Article

MultiFuzz: A Coverage-based Multiparty-protocol Fuzzer for IoT Publish/Subscribe Protocols

Yingpei Zeng [1,2], Mingmin Lin [1], Shanqing Guo [3], Yanzhao Shen [1,4], Tingting Cui [1], Ting Wu [1,5,*], Qiuhua Zheng [1] and Qiuhua Wang [1]

1 School of Cyberspace, Hangzhou Dianzi University, Hangzhou 310000, China; yzeng@hdu.edu.cn (Y.Z.); lin_mingmin@163.com (M.L.); yanzhaoshen@hdu.edu.cn (Y.S.); cuitingting@hdu.edu.cn (T.C.); zheng_qiuhua@163.com (Q.Z.); wangqiuhua@hdu.edu.cn (Q.W.)
2 State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210000, China
3 School of Cyber Science and Technology, Shandong University, Jinan 250000, China; guoshanqing@sdu.edu.cn
4 Science and Technology on Communication Security Laboratory, Chengdu 610041, China
5 Hangzhou Innovation Institute, Beihang University, Hangzhou 310000, China
* Correspondence: wuting@hdu.edu.cn (T.W.)

Version June 24, 2023 submitted to Sensors

Abstract: The publish/subscribe model has gained prominence in the IoT (internet of thing) network, and both MQTT (Message Queue Telemetry Transport) and CoAP (Constrained Application Protocol) support it. However, existing coverage-based fuzzers may miss some paths when fuzzing such publish/subscribe protocols, because they implicitly assume that there are only two parties in a protocol, which is not true now since there are three parties, i.e., the publisher, the subscriber and the broker. In this paper, we propose MultiFuzz, a new coverage-based multiparty-protocol fuzzer. First, it embeds multiple-connection information in a single input. Second, it uses a message mutation algorithm to stimulate protocol state transitions, without the need of protocol specifications. Third, it uses a new *desockmulti* module to feed the network messages into the program under test. *desockmulti* is similar to *desock* (Preeny), a tool widely used by the community, but it is specially designed for fuzzing and is 10x faster. We implement MultiFuzz based on AFL, and use it to fuzz two popular projects Eclipse Mosquitto and libcoap. We reported discovered problems to the projects. In addition, we compare MultiFuzz with AFL and two state-of-the-art fuzzers, MOPT and AFLNET, and find it discovering more paths and crashes.

Keywords: coverage-based fuzzing; network protocol; publish/subscribe; multiparty-protocol fuzzer; MQTT; CoAP; IoT; Preeny; security; desock

1. Introduction

Fuzzing [1,2] is an important way to discover vulnerabilities in programs. The basic idea of fuzzing is to feed different inputs into a program under test (PUT) and keep monitoring its status for any misbehavior. Coverage-based fuzzing [3–5] is categorised as greybox fuzzing [2,6]. Different from traditional blackbox fuzzing [7,8], coverage-based fuzzing monitors the internal execution paths of inputs, and saves the inputs as further mutation seeds if they exercise any new and interesting paths. Though coverage-based fuzzing usually does not need sophisticated program analysis or the grammar of program input like whitebox fuzzing [9], it is shown to be able to gradually exercise different parts of the program and discover many vulnerabilities [3]. Now, coverage-based fuzzing is being both used by the security industry [3,10] and researched by the academia [5,11–16].

Different network protocols are proposed in the internet of thing (IoT) domain [17–20] to fit the unique requirements of IoT, and several important protocols support the publish/subscribe model [19].

For example, the MQTT (Message Queue Telemetry Transport) protocol [21] uses the publish/subscribe model as its core design, and the CoAP (Constrained Application Protocol) protocol [22] supports to "observe" resources (a similar publish/subscribe model) by using a protocol extension [23]. The publish/subscribe model provides loose coupling and scalability to the IoT network [19]: (i) publishers and subscribers do not need to know the existence of each other, and do not need to be online at the same time, (ii) one publisher could publish data to many subscribers and one subscriber could subscribe data from many publishers, i.e., supporting a many-to-many communication model. In this paper we focus on the fuzzing of two publish/subscribe protocols, MQTT and CoAP. They are widely used in the IoT network, and are also supported by IoT cloud providers like Amazon and Microsoft [24,25].

Currently, there are mainly two methods for network protocol fuzzing. The first method belongs to blackbox fuzzing [19]. It includes general fuzzing tools like Boofuzz (Sulley) [26] and Peach [8], and tools designed for specific protocols like TLS-Attacker [27]. These tools usually need users to write scripts or codes to describe the formats of network messages and the transitions of protocol states. They require the expertise on the protocols to get good fuzzing results, and the scripts and codes need to be updated accordingly when the protocols have new versions. The second method belongs to greybox fuzzing, which is to adapt coverage-based fuzzing tools to the fuzzing of network protocols. The method is more promising since it generally does not need to know protocol specifications or write codes. However, it needs a way to feed fuzzing inputs into the network program under test. One way is to use *desock* (a module of Preeny) [28] to hook the socket functions like `socket()` and `accept()` [1] (recommended by AFL [3] for no code modifications required), another way is to send inputs through ordinary sockets like AFLNET [15], and the last way is to modify the program source codes to make the program read packets directly from memory buffers other than real network interfaces, like fuzzing openssl in the Google OSS-Fuzz project [10]. The first two ways may introduce performance bottleneck (as we later shown in Section 5.4). The last way may be not trivial since the modification may need code refactoring.

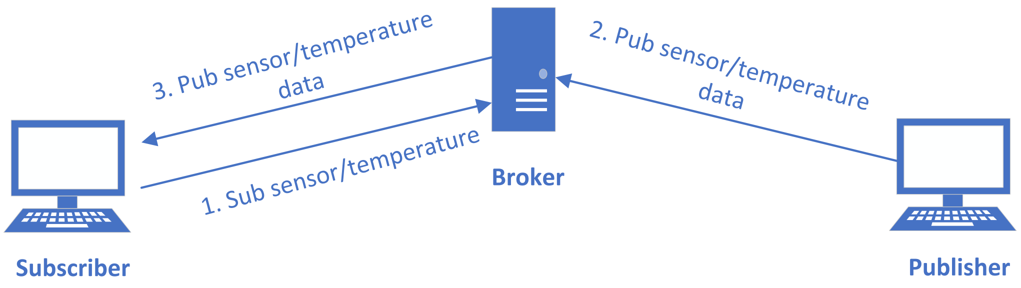

3. Pub sensor/temperature data
1. Sub sensor/temperature
Broker
2. Pub sensor/temperature data
Subscriber
Publisher

Figure 1. A typical publish/subscribe process of the MQTT protocol (some CONNECT and ACK messages are omitted for simplicity).

We can see that coverage-based fuzzers need less preparation before fuzzing, however, there is another special problem when using them to fuzz the publish/subscribe protocols in IoT. Existing coverage-based fuzzers implicitly assume that there are only two parties in the network protocols (the fuzzers pretend to be one party when fuzzing another party), but there are three parties in the publish/subscribe protocols. Considering a typical process of the MQTT protocol [21,29] in Figure 1, a subscriber subscribes to the sensor/temperature topic first. When a publisher publishes a value to the topic, the subscriber will receive the published value later. Such a process cannot be simulated by

[1] The hooking is usually done by LD_PRELOAD, and the sockets returned to the PUT are hijacked to send (or receive) data to (or from) *stdout* (or *stdin*). We further explain the design of *desock* later in Section 4.4.

existing coverage-based fuzzers, this is because only the messages of a single connection are included in a fuzzing input, and the fuzzer initiates only a single connection for each input [3,10,15]. Thus, even if the fuzzer successfully simulates as a subscriber and sends the first SUBSCRIBE message in the process, it cannot receive the third PUBLISH message, because there is no publisher sending the second PUBLISH message during a fuzzing execution (also the broker server under fuzzing is restarted for each input). The fuzzer may receive the PUBLISH message if it happens to subscribe to some built-in topics, but it cannot make the broker server run the whole dynamic publish/subscribe process shown in Figure 1. If there is a vulnerability in the corresponding execution path in the broker server, the fuzzer cannot discover it. So, in general existing coverage-based fuzzers are not sound for such multiparty protocols.

In this paper, we propose a coverage-based multiparty-protocol fuzzer called MultiFuzz. We compare MultiFuzz with existing fuzzers in Table 1. MultiFuzz does not need any protocol specifications, or any coding by the users. It can initiate multiple connections to a PUT, which enables it to fuzz the IoT publish/subscribe protocols like MQTT and CoAP [2]. MultiFuzz has a new module *desockmulti* to feed network messages into the PUT, and is 10x faster than *desock* (Preeny). In order to stimulate the state transitions of network protocols, MultiFuzz uses a message mutation algorithm to mutate inputs at a higher level first. MultiFuzz is coverage-based hence it belongs to the greybox category.

Table 1. Fuzzer comparisons. "Partial" means some knowledge or work is needed.

