Fuzz The Power: Dual-role State Guided Black-box Fuzzing for USB Power Delivery

Kyungtae Kim and Sungwoo Kim, *Purdue University;*
Kevin R. B. Butler, *University of Florida;* Antonio Bianchi,
Rick Kennell, and Dave (Jing) Tian, *Purdue University*

https://www.usenix.org/conference/usenixsecurity23/presentation/kim-kyungtae

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

August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

Fuzz The Power: Dual-role State Guided Black-box Fuzzing for USB Power Delivery

Kyungtae Kim[†], Sungwoo Kim[†], Kevin R. B. Butler[§], Antonio Bianchi[†], Rick Kennell[†], Dave (Jing) Tian[†]

[†]*Purdue University,* [§]*University of Florida*
[†]*{kim1798,sk,antoniob,rick,daveti}@purdue.edu,* [§]*butler@ufl.edu*

Abstract

USB Power Delivery (USBPD) is a state-of-the-art charging protocol for advanced power supply. Thanks to its high volume of power supply, it has been widely adopted by consumer devices, such as smartphones and laptops, and has become the de facto USB charging standard in both EU and North America. Due to the low-level nature of charging and the complexity of the protocol, USBPD is often implemented as proprietary firmware running on a dedicated microcontroller unit (MCU) with a USBPD physical layer. Bugs within these implementations can not only lead to safety issues, e.g., over charging, but also cause security issues, such as allowing attackers to reflash USBPD firmware.

This paper proposes FUZZPD, the first *black-box* fuzzing technique with *dual-role state* guidance targeting off-the-shelf USBPD devices with closed-source USBPD firmware. FUZZPD only requires a physical USB Type-C connection to operate in a *plug-n-fuzz* fashion. To facilitate the black-box fuzzing of USBPD firmware, FUZZPD manually creates a dual-role state machine from the USBPD specification, which enables both state coverage and transitions from fuzzing inputs. FUZZPD further provides a multi-level mutation strategy, allowing for fine-grained state-aware fuzzing with intra- and inter-state mutations. We implement FUZZPD using a Chromebook as the fuzzing host and evaluate it against 12 USBPD mobile devices from 7 different vendors, 7 USB hubs from 7 different vendors, and 5 chargers from 5 different vendors. FUZZPD has found 15 unique bugs, 9 of which have been confirmed by the corresponding vendors. We additionally conduct a comparison between FUZZPD and multiple state-of-the-art black-box fuzzing techniques, demonstrating that FUZZPD achieves code coverage that is 40% to 3x higher than other solutions. We then compare FUZZPD with the USBPD compliance test suite from USBIF and show that FUZZPD can find 7 more bugs with 2x higher code coverage. FUZZPD is the first step towards secure and trustworthy USB charging.

1 Introduction

With its usability and versatility, USB Type-C (a.k.a., USB-C) has quickly replaced traditional USB ports (e.g., Type-A and B) as the default USB connector in a large number of consumer devices, including smartphones, laptops, etc. USB Power Delivery (USBPD or PD) is a pivotal function over the USB-C interface. Thanks to the extra pins available in USB-C, USBPD offers "*fast charging*" by providing a power negotiation capability and a higher power transmission for each VBUS line. It keeps evolving with new revisions supporting more powerful and various functionalities, such as further increased power supply (up to 240W) and authentication capability [1]. USBPD has become the de facto USB charging standard in both EU and North America.

Due to the low-level nature of charging and the complexity of the protocol, USBPD is often implemented as proprietary firmware running on a dedicated microcontroller unit (MCU) with a USBPD physical layer. Bugs within these implementations can not only lead to safety issues, e.g., over charging [2], but also cause security issues, such as allowing attackers to reflash USBPD firmware or launch denial of service [3]. While software fuzzing techniques have been proven effective to USB bug discovery [4–6], all of them target USB stacks within host operating systems assuming a white-box environment with some instrumentation for coverage as the feedback for fuzzing. On the contrary, USBPD stacks are usually implemented apart from host operating systems and provided as a binary blob (firmware) without source files publicly available, yielding all existing USB fuzzing techniques futile.

In this paper, we propose FUZZPD, the first *black-box* fuzzing technique with *dual-role state* guidance targeting off-the-shelf USBPD devices with closed-source USBPD firmware. FUZZPD only requires a physical USB Type-C connection to operate in a *plug-n-fuzz* fashion without the burden of hardware emulation and firmware rehosting. As a result, FUZZPD is ready to fuzz *any* commodity USBPD devices found in the wild. FUZZPD facilitates the black-box fuzzing of USBPD firmware by manually constructing a dual-role state machine from the USBPD specification, enabling both state coverage and transitions from fuzzing inputs. Specifically, we take the advantage of standard PD functional sequences that exchange orderly to carry out different PD functions. We obtain these PD functional sequences from the USBPD standard [1] and consolidate the sequences into a state machine with transitions populated with the functional sequences. Our state machine covers dual-role, e.g., power sink and source, defined by the specification, and FuzzPD supports on-the-fly power role switching during fuzzing. FuzzPD further provides a multi-level mutation strategy, allowing for a fine-grained state-aware fuzzing with intra- and inter-state mutations, leveraging the dual-role state coverage as the guidance and PD message seeding.

We implement FuzzPD using Chromium EC [7] on a Chromebook [8] as the fuzzing host and evaluate it against 12 USBPD mobile devices from 7 different vendors, 7 USB hubs from 7 different vendors, and 5 chargers from 5 different vendors. FuzzPD have found 15 unique bugs, ranging from PD compliance violations to over-voltage issues and out-of-bounds bugs within firmware. All of the bugs have been reported to the corresponding vendors and 9 of them have been confirmed by the vendors. We further compare FuzzPD against multiple state-of-the-art black-box fuzzing solutions [9–11] in terms of fuzzing efficacy. We demonstrate that FuzzPD outperformed all other techniques in terms of execution coverage, which ranges from 40% to 3x higher, using an open-source USBPD firmware as the ground truth. We then compare FuzzPD with the USBPD compliance test suite from USBIF [12] in terms of bug detection capability, and show that FuzzPD can find 7 more bugs with 2x higher code coverage. We open-source FuzzPD to facilitate the USBPD security research in the community [13].

The key contributions of this paper are as follows:

- FuzzPD is the first black-box USBPD fuzzing technique targeting off-the-shelf USBPD devices.
- We design a dual-role state machine by extracting and abstracting PD functional sequences from the USBPD specification to guide the black-box fuzzing in FuzzPD. We further devise a two-stage mutation strategy for a fine-grained intra- and inter-state state explorations considering both the dual-role state coverage and PD message seeding
- We implement FuzzPD using Chromium EC on a Chromebook as the fuzzing host, ready to fuzz any commodity USBPD devices found in the wild.
- We evaluate FuzzPD using 24 different USBPD-capable devices across 19 different vendors and found 15 unique bugs. Comparing to multiple state-of-the-art black-box fuzzing solutions and the USBPD compliance test suite from USBIF, FuzzPD outperforms other techniques in terms of code coverage, ranging from 40% to 3x higher, and also find 7 more bugs with 2x higher code coverage respectively.

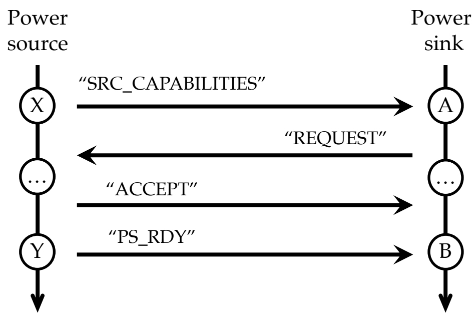


Figure 1: A functional sequence for power contract.

2 Background

USB Type-C (a.k.a. USB-C) [14] has become mainstream for a variety of peripherals and smart devices, thanks to its versatility and enhanced usability. In parallel, USB Power Delivery (USBPD or PD), the power delivery standard carried over USB-C, has been widely accepted and adopted for its rapid power charging capability, such as the ability of providing up to 240W [1]. Apart from increased power supply, USBPD also supports the alternate mode defined by USB-C, which enables USB-C ports to transfer non-USB data via multiple interfaces, such as DisplayPort and Thunderbolt. USB4 also relies on USBPD for power establishment.

USBPD communication involves two parties in different roles: a *power source* (providing charge) and a *power sink* (accepting charge). The specification [1] mandates how both roles should be implemented, detailing a finite state machine representing the different states a USBPD device can assume, and the possible transitions between these states.

In USBPD, the data communication between the two connected parties is achieved through the exchange of *PD messages*. Each PD message comes with a header, and optionally includes data objects depending on the type of the message — *data* and *extended* message types have up to 7 data objects, while messages of the *control* type do not have any data object. The specification also allows vendors to implement vendor-specific messages (VDMs) that can be exchanged between devices.

A *functional sequence* is a sequence of PD messages to be exchanged between the two roles, as illustrated in Figure 1. Both roles interact with each other by receiving and sending corresponding messages in a designated order within a specific functional sequence. Different functional sequences are used for different (unique) PD functionalities. For example, the "power contract" sequence in Figure 1 is used for negotiating the charging power to deliver, whereas the "power role swap" sequence is needed for switching the power roles between two devices.

