CEFI: Command Execution Flow Integrity for Embedded Devices

Anni Peng[1], Dongliang Fang[2], Wei Zhou[3], Erik van der Kouwe[4], Yin Li[1], and Yuqing Zhang✉[1,5]

[1] National Computer Network Intrusion Protection Center, UCAS, China
[2] Institute of Information Engineering, CAS, China
School of Cyber Security, UCAS, China
[3] School of Cyber Science and Engineering, HUST, China
[4] Vrije Universiteit Amsterdam
[5] School of Cyberspace Security, Hainan unversity, China
zhangyq@nipc.org.cn

Abstract. As embedded devices are widely used in increasingly complex settings (e.g., smart homes and industrial control systems), one device is usually connected with multiple entities, such as mobile apps and the cloud. Recent research has shown that *privilege separation* vulnerabilities, which allow violations of authority between different entities, are occuring in IoT systems. Because such vulnerabilities can be exploited without violating static control flow and data flow, existing CFI and DFI solutions cannot prevent them. We present *CEFI*, the first method to enforce integrity of command execution on embedded devices after deployment. *CEFI* provides fine-grained *Command Execution Flow Integrity* by preventing external commands from being executed on control flow paths belonging to interaction channels that are not authorized to perform them. Using minimal manual annotations as a starting point, *CEFI* statically determined the legal path set (from the start to the end point) and instruments the program to verify the legitimacy of the command execution at runtime by checking whether the calling context is consistent between the runtime executed path and statically obtained legal path set. We evaluate our prototype with five real-world firmware samples, and show that *CEFI* has an average performance overhead of just 0.18%, an average memory overhead of 0.19%, and that *CEFI* can effectively protect embedded devices against attacks on privilege separation vulnerabilities even if they do not violate control flow.

Keywords: Internet of Things (IoT) · embedded devices · enforcement.

1 Introduction

With the development of the Internet of Things (IoT), the application scenarios of embedded devices are becoming broader and more complicated. For example, embedded devices have long been restricted to closed environments, such as industrial plants and vehicle communication systems, but nowadays are increasingly connected, communicating with external systems to carry out their

own functionality. Embedded systems often communicate with multiple external systems in different roles. For example, a smart watch might communicate with one server over WiFi to receive updates, with another over a 4G cellular network to share location data, and also handle diagnostics and configuration commands received by traditional SMS. The attack surface has greatly increased over time. Attacks can send malicious data to an interaction channel by exploiting many low-level security bugs (such as buffer overflows like CVE-2020-25066, CVE-2020-27337, CVE-2020-27338, etc.) in firmware. Furthermore, attackers also leverage missing checks among different interaction channels to perform unauthorized functions.

Taking a smart home scenario as an example, a smart lock interacts with the IoT cloud and mobile app simultaneously. Specifically, the smart lock can receive operation commands both from the remote cloud and the local mobile app. The remote cloud can control the smart lock to update its firmware, and the local app can control the smart lock to perform lock operations (e.g., lock or unlock the door). Typically, different interaction channels are designed to serve different purposes. For this example, firmware updates can *only* be initiated from the trusted cloud. However, if firmware fails to properly verify its interaction channels, a local attacker can issue a malicious firmware update command to the device. This type of attack has been demonstrated in the previous research, which has uncovered 69 similar bugs [30]. Even worse, such an attack is stealthy, as a firmware update is considered to be a normal device operation. The received command does not deviate from normal ones, and there is no violation from the viewpoint of control-flow integrity. Although there is a large body of research on protecting low-end embedded devices [24, 20, 1, 16, 7, 6], they only focus on basic security properties like control-flow integrity and data-flow integrity. Most recently, OAT [24] provides an attestation method that prevents both control-flow and data-only attacks on embedded devices. However, newly discovered hazards [30, 36, 35] involved in IoT interaction channels have enlarged attack surfaces of embedded devices. Moreover, as seen in the example, such attacks do not carry abnormal data or violate the control flow, so none of the previous works can detect such attacks on the device side. Note that this attack differs from data-only attacks [13], which usually require the exploitation of memory corruption bugs. The root cause of our example is a *logical flaw* in the design or implementation of the product, which remains unknown to both vendors and users. It is different from the issue of implicit authorization, as discussed in SmartAuth [26]. Implicit authorization refers to the mobile app gaining more privileges without notifying the user, which is a problem that vendors are aware of but users are not. This also differs from research that primarily analyzes the mobile applications to identify and exploit potential security issues, as seen in studies such as [11, 12]. Although there has been considerable research on security issues in mobile applications, we found mitigating logic flaws in IoT embedded devices have not yet been systematically studied in the literature.

In this work, we propose the first interaction command based attestation method that verifies whether the requested operation is trustworthy when it

carries out one received command execution, even when strictly following its designed purpose. We automatically enforce this verification even if part of the call stack is shared between command handlers, requiring only minimal manual annotations to indicate which commands are allowed on which interfaces. We assign a unique code (i.e., an integer value) to each different code path, and automatically generate and enforce an allowlist that specifies all legal code paths. To prevent attackers from manipulating the unique code and the allowlist, we store both in secure memory on ARM-based devices using on the widely deployed TrustZone extension. This allows us to prevent attackers from executing commands from contexts where they are not authorized, even if executing them would not violate control-flow and data-flow integrity.

Contributions Our work makes the following contributions:

- We propose **C**ommand **E**xecution **F**low **I**ntegrity (CEFI), the first method to enforce integrity of command execution on embedded devices after deployment, even against attacks that violate neither static control flow nor static data flow.
- We apply a calling context encoding algorithm to classify each unique control flow of the program, which is lightweight and suitable for resource-constrained embedded devices.
- We implement *CEFI* and conduct evaluation over five real-world embedded programs that broadly cover multiple use cases in IoT devices, demonstrating the practicality of *CEFI* in real-world application scenarios. *CEFI* is available at `https://github.com/mituanzi/CEFI`.