Fuzzer	Need Spec.	Need Coding	Support Multiparty	Message-aware	Taxonomy
Boofuzz (Sulley) [26]	Yes	Yes	**Yes**	**Yes**	blackbox
AFL [3]	**No**	**No**	No	No	greybox
MOPT [14]	**No**	**No**	No	No	greybox
AFLNET [15]	Partial	Partial	No	**Yes**	greybox
MultiFuzz (this paper)	**No**	**No**	**Yes**	**Yes**	greybox

Specifically, our paper makes the following contributions:

- We propose a multiparty-protocol fuzzer, MultiFuzz, to soundly support the fuzzing of publish/subscribe protocols. The fuzzer could initiate multiple connections to a PUT, and has a new seed format for storing all messages of the connections in a single seed input.
- We propose a message mutation algorithm to mutate message sequences in a seed input, to efficiently stimulate the state transitions of protocols. The mutation algorithm considers the multiple connections stored in a seed as well.
- We design and implement *desockmulti* for feeding network messages to a PUT. Previously the community usually uses the *desock* module of Preeny together with AFL to fuzz network services, but *desock* supports one connection only. We use a new design to support more than one connection, and further optimize *desockmulti* to be 10x faster than the widely used *desock* tool (Section 5.4) [3].
- We implement MultiFuzz based on AFL, and use MultiFuzz to fuzz two popular projects, Eclipse Mosquitto (an MQTT broker) [29], and libcoap (a CoAP library) [30]. We reported our found vulnerabilities to the projects and were acknowledged (Section 5.5). We also show that MultiFuzz outperforms AFL, and state-of-the-art fuzzers MOPT [14] and AFLNET [15] in finding program

[2] Please note that existing greybox fuzzers like AFL [3] and AFLNET [15] could still be used to fuzz the publish/subscribe protocols (i.e., initiating a single connection to the PUT to simulate one party in the protocols), although they may intrinsically miss some paths as we illustrated previously. Blackbox fuzzers like Boofuzz (Sulley) [26] do not restart the PUT after each fuzzing input; therefore, they naturally support multiparty protocols since they may simulate the multiple connections to the PUT by using multiple fuzzing inputs.

[3] We plan to opensource *desockmulti* after the publication of this paper.

paths and crashes (e.g., finding program paths 44.6% more than AFLNET, 126.6% more than AFL, and 125.4% more than MOPT, when fuzzing Eclipse Mosquitto).

The rest of the paper is organized as follows. In Section 2 we review related work. Then in Section 3 we give a brief introduction to MQTT and CoAP. In Section 4 we describe MultiFuzz in detail. We give experiment results in Section 5, and conclude the paper in Section 6.

2. Related Work

Fuzzing [1,2,6,31] now is a widely used technique to discover vulnerabilities in programs. The basic idea of fuzzing is to feed different and even abnormal inputs into a PUT, and keep monitoring its status to check if the program is crashed or misbehaving [1]. Fuzzers could be classified into three categories: blackbox, greybox and whitebox [2,6]. Blackbox fuzzers [7,8] could only monitor the input/output of a PUT, and some of them like Peach may know the structure of inputs [8]. Most traditional fuzzers are in this category [2]. Whitebox fuzzers [9,32] use much more internal information of a PUT, e.g., through symbolic execution. Greybox fuzzers take the middle way that they collect some internal information from a PUT. For example, coverage-based fuzzers collect the coverage information of inputs [2,6]. Greybox fuzzers usually run faster than whitebox fuzzers and utilize more information than blackbox fuzzers [6], and are good choices if we want higher coverage and discovering "hidden" bugs [6]. Modern fuzzers like AFL [3], libFuzzer [4] belong to this category. Greybox fuzzing (mainly coverage-based) has already been widely used in the security industry [3,10]. It is also a hot research topic; many fuzzers like AFLFast [5], CollAFL [11], Angora [12], QSYM [13], MOPT [14], and IJON [16] are proposed.

Fuzzing network protocols (i.e., network services/programs) is known to be difficult [15]. This is because the input is a sequence of messages, but not a single file as in traditional fuzzing, and it needs a way to feed the input into a PUT. Existing network protocol fuzzing approaches could be divided into two categories:

- **Blackbox network-protocol fuzzing** [19]. It includes general protocol fuzzing tools like SPIKE [33], PROTOS [34], SNOOZE [35], LZFuzz [36], Boofuzz (Sulley) [26], and Peach [8], as well as tools specially designed for some protocols, like TLS-Attacker [27] for the TLS (Transport Layer Security) protocol, and MTF [37] for the Modbus protocol. Most of these tools need users to tell the formats of network messages, and some of them also support users to provide the transition rules of protocol states [8]. For the general protocol fuzzing tools, users need to provide such information by scripts [26,33,34] or xml files [8,35]. For the tools specially designed for some protocols, such information is provided by tool developers. Most of these tools pretend as clients to feed inputs into the network programs, and some tools like LZFuzz [36] act as man-in-the-middle (MITM) proxies to modify messages between the client and the server. Blackbox fuzzing usually requires to write scripts, xml files, or codes following the protocol specifications, and needs to update them accordingly when the protocols have new versions. Also, blackbox fuzzing may be more suitable for discovering "shallow" bugs, comparing with greybox and whitebox fuzzing [6].
- **Greybox network-protocol fuzzing** [3,10,15]. The general coverage-based fuzzing tools like AFL [3] and libFuzzer [4] are used to fuzz network protocols as well. Usually users do not need to know protocol specifications or write any scripts/codes, instead, they prepare (e.g., by recording) some messages as seed inputs. However, since tools like AFL [3] were used to fuzz programs using files/*stdin*/memory buffer as the input source, they need some way to feed fuzzing inputs into the network programs. There are three known ways now. AFL recommends to use Preeny (*desock*) [28], a hook-based tool, to simply redirect *stdin* to sockets hijacked by the tool [3], AFLNET [15] sends inputs through ordinary sockets to the network programs, and users could also modify the program source codes to make the programs read packets directly from memory buffers other than real network interfaces, like the Google OSS-Fuzz project does to openssl [10]. The

first two ways may limit the execution speed of fuzzing (comparing with the *desockmulti* tool proposed in this paper). The third way may be difficult if the original developers of the programs do not expect such modification, and is also impossible for closed source programs. A very recent work AFLNET [15] proposed to combine coverage-based fuzzing with automated state model inferencing. While the fuzzing generates new message sequences to cover new states, the inferred state model guides how to do the fuzzing. AFLNET is shown to outperform Boofuzz [26] and AFL [3] in both code coverage and vulnerability discovery [15]. However, it requires users to write codes to extract partial information like the response codes from messages.

There are also some recent studies on the fuzzing in IoT domain. IoTFUZZER [38] is a new blackbox fuzzer which utilizes the mobile apps controlling IoT devices to do protocol fuzzing without protocol specifications. It indirectly mutates the protocol fields by mutating at data sources (e.g., string constants and inputs from system APIs). IoTFUZZER needs the mobile apps to fuzz network protocols, and is also limited to the fuzzing of functionalities related to the mobile apps. FIRM-AFL [39] uses AFL to fuzz IoT firmware, and uses augmented process emulation to fuzz programs at a higher speed. It mainly focuses on the fuzzing of ordinary programs in IoT firmware but not the network protocols. In [40], the authors propose a template-based fuzzing method to fuzz the MQTT protocol. The fuzzer is at the man-in-the-middle position between the client and the broker, and it selectively mutates the packets that match the specified types (e.g., PUBLISH messages). It provides templates to users to decide which fields to mutate, to alleviate the burden of writing codes like in Boofuzz [26]. However, users still need to know the specification (e.g., packet types) of the protocol. mqtt_fuzz [41] is an open-source tool designed for fuzzing the MQTT broker server. It could generate most of the MQTT packets for fuzzing. However, it has not been updated for five years, and does not support MQTT version 5.0 released in 2019 [21], not to mention the CoAP protocol that is also studied in this paper.

3. An Introduction to MQTT and CoAP

MQTT [21] and CoAP [22] are two important application-layer network protocols proposed in IoT [17–20]. MQTT is a publish/subscribe model protocol, and CoAP supports both the request/reply and publish/subscribe models [19,42,43]. The publish/subscribe model provides benefits that are crucial to the IoT network, like loose coupling and great scalability [19]. The two protocols (especially MQTT) are now widely used in the IoT network [44,45], and are also supported by IoT cloud providers like Amazon, Microsoft, and Google [24,25].

MQTT [21] is a publish/subscribe messaging transport protocol. It is light weight and has a simple design. It requires a small code footprint and limited network bandwidth. The protocol runs over TCP by default. Its clients could be publishers who publish application messages that other clients might be interested in, or subscribers who request application messages that they are interested in. Its server acts as an intermediary (broker) between clients which publish application messages and clients which have made subscriptions (please refer to Figure 1 for its architecture). The information delivered by MQTT is based on topics, and topics use topic level separator (i.e., "/") to introduce structure into the topic name. When the subscribers send subscription, they use topic filters, which may contain wildcards (i.e., multi-level wildcard "#", or single-level wildcard "+") so they could subscribe to multiple topics. However, when publishers send publish messages, they could only use topic name (no wildcards included).