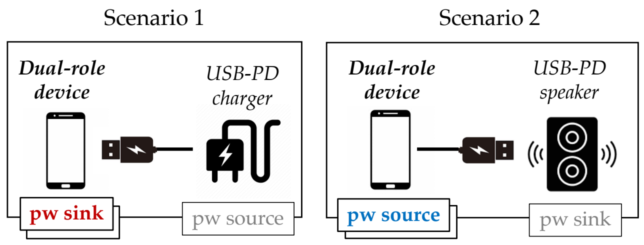


Figure 2: A dual-role USBPD device.

Functional sequences are closely coupled to state transitions of both power roles. For instance, as seen in Figure 1, the functional sequence induces a state change from state "X" to state "Y" for the power source device, and, simultaneously, from state "A" to state "B" for the power sink role device.

A sequence is considered as `active` by a device if it is initiated by the device itself. Otherwise, it is considered as `passive`. From the viewpoint of a power source role, for example, the power contract sequence falls into an active sequence because the source role is the one exclusively initiating this sequence. Conversely, the power contract sequence is a passive sequence from the sink role's point of view.

While there are USBPD devices supporting only a single role by nature (e.g., USB power chargers are typically source-only), modern USBPD devices can support the dual-role capability (Table 2). Dual-role devices are capable of serving as either power roles, depending on the usage scenario. For example, a PD-enabled smartphone could be charged as a power sink if a USB charger is plugged to it, whereas it becomes a power source and delivers power when connected to a USB speaker (see Figure 2). Furthermore, dual-role devices can swap their current roles at any point of their communication, while being connected to another dual-role device.

3 Security Model

We trust the USB hardware parts and especially the physical layer of target PD devices, and assume they are free of hardware defects (e.g., over-voltage caused by a rogue cable) and operate properly in terms of functionality. We further limit the attack surface to the Control Channel (CC) lines of USB-C, where PD messages are exchanged, instead of worrying about typical USB attacks launched via data pins, e.g., BadUSB attacks [15].

We assume that PD firmware within target PD devices contains software defects and vulnerabilities. In this scenario, adversaries can exploit these vulnerabilities, which only requires a physical USB-C connection to cause safety, security, and privacy violations. For example, an adversary can install a malicious USB-C charging port (or compromise existing ones) in an airport. After connecting to a victim PD device (e.g., smartphone), the illegitimate PD logic within the malicious USBPD controller can ignore the power negotiation and provide over-current and over-voltage power supply to physically damage the target PD device. This rogue USB-C charging port can also issue malformed and even malicious PD messages to attack the target PD device, e.g., exploiting a buffer overflow within target PD firmware implementation to launch denial of service attack, or sending a firmware reflashing command to the target PD device.

4 Motivation and Challenges

Existing works [4–6, 16–18] have demonstrated the efficacy of fuzzing techniques on detecting bugs and vulnerabilities within USB stack implementations. State-of-the-art USB fuzzing solutions [4, 6] have managed to get rid of the hardware dependency via emulation and leveraged code or state coverage as the feedback for efficient fuzzing input mutations. A naive approach would be applying existing USB fuzzing techniques to USBPD, which unfortunately does not work due to a number of unique challenges imposed by real-world USBPD implementations.

C1: Diverse and closed-source ecosystems. Unlike typical USB stack implementations found in operating systems, USBPD functions are usually implemented in firmware, running on dedicated PD controllers (e.g., MCUs) and physical layers. Different vendors have their own USBPD implementations and keep both firmware and hardware as proprietary. It is possible to build an efficient firmware fuzzing environment using firmware rehosting, which emulates the hardware (both MCU and peripherals) to run firmware locally [19–22]. Using this approach, we can run various firmware samples on a single architecture at-scale (e.g., on a server dedicated to fuzzing) and even instrument the firmware to support features such as runtime coverage feedback.

Despite such benefits, firmware rehosting does not scale for PD implementations. Specifically, hardware emulation requires a significant amount of engineering effort to support one single architecture and peripherals within an System-on-Chip (SoC), let alone dozens of different USBPD implementations. More importantly, acquiring and unpacking PD firmware are often challenging. For instance, vendors often disable the firmware dumping capability within final products, and apply obfuscation and encryption schemes to protect against firmware analysis. In the end, what we need is a "plug-n-play" fuzzing technique that can fuzz off-the-shelf USBPD devices with minimum requirements.

C2: Fully stateful communications. If the statefulness of USB communication is mainly reflected at the enumeration phase, USBPD communication is fully driven by the state machines defined by the specification. Since PD message is a key building block of USBPD communication (see Figure 2), a naive approach is to simply fuzz arbitrary PD messages by mutating individual messages with different message types, as shown in Figure 3 (1). Despite following the format of PD

Fuzzer	**Domain**	**Target Role**	**Target Stack**	**Type**	**Hardware**	**Stateful**	**Mutation Guidance**	**In-State Mutation**
FaceDancer [16]	USB	Single (Host)	OS	Blackbox	On-device	×	–	–
Umap2 [17]	USB	Single (Host)	OS	Blackbox	On-device	×	–	–
vUSBf [18]	USB	Single (Host)	OS	Greybox	Emulation	×	–	–
Syzkaller [4]	USB	Single (Host)	OS	Greybox	Emulation	△	Code cov	Code cov
USBFuzz [5]	USB	Single (Host)	OS	Greybox	Emulation	×	Code cov	–
FuzzUSB [6]	USB	Single (Gadget)	OS	Greybox	Emulation	○	State cov	Code cov
Snipuzz [11]	IoT	Single	Firmware	Blackbox	On-device	×	–	–
DIANE [10]	IoT	Single	Firmware	Blackbox	On-device	×	–	–
IoTfuzzer [9]	IoT	Single	Firmware	Blackbox	On-device	×	–	–
FUZZPD	USBPD	Dual (Src/Sink)	Firmware	Blackbox	On-device	○	State cov	Seeding

Table 1: State-of-the-art USB/Black-box fuzzing techniques. In the **Stateful** column, the symbol '×' indicates that fuzzers do not consider the statefulness when fuzzing targets, whereas fuzzers with the symbol '○' achieve state-aware fuzzing. Meanwhile, Syzkaller is represented by the symbol '△' because it has implicit and limited consideration of the target's statefulness, using input templates.

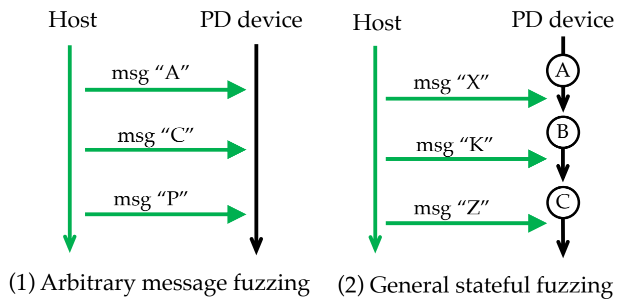


Figure 3: USBPD fuzzing scenarios.

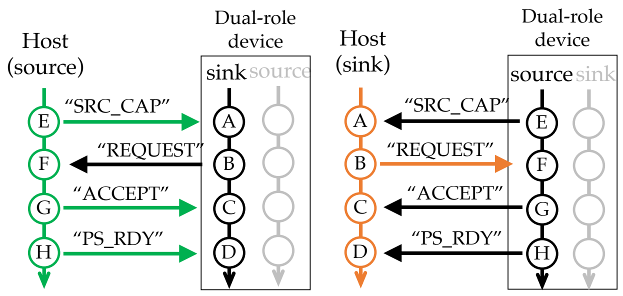


Figure 4: USBPD communication for a dual-role PD device.

messages, this state-agnostic approach only discloses trivial and shallow bugs within PD implementations, and might not even get the fuzzing target connected after hours of fuzzing.

A better idea might be leveraging the statefulness of PD communications and using states as guidance during fuzzing. For instance, we can employ the standard PD state machine defined by the specification [1] and mutate input messages in the hope of triggering different state transitions, as shown in Figure 3 (2), similar to existing state-guided fuzzers [6, 23–26]. This approach enables a fuzzer to provide mutated messages in different states of the state machine towards new state exploration. Unfortunately, such a textbook implementation of state-guided fuzzing does not work for real-world PD implementations for two reasons. First, PD state machine defined by the specification contains states related to non-deterministic or non PD message triggers. For instance, a state transition might depend not only on the PD messaging but also the timer interrupt, which is out of the control of a fuzzer and yields a non-deterministic transition. Second, it is unlikely to exactly keep track of actual state changes of the fuzzing target during PD communication without accessing and analyzing the target PD firmware (which is often infeasible, as mentioned in **C1**). As a result, to reach the core logic of PD implementations during fuzzing, we need a PD state machine as the guidance, with transitions fully controlled by fuzzers and the ease of tracking fuzzing target's state without understanding the target firmware.

C3: Dual-role capability. Existing USB fuzzing solutions [4–6, 16–18] only target single-role fuzzing, e.g., USB host fuzzing or USB gadget fuzzing. In fact, most USB fuzzers aim for USB host fuzzing except FuzzUSB [6] targeting USB gadget stacks. The reason is that a big portion of existing USB devices are not dual-role device, e.g., laptops. Even for dual-role USB devices, USB host stacks and USB gadget stacks are independent from each other without sharing much code base. As discussed in Figure 2, however, modern USBPD devices are usually dual-role capable, and both roles are likely to share the same code base due to their tight coupling. Therefore, conventional single-role fuzzing solutions are unlikely to examine the other role of USBPD implementations. For instance, given the functional sequence example in Figure 4, its 2nd message `"REQUEST"` in the sequence is only triggered while in the power sink role, Likewise, the remaining three messages in the sequence can be solely sent (or mutated) from the power source role.

