with label  $\text{WILL} \oplus \sigma_o$  transitions to state  $\langle C(\text{WILL}) \mid S(\text{WILL}) \rangle$  or  $\langle C(\text{WILL}) \mid S(\text{WONT}) \rangle$ . If not, a bug exists.

## 9 Conclusion

In this paper, we present BLEEM, a *packet-sequence-oriented* protocol fuzzer that applies an evolutionary approach to explore the massive protocol state space: it accesses the system feedback by analyzing the output sequences and dynamically tunes the exploration direction by applying the proposed guided fuzzing strategy. Meanwhile, BLEEM generates highly protocol-logic-aware packet sequences by leveraging the observed interactive traffic. Compared to state-of-the-art fuzzers, BLEEM can achieve higher coverage and detect more bugs in real-world protocol implementations. BLEEM is fully automatic and can be applied to test the implementations of most general protocols in the black-box setting.



## Acknowledgments

We thank the anonymous reviewers for their constructive feedback and suggestions. This research is sponsored in part by the National Key Research and Development Project (No. 2022YFB3104000, No2021QY0604) and NSFC Program (No. 62022046, 92167101, U1911401, 62021002).

## References

- [1] OSS-Fuzz/Gnutls. <https://github.com/google/oss-fuzz/tree/master/projects/gnutls>.
- [2] OSS-Fuzz/Libressl. <https://github.com/google/oss-fuzz/tree/master/projects/libressl>.
- [3] RFC 4346. The transport layer security (TLS) protocol. section f.1.1.2: Rsa key exchange and authentication. Website. <https://www.rfc-editor.org/rfc/rfc4346#appendix-F.1.1.2>.
- [4] RFC 6347. Datagram transport layer security version 1.2. section 4.1.2.6: Anti-replay. <https://datatracker.ietf.org/doc/html/rfc6347#section-4.1.2.6>.
- [5] aflnet. AFLNet: A greybox fuzzer for network protocols. <https://github.com/aflnet/aflnet>.
- [6] Pedram Amini and Aaron Portnoy. Sulley. 2012. <https://github.com/OpenRCE/sulley>.
- [7] Dana Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75, 1987.
- [8] Jinsheng Ba, Marcel Böhme, Zahra Mirzamomen, and Abhik Roychoudhury. Stateful greybox fuzzing. *2022 Usenix Security Symposium*.
- [9] Domagoj Babic, Stefan Bucur, Yaohui Chen, Franjo Ivancic, Tim King, Markus Kusano, Caroline Lemieux, László Szekeres, and Wei Wang. FUDGE: Fuzz driver generation at scale. *2019 ACM ESEC/FSE*.
- [10] bajinsheng. SGFuzz: Stateful greybox fuzzer. <https://github.com/bajinsheng/SGFuzz>.
- [11] BoringSSL. Fuzzer mode. <https://github.com/google/boringssl/blob/master/FUZZING.md>.
