U-Fuzz: Stateful Fuzzing of IoT Protocols on COTS Devices

1st Zewen Shang
ASSET Research Group
SUTD
Singapore
zewe_shang@mymail.sutd.edu.sg

2nd Matheus E. Garbelini
ASSET Research Group
SUTD
Singapore
matheus_garbelini@sutd.edu.sg

3rd Sudipta Chattopadhyay
ASSET Research Group
SUTD
Singapore
sudipta_chattopadhyay@sutd.edu.sg

***Abstract*—Internet-of-Things (IoT) devices have become widely popular and are being increasingly utilized in both home and industrial environments. Such devices use a variety of different protocols for communication. Considering the complex and stateful nature of these protocols, their implementations may contain security vulnerabilities and are subject to remote exploitation. To address this, we present U-FUZZ, a framework to systematically discover and replicate security vulnerabilities on arbitrary wired and wireless IoT protocol implementations. Given only a network capture file which contains the packet traces of normal (i.e., benign) communication, U-FUZZ automatically constructs a protocol state machine. Subsequently, this state machine is leveraged via a stateful fuzzing engine to arbitrarily manipulate and replay communicated packets. U-FUZZ carefully disintegrates the design of state machine construction from the fuzzing actions and optimizations, allowing U-FUZZ to work with an arbitrary number of protocols without any change in the stateful fuzzing engine. U-FUZZ does not require any access to the source code of the protocol and it also does not involve any instrumentation. This makes U-FUZZ to applicable out-of-the-box for fuzzing arbitrary IoT devices employing a variety of protocols. We implemented U-FUZZ and applied it against ten subject implementations including implementations on five commercial-off-the-shelf (COTS) devices employing three popular IoT protocols: 5G NR, Zigbee, and CoAP. As of today, U-FUZZ discovered a total of 11 new vulnerabilities (out of 16) and CVEs have already been assigned to all of them.**

I. Introduction

The massive progress of internet-of-things (IoT) has resulted in a significant number of applications across various domains including smart homes, healthcare and robotics. However, security vulnerabilities in IoT communication protocols may lead to severe consequences. Moreover, since many IoT protocols are wireless, such security vulnerabilities can be exploited *over-the-air*, leading to denial-of-service (DoS), security bypass, and information leakage [3], [16]. Therefore, it is of critical importance to systematically detect security vulnerabilities in IoT protocol implementations. This is also significantly challenging, as such implementation in commercial-off-the-shelf (COTS) devices is closed-source, rendering any instrumentation of its code impractical.

In this paper, we propose U-FUZZ, a framework to systematically discover and replicate security vulnerabilities in IoT protocol implementations. U-FUZZ works on

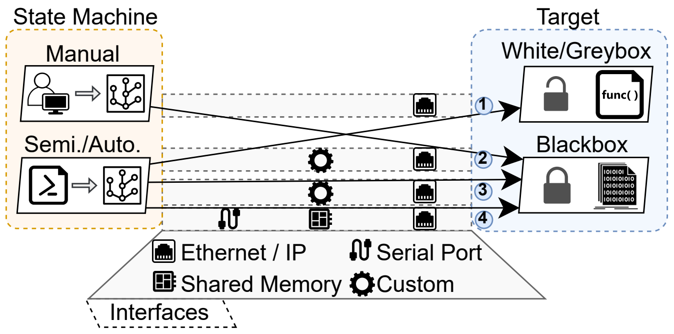


Fig. 1. Positioning of U-FUZZ in relation to previous state-of-the-art IoT fuzzers. *Semi.* means semi-automatic state machine generation, while *auto.* means automatic.

both wired and wireless protocols and it can be employed out-of-the-box for protocol implementations on COTS IoT devices. The key idea embodied in U-FUZZ is a generic state mapping technique, that takes as input a normal (i.e., benign) communication trace and automatically creates an abstract state machine for arbitrary (IoT) protocol implementations. Along with the state machine, U-FUZZ also creates a condition, which, is used during fuzzing to map an arbitrary packet to a protocol state. Such an on-the-fly state mapping then guides the fuzzing process to optimize a user-configurable cost function e.g., maximizing the state machine transitions. U-FUZZ fuzzing engine is placed as a *man-in-the-middle*, which only intercepts regular packets and systematically manipulates the communication traffic as guided by the cost function. In this fashion, U-FUZZ aims to provide a general methodology for stateful fuzzing of arbitrary IoT protocols even without any access to the source code. To minimize developer effort, U-FUZZ employs a simple heuristic to create an exploit script with minimal malformed communication. Such an exploit replicates each discovered vulnerability via fuzzing.

Figure 1 illustrates the positioning of our work with respect to the existing works on (stateful) fuzzing. Existing works on protocol fuzzing have only focused on specific protocols e.g., DTLS [10] [11], Bluetooth Low Energy [29], MQTT [1], EDHOC [36] to learn the state machine automatically (indicated in Figure 1 by arrow 3 and the custom interface icon). Furthermore, other works on protocol fuzzing demand: ① the availability of source code and code instrumentation [4], [30], ② manually creating state machines for fuzzing [15], [16] and ③ manually creating state mapping rules for stateful fuzzing [13]. Finally, even though a recent work ③ supports multiple IP-based protocol fuzzing [9], [21], it is not applicable for fuzzing many widely used wireless protocols in IoT such as Zigbee that uses the ZNP protocol with serial interface, and 5G NR protocol that is shared-memory controllable. Thus, extending such work [9], [21] for COTS IoT protocol implementations involves significant engineering to mention the least. Moreover, the unavailability of the implementation [21] makes any extension to the work infeasible in practice. In contrast, our U-Fuzz approach (④ in Figure 1) provides an open platform, it applies to arbitrary wired and wireless protocol implementations using generic fuzzing interfaces and systematically explores the state-space of IoT protocols without having access to the source code and without any instrumentation. The open platform also makes it easy to extend U-Fuzz for other IoT protocols. After providing a brief overview of U-Fuzz (Section II), we make the following contributions:

1) We present U-Fuzz state mapping technique, which takes as input only a network capture file and constructs the state machine and conditions for state mapping for arbitrary IoT protocols (Section III-A).
2) We present U-Fuzz stateful, man-in-the-middle fuzzing engine that seamlessly integrates with the proposed state mapping technique, thus, allowing guided stateful fuzzing of IoT protocol implementations. Such a disintegration of the state mapping and stateful fuzzing engine easily allows to reuse the same fuzzing engine across several protocols (Section III-B).
3) We present our simple heuristic to reproduce IoT vulnerabilities with minimal manipulation to communication traffic (Section III-D).
4) We present our implementation and integration of U-Fuzz across popular IoT protocols with varying complexities i.e., CoAP, Zigbee, and 5G NR. (Section IV).
5) We evaluate U-Fuzz against implementations of both IP-based and wireless protocols i.e., 5G NR, CoAP and Zigbee using ten subject implementations including implementations on five commercial-off-the-shelf (COTS) devices. Our evaluation reveals 11 new vulnerabilities (out of a total 16), out of which nine CVEs are already assigned. (Section V).
6) We launch concrete attacks on off-the-shelf IoT devices exploiting the vulnerabilities found by U-Fuzz and discuss their impact (Section V-E).

After discussing the related work (Section VI), we conclude in Section VIII.

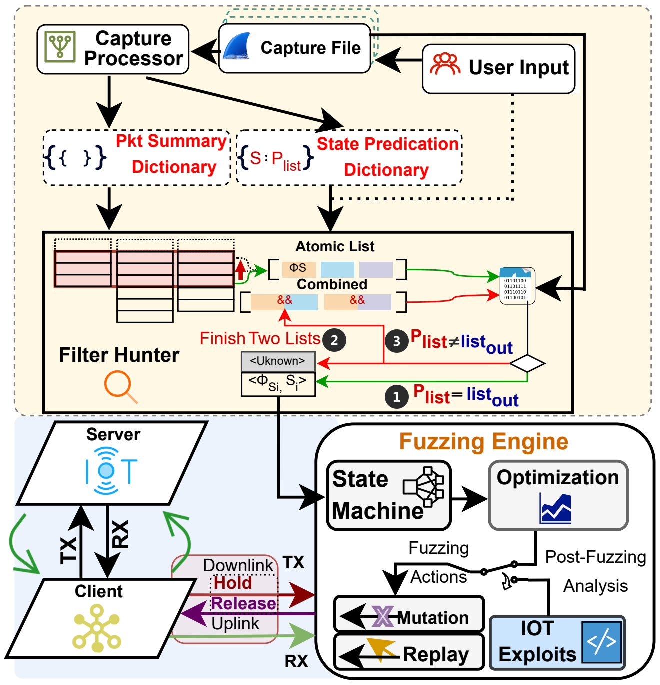


Fig. 2. Illustration of the U-Fuzz approach depicting key components: a multi-protocol state mapper (highlighted in light yellow), communication between IoT APP and SUT (in blue), interception of packets (via green arrows) and fuzzing engine (highlighted in white) that mutates or injects packets towards the SUT.