To maximize the fuzzing coverage, a PD fuzzer needs not only to support both power roles, but also to be aware of the constraints under each power role, e.g., what functionality could be conducted under a certain role. Recall **C2** to achieve stateful fuzzing using the PD state machine. This means the PD state machine has to cover dual-role with all the constraints encoded for each state transition, assuming a power role switch might happen at any time during PD communication.

C4: In-state fuzzing input mutation. As discussed in **C2**, state guidance is an essential requirement to realize USBPD fuzzing. However, state guidance alone does not suffice for optimizing mutation generation. Specifically, state-guided fuzzing particularly aims at exploring a new state within the given state machine, but it cannot suggest further directions for *in-state* mutation. For example, Syzkaller [4] and FuzzUSB [6] leverage the code coverage as the feedback to help in-state fuzzing input mutations. However, code coverage requires instrumentation, which, in-turn typically requires emulation capability or source code availability, which is impractical for PD firmware, as discussed in **C1**.

The message syntax of USBPD is well-defined in the specification, providing valid templates (and outline) for fuzzing input generation that can be used for individual message mutation within a state. However, without any feedback from in-state fuzzing, the fuzzing input mutation is still pure random, leading to a randomly generated PD message for fuzzing without the awareness of the current power role or state, and downgrading a stateful fuzzer to a state-agnostic fuzzer. Therefore, we need an efficient way to generate (mostly) valid inputs for in-state fuzzing even without any feedback to improve mutation generation and with the understanding of the constraints imposed by the current power role and state.

As shown in Table 1, we did a mini systematization of existing USB fuzzing solutions and state-of-the-art black-box fuzzing techniques to highlight and distinguish this work from all others. Compared to all USB fuzzing solutions, this work is the first targeting USBPD firmware and supporting dual-role during fuzzing. By comparing recent black-box fuzzing techniques, this work introduces state coverage as the feedback and incorporates in-state mutation.

5 Design

To address the challenges of USBPD fuzzing discussed in §4, we present a new USBPD fuzzing technique, FUZZPD. FUZZPD adopts a black-box fuzzing approach, and tackles stateful USBPD communication for dual-role PD devices based on a dual-role state machine. FUZZPD also improves its mutation efficiency relying on multi-staged and message seed guided mutations.

5.1 Overview of FUZZPD

In this section, we describe the overall design of FUZZPD.

Generic black-box USBPD fuzzing. FUZZPD is the first black-box fuzzing technique for USB Power Delivery. As presented in Figure 5, FUZZPD adopts an on-device fuzzing design, examining real USBPD devices with a connection to a dedicated USB-C cable. Since USBPD operates over USB-C

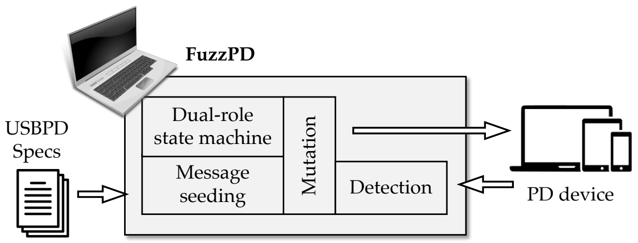


Figure 5: Overview of FUZZPD

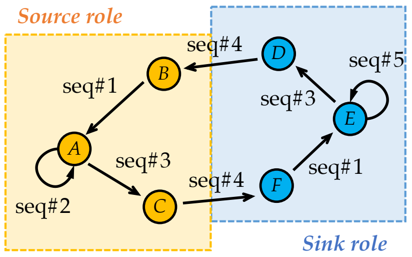


Figure 6: A dual-role state machine

that defines a physical interface, our approach of on-device fuzzing makes FUZZPD compatible with any of USBPD capable devices and carry out fuzzing in a generic way without requiring expensive device emulation and tedious firmware analysis (**C1**).

Dual-role state guidance. To tackle stateful USBPD communication (**C2**) and examine dual-role PD functionality of USBPD devices (**C3**), we design a customized dual-role state machine (§5.2). We achieve a fully controllable dual-role state machine with PD functional sequences for its state transitions. Given the dual-role state machine, triggering its functional sequences can move states implicitly for a black-box target without the knowledge of its complex internals. Meanwhile, the dual-role state machine seamlessly encompasses a dedicated functional sequence for power role swap. Thus, dual-role switch is offloaded under control of the dual-role state machine, and state transitions can swap power roles at any time during fuzzing, thereby reaching different PD functionality in different roles.

Multi-level mutation using message seeding. To enhance the effectiveness of mutations by maximizing fuzzing scope (**C4**), FUZZPD uses two-staged stateful mutations that deal with inter- and intra-state fuzzing scope (§5.4). Other than state coverage guided mutation for inter-state exploration, we leverage message seeding for seed message inputs, which provides further guidance for an in-state message mutation. Our message seeding, originating from USBPD compliance test rules, not only enables functional sequence aware mutation, but also generates high-quality input messages to explore deeper functional USBPD logic (§5.3).

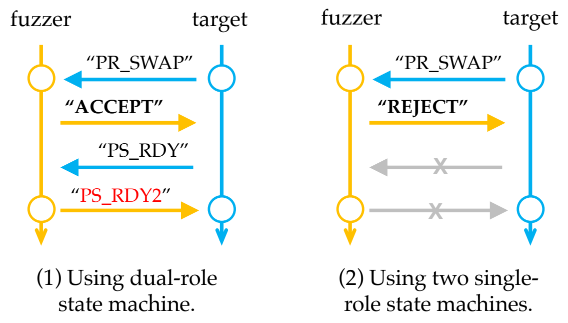

fuzzer
target
"PR_SWAP"
"ACCEPT"
"PS_RDY"
"PS_RDY2"
(1) Using dual-role state machine.
fuzzer
target
"PR_SWAP"
"REJECT"
(2) Using two single-role state machines.

Figure 7: Valid messages within the power role swap sequence in two fuzzing approaches.

5.2 Dual-role State Machine

To tackle the challenges in using the standard state machine (**C2**) and handling dual-role characteristics of PD devices (**C3**), we develop a specialized dual-role state machine, as illustrated in Figure 6. The key insight of our dual-role state machine is in leveraging *functional sequences* to build a fully controllable and consolidated state machine. As discussed in Figure 2, functional sequences are solely executed by USBPD messages (i.e., fuzzing inputs) and tightly coupled with the stateful communication of both connected devices, including the handling of power role swaps during USBPD operations. As such, our dual-role state machine makes use of functional sequences to transition between states, which facilitates state-aware fuzzing in a black-box mode. Specifically, the dual-role state machine triggers functional sequences to implicitly change states of target PD devices, without requiring any knowledge of the target's internals. Additionally, it handles both power roles in a dual-role PD device using a dedicated role switch functional sequence seamlessly incorporated. It is worth noting that our dual-role state machine enables a fuzzer to test both communication parties without disrupting the ongoing fuzzing by requesting (or handling) runtime role swaps at any time during the fuzzing campaign. This capability cannot be achieved with traditional state machines focusing on a single communication entity (e.g., host side).

Moreover, since a single-role state machine does not inherently support role swaps at runtime, simply using two different (single) role state machines cannot handle runtime role swaps either. Consequently, a fuzzer using two single-role state machines is limited in various message mutations. For instance, it is unable to fuzz the last message (`PS_RDY2`) of the role swap sequence because it rejects the initial role swap request due to the lack of the role swap support as shown in Figure 7 (2). In contrast, our fuzzer is capable of fuzzing the message `PS_RDY2` after exchanging the first three messages, with the support of the role swaps sequence in the dual-role state machine, as illustrated in Figure 7 (1). If the code dealing with `PS_RDY2` is buggy or non-compliant within the tested device, FUZZPD can have an opportunity to uncover such violations, whereas the single-role state machines based fuzzer cannot, as we will further discuss in §7.1.

State machine generation. Our dual-role state machine consists of two essential components: 1) states that are abstracted away from standard PD state machine and trigger functional sequences, and 2) transitions that are essentially functional sequences. Figure 8 summarizes a procedure of our dual-role state machine construction. We first extract all the functional sequences listed in the USBPD spec. For each functional sequence, we record its entry and exit states for both roles (①) and abstract away all intermediate states in between these two states due to their non-controllable nature (see Figure 4). We then split each sequence into per-role sequences to distinguish different roles (②). Lastly, we construct a single state machine by merging the same states for all different functional sequences (③). As a result, its states are abstracted from the standard PD state machine, and its transitions are represented as functional sequences. Note that we seamlessly incorporate state transitions for power role swap because a functional sequence of power role swap (i.e., `seq#4` in Figure 6) is merged into the state machine. The whole state machine can be traversed by triggering functional sequences under the control of our message mutation.

One unique feature of our state machine is that due to its combined nature, the state machine is broadly broken down into two distinct partitions with different roles, and both are bridged by role swap functional sequences (i.e., `seq#4`). Accordingly, the whole state machine (especially its transitions) looks symmetric because each functional sequence usually involves both roles. Furthermore, the availability of functional sequence execution depends on a state. As we will explain in §5.4, to fuzz a message in a particular functional sequence, the current state should be a state where the target functional sequence can be triggered. Taking the example of Figure 6, to fuzz any of message within the sequence `#1`, the current state must be state `B` or state `F` depending on the required power role.