- [12] Georges Bossert, Frédéric Guihéry, and Guillaume Hiet. Towards automated protocol reverse engineering using semantic information. *9th ACM symposium on Information, computer and communications security*, 2014.
- [13] Clang. Clang 15.0.0 documentation, undefinedbehaviorsanitizer. <https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html>.
- [14] Paolo Milani Comparetti, Gilbert Wondracek, Christopher Krügel, and Engin Kirda. Prospex: Protocol specification extraction. *IEEE Symposium on Security and Privacy*, 2009.
- [15] CVE-2014-0160. Heartbleed - a vulnerability in openssl. 2014. <http://heartbleed.com>.
- [16] US National Vulnerability Database. Common vulnerabilities and exposures (CVE).
- [17] Joeri de Ruiter and Erik Poll. Protocol state fuzzing of TLS implementations. *2015 USENIX Security Symposium*.
- [18] Michael Eddington. Peach fuzzing platform. <https://gitlab.com/gitlab-org/security-products/protocol-fuzzer-ce>.
- [19] facebookincubator. A known problem in the CloningScheduler in Facebook mvfst. <https://github.com/facebookincubator/mvfst/blob/421196ec98a9abd69c7a4353c555a0c981a69109/quic/api/QuicBatchWriter.cpp#L17>.
- [20] Xiaotao Feng, Ruoxi Sun, Xiaogang Zhu, Minghui Xue, Sheng Wen, Dongxi Liu, Surya Nepal, and Yang Xiang. Snipuzz: Black-box fuzzing of IoT firmware via message snippet inference. *2021 ACM SIGSAC CCS*.
- [21] Tiago Ferreira, Harrison Brewton, Loris D'antoni, and Alexandra Silva. Prognosis: Closed-box analysis of network protocol implementations. *2021 ACM SIGCOMM*.
- [22] Paul Fiterau-Brostean, Ramon Janssen, and Frits W. Vaandrager. Combining model learning and model checking to analyze TCP implementations. *2016 CAV*.
- [23] Paul Fiterau-Brostean, Bengt Jonsson, Robert Merget, Joeri de Ruiter, Konstantinos Sagonas, and Juraj Somorovsky. Analysis of DTLS implementations using protocol state fuzzing. *2020 USENIX Security Symposium*.
- [24] Matheus E. Garbelini, Vaibhav Bedi, Sudipta Chattopadhyay, Sumei Sun, and Ernest Kurniawan. Braktooth: Causing havoc on bluetooth link manager via directed fuzzing. *2022 USENIX Security Symposium*.
- [25] Matheus E. Garbelini, Chundong Wang, Sudipta Chattopadhyay, Sumei Sun, and Ernest Kurniawan. Sweyn-Tooth: Unleashing mayhem over bluetooth low energy. *2020 USENIX Annual Technical Conference*.
- [26] Hugo Gascon, Christian Wressnegger, Fabian Yamaguchi, Dan Arp, and Konrad Rieck. Pulsar: Stateful black-box fuzzing of proprietary network protocols. *SecureComm*, 2015.