In a publish message of MQTT, it could indicate one of the three QoS levels: 0 for at most once delivery (messages are delivered using the best efforts and message loss can occur), 1 for at least once delivery (messages are assured to arrive but duplicates can occur), and 2 for exactly once delivery (messages are assured to arrive exactly once). An MQTT control packet consists of up to three parts [21]: Fixed Header that presents in all MQTT control packets, Variable Header and Payload that present in some MQTT control packets. In MQTT v5.0 there are 15 types of MQTT control packets in total (the AUTH type is newly added in v5.0). We list them in Table 2. We can see that most of these packets are in pairs (i.e., a CONTROL command and its ACK), except that for QoS 2, there are 3 ACKs for ensuring the exactly once delivery.

Table 2. MQTT control packet types [21].

Name	Direction of flow	Description
CONNECT	Client to Server	Connection request
CONNACK	Server to Client	Connect acknowledgment
PUBLISH	Client to Server or Server to Client	Publish message
PUBACK	Client to Server or Server to Client	Publish acknowledgment (QoS 1)
PUBREC	Client to Server or Server to Client	Publish received (QoS 2 delivery part 1)
PUBREL	Client to Server or Server to Client	Publish release (QoS 2 delivery part 2)
PUBCOMP	Client to Server or Server to Client	Publish complete (QoS 2 delivery part 3)
SUBSCRIBE	Client to Server	Subscribe request
SUBACK	Server to Client	Subscribe acknowledgment
UNSUBSCRIBE	Client to Server	Unsubscribe request
UNSUBACK	Server to Client	Unsubscribe acknowledgment
PINGREQ	Client to Server	PING request
PINGRESP	Server to Client	PING response
DISCONNECT	Client to Server or Server to Client	Disconnect notification
AUTH	Client to Server or Server to Client	Authentication exchange

CoAP [22] is a specialized web transfer protocol for constrained environments. It follows the REST (Representational State Transfer) architecture of the Web, but is optimized for machine-to-machine (M2M) applications. It is bound to UDP by default, but could be bound to TCP as well [46]. It can be logically considered as a two-layer protocol, a CoAP messaging layer used to deal with UDP and the asynchronous interactions, and a request/response layer for the REST-style methods and response codes. In the messaging layer CoAP defines four types of messages: Confirmable, Non-confirmable, Acknowledgement, Reset. For example, marking a message as Confirmable (CON) could provide reliability to the up-layer. In the request/response layer, CoAP defines and uses GET, PUT, POST, and DELETE methods like HTTP, and uses a token field to match responses to requests independently from the underlying messaging layer (which uses a Message ID field for the similar purpose). CoAP also defines a URI scheme like HTTP, with the prefix "coap://" (or other variants like "coap+tcp://" for the TCP transport layer case).

The CoAP protocol also supports options, and it uses an Observe option to make CoAP clients can "observe" resources in a publish/subscribe model [23]. The process could be as follows. A client sends an extended GET request (the Observe option is set to 0) to the server to register its interest in a resource. Whenever the state of the resource changes (e.g., by a PUT request from others), the server notifies each observing client by a response. In the response the token is the same as the token in the original GET request, and the Observe option is set to a sequence number for reordering detection.

The security of the MQTT and CoAP protocols are very important, since they may be deployed in hostile environments. Both of them can be secured by either TLS or DTLS (Datagram Transport Layer Security), depending on whether TCP or UDP is used as the transport layer protocol [21,22]. Mutual authentication between the client and the server can also be added [21]. Researchers also formally verified the protocols [47], studied the possible attacks [48], and proposed intrusion detection for them [49].

4. MultiFuzz

In this section, we give an overview of MultiFuzz first, and describe its techniques in detail in later subsections.

4.1. Overview

The basic idea of MultiFuzz is to make the fuzzer support multiple connections when fuzzing a single seed. Thus, it could simulate the processes of the IoT publish/subscribe protocols (as well as other multiparty protocols), since during fuzzing, each connection could represent any party in the protocols. We make necessary changes from the seed input to the execution of a PUT.

We show the architecture of MultiFuzz in Figure 2. It is similar to other coverage-based fuzzers like AFL [3] (MultiFuzz is implemented based on AFL), and can be divided into four modules as well. The different parts are highlighted by yellow grids. First, the seed pool stores all initial seed inputs and newly found interesting inputs. MultiFuzz includes a new seed format to store the information of multiple connections in a seed. Second, the scheduling module selects seeds from the seed pool and sends to the mutation module. The scheduling module also later checks with the execution & monitoring module for any new inputs that have interesting paths, and saves such new inputs into the seed pool for future use. The scheduling module is unchanged in MultiFuzz. Third, the mutation module mutates the seed inputs and sends to the execution & monitoring module. The mutation module can be further divided into three stages in AFL [3]: the deterministic stage (using some predefined operations like "bitflip" and "arithmetic inc/dec"), the havoc stage (making *stacked* changes using previous operations), and the splicing stage (splicing the seed with another randomly selected seed). The deterministic stage takes a long time and can be skipped by using the "-d" option. MultiFuzz adds a new message mutation stage before the havoc stage to make message-aware mutation at a higher level. The execution & monitoring module executes the PUT and feeds inputs into it, and monitors the results like any crashes or other misbehavior. The module previously may use *desock* (Preeny) [28] to hook socket functions in order to feed inputs into the PUT, but MultiFuzz uses a faster and multi-connection-oriented tool *desockmulti* instead.

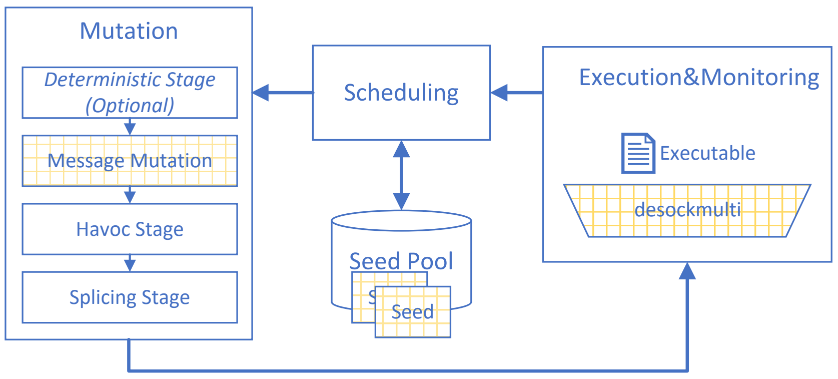


Figure 2. The architecture of MultiFuzz. It has the same architecture as other coverage-based fuzzers like AFL [3], with the changes highlighted by yellow grids.

4.2. Augmenting Seeds with Multi-Connection Information

We need a new seed format for storing the information of multiple connections. Existing fuzzers [3,10,15] assume that only a single connection is made to the PUT when fuzzing with a seed, thus, they could directly store all raw messages in a seed, without any extra information. During fuzzing, the fuzzers start a connection (either really [15], by hooking [3,28], or virtually [10]), and send all the messages to the PUT through the connection. In MultiFuzz, however, we need to start multiple connections for a seed input, so which messages belong to each connection must be determined. We also want the determination to be definite but not random, because we want the fuzzer to be stable (i.e., running the PUT with the same seed multiple times exercising the same path). We once considered embedding the meta information (like the number of connections and the lengths of messages) in the file name of a seed, and storing the real messages in the seed file. However, we do not use the method because: (i) file names have limited lengths in some file systems (e.g., 255 bytes in ext4), which limit the number of messages in a seed, (ii) the file names may already have meanings, for example, AFL [3] embeds id, parent, and mutation operation in the seed file name, (iii) sometimes we may want to fuzz the meta information like the length of a message, then we need to design extra fuzz operations. We finally decide to embed the meta information into the seed content as well, and experiment results confirm that it works well (Section 5).

We design a new seed format for embedding multi-connection information and show it in Figure 3. We want to keep the seed compact so we use a binary form format. Each seed now contains a HEADER and one or more MESSAGEs. The HEADER occupies two bytes, and each represents an unsigned value ([0-255]). The first byte is the number of sockets that connect to the PUT's accepting (i.e., listening) socket. The second byte is the number of sockets that the PUT connect to others [4]. In each MESSAGE, there is a one-byte unsigned value ([0-255]) representing the index of the socket the message belongs to (i.e., which socket the message sent through), a two-byte unsigned value ([0-65535], little endian) representing the length of the message len_i, and len_i bytes representing the message content. In a seed, different MESSAGEs could have the same socket index and all of them will be sent through the socket. We show a seed example in Figure 4. Its accept num is 2, so the fuzzer will initiate two sockets connecting to the PUT's accepting socket. The fuzzer will send the four-byte message "00 11 22 33" through the first socket (socket index = 0), and send the two-byte message "FF EE" through the second socket (socket index = 1).