5.3 Message Seeding

To support finer-grained guidance for mutation of FUZZPD, we employ message seed guidance for PD message input generation. As we will explain in §5.4, message seeding is used to guide mutation within a state, which not only helps generate high quality message inputs, but also allows mutated messages to reach the core logic of PD functions without being filtered by message validity checks. Message seeding has two types of information, *a meta data* and *a seed message*. A meta data indicates the information of a functional sequence for its seed message, which contains a sequence ID that the seed message belongs to and an order of the seed message in the sequence. Using the meta data, we can mutate the seed message in consideration of a PD device's state, in a

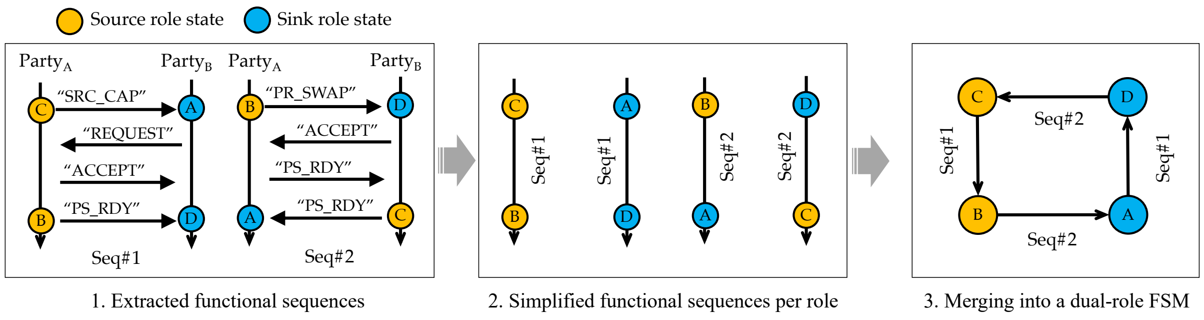


Figure 8: Dual-role state machine construction

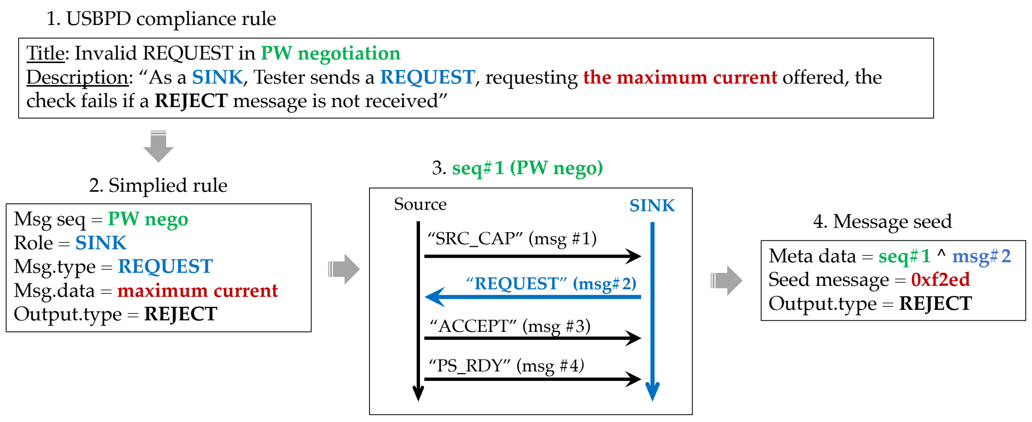


Figure 9: Simplified message seeding extraction from USBPD compliance rule

way that the seed message is sent in the right order in the right sequence. Meanwhile, a seed message presents certain message values. Similar to general coverage guided fuzzing, a seed message is used as a base message to mutate.

Figure 9 simplifies the procedure of our message seed extraction. As mentioned in §5.1, all message seeds originate from USBPD compliance test suite [12]. The test suite enumerates compliance rules, and each compliance rule that corresponds to a certain functional sequence specifies a target message to be tested within a relevant sequence. We inspect the compliance rules one-by-one from the test suite, and then extract and put together a list of message seedings. For each rule, we first extract a relevant functional sequence, which is easily noticeable according to the title of the rule or can be inferred from the contents of the description (1). Then we find out its meta information from its description, such as the role and the type of the target message (2). Given this information along with a list of functional sequences (see §5.2) (3), we extract and record its sequence number as well as message number in the sequence, along with its seed message (4).

5.4 Mutation Strategy

As aforementioned, we leverage a dual-role state machine for USBPD fuzzing by taking advantage of the stateful nature of USBPD functional sequences. With this state machine, we devise a new mutation scheme that enables effective and fine-grained USBPD fuzzing. FuzzPD's mutation scheme has two primary objects: 1) to explore the dual-role state machine for *inter state mutation*, and 2) to provide finer-grained guidance for *intra state mutation*.

Inter-state mutation. This mutation phase aims to explore the dual-role state machine. To trigger state transitions, we execute the corresponding functional sequences assigned for the state transitions in the state machine. In this phase, we employ a general state-aware mutation approach, which explores the state machine to prioritize new state or transition coverage [6, 27]. Once all the states in the state machine are visited, we switch to a random state exploration mode to continue the subsequent intra-state mutation phase.

Intra-state mutation. Apart from across-state exploration, we further enhance our fuzzing with in-state mutation. To generate better message inputs within a state, we take advantage of the message seeding as described in §5.3. Specifically, given a message seed, a fuzzer first triggers the designated functional sequence by exchanging its messages in order until the target seed message is sent. Then, the fuzzer mutates the seed message and delivers it to the device. Similar to general coverage-guided mutation, we leverage typical mutation operations, such as genetic algorithms, at this point.

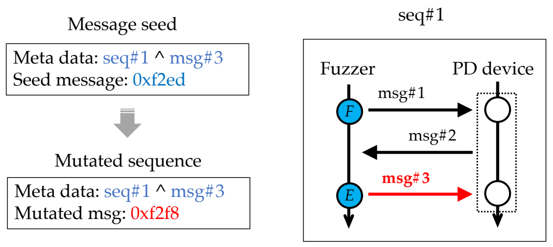


Figure 10: Running example of intra-state mutation

Algorithm 1 Fuzzing Execution.

```
Initial: $Cur\_seq \leftarrow \{\}$ /* initialize current sequence */
 1: while True do
 2:     /* 1. message receive */
 3:     if IsMessageRecv() then
 4:         if $Cur\_seq$ = NULL then /* no current sequence */
 5:             $Cur\_seq \leftarrow$ IntraStateMutate(passive)
 6:         if $Cur\_msg$.type = Msg.type then /* received valid msg */
 7:             update $Cur\_msg$ count
 8:         else/* received invalid msg in the current sequence */
 9:             further analysis for detection
10:     /* 2. message send */
11:     if $Cur\_seq$ = NULL then
12:         if Need inter-state mutate then
13:             $Cur\_seq \leftarrow$ InterStateMutate()
14:         else
15:             $Cur\_seq \leftarrow$ IntraStateMutate(active)
16:     if there are remaining msgs to send in the $Cur\_seq$ then
17:         deliver a next msg in order within $Cur\_seq$
```

Aside from the mutation strategy with message seeding, we also support message generation for in-state mutations without the assistance of seed messages. We use several different strategies. For example, we randomly choose one out of functional sequences that can be triggered from the current state and then mutate an arbitrary message in the sequence. We also perform out-of-order message mutation in a functional sequence, switching messages or randomizing the order of message exchanges. Once a new mutated sequence is ready, this sequence can be exchanged like seed based mutation above.

Figure 10 steps through FuzzPD's mutation when using message seeding. In the example, we try to mutate a seed message in `seq#1`. Suppose the current state is `F` after inter-state mutation, where `seq#1` can be triggered. To perform intra-state mutation subsequently, we first mutate the seed message (`0xf2ed`) and then obtain a mutated message value (`0xf2f8`). Then we initiate this mutated sequence `seq#1` by sending its first message `msg#1` to the device. Since then, we expect to receive a message `msg#2` from the device according to `seq#1`. Once receiving that message, we transfer the newly mutated message at this point, i.e., in the third order (i.e., `msg#3`) of the sequence.

5.5 Fuzzing

Given the dual-role state machine and message seeds, FuzzPD exercises multi-staged mutations during the entire fuzzing execution. In this section, we elaborate on the fuzzing execution workflow of FuzzPD by taking individual steps from the beginning.

Preparation. As we highlight in §5.1, the essence of FuzzPD is that FuzzPD examines and fuzzes real USBPD devices. In this sense, all the requirements to carry out our fuzzing is physical access and connection to each test PD device. After connecting to our fuzzer machine, the entire fuzzing process begins with a pre-processing phase, which follows the steps outlined below.

Pre-processing. The availability of USBPD functions varies by PD device. For example, some USBPD power adaptors offer more advanced power supply capabilities, through Programmable Power Supply (PPS)[1], which is typically unavailable in smartphones. In this regard, it makes more sense to extensively examine PPS functionality for USB chargers, rather than smartphones. To enhance the mutation efficiency, we prioritize functional sequences that are supported by tested PD devices, and grant them higher priority to perform more extensive mutation for these sequences. We achieve this by identifying all PD functionalities (i.e., functional sequences) available to a target device shortly after a new connection is made. Specifically, we send a series of certain functional sequences that retrieve the target's capabilities, such as `GET_STATUS` sequence. The collected functional sequences would be used for future message mutation with higher priority.