2 Background

2.1 IoT Architecture

IoT architectures typically involve multiple types of entities [36, 35] including IoT device, cloud backend, and the companion mobile apps running on smartphones (see Fig. 1). Each entity has different responsibilities and design goals. The *IoT device* is designed to interact with the physical world through sensors and actuators. It sends collected real-time information (e.g., device status and environment events) to the cloud or the mobile application. The *cloud backend* manages devices and mobile app user accounts, including the binding between devices and user accounts. In addition, device firmware can be updated from the cloud when needed. When users are not in the same LAN with the devices, the cloud can act as a proxy to forward device control commands from remote users, and forward the device status or command execution results back to the app. *Mobile apps* provide users with an interface to manage devices (e.g., binding the device, viewing device status, and issuing control commands). Generally, mobile apps can control the device in two ways: 1) directly send the commands to the device if they are in the same LAN, or 2) indirectly send the requests to the device via the cloud remotely. There are bidirectional interaction channels between each pair of entities (see Fig. 1).

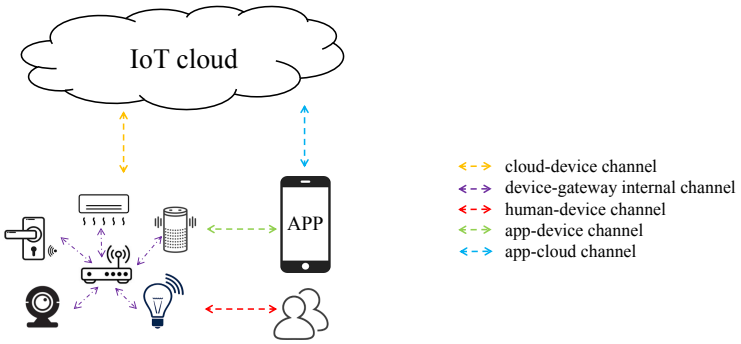

Fig. 1. Interaction model of IoT platform

2.2 Interaction Channels on IoT platform

In the IoT platform, each entity plays a different role and takes on different responsibilities. As a result, the interaction channels between these entities present a certain complexity. Specifically, i) There are many interaction channels derived from multiple entities. First, the relationship between the entities is not only in a one-to-one pattern, but also a one-to-many or many-to-many pattern. For example, a device can be accessed by multiple different users and delegated to multiple different third-party platforms [31](e.g., Philips Hue, LIFX, Google cloud, etc.). Second, entities may also have many interactions inside. For example, a smart hub can interact with multiple smart lights via ZigBee. Finally, a human may also participate in the interaction model directly and bring new interaction channels, such as controlling the device based on the human voice or directly controlling the device with a physical touch screen. ii) Each channel has a different design purpose, which is mentioned in section 2.1. In order to suit the situation, the interaction channels involve a variety of different communication protocols, such as Bluetooth, Zigbee, MQTT, HTTP, etc. iii) There may be functional overlap between different interaction channels. For example, both the cloud and the mobile app have the same functionality in controlling the device (e.g., turn on/off the device), recall that the cloud can act as a proxy to forward control commands from the mobile app. We also note that the app→device channel and cloud→device channel have different responsibilities to manage the device account [30].

The complex interaction model among multiple entities in IoT platforms makes maintaining design purpose more complicated, increasing the security risk of many critical tasks, such as authentication [22, 36], privilege separation [30, 15, 31], state synchronization [21], and task isolation [14], etc. This paper focuses on mitigating *Privilege Separation Vulnerabilities* (PSVs) that violate the privilege separation model [30] (see Section 3.1).

2.3 ARM TrustZone

Our system relies on ARM TrustZone to protect critical state. TrustZone offers a trusted execution environment on ARM. It is available even on Cortex-M microcontrollers, which allows our system to be used even on low-cost CPUs optimized for ultra-low power embedded applications.

TrustZone introduces two protection domains with different permissions at the processor level, the Secure World and the Normal World. The two worlds are completely isolated by the hardware and have different permissions. While code executed from Secure World can access memory in both secure and non-secure regions, both applications and operating systems running in the Normal World are prevented from accessing the resources of the Secure World. Access is only possible through API interfaces specifically offered for this purpose by software in the Secure World. Properties such as hardware isolation and different permissions between the two worlds provide an effective mechanism for protecting an application's code and data, even in the face of a compromised operating system kernel. Therefore, TrustZone can be used to protect the state that is critical for security of our system from tampering.

3 Motivation

3.1 Problem Statement

In this section, we show the dangers of privilege separation vulnerabilities, and the need for a lightweight system to mitigate them, especially on low-powered IoT devices. Following Yao et al. [30], we define privilege separation vulnerabilities as vulnerabilities that violate the privilege separation model. The privilege separation model defines the privileges of each involved role (e.g., remote cloud, local app), which can be inferred from the specification, program context, empirical knowledge, etc.

In order to explain the vulnerabilities in detail, we abstracted a piece of pseudocode from a real example, as shown in Listing 1.1. It demonstrates two types of privilege separation vulnerabilities, based on examples in prior work [30]. The code implements two independent handlers (i.e., cloud_handler and local_handler) to process data from the remote cloud and the local app respectively (line 5). Both handlers use similar processing logic (line 10-39): they receive data over a network, perform authentication, and parse the data. Both invoke the `extract_cmd` function whenever a command is specified. However, `cloud_handler` and `local_handler` may use different receive functions (e.g., `ssl_recv` and `tcp_recv`), different protocols (e.g., MQTT and HTTP), and different data formats (e.g., encrypted format and JSON format). The `extract_cmd` function extracts the specific command and its parameters, and after some checks (e.g., format and value range checks), it executes the command by invoking the corresponding execution functions.