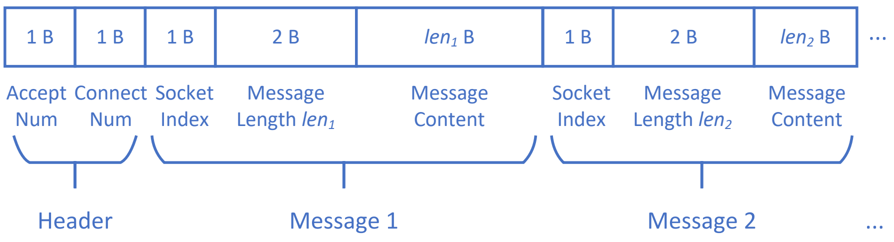
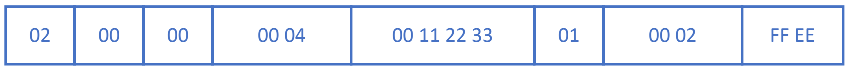


Figure 3. The new seed format.



02	00	00	00 04	00 11 22 33	01	00 02	FF EE

Figure 4. An example seed in the new format (hex encoding).

4.3. The New Message Mutation Stage

Existing mutation stages are not message-aware and are inefficient in stimulating the state transitions of protocols. We would like to design a mutation algorithm that does not need users to do extra work, but still provides good results. Our basic idea is to make full mutation on the message sequences, and rely on the existing evolution mechanisms in coverage-based fuzzing to approach new protocol states. For example, suppose currently there are three messages A, B, and C as separated seeds, there are `handleA`, `handleB`, and `handleC` functions handling these kinds of messages respectively (a common design pattern), and the PUT enters $state_A$, $state_B$, and $state_C$ after handling messages A, B, and C respectively and only the transitions $state_A$->$state_B$, and $state_B$->$state_C$ are allowed. Then, if there is a bug that only occurs in $state_C$ (i.e., after processing A, B, and C), it would be nearly unlikely

[4] The connect num is not used yet since the servers we fuzzed do not initiate connections to others.

Algorithm 1 Message mutation algorithm

```
Input: seed (its connection information already parsed)
Output: output buffer
1: use_stacking = 1 << (1 + UR(MF_STACK_POW2))    // UR is random function
2: while use_stacking > 0 do
3:     Choose 1 out of below 5 operations randomly:
4:       1. choose 1 out of the 4 operations randomly: to add/minus the accept num/connect num
5:       2. choose to remove or to add a connection (if to add, further choose to add a single-message
         connection or to duplicate a full connection from a random seed)
6:       3. add a random message to, or remove a message from the end of a random connection in the
         seed
7:       4. like operation #3 but the change position is random in the connection
8:       5. switch two randomly chosen messages of a connection in the seed
9:     use_stacking = use_stacking - 1
```

to generate seed A‖B‖C ("‖" represents concatenation) with existing mutation stages. However, with our message mutation algorithm proposed next in this subsection, it would quickly generate input A‖B based on existing seed A and seed B (and would be saved as a new seed since it has a new path), also quickly generate the wanted A‖B‖C input (either based on the new seed A‖B, or directly based on A, B, and C). We also need to consider the special mutation requirements for our multi-connection scenarios.

We design a new message mutation algorithm, which makes stacking changes like the existing havoc stage in AFL [3], and show it in Algorithm 1. We add the message mutation stage before each havoc stage of a seed. We add it here because the deterministic stage is optional and called only once for a seed, and the splicing stage also reuses the havoc stage to mutate inputs after each splicing. In the algorithm, we first decide the number of stacking changes with a constant MF_STACK_POW2 (the constant is set to 3 now, so the maximum number of stacking changes is 16). Then each time we randomly choose one operation out of the five possible ones. The first operation is to add or minus the numbers (i.e., accept num or connect num) in the header of the seed. The second operation is to add or remove a whole connection. Note that in the operation (and next operation) we may need messages for stuffing, and we get them from randomly chosen seeds. We do not limit to the initial seeds because new types of messages may be discovered during fuzzing. The third operation is a sequential change, which is to add or remove a message at the end of a connection. The fourth operation is like the third one, but is to add or remove a message at a random position in the connection. The fifth operation is to choose a random connection from the seed and switch the positions of two random messages in the connection.

4.4. desockmulti, *A Fast and Multi-Connection-Oriented De-Socketing Tool*

Since the seed input of MultiFuzz has a different format and is multi-connection-oriented, we need a new tool to feed the input into a PUT. The community usually uses the *desock* module of Preeny [28] to work with AFL [3] (recommended by AFL [3] for no code modifications needed). *desock* uses LD_PRELOAD to hook the `socket()`, `bind()`, `listen()`, and `accept()` functions. It uses two threads to synchronize a `socketpair` to *stdin* and *stdout*. However, its design makes it unable to accept multiple connections for a server (since all new sockets will be duplicated from the `socketpair` through `dup()` calls). Also, the original purpose of Preeny is to interact with binaries locally; therefore, it is not optimized for fuzzing. For example, *desock* uses `poll()` to keep reading from *stdin*. Then, first it needs an extra thread to keep calling `poll()`. Second, it is unnecessary since in fuzzing the whole input is provided at a time and no `poll()` calls are needed. We find the performance of *desock* is indeed limited in fuzzing (Section 5.4).

We design and implement a new tool, *desockmulti*. It has the following advantages:

- *desockmulti* supports the new seed format.
- *desockmulti* can initiate multiple connections (i.e., one or more) to a PUT, and it could be a replacement of *desock*, which can only initiate one connection.
- *desockmulti* is optimized for fuzzing, and is 10x faster than *desock*.

We describe the design of *desockmulti* in detail. It also uses LD_PRELOAD for hooking the major socket functions `socket()`, `bind()`, `listen()`, and `accept()`, and uses UNIX sockets to simulate the original INET/INET6 sockets. However, the relationships of the sockets in *desock* and *desockmulti* are different, as shown in Figure 5. In *desock*, only a single socket pair created by the `socketpair()` system call is used. One socket of the pair is returned to the PUT (though the socket may be duplicated to other file descriptors by `dup()` in the hooked `accept()` function), and the other socket's read stream and write stream are synchronized to *stdout* and from *stdin* respectively by two threads. In *desockmulti*, multiple socket pairs are created for multiple connections, and one socket of each pair is returned to the PUT. However, these socket pairs are not created by the `socketpair()` system call but by ordinary `connect()` and `accept()` calls. This is because multiple `accept()` calls are the only valid way to make the PUT process multiple new connections as usual. Also, other hooked socket functions work in more "real" ways as well. For example, in the hooked `bind()`, we really bind the socket at an address.

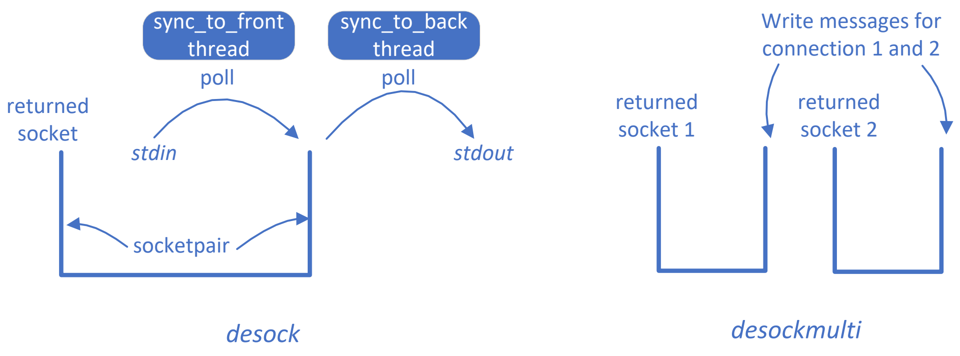


Figure 5. The design of *desock* and *desockmulti*.

We improve the performance of *desockmulti* mainly with the following optimizations: (i) we read all the content of a seed *at a time* without using `poll()`, since no interaction is needed during fuzzing, (ii) we removed the calls to `dup2()` which seems to be slow, (iii) we use the *abstract* socket address [5] in Linux system to remove the socket's relation with ordinary filesystem, (iv) and we remove the use of threads which is unnecessary in the new design, even we are using `connect()` and `accept()` to create socket pairs. We discovered some optimizations (e.g., (ii) and (iv)) through profiling (e.g., *strace* and detailed logging). Comparing with *desock*, *desockmulti* has more optimizations like (i), (ii), and (iv), which make it 10x faster in fuzzing (Section 5.4).

4.5. Other Implementation Details

We share some other implementation details here. We implement MultiFuzz based on AFL [3] and we add an option "-l" for enabling MultiFuzz. Since the seeds are mutated randomly and some numbers like the socket index may become out of their bounds (i.e., accept num + connect num), we use the numbers modulo their bounds, instead of treating the seeds as invalid. We add a method `multifuzz_generate()` for the message mutation of MultiFuzz, and call it before each of the stacking havoc mutation. We make *desockmulti* support the original seed format by an environment variable

[5] https://www.man7.org/linux/man-pages/man7/unix.7.html

USE_RAW_FORMAT. When the variable is set *desockmulti* behaves like a faster *desock*. MultiFuzz uses messages as seed inputs like other coverage-based fuzzers. We use Wireshark [6] to capture the messages, and develop a Python script to dump messages from the captured pcap files.