II. Background and Framework Overview

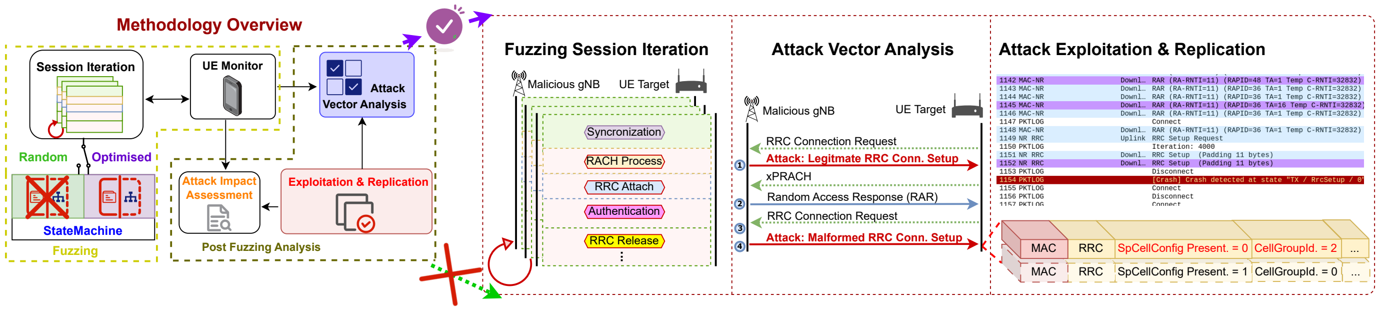
A. Background

In this section, we introduce the background knowledge for the IoT protocols targeted by our U-Fuzz approach.

1) **5G NR:** 5G cellular network architecture consists of three key components: The gNodeB (gNB), User Equipment (UE), and Core Network. The gNB is also known as the base station in the traditional cellular network. It serves as the access point for wireless communication between the UE and the 5G core network. The UE refers to end-user devices, such as smartphones. Lastly, the Core Network acts as the backbone of the 5G architecture by providing control and management functions, including authentication, security, mobility management, session establishment, and data routing between network entities.

Multiple protocols including Radio Resource Control (RRC), Non-Access Stratum (NAS), Medium Access Control (MAC), Packet Data Convergence Protocol (PDCP) and Radio Link Control (RLC) from both network layer (OSI layer 3) and data link layer (OSI layer 2) are involved to ensure that the connection is securely established. We employ our U-Fuzz approach for downlink (i.e., the communication from the gNB to the UE) fuzzing.

2) **Constrained Application Protocol (CoAP):** The *Constrained Application Protocol* (CoAP) is a specialized web protocol designed for computing environments with constraints on processing power, storage, memory, and battery life. Hence, such protocols are suitable for low power IoT



```
1142 MAC-NR        Downl…  RAR (RA-RNTI=11) (RAPID=48 TA=1 Temp C-RNTI=32832)
1143 MAC-NR        Downl…  RAR (RA-RNTI=11) (RAPID=36 TA=1 Temp C-RNTI=32832)
1144 MAC-NR        Downl…  RAR (RA-RNTI=11) (RAPID=36 TA=1 Temp C-RNTI=32832)
1145 MAC-NR        Downl…  RAR (RA-RNTI=11) (RAPID=36 TA=16 Temp C-RNTI=32832)
1146 MAC-NR        Downl…  RAR (RA-RNTI=11) (RAPID=36 TA=1 Temp C-RNTI=32832)
1147 PKTLOG                Connect
1148 MAC-NR        Downl…  RAR (RA-RNTI=11) (RAPID=36 TA=1 Temp C-RNTI=32832)
1149 NR RRC        Uplink  RRC Setup Request
1150 PKTLOG                Iteration: 4000
1151 NR RRC        Downl…  RRC Setup  (Padding 11 bytes)
1152 NR RRC        Downl…  RRC Setup  (Padding 11 bytes)
1153 PKTLOG                Disconnect
1154 PKTLOG                [Crash] Crash detected at state "TX / RrcSetup / 0'
1155 PKTLOG                Connect
1156 PKTLOG                Disconnect
1157 PKTLOG                Connect
```

Fig. 3. An illustration of the U-FUZZ workflow in discovering and replicating the zero-day vulnerability **V1** (see Table V)

devices. To meet the resource constraints, CoAP aims to minimize the amount of data sent over the network. This prevents issues like packet fragmentation, which can significantly reduce the successful delivery rate of the data. CoAP is designed in a familiar client-server model used by the *Hypertext Transfer Protocol* (HTTP). In this model, clients send requests to the server using specific *method codes* to perform actions on server resources, identified by *Uniform Resources Identifiers* (URIs). CoAP employs a set of method codes that align with the principles of *Representational State Transfer* (REST). These methods include `GET` (used to retrieve information from a resource), `POST` (typically used for creating or updating resources, with the specific action determined by the server), `PUT` (used to update a resource), and `DELETE` (used to remove a resource).

3) **Zigbee:** Zigbee is a wireless communication protocol designed for short-distance, low-power networks. It's distinguished by forming mesh networks, where devices cooperate to relay messages, extending network reach and reliability. Zigbee shines in smart homes, industrial automation and healthcare, offering secure and scalable solutions for a variety of IoT needs. Within the Zigbee ecosystem, devices fall into three categories: Coordinators, which manage the network; Routers, responsible for data relay and specific tasks; and End Nodes, which communicate with their parent nodes and can use batteries.

B. U-FUZZ *workflow*

Figure 2 outlines the workflow of U-FUZZ. Broadly, U-FUZZ consists of three steps: *State Machine Generation*, *Fuzzing*, and *Post-Fuzzing Analysis*.

One of the key contributions of our U-FUZZ approach is to automatically construct the state machine of arbitrary (IoT) protocols (see the box highlighted in yellow in Figure 2). Such is accomplished by subjecting our novel state mapper to traces of network packet captures. The state mapper embodied in U-FUZZ integrates all protocols supported by Wireshark, as it relies on the Wireshark dissectors for packet decoding. Nonetheless, custom protocols can also be integrated within the U-FUZZ framework as long as a custom dissector is provided. For example, in our evaluation, we show the extensibility of U-FUZZ framework beyond Wireshark by implementing a custom dissector for Zigbee Network Processor (ZNP) interface. We detail the state mapping process in Section III-A.

Once the state machine is generated by our state mapping process, it is leveraged systematically via a stateful fuzzing engine (see “Fuzzing Engine" in Figure 2). Such a fuzzing engine has two key functions: firstly, to generate adversarial communication scenarios via packet mutation and replay, as illustrated in Figure 2, and secondly, to perform an evolutionary search on the possible packet mutations by employing a variety of cost functions related to the state machine, as generated by our state mapping process. For instance, we guide the evolutionary fuzzing process to maximize the transition coverage of the state machine. While we use the transition coverage to compute the test adequacy of our fuzzing sessions, other cost functions can easily be provided within the U-FUZZ framework. We outline the key characteristics of our stateful fuzzing engine in Section III-B.

It is worthwhile to mention that U-FUZZ incorporates strategies to automatically detect crashes for effective fuzzing. For 5G implementations on smartphones, the error code and assertion violations can be easily collected from device logs, whereas for CoAP, the server crash is also indicated via the error code. However, for Zigbee devices, such internal device logs are not available. Thus, we implement custom crash detection techniques to reliably detect Zigbee device crashes. All our crash detection techniques are outlined in Section III-C. As a byproduct of crash detection, U-FUZZ also produces a communication trace of the resulting crash. This communication trace is a network capture that highlights the crash which subject to the post-fuzzing analysis (see “Post-Fuzzing" analysis in Figure 3) for automatically replicating SUT crashes obtained during a fuzzing session. We discuss the details of our post fuzzing analysis methodologies in Section III-D.

Running Example: Figure 3 illustrates U-FUZZ approach in discovering and replicating vulnerability **V1** – `Invalid CellGroupConfig` (see Table V) on *OnePlus Nord CE 2* smartphone employing *MediaTek Dimensity 900 5G* modem. Initially, 5G communication traces were provided to the U-FUZZ for the 5G state machine generation. We note that such a communication trace was valid packet sequences captured during legitimate communication. After the 5G state machine was generated, it was leveraged to guide the fuzzing process to reach deeper states that result in higher state machine coverage. In this way, U-Fuzz is also likely to find vulnerabilities that demand covering many transitions before reaching a vulnerable state. U-Fuzz achieved ***69.6%*** of model coverage by performing the stateful evolutionary fuzzing, while the fuzzing without any state machine guidance (see Figure 3) obtained ***60.9%*** coverage. More importantly, the higher coverage obtained by the U-Fuzz approach resulted in finding multiple unique 5G vulnerabilities including ***V1, V2, V4, V5*** (see Table V), none of which were discovered without the state machine guidance. An example of such a vulnerability is illustrated in Figure 3. Without the state machine guidance (as shown via "Random" in Figure 3), the zero-Day vulnerability **V1** could not be found.