Fuzzing execution. Algorithm 1 details FuzzPD's fuzzing execution. At a high level, the fuzzing execution works through two consecutive phases in a loop, message receiving (line 3) and message sending (line 11). Note that the current functional sequence (i.e., *Cur_seq*) can have either a normal functional sequence or a mutated functional sequence (containing a single mutated message), depending on different mutation strategy. First, we try to select a new functional sequence to exercise in each phase if no sequence is currently being tracked (line 4 and 11). In the message sending phase, at this point, we carry out either inter- or intra-state mutation depending on the mutation policy or probability. When performing an inter-state mutation, a normal functional sequence is chosen for the current sequence (line 13), which will trigger a state transition. Otherwise, we execute an intra-state mutation, and a new sequence with a mutated message is assigned into the current sequence (line 15). Meanwhile, the message receiving phase is responsible for mutating passive functional sequences that are triggered exclusively by a PD device, not by a fuzzing host (see Figure 2). Due to such non-triggerable nature, we grant this passive sequence mutation a priority by executing it (line 5) ahead of active sequence mutation (line 15) in a loop, to increase the opportunity of taking passive sequence mutation. Once a new functional sequence is assigned and gets started to exercise, we send corresponding messages in an orderly manner as specified in the current functional sequence.

Bug detection. Unlike the greybox approach, sanitizing techniques, such as ASAN [28] and MSAN [29], are usually inapplicable for black-box fuzzing. Instead, we make use of the following ways for bug detection. First, we ensure all message exchanges are compliant with standard functional sequence according to the USBPD spec, when performing normal functional sequences, such as state transitions. We then carry out further analysis to detect bugs if any discrepancy is observed. Also, we check any output violation when exercising regular compliance testing during fuzzing. For a mutated message sent to a target, we conduct an investigation into every response message to check for its validity. For example, as detailed in §7.1, we ensure that a target device rejects request messages that have been intentionally mutated to induce over-voltage.

6 Implementation

The prototype of FuzzPD is built upon Chromium OS Embedded Controller (EC) software [7]. We deploy FuzzPD's core logics, such as the dual-role state machine and the fuzzing engine, on EC's USBPD subsystem module. Then, we compile the extended EC implementation and flash its binary into the EC ROM to reflect the change to a fuzzing host, Chromebook Spin 713 [8], which we used throughout all our experiments (§7). We basically piggyback on underlying resources and innate capabilities of the Chromebook, such as power supply capability, for fuzzer's base capability. We additionally simulate further capabilities during message exchanges to draw a variety of partner's responses and their PD functionality. FuzzPD is composed of 2,400 lines of C code in total: 1800 lines for the dual-role state machine and fuzzing engine, and 600 lines for message seeding, etc.

7 Evaluation

In this section, we evaluate different aspects of FuzzPD. First, we show how USBPD can be used to find new bugs, and we provide details of some of the found bugs in dedicated case studies (§7.1). Next, we examine the effectiveness of FuzzPD when using different methods for coverage tracking (§7.3). We compare the performance of FuzzPD and the USBPD compliance test (§7.4).

Experimental setup. To evaluate FuzzPD's prototype, we use a Chromebook Spin 713 [8], which is equipped with an Intel Core i5-10210U 1.60GHz and 8 GB RAM. This system runs Chrome OS and it has two USB-C ports supporting USBPD 3.0 technology. We perform all our evaluations via a connection to each PD device under test, using a dedicated USBPD cable supporting fast power charging, as well as advanced data communication, e.g., Thunderbolt4. To evaluate FuzzPD, we used 24 USBPD devices from 18 vendors, ranging from smart devices and USB chargers to USB docking stations, as summarized in Table 2. Note that we also used the Chromebook Spin 713 as a tested device. In this case, we connected its two USB-C ports with each other, and used one port as the fuzzing host and the other as the PD device under test.

7.1 Findings

We thoroughly inspected the USBPD specification and test suites, extracting functional sequences as well as message seeds. Out of the functional sequences and message seeds available in the USBPD standard, we excluded the ones that are not suitable for our purpose, such as sequences (or seeding) for USB-C cables or unsupported by any of tested PD devices. As a result, we obtained 40 standard functional sequences and 45 message seeds.

As mentioned in §5.5, USBPD devices do not usually support all 40 functional sequences. They implement a limited set of functional sequences that are needed to support their functionalities. To retrieve valid functional sequences for the tested devices, we send a series of messages (e.g., `GET_STATUS`) in the pre-processing step of fuzzing (see §5.5), which takes approximately 5 seconds for each device. Table 2 summarizes their key capabilities. Note that the seventh and eighth columns denote the number of functional sequences available in each device and the total number of messages in individual sequences, respectively. As mentioned in Figure 2, different functional sequences represent different PD functions. For this reason, the devices that support USBPD revision 3.0 have more functions than the ones supporting revision 2.0. Additionally, smart devices (`S.X`) tend to provide richer PD functions to be examined, in comparison with the others (`H.X` and `A.X`). We observe that all the tested PD devices except USB chargers are capable of dual power roles.

We conducted extensive fuzzing campaigns using the experimental setup explained above. As listed in Table 3, we have discovered 15 unique bugs, reported all the bugs to the corresponding vendors, and got 9 bugs confirmed. Table 5 presents the bug distribution for the tested devices by each bug type (ID). The bugs ranged from over-voltage issues to out-of-bounds memory access within the USBPD firmware. Overall, the high-end devices equipped with richer PD features, such as smart devices, tend to be more problematic. However, Surface Pro 8 (`S.5`) turned out to be resilient to all the mutated inputs and has shown much stronger security in its USBPD system compared to other devices, despite its vast range of supported PD functionality. Another observation is that most of the bugs were discovered while fuzzing the sequence of the power contract. This is because all the USBPD devices have at least this functionality for their rapid charging.

ID	Vendor	Device	Spec					Message Coverage			
			PD Rev.	D-Role	VDMs	# Seq	# Msg	PDf$_M$	PDf$_S$	PDf$_K$	FuzzPD
S.1	Samsung	Galaxy S21,S23Ultra	3	✓	✓	34	71	0	34	37	71
S.2	Samsung	Galaxy A13	3	✓		34	71	0	34	37	71
S.3	Apple	MacBook Pro 13	2	✓	✓	40	72	0	40	32	72
S.4	Apple	iPad Pro	3	✓	✓	40	72	0	40	32	72
S.5	Microsoft	Surface Pro 8	3	✓		36	64	0	36	28	64
S.6	Xiaomi	Redmi Note 11 Pro	3	✓		34	71	0	34	37	71
S.7	Xiaomi	Mi 11 Lite 5G	3	✓	✓	30	63	0	30	33	63
S.8	Google	Pixel 5a	3	✓		21	46	0	21	25	46
S.9	Google	Pixelbook Go	2	✓	✓	19	43	0	19	24	43
S.10	Acer	Chromebook Spin 713	3	✓	✓	33	61	0	31	30	61
S.11	Nintendo	Switch	2	✓		14	36	0	14	22	36
H.1	QGeeM		3	✓		21	45	0	21	24	45
H.2	IPTIME		3	✓	✓	23	53	0	23	30	53
H.3	YCCTEAM		2	✓	✓	13	45	0	13	32	45
H.4	Anker	USB-C hub	3	✓		21	45	0	21	24	45
H.5	TrendNet		2	✓		13	45	0	13	32	45
H.6	UtechSmart		3	✓	✓	21	45	0	21	24	45
H.7	D-link		3	✓		13	45	0	13	32	45
A.1	AOXEY		3	src-only		12	13	0	–	13	13
A.2	RavPower		3	src-only		10	10	0	–	10	10
A.3	WeWatch	USB-C charger	3	src-only		12	13	0	–	13	13
A.4	syncwire		3	src-only		6	6	0	–	6	6
A.5	Blechmeki		3	src-only		12	13	0	–	13	13

Table 2: Tested devices with corresponding used PD specification.

FuzzPD found a non-compliant violation in the power role swap sequences (Bug ID 15), which remains undisclosed by a fuzzer using two single-role state machines because of unsupportive role swap sequence, as discussed in §5.2. Note that we discovered a total of 8 messages within 3 different USBPD functional sequences, including power role swaps, which can be exclusively fuzzed when employing our dual-role state machine.

7.2 Dual-role State Machine Construction

As described in §5.2, we take several steps to construct a dual-role state machine from USBPD spec. In this section, we quantify each step of the state machine generation. First, we extract all the USBPD functional sequences and their corresponding messages. This is a straightforward process that takes 2 hours as USBPD spec provides a well-described list of the functional sequences in the spec document. Next, we associate each functional sequence with start and end states, which is a more time-consuming task. We use two different approaches for this step, depending on the functional sequence. If the description of functional sequences includes the relevant states, e.g., power contract functional sequence, we extract them from this description directly. Otherwise, we manually identify the start and end states from a standard state machine based on functional sequence messages, which takes 5 hours. After extracting the functional sequence with their states, we split and merge them to create a complete dual-role state machine, which takes approximately 2 hours. In total, the entire process of building a complete dual-role state machine takes around 9 hours.

7.3 Coverage

In this section, we examine fuzzing coverage achieved by FuzzPD in different configurations. We take a measurement in two different ways to complement limited black-box coverage measurement. To better evaluate the different features of FuzzPD, we implemented three fuzzing configurations for baselines, shown in Table 4, in which we selectively disable the key features of FuzzPD. Specifically, PDf$_M$ is a basic form of USBPD fuzzer that is aware of USBPD message syntax, but it is state-agnostic. On the other hand, PDf$_S$ and PDf$_K$ are advanced fuzzers equipped with a state machine (with different role) along with functional sequence-based mutations, but limited to a single power role, source for PDf$_S$ and sink for PDf$_K$.