5. Evaluation

5.1. Experiment Settings

In order to evaluate MultiFuzz, we select two famous MQTT and CoAP protocol implementations, Eclipse Mosquitto [29] and libcoap [30]. Eclipse Mosquitto [29] is a message broker that implements the MQTT protocol versions 5.0 [21], 3.1.1, and 3.1. It also provides mosquitto_pub and mosquitto_sub command line MQTT clients. We use its latest release version 1.6.10 for fuzzing. We build Mosquitto by not enabling TLS (`make WITH_TLS=no`) since we do not want to fuzz the TLS codes. When collecting fuzzing seeds, we use the mosquitto_sub and mosquitto_pub clients to connect to the local broker server, and send a subscribe request and a publish request respectively. Then, we use Wireshark to capture the packets and use a custom Python script to dump messages as we mentioned before. We get 10 seeds for Mosquitto. libcoap [30] is a C-Implementation of CoAP that provides core functionality for the development of resource-efficient CoAP servers and clients. It supports extensions like resource observation [23], TCP [46], block-wise transfer, FETCH/PATCH, and No-Response. It provides coap-client and coap-server and we use coap-server as the PUT. We use its latest release version 4.2.1 for fuzzing. We also build with TLS disabled. Since our *desockmulti* does not support UDP yet (*desock* does not support as well), we use TCP as the transport layer protocol for coap-client and coap-server. For collecting seeds, we use two coap-client instances to observe and put a resource respectively (realizing the resource observation). Following the same method, we get 31 seeds for libcoap. We also build both projects without ASan (Address Sanitizer) [50] enabled for faster fuzzing speeds, and we later build them with ASan enabled for analyzing crashes and program paths [7].

Similar to AFL [3] (working with *desock*), MultiFuzz can be run with following command in the console: `LD_PRELOAD=/path/to/desockmulti/desockmulti.so ./afl-fuzz -l 0 -d -i testcase_dir -o findings_dir -- /path/to/program [...params...]`, where "-l" is for enabling MultiFuzz as mentioned before, and "0" is for ordinary-initial-seed case ("1" is for new-format-initial-seed case, i.e., format shown in Figure 3).

We choose AFL [3], and two state-of-the-art fuzzers MOpt [14] and AFLNet [15] to compare with MultiFuzz. AFL [3] is one of the most famous coverage-based fuzzers, and we use its latest version 2.52b. MOpt is a recently proposed fuzzer that uses Particle Swarm Optimization (PSO) algorithm to find the optimal probability distribution of mutation operators [14]. AFLNet is a more recently proposed fuzzer that is based on AFL as well, but uses automated state model inferencing to work with coverage-based fuzzing [15]. AFLNet needs users to write codes to extract response codes from messages, and we implement them as required. Basically, we use the higher 4 bits of the first byte of a MQTT message, and the second byte of a CoAP message, as the response code respectively [21,22]. We run AFLNet with its default settings (i.e., `-D 10000 -q 3 -s 3 -K -R`). We skip the deterministic stage during the mutation (which is the default configuration of AFLNet) for all the fuzzers (including the MOpt fuzzer, which automatically skips the deterministic stage after a while [14]). We use the *desock* module of Preeny [28] to work together with AFL and MOpt. We also use the same seeds set for all fuzzers.

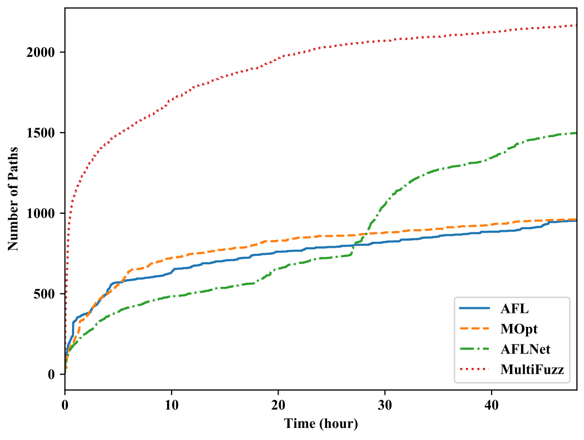
[6] https://www.wireshark.org/

[7] We do not consider the detection of data race or race condition in this paper, hence we do not use sanitizers like ThreadSanitizer, or specially design any mechanisms to boost concurrency.

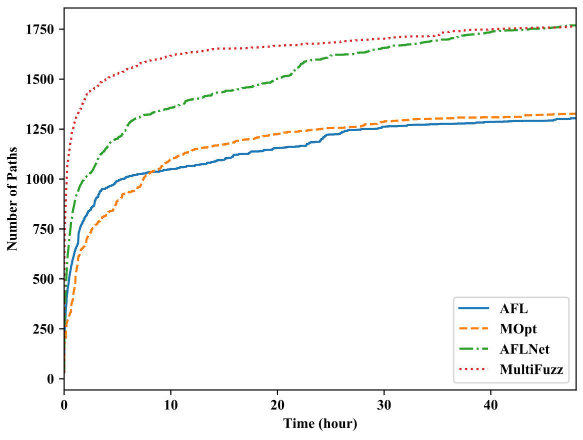
All the experiments run on a server configured with 2 Intel(R) Xeon(R) CPU E5-2640 v4 @ 2.40GHz processors, 64GB RAM memory, and with 64-bit Ubuntu 20.04 LTS. All the fuzzers run 2 days with a single fuzzer instance, since it is recommended to run more than 24 hours [31].

5.2. Path and Crash Discovery

The number of discovered program paths is one of the most important indicators of a fuzzer's ability, and we show the experiment results in Figure 6 (we show the results of one run and different runs have similar results). We can see that all fuzzers have a similar trend: finding more paths quickly at the beginning and reaching a plateau later. An exception is AFLNET in fuzzing Mosquitto, which has a sudden growth in the middle (because of the big seeds produced at that time, and we will further analyze it later in this subsection). In both projects, we can see that MultiFuzz discovers paths much faster than other fuzzers at the beginning. When later most paths have already been found, the discovery of MultiFuzz slows down and AFLNET may catch up. However, it seems difficult for AFL and MOPT to find the same paths even fuzzing for a long time. MultiFuzz eventually discovers 2166 paths when fuzzing Mosquitto, which is 44.6% more than AFLNET (1498), 126.6% more than AFL (956), and is 125.4% more than MOPT (961). In the case of libcoap, the results are similar. MultiFuzz discovers 1763 paths, which is similar to AFLNET (1769), but is 35.2% more than AFL (1304), and is 32.9% more than MOPT (1327). We also checked the found paths (i.e., seeds/queue entries) and confirmed that MultiFuzz does find new message types that are not in the original seed set, e.g., the PUBCOMP [21] message. The results confirm the hardness of fuzzing network protocols, since even MOPT could optimize the selection of mutation operators, it cannot discover much more paths than AFL. The AFLNET fuzzer indeed could discover more paths than AFL, which proves that it utilizes the message response codes we returned in codes very well. Finally, as a fuzzer that does not need extra information or codes from users, MultiFuzz discovers much more paths than similar fuzzers AFL and MOPT.



(a) Mosquitto



(b) libcoap

Figure 6. The paths discovered by different fuzzers.

We compare the number of crashes found by different fuzzers in Figure 7. Since all the fuzzers cannot find any crashes in Mosquitto, we only show the results of libcoap. MultiFuzz also finds crashes more quickly than other fuzzers. MultiFuzz eventually finds 198 unique crashes, which is 70.7% more than AFL (116), 55.9% more than MOPT (127), and 273.6% more than AFLNET (53). The results indicate that MultiFuzz can search for bugs more quickly than the fuzzers that are not optimized for the fuzzing of network protocols. AFLNET does not perform well here, which may be due to it focuses more on higher level message mutations, but not on the detail program logics.