Figure 3 also illustrates the post-fuzzing process. Specifically, a vulnerable communication trace containing several mutated packets (highlighted in purple) are shown in Figure 3: *(i)* `Malformed RRC Conn. Setup`, *(ii)* `Malformed MAC-NR RA-RNTI` and *(iii)* `Malformed MAC-NR RA-RNTI`. After analyzing the attack vector, U-Fuzz determines that the same vulnerability can be reproduced even if the `MAC-NR RA-RNTI` packets are not malformed. Indeed, during our evaluation, the attack vector of ***V1*** was computed to be just one malformed message, as highlighted by `Malformed RRC Conn. Setup` plus one legitimate connection process before sending the Malformed RRC Conn. Setup. Consequently, U-Fuzz facilitates development of an exploit script that intercepts the `RRC Conn. Setup` message and modifies the packet fields `CellGroupId` and `SpCellConfig Present` values to 2 and 0, respectively after a successful connection with the base station (see the malformed `RRC Conn. Setup` Figure 3). These modified packets are then released to the target UE to reliably reproduce the vulnerability ***V1***.

III. Design of U-Fuzz

A. Multi-Protocol State Mapper Design

Creating State Labels: As shown in Figure 2, the Multi-Protocol State Mapper only requires one Wireshark capture file from the user. Each packet in the capture file is parsed and summarized into a nested dictionary $\mathbb{P}_D$. As shown in Figure 4, the first key of $\mathbb{P}_D$ is the packet number (e.g., *Packet 1, Packet 2* etc.) and each entry contains the nested dictionary of all the packet layers (e.g., *udp* and *mac-nr*). The last node of the nested dictionary contains the value of the protocol fields specified by its key (shown as *udp.dstport=9999* in Figure 4).

The Capture Processor also computes all the potential states which ought to be included in the state machine. To this end, it first considers the *Wireshark* description of each packet and groups the packets by the same Wireshark

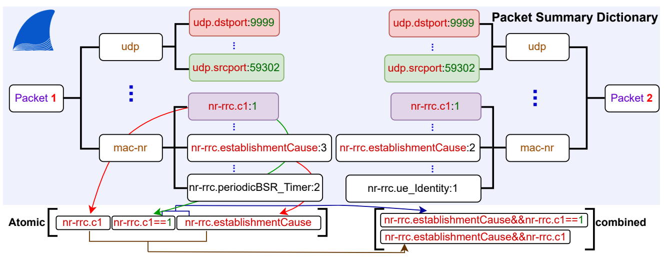

Packet Summary Dictionary
Packet 1
udp
mac-nr
udp.dstport:9999
udp.srcport:59302
nr-rrc.c1:1
nr-rrc.establishmentCause:3
nr-rrc.periodicBSR_Timer:2
udp.dstport:9999
udp.srcport:59302
nr-rrc.c1:1
nr-rrc.establishmentCause:2
nr-rrc.ue_Identity:1
udp
mac-nr
Packet 2
Atomic
nr-rrc.c1
nr-rrc.c1==1
nr-rrc.establishmentCause
nr-rrc.establishmentCause&&nr-rrc.c1==1
nr-rrc.establishmentCause&&nr-rrc.c1
combined

Fig. 4. Example of the generated packet summary

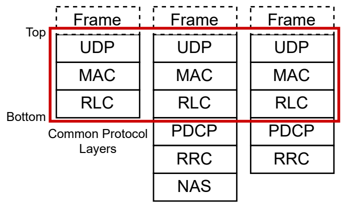


Fig. 5. Illustration of the common protocol layers of 5G NR Packets selected by *Filter Hunter*, prior to enumeration of the *Potential filter list*

description. Concretely, this is done via a *State Predication Dictionary* $\mathbb{S}_P$. For each $\langle S, P_{list}\rangle \in \mathbb{S}_P$, S denotes the state label created from the set of packets P_{list} grouped by the same Wireshark descriptor. $\mathbb{S}_P$ can also be optionally refined by the user for fine-grained customization of the state machine based on domain-specific knowledge.

Formalizing State Filters: We note that merely creating the state labels is not sufficient for fuzzing arbitrary IoT protocols in a stateful fashion. Since U-Fuzz employs stateful and man-in-the-middle fuzzing (see Section III-B), we need to map an arbitrary packet to a state, for guiding the fuzzing session. This, in turn, requires to create a condition that could effectively map any communicated packets to a state label, as captured in the State Predication Dictionary $\mathbb{S}_P$. To this end, the Packet Summary Dictionary and the State Predication Dictionary are leveraged by the *Filter Hunter* component to search for such a condition (i.e., *filter*). Concretely, consider an arbitrary item $\langle S, P_{list}\rangle \in \mathbb{S}_P$. Our objective is to find condition Φ_S as follows:

$$\Phi_S\left(\bigcup_{i=1}^{n} P_i\right) = P_{list} \tag{1}$$

where $\bigcup_{i=1}^{n} P_i$ is the set of all packets in the input capture file and $\Phi_S(P)$ returns the set of packets from P that satisfies the condition Φ_S.

Generating State Filters: As shown in Equation 1, we aim to generate the set of filters Φ_S for each $\langle S, P_{list}\rangle$ appearing in the State Predication Dictionary. For a large enough network capture file, this is a challenging task due to the massive search space involving potential filters. We employ simple, yet effective heuristics to address such challenges.

The search for a common filter, to classify the packets in an arbitrary state $\langle S, P_{list}\rangle$, starts by searching the common layers in all packets $P \in P_{list}$. Once such a set of common layers, say $layers(P_{list})$ are found, we attempt to find the filtering condition starting from the bottom-most layer in $layers(P_{list})$. This is because such bottom layer is often the most relevant to informing the state of the packet during communication. For example, the set of layers for three 5G packets in a communication is illustrated in Figure 5. These three packets have the same Wireshark descriptor and hence, are grouped into the same state. While the *frame* layer is an implementation-specific tweak in Wireshark and is irrelevant, we note that $layers(P_{list})$ in Figure 5 corresponds to $\{UDP, MAC, RLC\}$. Since RLC is the bottom-most layer in $layers(P_{list})$, the search process is attempted on the fields of RLC layer first.

Let us assume $L \in layers(P_{list})$ and we aim to find the characteristics of fields in the layer L that accurately classify the set of packets P_{list}. To this end, U-FUZZ first creates a set of atomic filters $Filter_{list}$ as follows:

$$cf_{list} = \bigcap_{P \in P_{list}} \{f \mid f \in fields(P, L)\} \tag{2}$$

$$cv_{list} = \bigcap_{P \in P_{list}} \{f : v \mid f \in cf_{list} \wedge v = value(P, f)\} \tag{3}$$

$$Filter_{list} = cf_{list} \bigcup cv_{list} \tag{4}$$

where $fields(P, L)$ is the set of fields in layer L of packet P and $value(P, f)$ is the value of field f in packet P. Intuitively, cf_{list} creates the list of common fields across all packets in P_{list}. Similarly, cv_{list} creates the list of common field, value pairs across packets in P_{list}. Finally, the set of atomic filters is created as a union of cf_{list} and cv_{list} i.e., the set of common field names and common field, value pairs, respectively. As illustrated in Figure 4 for `Packet 1` and `Packet 2`, cf_{list} is $\{nr-rrc.c1, nr-rrc.establishmentCause\}$. However, cv_{list} is $\{nr-rrc.c1 : 1\}$, as only the field $nr-rrc.c1$ contains the same value (one) across both `Packet 1` and `Packet 2`.

Once $Filter_{list}$ is generated, the final step is to scan through this list to find the accurate filter Φ_S such that $\Phi_S\left(\bigcup_{i=1}^{n} P_i\right) = P_{list}$ (Equation 1) holds. To this end, U-FUZZ first takes $\Phi_S = filter \in Filter_{list}$ and checks whether Equation 1 holds. If no such *filter* is found, then U-FUZZ tightens the condition Φ_S by combining multiple elements from $Filter_{list}$shown as ③ in Figure 2. For example, consider the illustration shown in Figure 4. If none of the atomic filters can accurately classify `Packet 1` and `Packet 2`, then we combine multiple filters $\{nr-rrc.c1,\ nr-rrc.establishmentCause\}$ (i.e., the presence of both fields $nr-rrc.c1$ and $nr-rrc.establishmentCause$) and $\{nr-rrc.c1 = 1,\ nr-rrc.establishmentCause\}$ (i.e., the presence of field value $nr-rrc.c1 = 1$ **and** the presence of field $nr-rrc.establishmentCause$). Intuitively, we seek a stricter condition to accurately filter `Packet 1` and `Packet 2` from the list of all packets. While the aforementioned process leads to combinatorial explosion, in our evaluation, we almost always found the accurate filters by combining at most two atomic filters.