7.3.1 Message Coverage

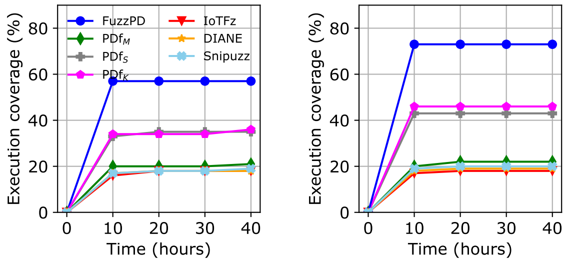
We try to measure the depth that our fuzzer, in its different configurations, is able to reach in the generated PD message sequences. To achieve this aim, we define a new coverage criteria for this purpose, which we call *message coverage*. This coverage represents the number of *covered* messages within each functional sequence. In this context, a covered message is a message that has been sent by the fuzzer after having received a message from the tested device, indicating

ID	Type	Description	Buggy seq	Msg type	Detected by	Confirmed
1	DoS	No response, repeat re-init	PW contract	PS_RDY	FuzzPD	
2	Out-of-bounds	invalid object position	PW contract	REQUEST	FuzzPD/ Test suite	✓
3	Non-compliance	Not response with soft_reset	PW contract	ACCPET	FuzzPD	
4	Non-compliance	Max and min voltage inversion	PW contract	SRC_CAP	FuzzPD	✓
5	DoS	Over-voltage	PW contract	SRC_CAP	FuzzPD	✓
6	Non-compliance	Max and op current inversion	PW contract	REQUEST	FuzzPD	
7	Non-compliance	Rev. violation	get status	GET_STATUS	FuzzPD/ Test suite	✓
8	Non-compliance	Rev. violation	get battery cap	GET_BAT_CAP	FuzzPD/ Test suite	✓
9	Non-compliance	Rev. violation	get battery status	GET_BAT_STATUS	FuzzPD/ Test suite	✓
10	Non-compliance	Rev. violation	get manifacturer info	GET_MANI_INFO	FuzzPD/ Test suite	✓
11	Non-compliance	Rev. violation	alert	ALERT	FuzzPD/ Test suite	✓
12	Non-compliance	Not response with soft_reset	PW contract	REQUEST	FuzzPD	
13	Non-compliance	Rev. violation	get status	STATUS	FuzzPD/ Test suite	✓
14	Non-compliance	Rev. violation	get src cap extension	GET_SRC_CAP_EXT	FuzzPD/ Test suite	
15	Non-compliance	Not response with soft_reset	PW role swap	PS_RDY	FuzzPD	

Table 3: 15 unique bugs discovered by FuzzPD. Table 5 (in Appendix) presents the details about the affected devices.

Baseline Fuzzer	Message Syntax	State Machine
PDf$_M$	✓	–
PDf$_S$	✓	Single (source)
PDf$_K$	✓	Single (sink)
FuzzPD	✓	Dual

Table 4: Properties of the considered baseline fuzzers.



(a) Pixelbook Go **(b)** Spin 713

Figure 11: Execution coverage for USBPD on two Chromebook laptops, Google Pixelbook Go, and Acer Spin713..

that the tested device reached an appropriate internal state. For instance, in the power contract sequence, a fuzzer is able to cover the second message (i.e., `REQUEST`) in its power sink role after receiving the previous message (i.e., `SRC_CAP`) from the target device. In fact, receiving the `SRC_CAP` message is an indication that the target device reached an internal state able to process the `REQUEST` message sent afterwards. Note that the designed message coverage metric is available for all the tested devices, while more traditional code coverage is not always available, given the closed source nature of most USBPD firmware.

We examine message coverage by issuing all available functional sequences on each tested PD device. Without carrying out mutations, we try to complete each available functional sequence by sending valid (right) messages. Then, we count the number of messages each fuzzer can trigger in the correct position within the sequence. We repeat all these steps for all the fuzzing configurations.

Table 2 summarizes, in the last four columns, the results of the coverage measurement. The results present the number of messages covered by each fuzzing technique introduced in Table 4. As shown, PDf$_M$ yielded the least coverage — only the first message in each functional sequence is covered. This happens because it does not consider message exchanges driven by functional sequences due to its sequence- and state-agnostic nature. On the other hand, PDf$_S$ and PDf$_K$, achieved better coverage results since they are sequence-aware. In fact, they exercise sequence-based communication and message mutations. Meanwhile, FuzzPD outperformed the others in most of the tested devices, except for USB chargers, in which we cannot benefit from our dual-role feature, since these devices do not support it. Instead, FuzzPD can overcome limited scope of power role and act as both power roles, maximizing coverage of both roles using our dual-role state machine.

7.3.2 Execution Coverage

Beside message coverage (as discussed in §7.3.1), we measure coverage improvements in terms of execution paths. We utilize two Chromebook machines, Spin 713 [8] and Pixelbook Go [30], for this experiment because their USBPD firmware source code (i.e., ChromiumOS EC code as explained in §6) is available, allowing us to measure their execution coverage [7]. To better highlight FuzzPD's superiority in path exploration, we compare it to state-of-the-art fuzzers. Since there are no USBPD fuzzers available in the literature, we employ three black-box firmware fuzzers [9–11] that are closer to ours in design concept. We simulate their features and deploy them on our fuzzing machine to conduct black-box USBPD fuzzing because their techniques cannot be directly applied to USBPD systems.

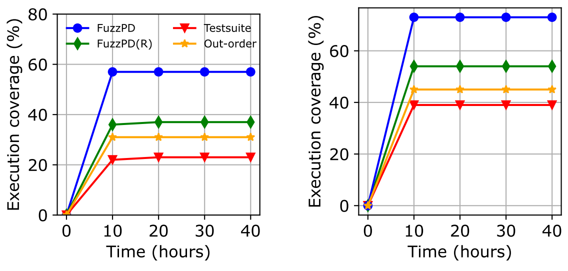
To track the execution coverage within the Chromebooks, we instrumented the EC code to collect coverage for USBPD functions and then reflashed each tested Chromebook machine. Note that fine-grained tracking methods, such as basic block (or edge) coverage, is not suitable in a real PD machine, because its performance overhead causes some PD functionality to fail due to time constraint violations (e.g., send message timeouts). For this reason, we settle on coarse-grained statement coverage tracking.

The result is presented in Figure 11. We measure average coverage after running each fuzzer for 40 hours three times. We use seed messages retrieved from the USBPD testsuite, as explained in §5.4. Since the Pixelbook Go supports less PD functionality due to its lower PD revision (2.0), the fuzzers tend to show less coverage than in the Spin713 laptop. Moreover, the three black-box fuzzers do not show meaningful coverage increase for USBPD, and their coverage is close to that of our message-aware fuzzer configuration (PDf_M). This is because these fuzzers focus mainly on identifying message formats for diverse IoT devices with fairly simple functions, such as light on/off. Thus, the approaches of their fuzzers are not suited for stateful and dual-role USBPD fuzzing. This limitation mainly blocks these fuzzers from increasing coverage beyond that of message aware fuzzers, whereas FUZZPD tackles more PD functions in different roles. Although this is a limited study with a small set of USBPD devices, we believe this can highlight and clearly show how FUZZPD's mutation with the dual-role state machine contributes to the overall coverage growth of the targets.

Our coverage measurement is based on the Chromium EC code, which is designed to implement all of the USBPD functionalities described in the USBPD specification. However, since the tested Chromebook machines support only a subset of these functionalities (as represented in the #Seq column for S9/S10 in Table 2), it is not possible to achieve full code coverage of the Chromium EC code, although FUZZPD was able to explore every supported functionality of the Chromebooks. The missing coverage in our testing came from the USBPD functionalities that are not supported by the target Chromebooks, such as USB authentication. We confirmed this finding manually by comparing the traced coverage with dedicated per-functionality coverage.

7.4 Comparison with Compliance Test Suite

As discussed in §5.4, the FUZZPD's mutation is based on seed messages from the USBPD compliance test suite. In this section, we further evaluate FUZZPD's effectiveness by comparing experimental results with the compliance test suite. First, we investigate our findings and distinguish between the bugs found by our mutation and the ones by testing the compliance rules. The last column of Table 3 indicates the source of each bug discovery. Our analysis shows that 7 bugs were discovered through our fuzzing approach, which could not be found through the compliance rule testing alone.



(a) Pixelbook Go (b) Spin 713

Figure 12: Execution coverage of FUZZPD, compared with compliance test suite (**Testsuite**), random intra-state mutation (**FuzzPD(R)**), and out-of-order mutation (**Out-order**).

Beside bug discovery, we additionally compare the two approaches in terms of execution coverage. Based on the two Chromebook laptops, we conduct a series of tests for the extracted USBPD compliance test suite by issuing message sequences relevant to each test case, and accumulate execution coverage data, similar to the coverage tracking in §7.3.2, until all the tests are complete. To highlight the usefulness of message seeding, we design two additional baselines with different mutation strategies. One employs a random inter-mutation strategy and the second uses an out-of-order mutation strategy without message seeding guidance, while still guided by the dual-role state machine.