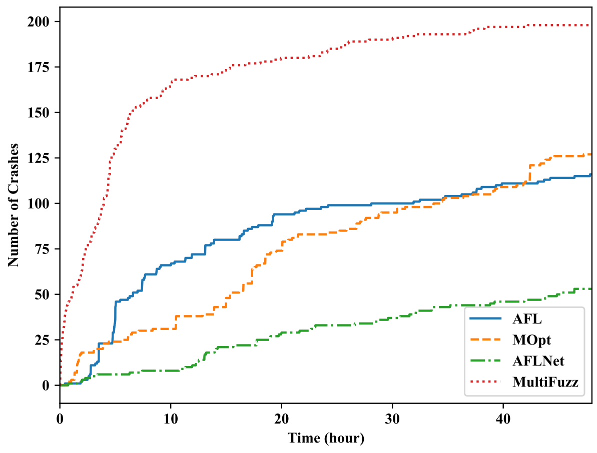
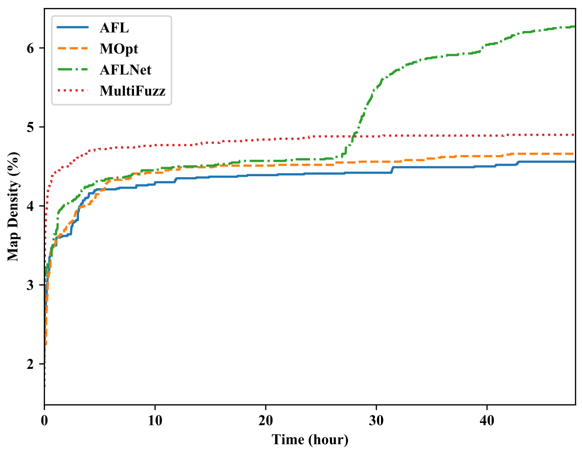
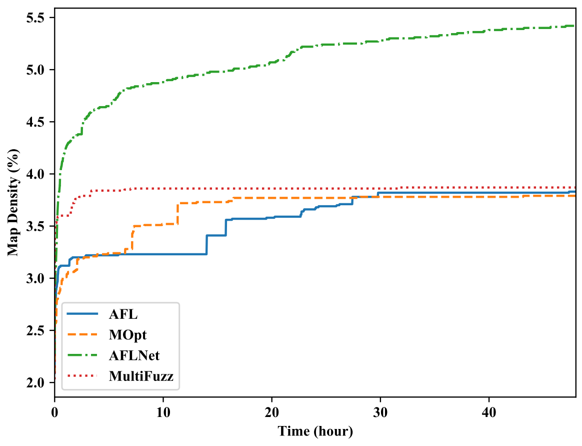


Figure 7. Crashes found in libcoap.



(a) Mosquitto



(b) libcoap

Figure 8. Bitmap density comparisons.

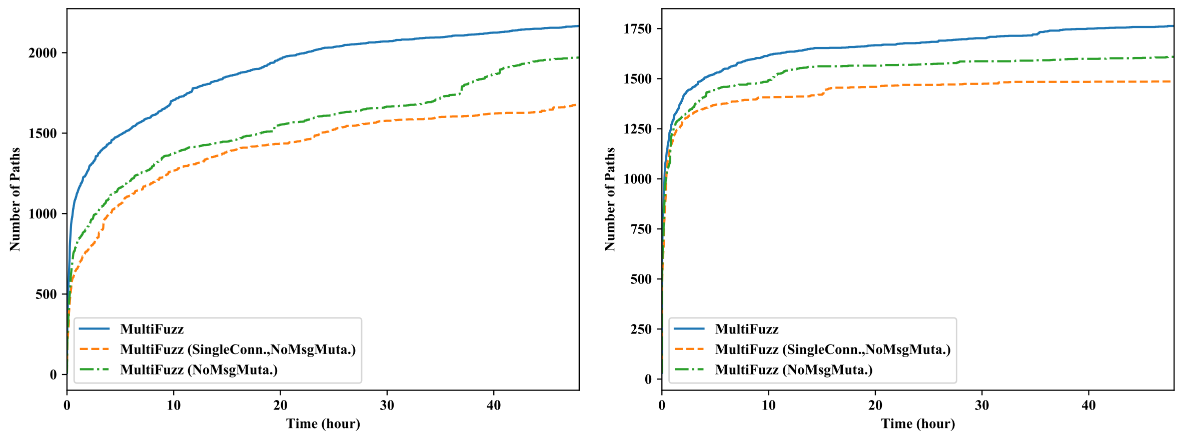
We use the bitmap density to check the code coverage of the fuzzers, and show the results in Figure 8. Coverage-based fuzzers usually use a fixed-size (e.g., 65536 bytes in AFL [3]) bitmap to store the coverage of all inputs. Each CFG (Control Flow Graph) edge of the PUT is mapped to a location in the bitmap (without considering collisions in high density cases [11]). Thus, the bitmap contains the accumulated edge coverage of all inputs. The bitmap density represents the ratio of locations in the bitmap that have values (i.e., edges travelled). We can see that MultiFuzz has a slightly higher bitmap density than AFL and MOPT, which is expected considering our message mutation algorithm.

However, AFLNET has a much higher bitmap density than other fuzzers, which shows that it can travel more edges. **UPDATE: This is because the bytes in the bitmap are overflowed, e.g., over 0xFF (255). The bug is fixed in AFL++.** First, this may be due to its ability to intentionally exercise rarely exercised protocol states. Especially for libcoap, we mainly generate publish/subscribe related message seeds, which may leave more room for the fuzzers to explore. Second, we further check the queue entries (i.e., seeds) of AFLNET, and find their sizes are much bigger than other fuzzers (for Mosquitto, the sizes grow a lot only around the middle of the fuzzing process, but for libcoap, the sizes grow quickly even at the beginning). Many seed sizes of AFLNET are several hundreds of KB (some are even over 1MB), and thousands of messages are inside a single seed. For comparison, the seed sizes of MultiFuzz and other fuzzers are usually less than 1 KB and are tens of KB at most, and only several or tens of messages are inside a single seed. Bigger size seeds may make the execution slower (Section 5.4). They also make the execution unstable (i.e., having different paths for the same seed) as well. We find the stability ratio (i.e., 1 - the ratio of variable bytes in the bitmap) of AFLNet drops to about 50% in Mosquitto and about 20% in libcoap (other fuzzers all are over 90%). However, on the other side, big size seeds may also introduce dynamic message processing behaviors (e.g., different read/write orders), which seem to be a plausible method to increase code coverage.

5.3. Effects of the Multi-Connection Design and the Message Mutation Algorithm

We know that the multi-connection design and the message mutation algorithm make the fuzzing of multiparty protocols sound and more efficient, however, it may be unclear that how much they improve the ability of MultiFuzz. We here make two variations of MultiFuzz: MultiFuzz (SingleConn., NoMsgMuta.) and MultiFuzz (NoMsgMuta.). In the first variation, we treat the seeds as in the original seed format that a seed only includes the messages of a single connection (i.e., by setting the USE_RAW_FORMAT environment variable when using *desockmulti*), and we disable the message mutation algorithm. In the second variation, we use the new seed format and *desockmulti* as normal, but disable the message mutation algorithm.

We show the paths discovered by MultiFuzz and its variations in Figure 9. We can see that MultiFuzz constantly outperforms its two variations, and MultiFuzz (NoMsgMuta.) constantly outperforms MultiFuzz (SingleConn., NoMsgMuta.). For Mosquitto, MultiFuzz eventually discovers 9.9% more paths than MultiFuzz (NoMsgMuta.), and 29.2% more paths than MultiFuzz (SingleConn., NoMsgMuta.). For libcoap, MultiFuzz discovers 9.6% more paths than MultiFuzz (NoMsgMuta.), and 18.6% more paths than MultiFuzz (SingleConn., NoMsgMuta.). We can clearly see that both the multi-connection design and the message mutation algorithm improve the path discovery ability of MultiFuzz, though they add some overhead at runtime.



(a) Mosquitto **(b)** libcoap

Figure 9. The paths discovered when disabling both the multi-connection design and the message mutation algorithm (MultiFuzz (SingleConn., NoMsgMuta.)), and disabling only the message mutation algorithm (MultiFuzz (NoMsgMuta.)).

We also show the number of unique crashes found in libcoap by these three types of MultiFuzz in Figure 10. MultiFuzz only slightly outperforms its two variations, and it eventually discovers 7 more crashes than MultiFuzz (SingleConn., NoMsgMuta.), and 12 more crashes than MultiFuzz (NoMsgMuta.). It seems that the multi-connection design and the message mutation algorithm do not improve the crash-finding ability too much.

5.4. The Comparison of Execution Speeds

We pay much attention to the performance of MultiFuzz (including the message mutation algorithm and the *desockmulti* module), and we show the execution speeds (executions/second) of different fuzzers, as well as the speed comparisons between MultiFuzz and others in Figure 11.

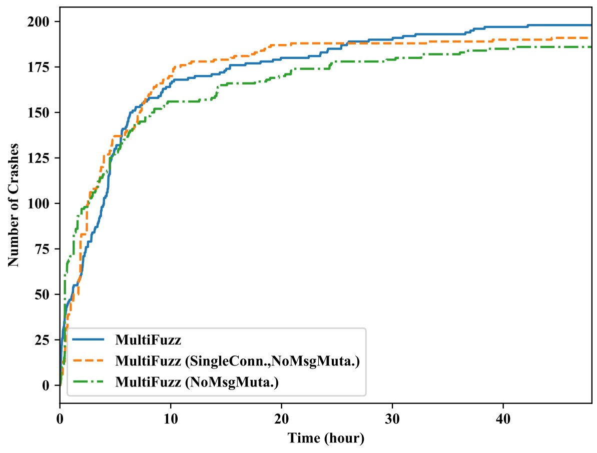


Figure 10. Crashes found in libcoap by MultiFuzz and its variations.