- [27] Google. OSS-Fuzz. <https://github.com/google/oss-fuzz>.
- [28] Serge Gorbunov and Arnold Rosenbloom. Autofuzz: Automated network protocol fuzzing framework. *Ijcsns*, 2010.
- [29] Yating Hsu, Guoqiang Shu, and David Lee. A model-based approach to security flaw detection of network protocol implementations. *IEEE International Conference on Network Protocols*, 2008.
- [30] Immor278. Snipuzz-py. <https://github.com/Immor278/Snipuzz-py>.
- [31] Kyriakos K. Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. FuzzGen: Automatic fuzzer generation. *2020 USENIX Security Symposium*.
- [32] Iyengar, J., Ed., and M. Thomson, Ed. QUIC: A udp-based multiplexed and secure transport, 2021. <https://www.rfc-editor.org/rfc/rfc9000.html>.
- [33] jtpereyda. BooFuzz: Network protocol fuzzing for humans. <https://github.com/jtpereyda/boofuzz>.
- [34] Jinho Jung, Stephen Tong, Hong Hu, Jungwon Lim, Yonghui Jin, and Taesoo Kim. WINNIE: Fuzzing windows applications with harness synthesis and fast cloning. *2021 NDSS*.
- [35] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael W. Hicks. Evaluating fuzz testing. *2018 ACM SIGSAC CCS*.
- [36] LLVM. SanitizerCoverage. <https://clang.llvm.org/docs/SanitizerCoverage.html#edge-coverage>.
- [37] Zhengxiong Luo, Feilong Zuo, Yu Jiang, Jian Gao, Xun Jiao, and Jiaguang Sun. Polar: Function code aware fuzz testing of ICS protocol. *ACM Trans. Embed. Comput. Syst.*, 18:93:1–93:22, 2019.
- [38] Zhengxiong Luo, Feilong Zuo, Yuheng Shen, Xun Jiao, Wanli Chang, and Yu Jiang. ICS protocol fuzzing: Coverage guided packet crack and generation. *ACM/IEEE Design Automation Conference (DAC)*, 2020.
- [39] T. Kohno M. Bellare and C. Namprempre. The secure shell (SSH) transport layer encryption modes. <https://datatracker.ietf.org/doc/html/rfc4344>.
- [40] Kenneth L. McMillan and Lenore D. Zuck. Formal specification and testing of QUIC. *ACM Special Interest Group on Data Communication*, 2019.
- [41] Robert McNaughton and Hisao Yamada. Regular expressions and state graphs for automata. *IRE Trans. Electron. Comput.*, 9, 1960.
- [42] MITRE. CVE-2021-40523.
- [43] Roberto Natella. StateAFL: Greybox fuzzing for stateful network servers. *ArXiv*, abs/2110.06253, 2021.
- [44] OpenBSD. Libressl. <https://www.libressl.org>.
- [45] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. AFLNET: A greybox fuzzer for network protocols. *2020 ICST*.
- [46] J. Postel and J. Reynolds. Rfc854, telnet protocol specification. <https://datatracker.ietf.org/doc/html/rfc854>.
- [47] Postel, J. and J. Reynolds. File transfer protocol, 1985. <https://www.rfc-editor.org/rfc/rfc959.html>.
- [48] Sergej Schumilo, Cornelius Aschermann, Andrea Jemmett, Ali Reza Abbasi, and Thorsten Holz. Nyx-net: network fuzzing with incremental snapshots. *Seventeenth European Conference on Computer Systems*, 2022.
- [49] secdev. Scapy Fuzzing API. <https://scapy.readthedocs.io/en/latest/usage.html#fuzzing>.
- [50] secdev. Scapy: Packet crafting for python2 and python3. <https://scapy.net>.
- [51] Kostya Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A fast address sanity checker. *2012 USENIX Annual Technical Conference*.
- [52] Juraj Somorovsky. Systematic fuzzing and testing of TLS libraries. *2016 ACM SIGSAC CCS*.
- [53] Synopsis. Defensics fuzz testing.
- [54] Peach Tech. Peach fuzzer configuration file (Peach Pit). Website. <https://peachtech.gitlab.io/peach-fuzzer-community/v3/PeachPit.html>.
- [55] Ken Thompson. Programming techniques: Regular expression search algorithm. *Communications of the ACM*, 1968.
- [56] Twistedmatrix. Twisted: building the engine of your network. <https://twistedmatrix.com>.
- [57] Yapeng Ye, Zhuo Zhang, Fei Wang, X. Zhang, and Dongyan Xu. NetPlier: Probabilistic network protocol reverse engineering from message traces. *2021 NDSS*.
- [58] Michal Zalewski. American fuzzy lop. 2015.
- [59] Cen Zhang, Xingwei Lin, Yuekang Li, Yinxing Xue, Jundong Xie, Hongxu Chen, Xinlei Ying, Jiashui Wang, and Yang Liu. APICraft: Fuzz driver generation for closed-source SDK libraries. *2021 USENIX Security Symposium*.