Figure 12 exhibits the result of the experiment. In comparison with the compliance test suite, FUZZPD achieved nearly 2x higher execution coverage on both Chromebooks. This is because all their test cases in the suite are not designed to maximize coverage, but to test limited USBPD functionalities. Moreover, our message seed-based mutation showed higher coverage than that of random intra-state mutation as random message generation failed to reach core functions of the USBPD module due to message sanitization within the Chromebook machine. The out-of-order mutations explored even lower coverage than random mutations because in most cases, it cannot reach deeper code blocks due to the violation of earlier time message type checks.

These experiments demonstrate that solely testing compliance rules in the test suite presents limited advantages, in terms of bug discovery and code coverage. In contrast, FUZZPD is more effective, yielding significantly superior results in the same aspects.

8 Discussion

Memory corruption attacks on USBPD. Memory-based attacks (e.g., control flow hijacking) have posed a significant threat to various types of firmware [31]. These firmware memory attacks could be more problematic because of their high privilege level and lack of mitigations (and protections) against exploitations, unlike general OS kernels. Similarly, USBPD firmware has been found to be vulnerable to memory based attacks due to incorrect implementation. As described in §7.1, one potential attack scenario on USBPD firmware is that attackers can exploit array index access violation, potentially leading to illegal memory reads or code execution [32]. Recent studies have demonstrated the feasibility of exploiting memory corruption in USBPD firmware of commercial USB-C chargers, which actually causes unintended firmware upgrading [33]. Thus, it is crucial to protect USBPD firmware against various memory attacks, like other firmware targeted attacks.

Automated state machine generation. The building procedure of a dual-role state machine is straightforward and one time task, but it could be tedious, requiring manual efforts currently although it does not have the big impact on the runtime efficiency and overall performance of FUZZPD. To complement and automate this phase, we could take advantage of existing solutions, such as recent state machine construction scheme leveraging natural language processing (NLP) [34].

USBPD functions untested. As discussed in §2, USBPD protocol provides a variety of functionalities, not only power supply related tasks, but also data communication. Besides, more PD functionality has become available with its new revisions, such as USB authentication. However, in reality, USBPD devices have limited USBPD functionality depending on their usage. For example, one of the latest PD features, Extended Power Range (EPR) [1], supporting the high volume of power up to 240W, is not even available by any of consumer devices in production at the time of the experiment. Thus, we were unable to examine all described PD functions in USBPD spec, and leave this for our future work.

USB-C hardware defects. There have been recent advanced attacks compromising errors and flaws within USB-C implementation. In their attacks, an attacker aims to launch side-channel attacks and leak sensitive data [35] using a rogue USB-C cable that secretly embeds a network chip. Others attempt to directly damage USB devices by overcharging them with hardware defected (or crafted) USB-C products [2]. Although their impact is non-trivial and severe enough to be addressed, such errors from hardware defects are orthogonal to USBPD buggy firmware, and it is outside the scope of our paper.

9 Related Work

USB security. Universal Serial Bus (USB) has large attack surface and complex codebase to support various functionality, and it has been an attractive target for many years. There has been a large volume of efforts to achieve secure USB subsystem. Specifically, recent fuzzing techniques have made significant contributions to finding massive USB bugs [4–6, 16–18, 36–39]. Against malicious USB devices, many of them try to find buggy code within USB kernel stacks [4, 5, 37, 39], and some of them focused on detecting malware within devices [38]. Despite massive discovery of security bugs from different aspects, none of existing USB fuzzing techniques can apply to USBPD firmware bug discovery because USBPD firmware implements entirely different functionality on its dedicated protocol stack. Another line of works apart from the efforts to find vulnerability, aims at protecting USB stacks from various runtime USB attacks [40–45]. They mostly develop a USB filtering system to block non-permissive or invalid packets from accessing target USB stacks [40, 41, 43]. Often, existing work tries to prevent data exfiltration, providing a tracking mechanism for USB data provenance with the help of hardware-assisted attestation [44]. Similary, none of them are designed for USBPD protocol, and unable to tackle the challenges regarding USBPD implementation.

Firmware fuzzing. As IoT ecosystem is fast growing, there is increasing need for firmware security. Security researchers have recently focused their efforts on building better firmware fuzzing techniques in this regard [9–11, 19–21]. A common practice in firmware fuzzing is to carry out its execution in an emulated environment [19–21]. They aim at building scalable firmware fuzzing environment by emulating device hardware to make firmware run on the emulation in a single test platform. Unfortunately, it is still challenging to acquire firmware images and implement emulation in diverse firmware platforms. Meanwhile, black-box fuzzing is a promising alternative, which is particularly used for IoT landscape [9–11]. The black-box approach overcomes the fundamental challenges of emulation based fuzzing without the need for firmware analysis and emulation efforts. Since the internals of devices are agnostic in a black-box mode, they mainly focus on retrieving unknown input syntax by taking advantage of domain knowledge, such as companion apps analysis [9, 10] or output similarity [11]. In comparison, FUZZPD achieves black-box USBPD fuzzing in a stateful manner with given message syntax, which encompasses dual-role state machines along with multi-layer mutation strategy.

10 Conclusion

Despite the wide adoption of USBPD, its implementation flaws have led to serious problems, not only safety issues, but also security breaches. In this paper, we present FUZZPD, the first black-box fuzzing technique for USBPD bug finding. FUZZPD fuzzes real USBPD targets with a simple physical USB-C connection. FUZZPD maximizes its fuzzing performance with a dual-role state machine enabling both state coverage and transitions from fuzzing inputs, along with a multi-level mutation strategy, implementing a fine-grained state-aware fuzzing with intra- and inter-state mutations. We evaluate FuzzPD on various USBPD devices, based on our implementation on a Chromebook, and found 15 new bugs. We demonstrate that FuzzPD achieves 40% to 3x higher execution coverage than state-of-the-art black-box firmware fuzzers. Compared with the USBPD compliance test suite, FuzzPD found 7 more bugs with 2x higher coverage growth.

Acknowledgments

We thank the anonymous reviewers for the valuable comments and suggestions. This work was supported in part by ONR under grants N00014-20-1-2128 and N00014-20-1-2671, and NSF under grants CNS-2145744 and CNS-2055123, and AFOSR under grant FA8650-19-1-1741. This material is also based on research sponsored by Wistron. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of our sponsors.

References

[1] "Usb pd specifications," USB Implementers Forum https://www.usb.org/document-library/usb-power-delivery, 2021.

[2] "Nintendo switch bricking," https://switchchargers.com/nintendo-switch-bricking-faq/, 2019.

[3] "Cve-2019-6176," https://cve.mitre.org/cgi-bin/cvename.cgi?name=2019-6176, 2019.

[4] D. Vyukov, "Syzkaller," https://github.com/google/syzkaller, 2015.

[5] H. Peng and M. Payer, "Usbfuzz: A framework for fuzzing usb drivers by device emulation," in *25th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 397–414.

[6] K. Kim, T. Kim, E. Warraich, B. Lee, K. R. Butler, A. Bianchi, and D. J. Tian, "Fuzzusb: Hybrid stateful fuzzing of usb gadget stacks," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022.

[7] Google, "Chromium os embedded controller," https://chromium.googlesource.com/chromiumos/platform/ec/+/HEAD/README.md.

[8] Acer, "Acer chromebook spin 713," https://www.acer.com/ac/en/GB/content/model/NX.HWNEK.001.

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

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

[11] X. Feng, R. Sun, X. Zhu, M. Xue, S. Wen, D. Liu, S. Nepal, and Y. 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*, 2021, pp. 337–350.

[12] "Usb power delivery compliance test specification," USB Implementers Forum https://www.usb.org/document-library/usb-power-delivery-compliance-test-specification-0, 2021.

[13] Https://github.com/purseclab/fuzzpd.

[14] "Usb type-c® cable and connector specification," USB Implementers Forum https://www.usb.org/usb-type-cr-cable-and-connector-specification, 2019.

[15] D. Kierznowski, "Badusb 2.0: Usb man in the middle attacks," *Retrieved from RoyalHolloway*, 2016.

[16] T. Goodspeed and S. Bratus, "Facedancer usb: Exploiting the magic school bus," in *Proceedings of the REcon 2012 Conference*, 2012.

[17] NCCGroup, "Umap2," https://github.com/nccgroup/umap2.

[18] S. Schumilo, R. Spenneberg, and H. Schwartke, "Don't trust your usb! how to find bugs in usb device drivers," *Blackhat Europe*, 2014.

[19] Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu, and L. Sun, "Firm-afl: High-throughput greybox fuzzing of iot firmware via augmented process emulation," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1099–1114.

[20] G. Hernandez, M. Muench, D. Maier, A. Milburn, S. Park, T. Scharnowski, T. Tucker, P. Traynor, and K. R. B. Butler, "FirmWire: Transparent Dynamic Analysis for Cellular Baseband Firmware," in *Symposium on Network and Distributed System Security (NDSS)* , 2022.

[21] M. Kim, D. Kim, E. Kim, S. Kim, Y. Jang, and Y. Kim, "Firmae: Towards large-scale emulation of iot firmware for dynamic analysis," in *Annual Computer Security Applications Conference*, 2020, pp. 733–745.

[22] E. Johnson, M. Bland, Y. Zhu, J. Mason, S. Checkoway, S. Savage, and K. Levchenko, "Jetset: Targeted firmware rehosting for embedded systems," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 321–338.

[23] J. De Ruiter and E. Poll, "Protocol state fuzzing of tls implementations," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 193–206.