Finally, we note that the above process is repeated for each common layer if a filter Φ_S is not found to hold $\Phi_S\left(\bigcup_{i=1}^{n} P_i\right) = P_{list}$. In the case the state mapper fails to find any common filter, such packets are mapped to an "Unknown" state and the fuzzer will not mutate such packets during the fuzzing session (shown as ② in Figure 2). The outcome of this state mapping process is a list of $\langle S_i, \Phi_{S_i}\rangle$ where S_i is the protocol state label and Φ_{S_i} is the respective filter (condition) to label a packet with state S_i. Thus, during the fuzzing process, if an arbitrary packet P satisfies the condition Φ_{S_i}, then P is mapped to state S_i (shown as ① in Figure 2). This is then used to guide the fuzzing process to improve state/transitions coverage.

B. Stateful Man-in-the-Middle Fuzzing

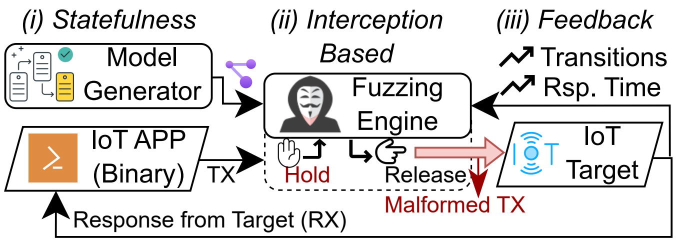


Fig. 6. U-FUZZ Fuzzing Engine Characteristics and Workflow.

The fuzzing engine of U-FUZZ employs three characteristics that are crucial to enable generalized fuzz testing of wired or wireless protocols. The first characteristic, namely ***(i) statefulness***, ensures that U-FUZZ can discern and prioritize mutation of messages that hold the protocol state (context) of the SUT. For instance, this allows to find vulnerabilities in the SUT that are only triggered after many valid messages have been exchanged. In contrast, a random fuzzer lacks discernment between messages type and therefore is unpractical to find attack vectors that *might* appear deep into the SUT communication. To this end, U-FUZZ track states during live communication based on the generated state-machine model, previously provided by the *Model Generator* component (see Figure 6).

Next, ***(ii) interception-based*** fuzzing enables the use of an existing binary program (black-box) containing the protocol stack to generate messages during the fuzzing session. As shown in Figure 6, this characteristic is depicted by the positioning of the fuzzing engine between the SUT and IoT APP (man-in-the-middle). This facilitates the fuzzing setup by not requiring bespoke message generation for the target IoT protocol as often required by previous work [15]. Instead, the user can simply provide an existent binary for U-FUZZ to intercept (see "IoT APP" of Figure 6). Furthermore, this approach inherently preserves the context of the SUT communication (i.e., dynamic field values) due to the fuzzing only mutating existent messages originating from the IoT APP. Consequently, this avoids early rejection of mutated messages by the SUT due to trivial context mismatches, as commonly reported by other works [13].

Lastly, ***(iii)*** **feedback-guided** feature guides the fuzzing exploration by steering mutation of packets towards states that might yield problematic SUT behavior. To this end, the fuzzing engine tries to optimize the value of a cost function obtained at the end of a fuzzing session (i.e., maximize state transitions, minimize the time between request and SUT response) to trigger vulnerabilities more efficiently. We use the particle swarm optimization (PSO) for optimizing the chosen cost function, as PSO is effective in situations that involve non-linear and stochastic behavior e.g., in wireless communication [13]. The choice of the cost function depends on the characteristics of the target protocol. For instance, protocols that exhibit many states during communication (i.e., Zigbee, 5G) are better explored by maximizing state transitions. In contrast, targets employing simplistic protocols (or a subset of it) exhibit less state transitions by design. Therefore, maximizing the value of a cost function that returns the time between requests and responses (i.e., "*Rsp. Time*" of Figure 6) helps U-FUZZ to quickly find malformed packets that can trigger a DoS in the SUT.

C. U-FUZZ *Target Monitor Design*

Since U-FUZZ tests multiple protocols, we develop a target monitor that is generic enough to capture SUT crashes or malfunctions for the evaluated protocols (*5G NR, CoAP, Zigbee*). In general terms, U-FUZZ monitors the health of the target during the fuzzing process by checking for SUT response timeouts. In particular, if the target does not send a response back to the IoT APP (see Figure 3), U-FUZZ indicates an SUT malfunction, which is then logged to a capture file for post-fuzzing analysis. This is particularly useful to detect misbehavior in COTS SUT, which may not provide an external means of detecting its health status. For example, Zigbee devices do not have any means of collecting traces other than its over-the-air responses. However, timeouts add caveats to the monitoring process as the time interval to exchange responses or status messages typically varies with protocols. As such, a timeout threshold T_h is employed to indicate the maximum time to wait for an SUT response before U-FUZZ indicates a timeout in the fuzzing logs. This is particularly useful for Zigbee devices, which advertise the status message *Link_status* in precisely 15 seconds. Hence, U-FUZZ can detect SUT malfunction by adjusting T_h higher than such 15-second interval.

Alternatively, if the SUT provides external interfaces, U-FUZZ can directly collect log traces from Android phones (used as our 5G SUT) via ADB interface. Similarly, CoAP targets often run in a user-controlled environment which can easily indicate to U-FUZZ whether the SUT has improperly exited its CoAP process (i.e., crashed).

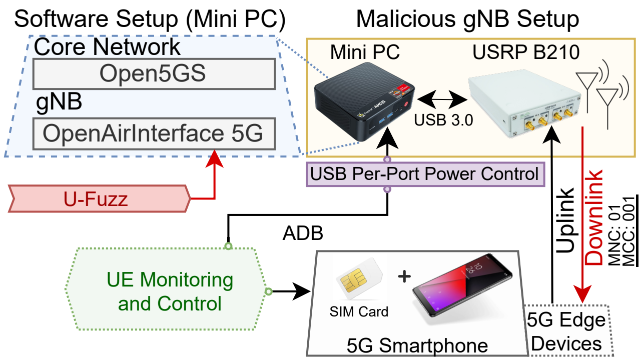


Fig. 7. Implementation Setup for U-FUZZ 5G testing and evaluation

D. *Post-fuzzing Analysis*

Given a vulnerable communication trace, as obtained via stateful fuzzing, the objective of post-fuzzing analysis is to systematically identify the minimal set of modifications (e.g., a minimal set of mutated or duplicated packets) by the fuzzer that results in the crash. After analyzing the attack vector in the vulnerable communication trace (see Figure 3), the U-FUZZ creates a simple C++ script to exploit the vulnerability on an arbitrary device (SUT) employing the respective protocol.

Firstly, our intuition is that the closer a mutated packet μ is with respect to the crash location (highlighted in red in Figure 3), the higher is the chance of μ to be the root cause of a crash. To this end, U-FUZZ starts a normal communication with the SUT and replicates the last mutated packet μ_n (i.e., last purple packet in sequence as shown in Figure 3). If the crash is not triggered by the selected μ_n, then U-FUZZ repeats the process for the next previously mutated packet in sequence i.e., μ_{n-1}. Finally, the replication process succeeds when the crash is triggered for an arbitrary μ. This heuristic allows us to reliably replicate the vulnerabilities found during our fuzzing session. This is of particular importance, as exploitation of targets in a wireless environment remains a time-consuming and non-trivial problem.

IV. IMPLEMENTATION AND EVALUATION SETUP

Evaluation Setup for 5G: Figure 7 outlines the software and hardware setup for 5G fuzzing, which is leveraged by a *AMD Ryzen 7 5800H* processor and Ubuntu 22.04 operating system. The software components include Open5GS to create the 5G Core network [25], OpenAirInterface for creating the base station (gNB) [24] and the U-FUZZ fuzzing engine and state mapper (see Figure 2). The *Software Defined Radio (SDR)* named *USRP B210* is used to enable communication with a 5G COTS smartphone (*OnePlus Nord CE 2*).