- [60] Mingrui Zhang, Jianzhong Liu, Fuchen Ma, Huafeng Zhang, and Yu Jiang. IntelliGen: Automatic driver synthesis for fuzz testing. *2021 ICSE-SEIP*.
- [61] Yong-Hao Zou, Jia-Ju Bai, Jielong Zhou, Jianfeng Tan, Chenggang Qin, and Shih-Min Hu. TCP-Fuzz: Detecting memory and semantic bugs in TCP stacks with fuzzing. *2021 USENIX Annual Technical Conference*.
- [62] Feilong Zuo, Zhengxiong Luo, Junze Yu, Ting Chen, Zichen Xu, Aiguo Cui, and Yu Jiang. Vulnerability detection of ICS protocols via cross-state fuzzing. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 2022.
- [63] Feilong Zuo, Zhengxiong Luo, Junze Yu, Zhe Liu, and Yu Jiang. PAVFuzz: State-sensitive fuzz testing of protocols in autonomous vehicles. *ACM/IEEE Design Automation Conference (DAC)*, 2021.

## A Implementation Details

The implementation is well-modularized. We defined uniform interfaces between the main fuzzing process and each module to facilitate scalability. In this way, we can easily extend BLEEM with new mutation operators, customized monitors (e.g., a semantically-aware monitor), and new protocol stacks (e.g., Bluetooth stack) by implementing corresponding required interface functions.

### A.1 Packet-Level Mutation Operators

*Scapy* has identified five general field types, including *NumberField*, *StringField*, *ListField*, *EnumerationField*, and *LengthField*. We devise respective mutation operators for these field types based on their features. More specifically,

- The *NumberField* mutation operator performs random addition or subtraction operations to the original value while considering the valid value range or returns a number randomly selected from the valid value range.
- The *LengthField* holds the length value of referenced field. The *LengthField* mutation operator inherits all the operations defined in the *NumberField* mutation operator. It also provides an additional operation of replacing the original value with extreme values (e.g., zero, negative values, the maximum, and the minimum) to manifest corner cases that cause memory errors, since their values usually affect the memory access in the program (as the bug case study given in Appendix C shows).
- The *StringField* mutation operator conducts a finite combination of string-splicing, substring-duplication, and substring-deletion on the original string.
- The *ListField* depicts the field holding a list with items of the same type. The *ListField* mutation operator performs a finite combination of element-duplication, element-

deletion, element-addition (based on the Oracle Map corpus), and list-order-shuffle on the original list.

- The *EnumField* represents the field whose possible values are taken from a given enumeration. For example, in HTTP, the `Method` field with possible values [“GET”, “POST”, “HEAD”, ...] is *EnumField*. The *EnumField* mutation operator selects a value from the valid enumeration set with high probability. It also provides value out of the valid set, with low probability, to manifest corner cases.

### A.2 Packet Instantiation Sub-Module

For target protocols on different layers, we implement corresponding proxies working on underlying layers to provide reliable underlying communication, including *TCP Proxy*, *UDP Proxy*, *IP Proxy*, and *Ether Proxy*. For example, when fuzzing SSL protocol, we can use a *TCP Proxy* that provides reliable TCP connections with the client and server so that we can focus on fuzzing the SSL packet instead of the full protocol stack. To this end, for a target protocol on layer  $\ell$ , the support proxy working on  $\ell - 1$  should: (i) provide network isolation on layer  $\ell - 1$  to intercept the exchanged payload of layer  $\ell$ ; (ii) maintain the two links with the client and server and synchronize their status; and (iii) support efficient concurrent interaction with the client and server on layer  $\ell$ .

We implement the first requirement by configuring the client with a proxy-provided service address, which differs from the server’s service address. For example, for the target SSL server listening on TCP port 4433, we start *TCP Proxy* and bind to TCP port 4432, and then configure the client with the service address of TCP port 4432. This step is noninvasive since the client is typically configurable regarding the service address to connect.

The second requirement is tailored for the *TCP Proxy* since TCP is connection-oriented. For target protocol running on TCP, the *TCP Proxy* is responsible for maintaining the two TCP links with the client and server and synchronizing their status, including the connection establishment and connection close. We implement it upon Twisted [56], which provides an event-driven programming paradigm for internet applications. Specifically, we design and implement two `Protocols` based on `twisted.internet.protocol.Protocol` to handle the data of these two links in an asynchronous manner.