By design, the IoT devices perform privileged operations that can only be initiated from specific interaction channels. Therefore, IoT devices should implement a strict *privilege separation model* when handling commands from different

```
void task_main(void) {
  /* initializations (e.g., variable declarations  */
  /* and definitions, memory allocations, etc) */
  // ......
  func_t handlers[2] = {cloud_handler, local_handler};
  /* register remote handler and local handler */
  // ......
}

void cloud_handler(void) {
  /*  initializations */
  // ......
  char *buf = remote_recv(); // e.g., ssl_recv
  if (auth(buf)) {
    parse_remote_data(buf);
    /* check whether it contains command */
    // ......
    if (remote_control) {
      extract_cmd(buf);
    }
    // ......
  }
  /* error handling and response */
}

void local_handler(void) {
  /*  initializations */
  char *buf = local_recv(); // e.g., tcp_recv
  if (auth(buf)) {
    parse_local_data(buf);
    /* check whether it contains command */
    // ......
    if (local_control) {
      extract_cmd(buf);
    }
    // ......
  }
  /* error handling and response */
}

void extract_cmd(char *buf) {
  cmd_t *cmd = parse_command(buf);
  parameter_t *para = parse_parameter(buf);
  /* check whether cmd and parameters are valid */
  /* (e.g., check its format, value range, etc) */
  // ......
  switch (cmd.type) {
    case OP_1_TYPE: exec_turnOn(cmd, para);  break;
    case OP_2_TYPE: exec_update(cmd, para);  break;
    case OP_3_TYPE: exec_turnOff(cmd, para); break;
    case OP_4_TYPE: exec_reboot(cmd, para);  break;
    // ......
  }
  // .....
}
```

Listing 1.1. Simplified Code Snippet on interaction channel processing logic.

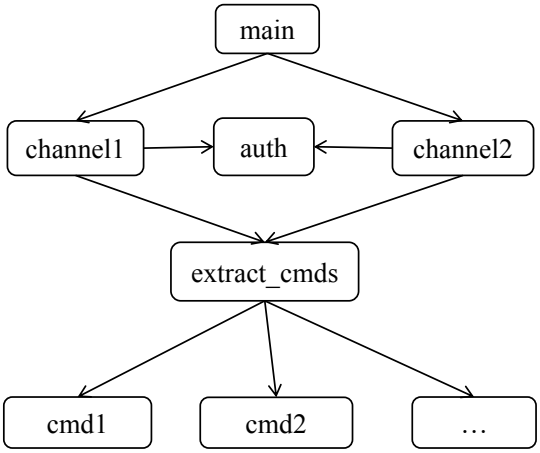

Fig. 2. Simplified Call Graph Illustration

channels (entities) [30]. For example, commands from the remote cloud are responsible for device management, such as binding or unbinding the device with the owner, and updating the device's firmware. Commands from the local apps are used to control interaction with the environment (physical world), e.g., turning on/off the switch, locking/unlocking the lock, adjusting the brightness of the light, etc. Different channels are not supposed to interfere with other's responsibilities. For example, the local app should not be able to update the device's firmware. These properties can be violated by PSVs. We will discuss two particular types: over-privilege vulnerabilities and authentication bypass vulnerabilities.

Over-Privilege Vulnerabilities Listing 1.1 shows an *over-privilege* vulnerability. The command handling function `extract_cmd` shared between the interfaces supports all commands, even those not authorized on some of them. There is a "valid" execution path (that is, one not violating CFI properties) from `local_handler` to `exec_update`, namely with the call trace `task_main` → `local_handler` → `local_recv` → `auth` → `parse_local_data` → `extract_cmd` → `exec_update`. However, based on the program context, it appears that the "valid" path is unexpected and violates the privilege separation model. We can infer this from the user app lacking a user interface (e.g., button) that can initiate the exec_update behavior. This behavior is not intended for the user app to perform. However, attackers can bypass the app interface and issue the command using scripts. This bug has been reported to the vendor and acknowledged [30]. Inferring expected behavior from the program context, including exposed user interfaces, is already used in existing research [26].

In this paper, we label such bugs as *over-privilege* vulnerabilities. Over-privilege vulnerabilities are common in IoT platforms, and previous research has uncovered many severe bugs [30, 36, 35]. Over-privilege are at the root of a number of CVEs, including CVE-2018-10691, CVE-2020-26072, CVE-2022-36782,

```
1   int auth(char *buf) {
2       /*extract user and password information from the buf*/
3       // ......
4       if ((strcmp(user, "GO") == 0) &&
5           (strcmp(pass, "ON") == 0))
6           return SUCCESS;
7       // ......
8       /* this is the real auth function, it checks */
9       /* user provided data with the credentials */
10      if (real_auth(user,pass))
11          return SUCCESS;
12      else
13          return FAIL;
14  }
```

Listing 1.2. Code Snippet from [22] with slight changes.

and CVE-2022-41627. To better understand the root cause of over-privilege vulnerabilities, we show a simplified call graph (see Fig. 2) abstracted from Listing 1.1. We assume there is a design goal that `cmd1` can only be reached through `channel1`, and `cmd2` can only be reached by `channel2`. From the graph, we can see that there are two unintended paths: `channel1` can reach `cmd2`, and `channel2` can reach `cmd1`. When taking a closer look at its root cause, we have two key observations: i) before the command execution functions (i.e., `cmd1`, `cmd2`) there is no check on the role or channel that issues the command, and ii) the unintended path uses a shared function `extract_cmds` to dispatch commands. This function mixes the relationship between channels and privileged operations.

Authentication Bypass Vulnerabilities In the Listing 1.1, the `auth()` function can also be bypassed without having knowledge of user's credentials. Generally, attacker may leverage control-flow hijack technique (e.g., based on some memory bugs) to bypass the authentication process. However, we do not target memory bugs in this paper, instead, we target the bugs that are derived from logic violations – the presence of hardcoded authentication credentials in the authentication routine. Specifically, we show a problematic implementation of `auth()` function in the Listing 1.2. There are hardcoded credentials ("GO" and "ON") in the `auth` function. The auth function can be bypassed (i.e., without calling the `real_auth` function) when the user's input is consistent with the hard-coded ones (line 2-4). Consequently, once an attacker analyzes and knows the relevant content of the hardcoded information, it is not hard for them to bypass the authentication and gain access to the device. Authentication bypass vulnerabilities are common, and are at the root of a number of CVEs, including CVE-2017-8226, CVE-2021-33218, CVE-2021-33220, CVE-2022-29730.