Evaluation Setup for Zigbee and CoAP: Figure 8 illustrates the U-FUZZ setup for fuzzing both Zigbee and CoAP implementations. Similarly to the 5G setup, U-FUZZ is deployed

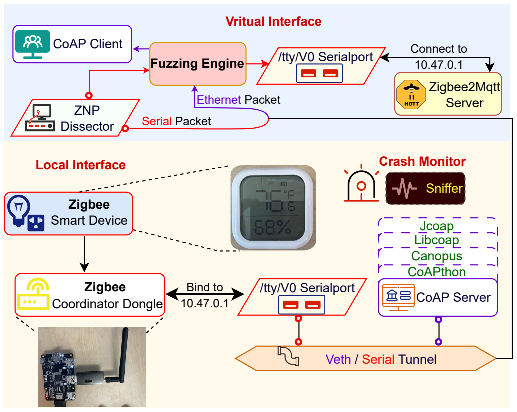


Fig. 8. Implementation Setup for U-Fuzz Zigbee and Coap Fuzzing.

TABLE I
LIST OF TARGETS AND THEIR RESPECTIVE SOFTWARE OR FIRMWARE VERSION

Protocol Implementation / Devices	Firmware / Commit Version
OnePlus Nord CE 2 5G	M_V3_P10
Zigbee2Mqtt	V1.30.1
Texas Instrument CC2531 USB Dongle	Z-Stack_Home_1.2
Sonoff Zigbee 3.0 USB Dongle	Z-Stack_3.0.x
Tuya Smart Plug	20B+TZSKT11BS107
Philips Hue Smart Light Bulb	V1.76.11
libcoap	V4.3.1
Jcoap	Commit 1c06936
Canopus	Commit e374f5b
CoAPthon	V4.0.2

to a host PC running the *Ubuntu 18.04* operating system. However, we attach a *Zigbee Coordinator* to the host PC, i.e., the *CC2531 USB Dongle* from *Texas Instruments*, to establish the communication between the IoT APP (*Zigbee2Mqtt*), and the Zigbee devices under evaluation (e.g., Philips Hue light). The dongle was programmed with the z-stack version *Z-Stack-home-1.2* to enable U-Fuzz communication with Zigbee SUTs via the serial-based ZNP protocol. Furthermore, a simple MQTT client is used alongside the IoT APP to continuously generate message requests to the SUT during the fuzzing sessions. Additionally, a *CC2531 Zigbee Sniffer Dongle* is used to assist detection of anomalies in the response of Zigbee devices as highlighted by the *Crash Monitor* component in Figure 8. In contrast to Zigbee, the CoAP fuzzing setup uses a virtual Ethernet interface and is entirely software-based, thus rendering unnecessary use of dedicated hardware during evaluation. This accelerates the CoAP fuzzing process and simplifies crash log retrieval.

Fuzzer Implementation and Subjects: All the SUTs used in our evaluation (i.e., subjects/targets) and their features are outlined in Table I. U-Fuzz is implemented in C++ (*4722 LoC*) and Python (*585 LoC*). This includes a custom dissector implementation for ZNP protocol, the multi-protocol state mapper and its integration with the stateful fuzzing engines. To generate reference traces of valid communication for the *Multi-protocol State Mapper* (Section III-A), we run the target SUT under a benign communication scenario for approximately 12 hours for each target. Then, the communication logs from the benign communication is leveraged to create the reference state machine. We note that the collection of such reference traces is a one-time effort and is not required *during* fuzzing campaign.

V. Evaluation Results

To evaluate U-Fuzz and showcase its capability, we answer the following research questions:

A. *RQ1: How effective is* U-Fuzz *fuzzer in terms of generating error-prone inputs?*

Table II outlines the effectiveness of U-Fuzz in finding vulnerabilities on multiple *COTS* IoT protocol implementations i.e., 5G NR, Zigbee and CoAP. For this set of experiments, we run the U-Fuzz fuzzing for 12 hours for each subject (i.e., an IoT device/software). In Table II, we classify a crash as a vulnerability when we were able to reliably replicate it with the same group of malicious packets. Each vulnerability name is identified with prefix ***V***. Moreover, due to the lack of internal logs in Zigbee devices, the time to replicate certain crashes may vary and therefore, the reproduction of these crashes is not stable. We recognize such cases as *anomalies* and anomalies are identified with prefix ***A***. As illustrated in Table II, our experiments are conducted on COTS devices e.g., 5G smartphone *OnePlus Nord CE2*, Zigbee smart devices like *Tuya Smart Plug*, *Philips Hue Smart Light Bulb* and Zigbee Coordinator USB Dongles such as *CC2531* and *SONOFF Zigbee 3.0*. In summary, U-Fuzz found sixteen vulnerabilities and two anomalies across all subjects Notably, U-Fuzz discovered 11 new (i.e., previously) unknown vulnerabilities, among which, nine have already received CVE identifiers (see Table II) and two are confirmed and pending CVE assignment from MediaTek. The results outlined in Table II show that U-Fuzz is effective in finding vulnerabilities not only in COTS devices for *5G NR* and *Zigbee* but also in rigid software implementations like libcoap.

B. *RQ2: How efficient is* U-Fuzz *fuzzer?*

Table III outlines both the model coverage and the average time to find the first crash/hangs for different protocol implementations on different subjects. Each experiment was run three times and the average time to find the first crash/hangs was calculated to reduce the randomness of the fuzzing session. For 5G, U-Fuzz finishes each fuzzing iteration by triggering re-connections via ADB whenever the UE completes the 5G procedures. However, fuzzing the data link can normally lead to UE unresponsiveness for several seconds (i.e., 2-4 seconds) without necessarily indicating a firmware issue. This is because 5G modems implement their own waiting states after receiving decoding errors (this happens as U-Fuzz fuzzing engine sends malformed packets) or handling expected failure states. Such inherent delays may increase the fuzzing time for 5G devices e.g., smartphones. Nonetheless, U-Fuzz finds the first 5G vulnerability only in 18 minutes.

TABLE II

EFFECTIVENESS OF U-FUZZ FRAMEWORK IN TERMS OF FINDING VULNERABILITIES IN OUR SUBJECTS. PREFIX ***V*** IS USED FOR VULNERABILITY WHEREAS PREFIX ***A*** IS USED FOR ANOMALY. THE VULNERABILITIES SHOWN IN RED ARE NEW AND WERE PREVIOUSLY UNKNOWN.

Protocol Under Test	Implementation Vulnerability	Affected Hardware/Software Implementation	Impact	CVE Status
5G	V1 - Invalid CellGroupConfig	OnePlus Nord CE 2	Crash	CVE-2024-20004
	V2 - Invalid CellGroupId	OnePlus Nord CE 2	Crash	CVE-2024-20003
	V3 - Invalid RLC Sequence	OnePlus Nord CE 2	Crash	CVE-2023-20702
	V4 - Invalid Uplink Config Element	OnePlus Nord CE 2	Crash	CVE-2023-32843
	V5 - Null Uplink Config Element	OnePlus Nord CE 2	Crash	CVE-2023-32845
Zigbee	V6 - Invalid Transaction and Cluster ID	Texas Instrument CC2531 USB Dongle Z-stack version: Z-Stack_Home_1.2 SONOFF Zigbee 3.0 USB Dongle-P Z-stack version: Z-Stack_3.0.x	Crash	CVE-2023-41388
	V7 - Invalid Transaction and Cluster ID	Zigbee2Mqtt Version:3.8	Control Service Failed	CVE-2023-41003
	V8 - Malformed AF_Data_Request	Zigbee2Mqtt Version:3.8	Crash	CVE-2023-42386
	V9 - Out of Sync State Information	Zigbee2Mqtt Version:3.8	Misleading State Information	CVE-2023-41004
	A1 - Skip Link Status	Tuya Smart Plug	Anomaly	Not applicable
	A2 - Skip Link Status	Philips Hue Smart Light Bulb	Anomaly	Not applicable
CoAP	V10 - NullPointerException	Jcoap	Crash	CVE-2023-34918
	V11 - Illegal_Argument_Exception_Invalid_Token_Length	Jcoap	Crash	CVE-2023-34920
	V12 - Slice_Bounds_out_of_Range	Canopus	Crash	CVE-2023-34919
	V13 - Bad Get Request	Canopus	Crash	CVE-2023-34921
	V14 - Invalid Size1 Size2 Options	libcoap	Crash	CVE-2023-33605
	V15 - Bad POST Request	CoAPthon	Crash	CVE-2018-12680
	V16 - Invalid Unicode Decoding	CoAPthon	Crash	CVE-2018-12680

TABLE III

EFFICIENCY AND COVERAGE OF U-FUZZ ACROSS VARIOUS SUBJECTS. THE MODEL COVERAGE FOR CERTAIN TARGETS IS SHOWN AS **N.A** AS THEIR RESPECTIVE VULNERABILITIES WERE FOUND DURING USE OF THE IOT APP (*Zigbee2Mqtt*) OR THE GENERIC INTERFACE (CC2531).