Third, to concurrently handle bidirectional traffic, we implement asynchronous interaction logic to separately manage the traffic of two directions, i.e., the server-to-client traffic and client-to-server traffic. The interaction logic of each direction continuously sniffs the traffic, conducts mutation on the received packet (as directed by the proposed strategy), and sends out the mutated packet.



Table 4: Published CVE IDs of protocol implementations in firmware and the emulation setting in our experiment

CVE ID	Device Type	Vendor	Model	Firmware Version	Vulnerable Binary	Protocol	Emulation Platform
CVE-2018-5767	Router	TENDA	AC15	15.03.1.16	bin/httpd	HTTP	QEMU user-mode
CVE-2020-25067	Router	NETGEAR	R8300	1.0.2.130	usr/sbin/upnpd	UPnP	QEMU full-system
CVE-2019-14457	IP Camera	VIVOTEK	CC8160	0100d	usr/sbin/httpd	HTTP	QEMU user-mode
CVE-2019-1663	Router	CISCO	RV130	1.0.3.44	usr/sbin/httpd	HTTP	QEMU full-system

### A.3 Monitors

We design and implement *Process Monitor* and *Network Monitors* to detect crashes. These monitors can be used in combination according to different scenarios.

**Process Monitor.** When the target process is locally accessible, the *Process Monitor* detects crashes by checking whether the process was terminated by a system signal (e.g., SIGSEGV). We can also enhance the program with ASan to detect memory corruption (in this case, the process will be terminated by SIGABRT when a memory error occurs).

**Network Monitor.** We implement the following *network monitors*, which can be employed locally or remotely.

- *TCP Monitor.* For the TCP-based protocols, we detect crashes by trying to conduct a TCP connection to the listening port since TCP is connection-oriented.
- *UDP Echo Monitor.* For the UDP-based protocols, this monitor provides an interface that allows users to identify the heartbeat packet of the under-test protocol. Then it works by assembling it with a UDP header, sending it to the listening port, and waiting for a response.
- *Ping Monitor.* The *Ping Monitor* validates a target is still alive by performing ICMP Echo tests.

## B Selected Targets in Evaluation

Table 4 shows the information on the selected closed-source targets, including the firmware information, corresponding vulnerable protocol binaries, and the emulation settings in our experiment.

Table 5: Selected open-source protocol implementations

Protocol	Subject	LOC	SUT (client, server)
TLS/SSL	BoringSSL	92K	tools/bssl client, tools/bssl server
DTLS	OpenSSL	884K	apps/openssl s_client, apps/openssl s_server
QUIC	mvfst	104K	samples/echo client, samples/echo server
PPTP	accel-ppp	61K	pptpsetup (Linux utility), sbin/accelpppd
IEC104	IEC104	3K	iec104_monitor, iec104_monitor
SSH	OpenSSH	136K	ssh, sshd
CoAP	libcoap	28K	examples/coap-client, examples/coap-server
DNS	Dnsmasq	32K	nslookup (Dnsutils, a Linux utility), dnsmasq
RTPS	CycloneDDS	226K	HelloWorldPublisher, HelloWorldSubscriber
MQTT	Mosquitto	45K	mosquitto_sub & mosquitto_pub, mosquitto
ICCP	libiec_iccp_mod	74K	client_example2, server_example1
RUDP	rudp	570	rudp.exe, rudp.exe
TLS/SSL	LibreSSL	596K	apps/openssl s_client, apps/openssl s_server
TLS/SSL	GnuTLS	429K	src/gnutls-cli, src/gnutls-serv
BGP	OpenBGPD	12K	bgpd, bgpd

Table 5 shows detailed information on target open-source protocol implementations, including the protocol type, protocol implementation name, lines of implementation code,

and selected SUT. To facilitate a fair comparison, the selected SUT (cf. the fourth column) all come from off-the-shelf utilities provided in the project, except for the client utility used to test `accel-ppp` and `Dnsmasq`, where we resorted to related utilities in the Linux community.