Limitations of Potential Solutions To defend against the privilege separation bugs, there are three potential solutions: i) Detection in advance by analyzing illegal path reachability. Ideally, we can use static analysis to detect illegal paths in advance and ensure that `cmd1` will never be reached through channel2. However, since static analysis faces some common challenges (e.g., it is hard to precisely resolve all indirect calls), it is very hard if not possible to accurately

exclude all paths from channel2 to `cmd1`. Moreover, any detected bugs still need code patches (defense solutions) to defend against potential attacks. ii) Blocking execution by checking input directly at the start point channel2. Recall the example in Listing 1.1, one may argue that we can easily block the execution if an "update" command is found at the `local_recv()`. We note that the received raw data at the entry point has various complex formats (e.g., JSON format and encrypted formats [30]) and not completely parsed yet at this point. Therefore, it is infeasible to directly filter illegal commands at the entry point (i.e., `local_recv()` and `remote_recv()`). Futhermore, different channels are not completely independent and often have some shared behaviors (e.g., both the cloud and user app are allowed to issue turn on/off commands) and call shared functions. Shared functions can easily lead to privilege separation vulnerabilities, as noted by Yao et al. [30]. Although the program can know which channel is involved at the entry point, it cannot predict the control flow, since the command has not yet been parsed at that point. iii) Applying traditional CFI solutions. However, logic bugs follow a "legal" path in the program implementation. There is a path from channel2 to `cmd1` which does not violate the CFG, so CFI solutions fundamentally cannot mitigate this type of vulnerability.

3.2 Threat Model

We focus only on privilege separation vulnerabilities, and assume that the attacker cannot hijack the original control flow and data flow. Note that we do not consider memory errors in the threat model, as existing work can mitigate them. Our approach is orthogonal to existing solutions addressing control flow hijacking (e.g., [1, 20]) and data-flow violations (e.g., [24]). We also assume that the attacker can access the IoT devices (e.g., via victim's LAN), so they have the ability to send the requests to the IoT devices directly or with a MITM attack. Moreover, we assume the attacker can exploit any logic errors to execute commands already present in the firmware without authorization. Our system will be applied by a programmer who has access to the firmware source code, and has knowledge of its high-level privilege-related design logic. This is realistic when our system is deployed by the original developer, and also for third parties in case of properly documented open source firmware. We also assume the trusted software in the TEE is bug-free and isolated from the Normal World firmware, as important metadata such as the allowlist is stored there. Considering the small code base of the TEE-side software and the limited attack surface, this is a reasonable assumption, well accepted by existing TEE-based work [1, 20, 24]. We do not consider low-level physical attacks, such as connecting to a JTAG debugger to re-program the firmware. Finally, we assume our compiler passes are free of bugs.

4 CEFI

In this section, we discuss the design and implementation of *CEFI*. Fig. 3 shows the overall design of *CEFI*. Developers can use *CEFI* to protect their firmware.

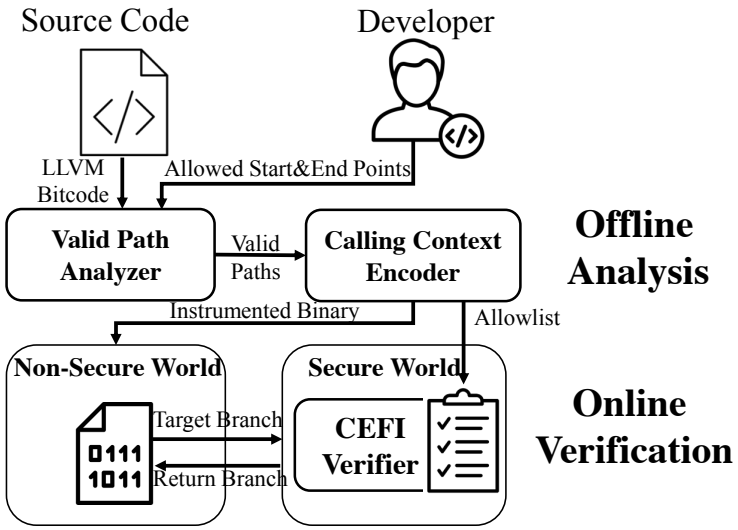

Fig. 3. The architecture of CEFI.

It acts as a compiler pass, and anyone who has access to the firmware source code can use it to generate a firmware binary hardened against privilege separation vulnerabilities. *CEFI* needs minimal manual annotations to specify the mapping between interfaces that can receive commands and commands permitted on those interfaces, and can then automatically instrument the program to enforce those policies at runtime, even in the face of logic bugs.

Our approach consists of two phases, which are discussed in this section. The static calling context encoding (CCE) phase (Section 4.1) happens at compile time, and performs static analysis and instrumentation to create a hardened binary. It uses the policies specified by the user in the form of minimal annotations to generate an `allowlist` that specifies the code paths that satisfy the policy, and are therefore valid contexts to execute particular commands, and instruments to code to be able to enforce this allowlist. The dynamic command execution flow verification phase (Section 4.2) offers the necessary support at runtime to perform these checks in a secure way. We take advantage of secure storage and isolation provided by Trustzone-M the `allowlist` at runtime. We use Trustzone to protect the security of the allowlist, as our approach relies on the guarantee that it is not illegally written to. Meanwhile, due to the security isolation of TrustZone, we avoid having to perform checks at all risky (e.g., pointer-based) write instructions, which is expensive.

4.1 Calling Context Encoding Instrumentation

We implement *CEFI* as two LLVM compiler passes, namely the Valid Path Analyzer and the Calling Context Encoder. We discuss them in this section.