Protocol Implementation / Devices	1st Crash/Hang	Model Coverage
OnePlus Nord CE 2 5G	18 min.	69.6%
Zigbee2Mqtt	7 min.	N.A
Texas Instrument CC2531 USB Dongle	7 min.	N.A
Tuya Smart Plug	1h.	57.1%
Philips Hue Smart Light Bulb	12h.	61.2%
libcoap	1h.	18%
Jcoap	5 min.	27%
Canopus	15 min.	8.6%
CoAPthon	3h.	28.3%

As for Zigbee, the time for finding the first crash/hang varies widely: from under 10 minutes to more than 12 hours. We recall that U-FUZZ fuzzed Zigbee through the customized *ZNP* protocol by *Texas Instrument* (see Figure 8). Under our implementation setup shown in Figure 8, both the coordinator dongle and the *Zigbee2Mqtt* were the direct fuzzing target, resulting in lower time (7 minutes) to find the first crash/hang. As for the Zigbee smart devices, the crash detection relies on the *Link Status* packet, which must be sent every 15 seconds. Notably, *Philips Hue Smart Light Bulb* requires 12 hours to find the first crash, whereas *Tuya Smart Plug* crashes in an hour. As for *CoAP*, the time to find the first crash primarily relies on the robustness of the implementation. For instance, compared to *CoAPthon* and *libcoap*, which took three hours and one hour, respectively, to exhibit the first crash, *Jcoap* and *Canopus* exhibited the first crash in 5 minutes and 15 minutes, respectively.

Table III also reports the model coverage, computed as the ratio between covered transitions and the total number of transitions in the generated state machine. Like **RQ1**, we also run all experiments for 12 hours for each subject to compute the model coverage. Moreover, All *CoAP* targets achieved relatively low model coverage compared to both *5G NR* and *Zigbee* by U-FUZZ. This is because U-FUZZ performs fuzzing during live communication (see Section III-B) and therefore, the number of transitions covered relies on the complexities of the CoAP client. As for our evaluation, we only implement a simple *CoAP* client for each CoAP server implementation. Nonetheless, U-FUZZ still finds unknown vulnerabilities for all the implementations (***V10-V16*** in Table II), which demonstrate the effectiveness of U-FUZZ.

C. *RQ3: How effective is* U-FUZZ *fuzzer compare to existing blackbox or greybox IoT Fuzzer?*

The design of U-FUZZ allows us to target arbitrary IoT protocols. In this research question, we aim to investigate the effectiveness of U-FUZZ with respect to tools that are specifically engineered for specific protocols. To this end, we choose three state-of-the-art fuzzing tools that most closely match the objective of our fuzzing process. In general, few works are applicable to fuzz protocol implementations on COTS IoT devices. Since our fuzzing objective is not to target the protocol simulation, tools that only perform fuzzing in a simulated environment are not selected. Concretely, we selected *SOTA-5G* [14] [13] for 5G NR, *SOTA-CoAP* [20], for *CoAP* and *SOTA-Zigbee* [35] for *Zigbee* as all these fuzzers are capable to fuzz COTS IoT devices and are specifically engineered for the respective protocols.

For a fair comparison, we run all competitive tools for each chosen representative device and subject for 12 hours. Figure 9 demonstrates our results. For the representative Zigbee devices, as shown in Figure 9, U-FUZZ discovers significantly more total and unique crashes as compared to *SOTA-Zigbee* [35]. This is not only for the targeted *Philips Hue Light bulb* (up to 5x) but also for the *Zigbee2Mqtt* and *Coordinator Dongle* (up to 58x). For the subjects implementing CoAP protocol i.e., libcoap and Canopus, U-FUZZ also outperforms *SOTA-CoAP* [20] by a significant margin, resulting in a total of 152 crashes and nine unique across these subjects compared to 102 total crashes and 2 unique which discovered by *SOTA-CoAP* [20]. As for 5G stateful fuzzing on *COTS* devices, there are no existing works available for comparison. Hence, we enhanced the generic stateful fuzzing engine proposed in earlier work [13] with 5G-specific state mapping rules illustrated in a recent work [14]. Our results in Figure 9 show that U-FUZZ is competitive to *SOTA-5G* [14] [13]. Indeed, *SOTA-5G* [13] [14] slightly outperforms U-FUZZ, as the state machine generated in *SOTA-5G* is more complete due to the manually constructed mapping rules. U-FUZZ constructs the state machine directly from the network capture file completely automatically which also makes the state machine incomplete, resulting in certain missed vulnerabilities detected by *SOTA-5G* [14] [13]. Nonetheless, our U-FUZZ approach discovered two vulnerabilities that were not discovered by *SOTA-5G* [14] [13]. Overall, our experiments show that U-FUZZ, despite having a generic approach for stateful fuzzing, is either competitive to or outperforms protocol-specific fuzzing tools.

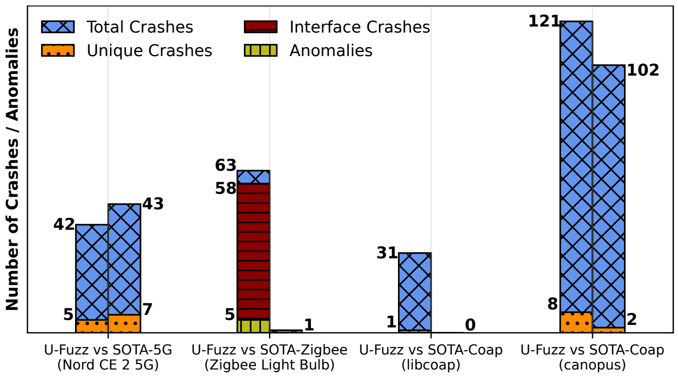


Fig. 9. Vulnerabilities found by U-FUZZ in comparison to state-of-the-art fuzzing tools for respective protocols.

D. *RQ4 (Ablation) - How much does the state mapping process contribute to vulnerability discovery?*

One of the major contributions for our U-FUZZ approach is the *Multi protocol State Mapper* (see Section III-A) which automatically constructs the state machine for arbitrary IoT protocols to guide the fuzzing process. In this research question, we perform an ablation study to concretely understand the contribution of our state mapper in finding security vulnerabilities in IoT protocols. To this end, we create a different version of U-FUZZ by switching off its state mapping component. Such a version, therefore, only performs random fuzzing. We run both U-FUZZ and the aforementioned version for 12 hours across different subjects.

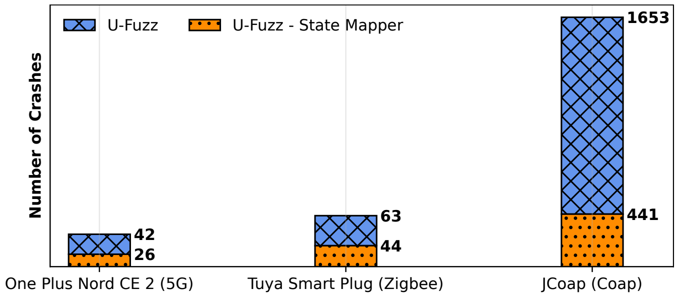


Fig. 10. Crashes found by U-FUZZ with and without State Mapper.

As shown in Figure 10, U-FUZZ discovers significantly more crashes across all studied protocols when the guidance is obtained from the covered state machine transitions. We also analyzed the number of unique crashes (with unique root cause) found for 5G (OnePlus Nord CE 2) and Zigbee protocol implementations (Tuya Smart Plug). Our analysis revealed that U-FUZZ discovered more unique crashes with the guidance obtained from state machine. While U-FUZZ discovered thousands of crashes for Jcoap (as compared to only 431 when state machine guidance was switched off), we did not manually analyze the thousands of crashes to identify the unique Jcoap crashes. Overall, our ablation study indicates that the multi-protocol state mapper and state-machine-based guidance are crucial to discover hidden crashes and vulnerabilities in IoT protocol implementations.

E. *Impact of* U-FUZZ *attacks*

By leveraging the setup depicted in Figure 7, U-FUZZ was able to fuzz a COTS SUT which employs a 5G modem from MediaTek (OnePlus Nord CE 2). In our experiment, U-FUZZ managed to find two new 5G modem vulnerabilities (out of five), namely ***V1*** and ***V2*** (see Table II). Moreover, the impact of ***V1-V5*** results in the SUT rebooting its 5G modem (*MediaTek Dimensity 900*) due to firmware reachable asserts (***V2-V5***) or invalid memory access violation (***V1***). Such issues would allow malicious base-stations within radio range to block the SUT from connecting to even 2G, 3G or 4G networks unless the user disables support for 5G connectivity.

Next, our experiments with CoAP (see Figure 8), was able to reveal 5 new vulnerabilities (out of seven) in reference *CoAP* implementations such as *libcoap, Jcoap, Canopus, CoAPthon* (see ***V10-V14*** in Table V). In summary, U-FUZZ found CoAP DoS vulnerabilities (process crash) which upon continuous abuse, allow an attacker to keep the CoAP server down.