## C Bug Case Study and Coordinated Disclosure

Table 6 shows the detailed information on previously unknown vulnerabilities exposed by BLEEM.

```

1 char working[DOMAIN_MAX_LEN + 1] = { 0 };
2 size_t i, wi = 0;
3 for (i = 0; i < len; i++) {
4     char c = candidate[i];
5     ... // checking validation of candidate[i]
6     if (wi > DOMAIN_MAX_LEN) // bug: the corner case is
7         // mistakenly handled
8         goto bad;
9     working[wi++] = c;
10    if (i == len - 1) {
11        candidate_domain = strdup(working); // buffer may
12        // lack '\0' termination when invoking strdup
13    }
14 }

```

Figure 10: Simplified code snippets related to CVE-2021-41581.

**Case Study of CVE-2021-41581.** This bug is revealed in the LibreSSL [44], the default TLS provider for OpenBSD and macOS. It is triggered at the certificate validation stage. Figure 10 presents the relevant code snippets of the flawed function, which walks through a given buffer candidate with length len, and checks its validation as a mailbox. In this process, the buffer working is firstly cleared for initialization (Line 1) and then is used to cache each checked character c in candidate (Lines 4, 8). Actually, the developers have considered the overflow but mistakenly handled the corner case (Line 6). If the len is equal to DML, the original '\0' termination of working would be overwritten when the variable wi equals DML (line 8), leading to the stack-based buffer over-read when invoking strdup (Line 10) due to lack of '\0' termination. Triggering this bug is non-trivial because the packet sequence should satisfy the following three conditions: (i) the program logic enters the certificate validation stage; (ii) the length len of the buffer candidate should be DML; and (iii) the characters in the buffer candidate should be valid to pass the check (Line 5). Otherwise, it would be rejected at the early process, and this case cannot be covered. If this bug is exploited, the devices that run this protocol may expose sensitive data. The three state-of-the-arts all failed to expose

Table 6: Previously unknown vulnerabilities exposed by BLEEM

Protocol	Subject	Information	Status	CVE ID	CVSS Score
TLS/SSL	LibreSSL	Stack buffer overflow in the x509_constraints_parse_mailbox in libcrypto/x509/x509_constraints.c	Assigned	CVE-2021-41581	5.5 MEDIUM
TLS/SSL	GnuTLS	MD_UPDATE does not prohibit zero-length input from illegal address, causing null pointer dereference	Assigned	CVE-2021-4209	6.5 MEDIUM
TLS/SSL	BoringSSL	The server tries to write to a socket that has been shut down even after the client has sent TLS Alert	Fixed	-	-
PPTP	accel-ppp	Stack-based out-of-bounds read in post_msg when processing a call_clear_request	Assigned	CVE-2021-42870	7.5 HIGH
PPTP	accel-ppp	Stack-based out-of-bounds read in the server if the client exits after authentication	Assigned	CVE-2021-42054	7.5 HIGH
PPTP	accel-ppp	Memory allocated to pool in ippool_init2 is not free when exits, which can cause a denial of service	Requested	-	-
IEC104	IEC104	Stack buffer overflow in the parameter Iec10x_Sta_Addr	Assigned	CVE-2020-20486	7.5 HIGH
IEC104	IEC104	Segmentation violation in the Iec104_Deal_FirmUpdate function	Assigned	CVE-2020-18731	7.5 HIGH
UDP	rdup	Memory leak in the rdup_delete function because memory allocated rdup is not freed	Assigned	CVE-2020-20665	7.5 HIGH
ICCP	libiec_iccp_mod	Heap buffer overflow in the ByteStream_readOctets in common/byte_stream.c	Assigned	CVE-2020-20490	7.5 HIGH
ICCP	libiec_iccp_mod	Heap buffer overflow in the parseIncomingMessage in mms/iso_cotp/cotp.c	Assigned	CVE-2020-20662	6.5 MEDIUM
ICCP	libiec_iccp_mod	Heap buffer overflow in the MmsConnection_create in mms_client_connection.c	Assigned	CVE-2020-20663	6.5 MEDIUM
BGP	OpenBGPD	Undefined behavior, a null pointer is passed as the first argument of memcpy in imsg.c	Fixed	-	-
BGP	OpenBGPD	Undefined behavior, an incorrect bitwise shift in imsg.c	Fixed	-	-
QUIC	mvfst	Heap buffer overflow in the EchoServerTransportFactory::make function in EchoServer.h	Requested	-	-
SUM			15	10 CVEs	-