Valid Path Analyzer To accurately enforce the execution path belonging to its expected interaction channel, *CEFI* needs developers to specify a legitimate pair set between the sensitive command function (i.e., end point) with its corresponding entry function (i.e., start point). There are some manual works annotating the start point and the end point, and representing their relationship. Taking Listing 1.3 as an example, function `func_1` and function `func_2` are two different entry functions (i.e., start point). The function `turn_on` and `update_firmware` are two different command execution functions (i.e., end point). The annotations specify a relationship between the start points and the end points, indicating which paths are allowed and which are not. Specifically, the end point `turn_on` can be reached from start point `func_1` and `func_2`, while end point `update_firmware` can only be reached by start point `func_1`. On this basis, we can obtain many allowed pairs, i.e., (`func_1`, `update_firmware`), (`func_1`, `turn_on`), (`func_2`, `turn_on`). After annotating, the path analyzer gathers all available paths from the start point to the end point of each pair based on the call graph with a function named `AllowPathAnalyzer()`. In this way, we can get the valid paths to each command execution function and filter out a lot of irrelevant code. Note that these valid paths have already filtered the paths that are not allowed to reach command execution functions, e.g., any paths from `func_2` → `update_firmware` are not regarded valid even if they exist.

```
1    void __attribute__((annotate("entry#role1"))) func_1();
2    void __attribute__((annotate("entry#role2"))) func_2();
3    void __attribute__((annotate("cmd#role1#role2"))) turn_on();
4    void __attribute__((annotate("cmd#role1"))) update_firmware();
```

Listing 1.3. Manual Annotation

Calling Context Encoder Calling context encoding is a lightweight technique to record dynamic calling path history, which has been widely used in many software development processes such as testing, event logging, and program analysis [4, 23, 32, 33]. Its basic idea is to instrument function calling point, so at the runtime, the instrumentation can dynamically update the ID such that the value of the ID represents the current calling context. To make the ID uniquely distinguish different contexts, the CCE algorithm solves many challenges, such as recursive calls, function pointers, etc. Since the CCE algorithm is not our contribution, we omit the details here. Instead, we directly integrate the work by Sumner et al. [23] in *CEFI*. The left-hand side of Fig. 4 (see [23] for the ID notation) illustrates how it works. In the calling graph, the CCE algorithm assigns a value to each edge between `StartPA` and `EndPA`. so the ID can be dynamically updated and each path to `EndPA` can be uniquely mapped to a value in the runtime. Specifically, before the start point `StartPA`, the id is initialized to 1, and at the end point `EndPA`, the id may have two different values (i.e., 1, 2), which can uniquely distinguish two different paths (*StartPA*→*f2*→*f4* →*EndPA* and *StartPA*→*f3*→*f4* →*EndPA*). When *CEFI* is deployed, it will instrument a secure gateway API call at each edge, but it only requires instrumentation at

Algorithm 1 Algorithm for Allowlist Generation, denote as AllowlistGenerate()

```
Input  : PairSet
Input  : CallGraph
Output: Allowlist
//PairSet: each item consists of (startPoint, endPoint)
//StartPA: entry function
//EndPA: command function
//Allowlist: a dict, the key is EndPA, the value is an AllowedIDSet
Item ← Head(PairSet)
do
    StartPA, EndPA ← GetTuple(Item)
    AllowPathSet ← ValidPathAnalyzer(StartPA, EndPA, CallGraph)
    EncodedAllowPathSet ← CallingContextEncoding(AllowPathSet)
    AllowedIDSet ← ComputeID(EncodedAllowPathSet)
    Allowlist ← AddTo(Allowlist, EndPA, AllowedIDSet)
    Item ← Next(PairSet);
while Item;
```

the edge *f3*→*f4*, as the process of "*id*+ = 0" does not alter the calling context ID, making instrumentation unnecessary at other edges (such as `StartPA`→*f2*, `StartPA`→*f3*, and `f2`→*f4*). *CEFI* is lightweight, which can be attributed to the fact that only a limited amount of instrumentation is required. Besides, the involved CCE algorithm is safe (i.e., different contexts are guaranteed to have different ID), reversible (i.e., calling context can be faithfully decoded and recovered) [23].

With the help of CCE, we can generate an `allowlist` for each end point (i.e., command execution function), as shown in Algorithm 1, which is denoted as `AllowlistGenerate()`. The algorithm takes the pair set of user-defined start and end points and the whole call graph of the firmware as inputs. The output is the `allowlist`. For each pair (i.e., StartPA, EndPA), the algorithm first gets an AllowPathSet using the `ValidPathAnalyzer()`, and each item in the set represents an allowed path from StartPA to EndPA. Then, it encodes each edge of an allowed path using `CallingContextEncoding()`, and accumulates the weights (IDs) of each allowed path to EndPA using `ComputeID`. Specifically, each call statement on the allowed path will be instrumented to update the value of context identifier, so that we can obtain a value at the EndPA and check it against the allowlist. Note that we assign one to the start point instead of zero as the traditional CCE algorithm does for quickly distinguishing the remaining paths to the command function with zero values without encoding them. After traversing every *item* in *pairSet*, we obtain the allowlist dictionary, in which the key is `EndPA` and the value is `AllowedIDset`.

Instrumentation with ARMv8-M Security Extension During instrumentation, the firmware reads the allowlist and resides in the read-only secure memory before initialization at runtime. Then, all allowed paths are encoded with CCE as mentioned before. Meanwhile, any ID update before the function call

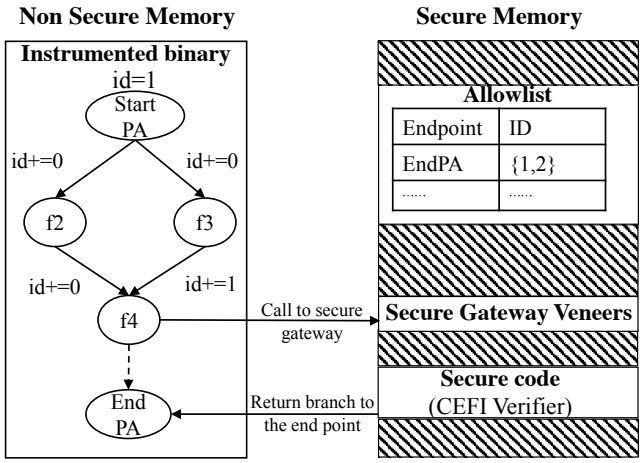

Fig. 4. A Running example of the CEFI verification process.

site will be transferred to TrustZone-protected Secure World, thus we can guarantee the security of dynamically computed calling context ID. To enforce the integrity check before command execution, we also intercept all call instructions to the command functions (end points) and redirect them to the CEFI verifier in the Secure World through secure gateway veneers [3]. If the verification is passed, the control flow returns back to the intended command function.