Similarly, U-FUZZ was able to target commercial Zigbee COTS through the common fuzzing setup as shown in Figure 8. However, U-FUZZ found multiple vulnerabilities in both the IoT APP (***V7-V9***) and the interface hardware used to interact to Zigbee SUTs during the fuzzing (***V6***). Such issues could be exploited to either prevent further interaction with the SUT due to timeouts (***V7***), DoS due to process crash (***V8***) or misleading information of the SUT (***V9***).

On the other hand, while most vulnerabilities result in crashes not applicable to the fuzzing targets (*Zigbee2Mqtt* online bridge and *Zigbee Coordinator Dongle*), there are still dozens of *Skip of Link Status* anomaly revealed, reflecting the misbehavior of the Zigbee devices. However, due to the resource limitation, further investigation could not be performed for the root cause.

VI. RELATED WORK

IoT vulnerabilities: Many IoT vulnerabilities have been found in the past few years. Such findings range from protocol design vulnerabilities [2], [3], [40], protocol implementation bugs [16], [38], [39] or product-specific vulnerabilities [5], [6], [23]. In this context, U-FUZZ alleviates the manual effort required to find similar implementation or product specific vulnerabilities by introducing a generalized fuzzing architecture. This provides essential tooling to systematically test the security of the IoT SUT. Furthermore, U-FUZZ automates the reproduction of bugs found during a fuzzing session (i.e., crashes, hangs). This is particularly useful to reduce the time to confirm bugs in COTS IoT devices, without requiring manually-constructed scripts as often required by works covering specific IoT vulnerabilities.

IoT & Network Fuzzing: Several works have been introduced to fuzz IoT Network Protocols. Such works target specific protocols such as Zigbee [33], Bluetooth [13], [15], CoAP [20], 5G [19], [22], [31] or network protocols that are only encapsulated over IP [8]–[10], [21]. However, U-FUZZ distinguishes itself by employing extensive multi-protocol support for both wireless and wired protocols *regardless of its encapsulation*. This gives U-FUZZ a higher degree of freedom to fuzz protocols transmitted in uncommon interfaces such as Zigbee via its ZNP serial-based interface or 5G NR via shared memory. Additionally, reproduction and confirmation of unique bugs is not systematically addressed in prior works and often require use of intrusive approaches [8] or address sanitizers running alongside the SUT firmware [9], which might not be always accessible to the user. Instead, U-FUZZ proposes an interface-agnostic target monitor and generalized and automated replication of crashes caused by complex attack vectors (see Sections III-D and III-C).

Stateful & Greybox Fuzzing: U-FUZZ is orthogonal to greybox fuzzers that aim to test software programs via code coverage such as AFL [41], which require some degree of access to the target source code for instrumentation. In comparison, U-FUZZ avoids reliance on code instrumentation by (i) extracting its fuzzing feedback directly from the response of the COTS target (see *Stateful Feedback* in Figure 1), which is often black-box and (ii) leveraging existing programs (IoT APPs) for *input generation*. These methodologies employed by U-FUZZ are particularly crucial for offering an out-of-the-box testing experience to the user as opposed to generation-based fuzzers that inherently require a manual implementation of the target protocol via model-based approaches [15]–[17], [27]–[29] or semi-automatic state machine construction [13], [14]. Furthermore, U-FUZZ differentiates itself from stateful protocol fuzzers that only support a small selection of protocols out-of-the box [4], [20], [30], [32] or does not support multiple wireless protocols due to lack of generic interfaces [21]. In contrast, U-FUZZ multi-protocol support is heavily extensible through both wired and wireless protocols via its generic *virtual ethernet, serial* and *shared memory* interfaces as discussed in Section II. Hence, U-FUZZ boasts support for ***over 3000*** Wireshark-supported protocols [12], thus granting U-FUZZ a considerable advantage over other multi-protocol fuzzers in terms of protocol coverage without requiring any implementation effort from users.

Test-case Generation: On the one hand, fuzz testing via test-case generation can tackle obscure issues in the SUT implementation due to its capability to cover specific protocol features [14], [26], [33], [37]. On the other hand, such approaches require developing a custom test-suite tailored to a limited set of protocols. Thus, test-case fuzzing requires expertise and manual effort from the user. Alternatively, U-FUZZ steers away from such manual effort by mutating messages of the communication between the user-provided IoT APP and the SUT, therefore not requiring the usage of a bespoke test-suite to fuzz the SUT.

Emulation-based Fuzzing: Approaches based on reverse engineering or emulation [18], [22], [33], [34] allow static and dynamic analysis of IoT implementations. However, such approaches involve a high degree of expertise to support new or additional SUT hardware architectures and protocols. Additionally, emulation demands reverse engineering (if at all possible), ultimately requiring manually replicating crashes on real hardware. In contrast, U-FUZZ does not require *any manual effort* to support arbitrary hardware architectures or protocols due to its black-box and over-the-air architecture.

VII. THREATS TO VALIDITY

Completeness of Constructed State Machine: Since U-FUZZ generates a state machine model from previously provided communication with the SUT, the completeness of the state machine depends on the diversity of messages contained in the capture file provided by the user [13]. During our experiments, this problem is minimized by selecting IoT APPs that are feature-complete and hence can exchange most message types with the SUT. By utilizing a relatively comprehensive state machine, U-FUZZ can delve into deeper states, uncovering additional vulnerabilities or anomalies. This also enhances the representativeness of the calculated model coverage.

Processing of Large Packet Captures: Currently, U-FUZZ does not handle packet capture files with more than $10K$ packets due to memory constraints. However, implementing more efficient packet processing algorithms in the *Capture Processor* (see Figure 2) could remove this limitation [7].

Coverage Improvement on IoT Protocols: The proposed U-FUZZ fuzzing engine and selected cost functions might not always improve SUT model coverage for all possible IoT protocols as compared to state-of-the-art fuzzers specialized in a specific protocol. However, our evaluation of U-FUZZ against targets communicating in heavily adopted IoT protocols (CoAP, Zigbee, 5G), reveals the flexible and yet competitive edge of U-FUZZ over prior IoT fuzzing tools. We also make our framework open source for researchers to extend and optimize U-FUZZ.

Comprehensiveness of the Target Monitor: The target monitor integrated in U-FUZZ is not guaranteed to detect crashes in all possible IoT SUTs. However, this shortcoming is easily addressed by providing several ways in which the user can configure or extend U-FUZZ generic interfaces.

VIII. CONCLUSION

In this paper, we propose U-FUZZ, a framework to automatically discover and replicate security vulnerabilities in both wired and wireless IoT protocol implementations on COTS IoT Devices. Compared to prior works, U-FUZZ brings some concrete advantages: *(i)* U-FUZZ provides a state mapping technique which only relies on network capture files as input to generate both the state machine and the conditions required for state mapping across diverse IoT protocols. This technique could be easily adapted by other stateful fuzzing tools. *(ii)* U-FUZZ stateful fuzzing targets both IP-based (e.g., CoAP) and non-IP-based protocols (e.g., Zigbee and 5G NR), showing concrete evidence on finding vulnerabilities in each such protocol implementations. Moreover, U-FUZZ opens possibilities for stateful fuzzing of over 3000 protocols (as supported by Wireshark) out-of-the-box. With such a possibility, we hope to significantly improve the state-of-the-art for automated security testing in both existing and next generation IoT protocols. In the furture, we plan to extend U-FUZZ to target not only Denial-of-Service (crash or hang), but more complex attacks e.g., leading to information leakage.

We hope the community uses U-FUZZ to test its limits and extends its support beyond the scope targeted in the paper. For the reproduction of research and to extend the capability of U-FUZZ, the source code and all experimental data are available in the following:

https://github.com/asset-group/U-Fuzz

ACKNOWLEDGMENT

This research is partially supported by National Research Foundation, Singapore, under its National Satellite of Excellence Programme "Design Science and Technology for Secure Critical Infrastructure: Phase II" (Award No: NRF-NCR25-NSOE05-0001). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore.

REFERENCES

[1] Bernhard K. Aichernig, Edi Muškardin, and Andrea Pferscher. Learning-based fuzzing of iot message brokers. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 47–58, 2021.

[2] Daniele Antonioli, Nils Ole Tippenhauer, and Kasper Rasmussen. Bias: Bluetooth impersonation attacks. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 549–562, 2020.

[3] Daniele Antonioli, Nils Ole Tippenhauer, and Kasper B. Rasmussen. The KNOB is broken: Exploiting low entropy in the encryption key negotiation of bluetooth BR/EDR. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1047–1061, Santa Clara, CA, August 2019. USENIX Association.

[4] Jinsheng Ba, Marcel Böhme, Zahra Mirzamomen, and Abhik Roychoudhury. Stateful greybox fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3255–3272, Boston, MA, August 2022. USENIX Association.