this bug because they have a small probability of success in entering this deep protocol state. Instead, our approach resorts to the packets provided by the SUT parties and applies guided fuzzing, thus can easily satisfy (i) and (iii), allowing for more efforts in the exploration of (ii).

We responsibly disclosed the vulnerabilities we found. Before publicly disclosing our findings, we reported the vulnerabilities to the respective vendors following their security procedures and coordinated appropriate disclosure periods with them, which aligns with industry standards. No vendor required us to redact our results prior to paper submission.

## D Enumeration-Type State Variable and Field

For BoringSSL, SGFuzz recognizes several state variables, including `tls12_server_hs_state_t` and `ssl_shutdown_t` as follows. These variables encode fine-grained program processing actions, including protocol-state-related (e.g., `tls12_server_hs_state_t`) and implementation-logic-related (e.g., `ssl_shutdown_t`). For example, the server program uses the `tls12_server_hs_state_t` to represent the protocol handshake state. It sets this variable to `statal2_send_server_hello` when generating and sending a Server Hello packet. Meanwhile, it uses the variable `ssl_shutdown_t` to represent the program shutdown state for the read half of the connection. If the server is configured not to send a `close_notify` packet, it sets this variable to `ssl_shutdown_close_notify`, indicating doing nothing.

```
enum tls12_server_hs_state_t {
    statel2_start_accept = 0,
    statel2_read_client_hello,
    statel2_read_client_hello_after_ech,
    statel2_select_certificate,
    statel2_select_parameters,
    statel2_send_server_hello,
    ...
};

enum ssl_shutdown_t {
    ssl_shutdown_none = 0,
    ssl_shutdown_close_notify = 1,
    ssl_shutdown_error = 2,
};
```

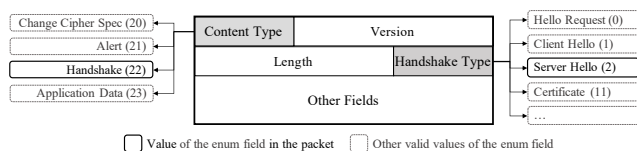


Figure 11: Enumeration fields in an SSL Server Hello packet

SGFuzz directly uses the different values of the state variables to identify different program states and captures state transitions by monitoring the assignment of recognized state variables based on instrumentation.

Correspondingly, Figure 11 shows an SSL Server Hello packet. The Content Type and Handshake Type fields are of type enumeration, and the Version and Length fields are of type number and length, respectively. These enumeration-type fields determine the packet type and indicate the protocol state. Specifically, the Content Type field, with four valid values, is set to Handshake (22), and the Handshake Type field is set to Server Hello (2), indicating this is a Server Hello packet exchanged during the handshake phase. Other fields, such as Version and Length, own low association with the protocol state. Hence, BLEEM abstracts this packet to `Handshake[Server_Hello]` by omitting other fields. Actually, from the program’s perspective, when a client or server receives a packet, it also first determines the packet type by parsing these fields and then takes corresponding actions (including setting state variables) according to the state machine.

Without instrumentation, BLEEM utilizes an on-the-fly approach. Since the enumeration fields typically identify the packet type and thus indicate the protocol state, BLEEM uses this key information to abstract concrete packets and constructs the *SUT States* by combining the *abstract packets* with bi-directional communication information, as shown in §4.