4.2 Command Execution Flow Integrity Enforcement

Figure. 4 illustrates the process of dynamic command execution flow integrity enforcement with CEFI verifier. When a call instruction to a command function is encountered in the firmware, the control flow will be transferred to the security gateway veneers and further forwarded to the CEFI verifier. It verifies the current calling context (current ID value) by looking up the valid ID set to the command function (e.g., EndPA in Figure. 4) from the allowlist. After verification, if the calling context is allowed, the control flow redirects back to the command execution. If the calling context is invalid (e.g., the ID value is not one or two in Figure. 4), the CEFI verifier will stall the execution and output the current calling context (ID value) via UART or other peripherals based on user specification.

5 Evaluation

To evaluate *CEFI*, we conducted experiments on five commonly-used programs to i) measure its performance in terms of runtime and memory overhead, ii) perform a security analysis to demonstrate its effectiveness, and iii) show the manual effort required to use *CEFI*.

Testing Environment *CEFI* is implemented on top of LLVM 10 and runs on Ubuntu 18.04. *CEFI* generates hardened binaries that can run on the STM32L562E-DK discovery kit, a popular IoT development board. This board features an ARM Cortex-M33 core with TrustZone support, along with 512 kB Flash memory and 256 kB SRAM. Our prototype does not rely on other board-specific features, making it adaptable to other ARM Cortex-M chips.

Benchmark Our benchmark consists of five embedded programs with practical application scenarios. They are small in size (the average size is 104.03 kB) compared with traditional software, which is representative of embedded programs in practical settings, as these CPUs cannot run larger programs. These programs have been used for evaluation in prior embedded device research [24, 10, 8].

- **Light Controller** is used in smart home applications. User can turn the light on/off remotely by sending control command. It is also used for evaluation by OAT [24].
- **Syringe Pump** is used in medical and production applications. The user can control a device to inject or withdraw fluid automatically by sending control command with user-provided amount. It is also used by C-FLAT [1] and OAT [24].
- **Thermostat** reads the temperature and humidity from a sensor. If the temperature is too far from a preset temperature it can, for example, trigger an air conditioning unit. It also accepts commands to retrieve the current temperature. This program is used by PRETENDER [10].
- **RF_door_lock** can be applied to smart door locks. Its commands include unlocking the door given the correct password, and setting a custom password. This program is used by PRETENDER [10].
- **Steering_control** is used in autonomous driving. It receives commands from the computer to control the steering and moving/motoring of the autonomous vehicle. This program is used by P2IM [8].

Table 1 presents details on the application layer logic of the programs. It does not consider the boot loader, board support package (BSP), device drivers, or any other components outside of the application layer. It includes the functions involved (#Functions), the number of annotations made (#Annotations), the number of all allowed paths from the StartPA to EndPA (#AllowedPaths), and the lines of code of application layer (#LoC).

Table 1. Statistics in Benchmark Programs

Program	#Functions	#Annotations	#AllowedPaths	#LoC
Light Controller [24]	33	4	7	286
Syringe Pump [1] [24]	51	4	8	569
Thermostat [10]	28	5	4	154
RF_door_lock [10]	25	4	5	219
Steering_control [8]	33	2	8	150

5.1 Performance Overhead

For defense mechanisms to be deployable, they must result in low performance overhead [25]. This is especially important for resource-constrained embedded devices. To study the runtime overhead introduced into a system by *CEFI*, we measure the execution time with and without *CEFI* for each test program. We record end-to-end overhead, based on the time between when a device receives an event and when a device completes the resulting action. The five programs we selected all have multiple execution paths that are triggered by different inputs. Therefore, we design different inputs to trigger each branch of the program. The reported runtimes are averages over ten executions of each input.

We present the runtime overhead and memory overhead in Table 2. The column #Trans lists the average transitions between the trusted and normal world of each programs. The results show that *CEFI* has very low overhead for each of the programs, with a geometric mean of just 0.18% over all of them.

Memory overhead consists of Flash overhead and RAM overhead. For Flash overhead, *CEFI* adds instrumentation to encode and decode the ID at each call site. In addition, before the specific function, we need instrumentation to send the ID to the Secure World to match it against the allowlist. These instrumentations increase Flash usage. The results show the Flash memory overhead of *CEFI*. The geometric mean overhead across all applications is just 0.19%. For RAM overhead, it mainly consists of the allowlist stored in the Secure World, costing less than 80 bytes since the biggest allowlist contains less than twenty legal integer IDs.