[24] R. Ma, T. Zhu, C. Hu, C. Shan, and X. Zhao, "Sulleyex: A fuzzer for stateful network protocol," in *Proceedings of the International Conference on Network and System Security (NSS)*. Springer, 2017, pp. 359–372.

[25] E. B. Yi, H. Zhang, K. Xu, A. Maji, and S. Bagchi, "Vulcan: Lessons in reliability of wear os ecosystem through state-aware fuzzing," in *Proceedings of the 18th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2020.

[26] H. Gascon, C. Wressnegger, F. Yamaguchi, D. Arp, and K. Rieck, "Pulsar: Stateful black-box fuzzing of proprietary network protocols," in *Proceedings of the International Conference on Security and Privacy in Communication Systems (SecureComm)*. Springer, 2015, pp. 330–347.

[27] Y.-H. Zou, J.-J. Bai, J. Zhou, J. Tan, C. Qin, and S.-M. Hu, "{TCP-Fuzz}: Detecting memory and semantic bugs in {TCP} stacks with fuzzing," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021, pp. 489–502.

[28] "Kernel address sanitizer," https://github.com/google/kasan/wiki, 2018.

[29] "Kernel memory sanitizer," https://github.com/google/kmsan, 2018.

[30] Google, "Pixelbook go," https://store.google.com/us/product/pixelbook_go?hl=en-US.

[31] U. F. Vulnerabilities, https://thehackernews.com/2022/07/new-uefi-firmware-vulnerabilities.html/, 2022.

[32] "Cwe-129: Improper validation of array index." https://cwe.mitre.org/data/definitions/129.html.

[33] ZDNet, "Badpower attack corrupts fast chargers to melt or set your device on fire," https://www.zdnet.com/article/badpower-attack-corrupts-fast-chargers-to-melt-or-set\protect\discretionary{\char\hyphenchar\font}{}{}your-device-on-fire.

[34] M. L. Pacheco, M. von Hippel, B. Weintraub, D. Goldwasser, and C. Nita-Rotaru, "Automated attack synthesis by extracting finite state machines from protocol specification documents," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022.

[35] I. B. Times, "Usb-c to lightning cable implanted with password stealing chip puts users at risk," https://www.ibtimes.com/usb-c-lightning-cable-implanted-password-stealing-chip\protect\discretionary{\char\hyphenchar\font}{}{}puts-users-risk-3287508.

[36] J. Patrick-Evans, L. Cavallaro, and J. Kinder, "Potus: Probing off-the-shelf usb drivers with symbolic fault injection," in *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, 2017.

[37] D. Song, F. Hetzelt, J. Kim, B. B. Kang, J.-P. Seifert, and M. Franz, "Agamotto: Accelerating kernel driver fuzzing with lightweight virtual machine checkpoints," in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020, pp. 2541–2557.

[38] G. Hernandez, F. Fowze, D. Tian, T. Yavuz, and K. R. Butler, "Firmusb: Vetting usb device firmware using domain informed symbolic execution," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2245–2262.

[39] D. Song, F. Hetzelt, D. Das, C. Spensky, Y. Na, S. Volckaert, G. Vigna, C. Kruegel, J.-P. Seifert, and M. Franz, "Periscope: An effective probing and fuzzing framework for the hardware-os boundary," in *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2019.

[40] D. J. Tian, N. Scaife, A. Bates, K. Butler, and P. Traynor, "Making usb great again with usbfilter," in *25th USENIX Security Symposium (USENIX Security 16)*, 2016, pp. 415–430.

[41] D. J. Tian, A. Bates, and K. Butler, "Defending against malicious usb firmware with goodusb," in *Proceedings of the 31st Annual Computer Security Applications Conference*, 2015, pp. 261–270.

[42] S. Angel, R. S. Wahby, M. Howald, J. B. Leners, M. Spilo, Z. Sun, A. J. Blumberg, and M. Walfish, "Defending against malicious peripherals with cinch," in *25th USENIX Security Symposium (USENIX Security 16)*, 2016, pp. 397–414.

[43] D. J. Tian, G. Hernandez, J. I. Choi, V. Frost, P. C. Johnson, and K. R. Butler, "Lbm: a security framework for peripherals within the linux kernel," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 967–984.

[44] D. Tian, A. Bates, K. R. Butler, and R. Rangaswami, "Provusb: Block-level provenance-based data protection for usb storage devices," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 242–253.

[45] K. Zhong, Z. Jiang, K. Ma, and S. Angel, "A file system for safely interacting with untrusted usb flash drives."

[46] G. Alendal, S. Axelsson, and G. O. Dyrkolbotn, "Exploiting vendor-defined messages in the usb power delivery protocol," in *Advances in Digital Forensics XV: 15th IFIP WG 11.9 International Conference, Orlando, FL, USA, January 28–29, 2019, Revised Selected Papers 15*. Springer, 2019, pp. 101–118.

A Case Studies

Over-voltage. As an initial step of the power contract, the power source provides a few available power options to the power sink. The sink then chooses the best available option, considering its acceptable power range. For safety reasons, the first voltage option must always be 5V, which is considered as a safe voltage for any connected device. When mutating and increasing the voltage provided as the first option, however, we observed 15 devices (out of 24 tested PD devices) that accept the first provided voltage, regardless of its value. We believe this issue is caused by the incorrect assumption that the first provided option is always 5V. This flaw is a violation of the PD specification, and it can cause over-charging and irreversibly damage to any PD device physically designed to only accept a low voltage.

Out-of-bounds. This bug occurs especially when fuzzing S.1 and H.3. As mentioned in the case above, in the normal power contract, the power receiver picks one of the voltage options provided by the connected power provider. For example, when a power source provides 4 available voltage options, the power sink should respond with an index ranging 1 to 4, for a valid response. Otherwise, the source should reject that power request. Since those devices (S.1 and H.3) provide only one voltage option to deliver, it should not accept any index different from 1. However, when mutating the response message to use the value 2 as voltage index, they accepted the request. We believe this behavior is caused by an implementation error, causing out-of-bounds memory accesses, and it is potentially exploitable.

B Vendor-defined Communication

As briefly mentioned in §2, USBPD offers vendor-specific communication capabilities via vendor-defined messages (VDMs). The messages of VDMs typically have private syntax and they are used for multiple purposes, such as analytics, firmware update, even malicious intent [46]. To show the effectiveness of FUZZPD, we try to analyze private message communications of VDMs. Since we lack the knowledge of their message formats and communications, we utilize random input generation for VDM payloads in the experiment. Specifically, our approach is to mutate each VDM payload and deliver this mutated input, along with a vendor ID, to the target device. We then analyze the responses.

As summarized in Table 2 (**VDMs** column), 9 devices responded to the VDM fuzzing in the experiment. We consider a response as valid if we receive VDM message(s) from the targets, rather than rejecting or ignoring our VDM request message. We observed some devices showed certain patterns in their message communication. For example, some smart devices, such as Samsung's, responded with the same output pattern (e.g., 0x21) when receiving a message with a certain number (e.g., 5) as its last digit. To learn the detailed semantics of each VDM communication, we further tried to extract and examine USBPD firmware that implements such vendor-specific messages exchanges.

We first learned the usage of the two Chromebooks' (Spin 713 and Pixelbook Go) VDMs by analyzing ChromiumOS EC's firmware code (55a0acd279 commit). It turns out that their VDMs are used for debug purposes, such as log message transfer, which is a common usage for VDMs. For other 5 devices that support VDMs, we attempted to directly extract their firmware from the devices since their USBPD firmware is unavailable in public. To retrieve their firmware images, we tried to utilize debug pins on a circuit board after disassembling the devices. Unfortunately, we were unable to acquire firmware as presented in Table 6. For smart devices (S.1 and S.7), it was almost impossible to find out detailed information of firmware (and even the PD controller) due to their custom circuit board implementations of USBPD without known debug interfaces. In other cases (H.2, H.3, and H.6), we could not find a debug interface due to lack of its datasheet, or only listen to noise without receiving valid ACKs from the target board.

In summary, while FUZZPD provides an infrastructure for testing VDMs, our limited study did not yield a complete understanding of the meaningful interpretation for the tested devices' VDM communications, To gain deeper understanding of VDM communications, we may need to perform more systematic and scalable studies, using various test devices with appropriate approaches. We leave this for our future work.

Bug ID	Device ID																						
	S.1	S.2	S.3	S.4	S.5	S.6	S.7	S.8	S.9	S.10	S.11	H.1	H.2	H.3	H.4	H.5	H.6	H.7	A.1	A.2	A.3	A.4	A.5	Total
1							1																	1
2	1													1										2
3														1										1
4			1	1					1	1	1													5
5	1	1	1	1		1	1		1	1	1	1		1	1	1	1	1						15
6																							1	1
7	1																							1
8	1																							1
9	1						1																	2
10	1																							1
11	1							1																2
12		1																						1
13								1																1
14							1																	1
15	1			1																				2
# bugs	8	2	2	3	0	1	4	2	2	2	2	1	0	3	1	1	1	1	0	0	0	0	1	37

Table 5: Distribution of the USBPD bugs.

Device	PD controller	Failure reasons
S.1,S.7	unknown	—
H.2	IO352BM	did not send valid ACKs
H.3	LDR6023SD	implemented in hardware
H.6	ag9311-maq	unknown debug ports (lack of datasheet)

Table 6: Experimental result of USBPD devices supporting VDMs.