We use tmpfs [8] to store fuzzing outputs for faster execution, and we build the binaries for fuzzing without ASan enabled as we mentioned before. The average execution speed of MultiFuzz is 1333.9 execs/s for Mosquitto and 1412.4 execs/s for libcoap. In contrast, the average execution speed of AFL is 128.8 execs/s and 94.6 execs/s respectively, which means MultiFuzz is 10.4x and 14.9x faster than AFL. MOPT is a bit faster than AFL, but MultiFuzz still is 10.3x and 11.6x faster than it. Since AFL and MOPT use the *desock* module of Preeny [28], most of the speed improvement of MultiFuzz is due the newly designed *desockmulti* tool. The execution speed of MultiFuzz (SingleConn., NoMsgMuta.) could be used as a quick reference as using AFL with *desockmulti*, since only a little initialization work is extra added in the variation. The average execution speed of MultiFuzz (SingleConn., NoMsgMuta.) is 1389.9 execs/s for Mosquitto and 983.8 execs/s for libcoap, which is 10.8x and 10.4x faster than AFL with *desock* as well. The execution speed of AFLNET is slower than other fuzzers. It is only 9.0 execs/s for Mosquitto and 22.6 execs/s for libcoap. We think it is mainly due to that AFLNET uses real INET network socket to connect to a PUT. The INET network socket is known to be much slower than the UNIX socket. Also, as we mentioned we find AFLNET generates much bigger size seeds (hundreds of KB) than other fuzzers (< 1KB), which also would slow down the execution speed.

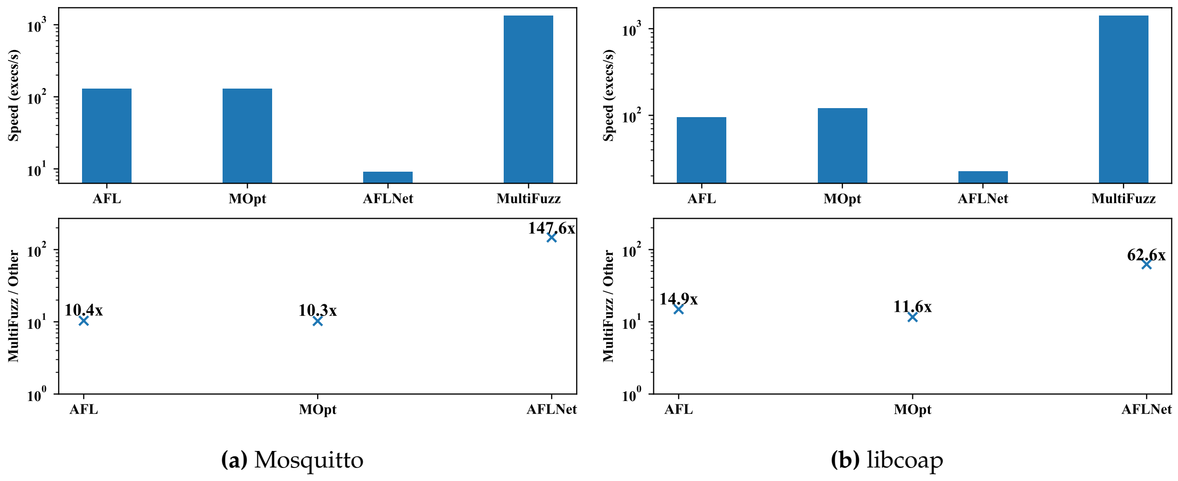


Figure 11. The execution speeds of different fuzzers, and the comparisons of MultiFuzz to other fuzzers.

[8] https://www.kernel.org/doc/html/latest/filesystems/tmpfs.html

5.5. Vulnerability Analysis

We manually investigate some of the found paths and crashes, and share the findings below.

Memory leaks in Eclipse Mosquitto. We find two memory leaks in Mosquitto, which could be repeatedly triggered by malformed requests and may cause DoS (Denial of Service) in the broker server. Both leaks are caused by the missing of `free()` calls during error processing. The first leak is in the `handle__subscribe()` method, at line 122 of `src/handle_subscribe.c`. When the broker processes a malformed MQTT v5.0 subscribe request, it calls `return MOSQ_ERR_PROTOCOL` directly, without calling `mosquitto__free(sub);mosquitto__free(payload);` as in other error processing cases. The second leak is in the `handle__publish()` method, at line 112 of `src/handle_publish.c`. When it processes a malformed MQTT v5.0 publish request, it calls `return rc` directly, without calling `mosquitto__free(topic)`. We reported the leaks to the Eclipse Mosquitto project and the developers fixed them instantly [9]. The fixes should be available in the next version 1.6.11.

`assert()` **failures and a memory leak in libcoap**. We find that most of the crashes are due to the failures of `assert()` calls, e.g., the failure of `assert(pdu->max_size > 0)` in the `coap_write_block_opt()` function, at line 77 of `src/block.c`. Such failures usually do not have impacts on production builds technically, since they should be built with the `NDEBUG` preprocessor macro defined, and the `assert()` method does nothing then. However, they could be reminders to the developers that unexpected things happen. For example, the memory leak we describe next has an assertion failure too. The memory leak is in the `handle_request()` method, at line 2208 of `src/net.c`. When the coap-server processes a malformed request in the `handle_request()` function, the call to the `coap_add_token()` method fails. However, only `coap_log(LOG_WARNING, "cannot generate response\r\n")` is called in the case, without a `coap_delete_pdu(response)` call to free the PDU timely. The memory leak can also be repeatedly triggered by malformed requests so it may cause DoS as well. We reported the leak to the libcoap project and was acknowledged as well [10].

6. Conclusions

This paper presented a coverage-based fuzzer MultiFuzz, which initiates multiple connections to a program under test, to soundly support the fuzzing of multiparty protocols like the publish/subscribe protocols in IoT. MultiFuzz contains a new seed format, a message mutation algorithm, and a new de-socketing module *desockmulti*. We used MultiFuzz to fuzz the Eclipse Mosquitto project and libcoap project, and reported our found vulnerabilities to the developers (all were acknowledged and fixed). We also showed that MultiFuzz found more paths and crashes, comparing with AFL, and two state-of-the-art fuzzers, MOPT and AFLNET. We think MultiFuzz is not limited to the fuzzing of IoT publish/subscribe protocols, and could be used to soundly fuzz other multiparty protocols as well. In addition, we believe the *desockmulti* module of MultiFuzz could benefit the community after open-sourcing, since it is similar to the widely used tool *desock* (Preeny) but is 10x faster.

Author Contributions: Conceptualization, Y.Z. and T.W.; methodology, Y.Z. and M.L.; software, Y.Z.; validation, Y.Z., M.L. and S.G.; formal analysis, Y.S.; investigation, Y.Z.; resources, Y.Z., T.W. Q.Z., and Q.W.; data curation, Y.Z.; writing–original draft preparation, Y.Z. and M.L.; writing–review and editing, Y.Z., S.G., Y.S. and T.C.; visualization, Y.Z.; supervision, Y.Z. and T.W.; project administration, Y.Z. and T.W.; funding acquisition, Y.Z., T.W., and Q.Z.. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported in part by the National Natural Science Foundation of China under Grant No. 61902098, No. 61902100, No. 61902148, and No. 91546203, in part by the Key Research Project of Zhejiang Province under Grant No. 2020C01078, No. 2019C01012, and No. 2017C01062, in part by Major Scientific and Technological Innovation Projects of Shandong Province, China No. 2017CXGC0704, No. 2018CXGC0708, No. 2019JZZY010132 and Qilu Young Scholar Program of Shandong University, in part by the Foundation of Science

[9] https://github.com/eclipse/mosquitto/commit/94d04136f8beedbd34ca16158b5be3111a9cb7b4
[10] https://github.com/obgm/libcoap/issues/535