Table 2. Runtime Overhead and Memory Overhead

Program	#Trans	Execution Time (ms)			Memory Consumption (bytes)		
		Baseline	*CEFI*	Overhead	Baseline	*CEFI*	Overhead
Light Controller	9	18.49	18.51	0.11%	102888	103172	0.28%
Syringe Pump	7	54.34	54.36	0.04%	110536	110772	0.21%
Thermostat	5	5.04	5.05	0.20%	108184	108332	0.14%
RF_door_lock	10	2.20	2.21	0.45%	102308	102480	0.17%
Steering_control	7	10.74	10.75	0.09%	108724	108888	0.15%
Geometric Mean				0.18%			0.19%

5.2 Effectiveness Analysis

Attack detection via *CEFI* Although privilege separation vulnerabilities are common (recent ones include for example CVE-2020-26072, CVE-2021-33220, and CVE-2022-36782), unfortunately firmware is rarely available open source. As such, there is no known vulnerable firmware available for us to test. Instead, we injected the vulnerability shown in Figure 2 into the test example, and verified that *CEFI* could prevent attacks that exploit this vulnerability while running the program. In order to construct the vulnerability, we added some functions to the test example, such as `switch_on`, `update_firmware`, `change_passwd`, and so on. For the experiment, we formulated the following rules: some commands can only

Table 3. Commands for Smart Light Example

Interaction Channel	Annotated Command Functions
Local client	switch_on, switch_off
IoT cloud	update_firmware, recovery_firmware, change_password

be issued by the cloud, and the others only by local clients. The details are shown in Table 3. However, we do not implement strict authentication, resulting in the vulnerability that the remote and local command sets can be mixed together.

In order to realize our enhancement scheme, we carried out the following steps: first, we annotate the command execution functions and entry functions, such as *switch_off*, *update_firmware*, etc. Then, the legal IDs of the paths between entry function and command execution function can be statically obtained. We traverse all legal operations, and store all legal ID information in TrustZone to form an allowlist. After getting the allowlist, we can send instructions to the device at will: for example, send the *update_firmware* command from the IoT cloud to the device, and the device can normally perform the corresponding operation, or send the *update_firmware* command from the local client to the device, and the device prompts that the operation cannot be performed. It can be seen from this that our scheme is effective. Although this vulnerability is constructed by ourselves, it tests the most important logic of the vulnerability, and our solution can indeed discover and defend against such vulnerabilities during program operation.

Security Analysis Our threat model does not assume that the attacker can hijack the original control flow and data flow, as outlined in section 3.2. To evade *CEFI*, attackers need to circumvent the validation checks of *CEFI*. They would need to achieve any of the following: 1) disable the instrumentation of the calling context ID updating logic and validation checks; 2) tamper with the metadata, which includes the calling context ID and the allowlist; or 3) discover a collision where a path from the channel (entry point) to **disallowed** commands (end point) can produce an calling context ID that is included in the allowlist.

However, 1) is ruled out by the assumption that existing control flow integrity enforcement has been effectively deployed, and attacks cannot bypass the execution of instrumented trampoline functions. *CEFI* prevents 2) by storing the crucial metadata (i.e., calling context ID and allowlist) in a TrustZone-protected secure world. The metadata can only be updated by a secure API call (i.e., instrumented trampoline function call). The assumption of control flow integrity ensures that secure API call cannot be hijacked and will be correctly executed. As a result, 2) is prevented as well. Lastly, 3) is prevented by the nature of CCE algorithm [23], which guarantees the uniqueness of the ID for each specific calling context.

5.3 Annotation Effort

CEFI requires annotation (see Listing 1.3) to express relationship between the interaction channel's entry point and the end point (i.e., command execution

function). Although this is a manual process, it requires only minimal effort (see Table 1). In our experiments, we anticipate that a few minutes are enough to complete annotations for a program, assuming that programmers have the knowledge of its design logic. While these programs may seem particularly small, this reflects the fact that most embedded programs are by nature required to be much smaller than regular software. We note that there are currently no standard benchmarks, but we chose these programs because they are also widely used for evaluation in related research.

6 Related Work

CEFI is the first approach to enforce integrity of command execution on embedded devices after deployment, even against attacks that violate neither static control flow nor static data flow, even though such vulnerabilities are common on embedded devices [35]. The only other work that can identify such vulnerabilities, Gerbil [30], uses symbolic execution to find them in the testing phase, but cannot prevent exploitation of residual vulnerabilities after deployment. *CEFI*'s very low overhead makes it particularly suitable for this purpose, especially on resource-constrained embedded devices.

Other existing work focuses mostly on detecting violations of control flow and data flow. In this section, we first focus on work that enables detection of such vulnerabilities on resource-constrained embedded devices. Since context-sensitivity is critical to *CEFI*'s ability to detect privilege separation vulnerabilities, we also discuss works that introduce context sensitivity to runtime detection of violations.

6.1 Control and Data Flow Integrity on Embedded Systems

Since embedded devices are resource-constrained, solutions to enforce control-flow integrity (CFI) and data-flow integrity (DFI) are only viable if they are very lightweight. For example, μRAI [2] uses LLVM compiler passes to enforce return address integrity by removing the need to spill return addresses to the stack. Silhouette [34] leverages an incorruptible shadow stack for hardening backward indirect jump and uses a label instruction for protecting forward indirect jump. CFI CaRE [20] leverages TrustZone to implement a shadow stack mechanism. DFI is used to protect the integrity of memory access (e.g., maintaining and checking bounds information for each memory read or write). DFI requires much more instrumentation than CFI, because it needs to perform checks at memory access points rather than just at indirect branches. As such, there are fewer solutions for DFI than for CFI for embedded systems, as pointed out in a recent survey [17]. One notable work is OAT [24], selectively protects critical data on embedded programs. Therefore, it reduces performance overhead by instrumenting critical variable access, but sacrifices protection. However, this solution does not affect the protection provided by *CEFI*. Existing work cannot prevent exploitation of privilege separation vulnerabilities without violating control and data flow properties.

6.2 Context Sensitive Defense Solutions

Context sensitivity allows defenses to be more restrictive than traditional CFI and DFI solutions can be, by considering not just static properties of the control and data flow graphs, but also the actual control flow path taken at runtime. Calling context is widely used in context-sensitive defenses [9, 18, 19, 27–29]. For example, PathArmor [27] conducts context-sensitive static analysis over the CFG on-demand, and provides context-insensitive CFI policies. However, PathArmor [27] does not protect against privilege separation vulnerabilities. Henry et al. [9] and David et al. [28] use the call stack and calling context for anomaly detection and system call trace consistency, but these methods are expensive [5]. Qiang et al. [33] propose HeapTherapy for lightweight trace collection and exploit detection, which aims to mitigate traditional heap buffer overflows. We use calling context encoding to defend against privilege separation vulnerabilities and logic bugs without control flow violations in resource-constrained embedded systems.