[5] Marco Casagrande, Riccardo Cestaro, Eleonora Losiouk, Mauro Conti, and Daniele Antonioli. E-spoofer: Attacking and defending xiaomi electric scooter ecosystem. In *Proceedings of the 16th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '23, page 85–95, New York, NY, USA, 2023. Association for Computing Machinery.

[6] Marco Casagrande, Eleonora Losiouk, Mauro Conti, Mathias Payer, and Daniele Antonioli. Breakmi: Reversing, exploiting and fixing xiaomi fitness tracking ecosystem. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(3):330–366, Jun. 2022.

[7] Danilo Cerović, Valentin Del Piccolo, Ahmed Amamou, Kamel Haddadou, and Guy Pujolle. Fast packet processing: A survey. *IEEE Communications Surveys & Tutorials*, 20(4):3645–3676, 2018.

[8] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing. In *NDSS*, 2018.

[9] Xiaotao Feng, Ruoxi Sun, Xiaogang Zhu, Minhui Xue, Sheng Wen, Dongxi Liu, Surya Nepal, and Yang Xiang. Snipuzz: Black-box fuzzing of iot firmware via message snippet inference. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 337–350, New York, NY, USA, 2021. Association for Computing Machinery.

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

[11] Paul Fiterău-Broştean, Bengt Jonsson, Konstantinos Sagonas, and Fredrik Tåquist. Dtls-fuzzer: A dtls protocol state fuzzer. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 456–458, 2022.

[12] Wireshark Foundation. Display filter reference. https://www.wireshark.org/docs/dfref/, October 2023.

[13] Matheus E. Garbelini, Vaibhav Bedi, Sudipta Chattopadhyay, Sumei Sun, and Ernest Kurniawan. BrakTooth: Causing havoc on bluetooth link manager via directed fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 1025–1042, Boston, MA, August 2022. USENIX Association.

[14] Matheus E Garbelini, Zewen Shang, Sudipta Chattopadhyay, Sumei Sun, and Ernest Kurniawan. Towards automated fuzzing of 4g/5g protocol implementations over the air. In *GLOBECOM 2022-2022 IEEE Global Communications Conference*, pages 86–92. IEEE, 2022.

[15] Matheus E. Garbelini, Chundong Wang, and Sudipta Chattopadhyay. Greyhound: Directed greybox wi-fi fuzzing. *IEEE Transactions on Dependable and Secure Computing*, 19(2):817–834, 2022.

[16] Matheus E. Garbelini, Chundong Wang, Sudipta Chattopadhyay, Sun Sumei, and Ernest Kurniawan. SweynTooth: Unleashing mayhem over bluetooth low energy. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 911–925. USENIX Association, July 2020.

[17] Hugo Gascon, Christian Wressnegger, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. Pulsar: Stateful black-box fuzzing of proprietary network protocols. In Bhavani Thuraisingham, XiaoFeng Wang, and Vinod Yegneswaran, editors, *Security and Privacy in Communication Networks*, pages 330–347, Cham, 2015. Springer International Publishing.

[18] Grant Hernandez, Marius Muench, Dominik Maier, Alyssa Milburn, Shinjo Park, Tobias Scharnowski, Tyler Tucker, Patrick Traynor, and Kevin Butler. Firmwire: Transparent dynamic analysis for cellular baseband firmware. In *Network and Distributed Systems Security Symposium (NDSS) 2022*, 2022.

[19] Eunsoo Kim, Dongkwan Kim, CheolJun Park, Insu Yun, and Yongdae Kim. Basespec: Comparative analysis of baseband software and cellular specifications for l3 protocols. *Proceedings 2021 Network and Distributed System Security Symposium*, 2021.

[20] Fredrik Liljedahl. Exploring the possibilities of robustness testing of coap implementations using evolutionary fuzzing. https://www.diva-portal.org/smash/get/diva2:1383128/FULLTEXT01.pdf, 2019.

[21] Zhengxiong Luo, Junze Yu, Feilong Zuo, Jianzhong Liu, Yu Jiang, Ting Chen, Abhik Roychoudhury, and Jiaguang Sun. Bleem: Packet sequence oriented fuzzing for protocol implementations. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 4481–4498, Anaheim, CA, August 2023. USENIX Association.

[22] Dominik Maier, Lukas Seidel, and Shinjo Park. Basesafe: Baseband sanitized fuzzing through emulation. In *Proceedings of the 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '20, page 122–132, New York, NY, USA, 2020. Association for Computing Machinery.

[23] Sen Nie, Ling Liu, and Yuefeng Du. Free-fall: Hacking tesla from wireless to can bus. https://www.blackhat.com/us-17/briefings/schedule/#free-fall-hacking-tesla-from-wireless-to-can-bus-6415, 2017.

[24] Navid Nikaein, Mahesh K. Marina, Saravana Manickam, Alex Dawson, Raymond Knopp, and Christian Bonnet. OpenAirInterface: A Flexible Platform for 5G Research. *SIGCOMM Comput. Commun. Rev.*, 44(5):33–38, October 2014.

[25] Open5GS. Open source implementation for 5G core and EPC. https://github.com/open5gs/open5gs.

[26] CheolJun Park, Sangwook Bae, BeomSeok Oh, Jiho Lee, Eunkyu Lee, Insu Yun, and Yongdae Kim. DoLTEst: In-depth downlink negative testing framework for LTE devices. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 1325–1342, Boston, MA, August 2022. USENIX Association.

[27] Haram Park, Carlos Kayembe Nkuba, Seunghoon Woo, and Heejo Lee. L2fuzz: Discovering bluetooth l2cap vulnerabilities using stateful fuzz testing. In *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 343–354, 2022.

[28] Joshua Pereyda. boofuzz: Network protocol fuzzing for humans. https://github.com/jtpereyda/boofuzz, April 2017.

[29] Andrea Pferscher and Bernhard K. Aichernig. Stateful black-box fuzzing of bluetooth devices using automata learning. In Jyotirmoy V. Deshmukh, Klaus Havelund, and Ivan Perez, editors, *NASA Formal Methods*, pages 373–392, Cham, 2022. Springer International Publishing.

[30] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. Aflnet: A greybox fuzzer for network protocols. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 460–465, 2020.

[31] Srinath Potnuru and Prajwol Kumar Nakarmi. Berserker: ASN.1-based fuzzing of radio resource control protocol for 4g and 5g. In *2021 17th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*. IEEE, oct 2021.

[32] Shisong Qin, Fan Hu, Zheyu Ma, Bodong Zhao, Tingting Yin, and Chao Zhang. Nsfuzz: Towards efficient and state-aware network service fuzzing. *ACM Trans. Softw. Eng. Methodol.*, 32(6), sep 2023.

[33] Mengfei Ren, Xiaolei Ren, Huadong Feng, Jiang Ming, and Yu Lei. Z-fuzzer: Device-agnostic fuzzing of zigbee protocol implementation. In *Proceedings of the 14th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '21, page 347–358, New York, NY, USA, 2021. Association for Computing Machinery.

[34] Jan Ruge, Jiska Classen, Francesco Gringoli, and Matthias Hollick. Frankenstein: Advanced wireless fuzzing to exploit new bluetooth escalation targets. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 19–36. USENIX Association, August 2020.

[35] Tom Rust. Fuzzing zigbee using z-stack. https://www.cs.ru.nl/bachelors-theses/2022/Tom_Rust___1040068___Fuzzing_Zigbee_using_Z-Stack.pdf, 2022.

[36] Konstantinos Sagonas and Thanasis Typaldos. Edhoc-fuzzer: An edhoc protocol state fuzzer. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2023, page 1495–1498, New York, NY, USA, 2023. Association for Computing Machinery.

[37] Zujany Salazar, Huu Nghia Nguyen, Wissam Mallouli, Ana R. Cavalli, and Edgardo Montes de Oca. 5greplay: A 5g network traffic fuzzer - application to attack injection. In *Proceedings of the 16th International Conference on Availability, Reliability and Security*, ARES '21, New York, NY, USA, 2021. Association for Computing Machinery.

[38] Ben Seri and Gregory Vishnepolsky. Blueborne: Unveiling zero day vulnerabilities and security flaws in modern bluetooth stacks. https://armis.com/blueborne/, 2017.

[39] Ben Seri, Gregory Vishnepolsky, and Dor Zusman. BleedingBit: The hidden attack surface within BLE chips. https://armis.com/bleedingbit/, 2018.

[40] Mathy Vanhoef and Frank Piessens. Release the kraken: New kracks in the 802.11 standard. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 299–314, New York, NY, USA, 2018. Association for Computing Machinery.

[41] Michal Zalewski. American Fuzzy Lop. https://github.com/google/AFL, April 2017.