and Technology on Communication Security Laboratory under Grant No. 6142103190105, and in part by the Natural Science Foundation of Zhejiang Province under Grant No. Q20F020063.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Miller, B.P.; Fredriksen, L.; So, B. An Empirical Study of the Reliability of UNIX Utilities. *Communications of the ACM* **1990**, *33*, 32–44. doi:10.1145/96267.96279.
2. Manes, V.J.M.; Han, H.S.; Han, C.; sang kil Cha.; Egele, M.; Schwartz, E.J.; Woo, M. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering* **2019**, pp. 1–21, [1812.00140]. doi:10.1109/TSE.2019.2946563.
3. Zalewski, M. AFL - American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/, accessed on 29 July 2020.
4. libFuzzer. http://llvm.org/docs/LibFuzzer.html, accessed on 29 July 2020.
5. Böhme, M.; Pham, V.T.; Roychoudhury, A. Coverage-based Greybox Fuzzing as Markov Chain. Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS), 2016, pp. 1032–1043. doi:10.1145/2976749.2978428.
6. Liang, H.; Pei, X.; Jia, X.; Shen, W.; Zhang, J. Fuzzing: State of the Art. *IEEE Transactions on Reliability* **2018**, *67*, 1199–1218.
7. zzuf. http://caca.zoy.org/wiki/zzuf, accessed on 29 July 2020.
8. Peach Tech. Peach Fuzzer. https://www.peach.tech, accessed on 29 July 2020.
9. Godefroid, P.; Kiezun, A.; Levin, M.Y. Grammar-based whitebox fuzzing. Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2008, pp. 206–215. doi:10.1145/1375581.1375607.
10. Google Security Team. A New Chapter for OSS-Fuzz. https://security.googleblog.com/2018/11/a-new-chapter-for-oss-fuzz.html, accessed on 29 July 2020.
11. Gan, S.; Zhang, C.; Qin, X.; Tu, X.; Li, K.; Pei, Z.; Chen, Z. CollAFL: Path Sensitive Fuzzing. IEEE Symposium on Security and Privacy (S&P), 2018, pp. 679–696. doi:10.1109/SP.2018.00040.
12. Chen, P.; Chen, H. Angora: Efficient Fuzzing by Principled Search. IEEE Symposium on Security and Privacy (S&P), 2018, pp. 711–725. doi:10.1109/SP.2018.00046.
13. Yun, I.; Lee, S.; Xu, M.; Jang, Y.; Kim, T. QSYM : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. 27th USENIX Security Symposium (Security), 2018, pp. 745–761.
14. Lyu, C.; Ji, S.; Zhang, C.; Li, Y.; Lee, W.H.; Song, Y.; Beyah, R. MOPT: Optimize Mutation Scheduling for Fuzzers. 28th USENIX Security Symposium (Security), 2019, p. 1.
15. Pham, V.t.; Boehme, M.; Roychoudhury, A. AFLNet: A Greybox Fuzzer for Network Protocols. Proceedings of the 13rd IEEE International Conference on Software Testing, Verification and Validation : Testing Tools Track, 2020, pp. 460–465. doi:10.1109/ICST46399.2020.00062.
16. Aschermann, C.; Schumilo, S.; Abbasi, A.; Holz, T. IJON: Exploring Deep State Spaces via Fuzzing. IEEE Symposium on Security and Privacy (S&P), 2020, pp. 1–16.
17. Al-Fuqaha, A.; Guizani, M.; Mohammadi, M.; Aledhari, M.; Ayyash, M. Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications. *IEEE Communications Surveys and Tutorials* **2015**, *17*, 2347–2376. doi:10.1109/COMST.2015.2444095.
18. Yassein, M.B.; Shatnawi, M.Q.; Al-Zoubi, D. Application layer protocols for the Internet of Things: A survey. Proceedings - 2016 International Conference on Engineering and MIS, ICEMIS 2016. IEEE, 2016. doi:10.1109/ICEMIS.2016.7745303.
19. Dizdarević, J.; Carpio, F.; Jukan, A.; Masip-Bruin, X. A survey of communication protocols for internet of things and related challenges of fog and cloud computing integration. *ACM Computing Surveys* **2019**, *51*, 1–30, [arXiv:1804.01747v2]. doi:10.1145/3292674.
20. Pereira, C.; Aguiar, A. Towards efficient mobile M2M communications: Survey and open challenges. *Sensors* **2014**, *14*, 19582–19608. doi:10.3390/s141019582.
21. Banks, A.; Briggs, E.; Borgendale, K.; Gupta, R., Eds. *MQTT Version 5.0*; OASIS Standard, 2019.
22. Shelby, Z.; Hartke, K.; C.Bormann. The Constrained Application Protocol (CoAP). *RFC 7252* **2014**.
23. Hartke, K. Observing Resources in the Constrained Application Protocol (CoAP). *RFC 7641* **2015**.

24. Team, T.H. Comparison of MQTT Support by IoT Cloud Platforms. https://www.hivemq.com/blog/hivemq-cloud-vs-aws-iot/, accessed on 29 July 2020.
25. Ptone. IoT Core CoAP proxy demonstration. https://cloud.google.com/community/tutorials/cloud-iot-coap-proxy, accessed on 29 July 2020.
26. Boofuzz: Network Protocol Fuzzing for Humans. https://github.com/jtpereyda/boofuzz, accessed on 29 July 2020.
27. Somorovsky, J. Systematic Fuzzing and Testing of TLS Libraries. Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS), 2016, pp. 1492–1504. doi:10.1145/2976749.2978411.
28. Preeny. https://github.com/zardus/preeny, accessed on 29 July 2020.
29. Eclipse Mosquitto. https://mosquitto.org/, accessed on 29 July 2020.
30. libcoap. https://libcoap.net/, accessed on 29 July 2020.
31. Klees, G.; Ruef, A.; Cooper, B.; Wei, S.; Hicks, M. Evaluating Fuzz Testing. Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS), 2018, pp. 2123–2138. doi:10.1145/3243734.3243804.
32. Godefroid, P.; Levin, M.Y.; Molnar, D. Automated whitebox fuzz testing. Network and Distributed System Security Symposium (NDSS), 2008, pp. 416–426.
33. Aitel, D. The Advantages of Block-Based Protocol Analysis for Security Testing. Technical report, Immunity Inc., 2002.
34. Roning, J.; Laakso, M.; Takanen, A. PROTOS presentations. https://www.ee.oulu.fi/research/ouspg/.
35. Banks, G.; Cova, M.; Felmetsger, V.; Almeroth, K.C.; Kemmerer, R.A.; Vigna, G. SNOOZE: Toward a Stateful NetwOrk prOtocol fuzZEr. Information Security, 9th International Conference (ISC), 2006, pp. 343–358. doi:10.1007/11836810_25.
36. Bratus, S.; Hansen, A.; Shubina, A. LZfuzz: a fast compression-based fuzzer for poorly documented protocols. Technical report, Department of computer science, Dartmouth college, 2008.
37. Voyiatzis, A.G.; Katsigiannis, K.; Koubias, S. A Modbus/TCP Fuzzer for testing internetworked industrial systems. IEEE International Conference on Emerging Technologies and Factory Automation, ETFA, 2015, Vol. 2015-Octob. doi:10.1109/ETFA.2015.7301400.
38. Chen, J.; Diao, W.; Zhao, Q.; Zuo, C.; Lin, Z.; Wang, X.; Lau, W.C.; Sun, M.; Yang, R.; Zhang, K. IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing. 25th Annual Network and Distributed System Security Symposium (NDSS), 2018.
39. Zheng, Y.; Davanian, A.; Yin, H.; Song, C.; Zhu, H.; Sun, L. FIRM-AFL: High-throughput greybox fuzzing of IoT firmware via augmented process emulation. Proceedings of the 28th USENIX Security Symposium, 2019, pp. 1099–1114.
40. Hernández Ramos, S.; Villalba, M.T.; Lacuesta, R. MQTT Security: A Novel Fuzzing Approach. *Wireless Communications and Mobile Computing* **2018**, *2018*. doi:10.1155/2018/8261746.
41. mqtt_fuzz. https://github.com/F-Secure/mqtt_fuzz, accessed on 29 July 2020.
42. Rodríguez-Domínguez, C.; Benghazi, K.; Noguera, M.; Garrido, J.L.; Rodríguez, M.L.; Ruiz-López, T. A Communication model to integrate the Request-Response and the publish-subscribe paradigms into ubiquitous systems. *Sensors* **2012**, *12*, 7648–7668. doi:10.3390/s120607648.
43. Davis, E.G.; Calveras, A.; Demirkol, I. Improving packet delivery performance of publish/subscribe protocols in wireless sensor networks. *Sensors* **2013**, *13*, 648–680. doi:10.3390/s130100648.
44. Akasiadis, C.; Pitsilis, V.; Spyropoulos, C.D. A multi-protocol IoT platform based on open-source frameworks. *Sensors* **2019**, *19*, 1–25. doi:10.3390/s19194217.
45. Larmo, A.; Ratilainen, A.; Saarinen, J. Impact of coAP and MQTT on NB-IoT system performance. *Sensors* **2019**, *19*. doi:10.3390/s19010007.
46. Bormann, C.; Lemay, S.; Tschofenig, H.; Hartke, K.; Silverajan, B.; Raymor, B. CoAP (Constrained Application Protocol) over TCP, TLS, and WebSockets. *RFC 8323* **2018**.
47. Houimli, M.; Kahloul, L.; Benaoun, S. Formal specification, verification and evaluation of the MQTT protocol in the Internet of Things. Proceedings of the 2017 International Conference on Mathematics and Information Technology, ICMIT 2017, 2017, pp. 214–221. doi:10.1109/MATHIT.2017.8259720.
48. Vaccari, I.; Aiello, M.; Cambiaso, E. SlowITe, a novel denial of service attack affecting MQTT. *Sensors* **2020**, *20*, 1–16. doi:10.3390/s20102932.

49. Granjal, J.; Silva, J.M.; Lourenço, N. Intrusion detection and prevention in CoAP wireless sensor networks using anomaly detection. *Sensors* **2018**, *18*. doi:10.3390/s18082445.
50. Serebryany, K.; Bruening, D.; Potapenko, A.; Vyukov, D. AddressSanitizer: A fast address sanity checker. USENIX Annual Technical Conference (ATC), 2012, pp. 309–318.