7 Conclusion

With the development of the Internet of Things (IoT), the application scenarios of embedded devices are becoming broader and more complicated. As a result, there are interactions between the various entities (i.e., cloud, the IoT device, mobile app). This causes the rise of a new type of vulnerability, privilege separation vulnerabilities, that can be exploited to launch attacks (e.g., device hijacking attacks) without control flow anomalies. Therefore, we propose *CEFI*- Command Execution Flow Integrity, to protect embedded devices against such attacks. Finally, we apply *CEFI* on five real-world programs. The evaluation shows that *CEFI* can effectively prevent this type of attack, with negligible runtime overhead of 0.18% and negligible memory overhead of 0.19%. In future work, we plan to evaluate *CEFI* on larger IoT systems.

Acknowledgements

We thank our shepherd Roland YAP Hock Chuan and anonymous reviewers for their valuable feedback. This work was supported by the National Natural Science Foundation of China (U1836210), the Key Research and Development Science and Technology of Hainan Province (GHYF2022010), the National Natural Science Foundation of China (No.62202188), the National Key R&D Program of China (No.2022YFB31033400), and the Netherlands Organisation for Scientific Research (NWO) through project “Vulcan”. Meanwhile, this work was partly done at VU Amsterdam. We thank the support provided by the China Scholarship Council (CSC) and the VUSec Group at VU Amsterdam.

References

1. Abera, T., Asokan, N., Davi, L., Ekberg, J.E., Nyman, T., Paverd, A., Sadeghi, A.R., Tsudik, G.: C-flat: control-flow attestation for embedded systems software. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 743–754 (2016)
2. Almakhdhub, N.S., Clements, A.A., Bagchi, S., Payer, M.: μrai: Securing embedded systems with return address integrity. In: 27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26 (2020)
3. ARM Ltd.: Arm compiler software development guide version 6.3 (2022), `https://developer.arm.com/documentation/dui0773/d/chunkpge1447084556319`
4. Ball, T., Larus, J.R.: Efficient path profiling. In: MICRO 29. pp. 46–57. IEEE (1996)
5. Bond, M.D., McKinley, K.S.: Probabilistic calling context. Acm Sigplan Notices **42**(10), 97–112 (2007)
6. Cerdeira, D., Santos, N., Fonseca, P., Pinto, S.: Sok: Understanding the prevailing security vulnerabilities in trustzone-assisted tee systems. In: 2020 IEEE Symposium on Security and Privacy (SP). pp. 1416–1432. IEEE (2020)
7. Clements, A.A., Almakhdhub, N.S., Bagchi, S., Payer, M.: Aces: Automatic compartments for embedded systems. In: USENIX Security 2018. vol. 2018, pp. 65–82 (2018)
8. Feng, B., Mera, A., Lu, L.: P2im: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling. In: USENIX Security 2020. pp. 1237–1254 (2020)
9. Feng, H.H., Kolesnikov, O.M., Fogla, P., Lee, W., Gong, W.: Anomaly detection using call stack information. In: 2003 Symposium on Security and Privacy, 2003. pp. 62–75. IEEE (2003)
10. Gustafson, E., Muench, M., Spensky, C., Redini, N., Machiry, A., Fratantonio, Y., Balzarotti, D., Francillon, A., Choe, Y.R., Kruegel, C., et al.: Toward the analysis of embedded firmware through automated re-hosting. In: RAID 2019. pp. 135–150 (2019)
11. Hassanshahi, B., Jia, Y., Yap, R.H.C., Saxena, P., Liang, Z.: Web-to-application injection attacks on android: Characterization and detection. In: ESORICS. pp. 577–598. Springer (2015)
12. Hassanshahi, B., Yap, R.H.C.: Android database attacks revisited. In: AsiaCCS 2017. pp. 625–639 (2017)
13. Hu, H., Shinde, S., Adrian, S., Chua, Z.L., Saxena, P., Liang, Z.: Data-oriented programming: On the expressiveness of non-control data attacks. In: 2016 IEEE Symposium on Security and Privacy (SP). pp. 969–986. IEEE (2016)
14. Huo, D., Cao, C., Liu, P., Wang, Y., Li, M., Xu, Z.: Commercial hypervisor-based task sandboxing mechanisms are unsecured? but we can fix it! Journal of Systems Architecture **116**, 102114 (2021)
15. Jia, Y., Xing, L., Mao, Y., Zhao, D., Wang, X., Zhao, S., Zhang, Y.: Burglars' iot paradise: Understanding and mitigating security risks of general messaging protocols on iot clouds. In: 2020 IEEE Symposium on Security and Privacy (SP). pp. 465–481. IEEE (2020)
16. Koeberl, P., Schulz, S., Sadeghi, A.R., Varadharajan, V.: Trustlite: A security architecture for tiny embedded devices. In: Proceedings of the Ninth European Conference on Computer Systems. pp. 1–14 (2014)

17. Mishra, T., Chantem, T., Gerdes, R.: Survey of control-flow integrity techniques for embedded and real-time embedded systems. arXiv preprint arXiv:2111.11390 (2021)
18. Newsome, J., Brumley, D., Song, D., Chamcham, J., Kovah, X.: Vulnerability-specific execution filtering for exploit prevention on commodity software. In: NDSS (2006)
19. Novark, G., Berger, E.D., Zorn, B.G.: Exterminator: automatically correcting memory errors with high probability. In: PLDI. pp. 1–11 (2007)
20. Nyman, T., Ekberg, J.E., Davi, L., Asokan, N.: Cfi care: Hardware-supported call and return enforcement for commercial microcontrollers. In: RAID. pp. 259–284. Springer (2017)
21. OConnor, T., Enck, W., Reaves, B.: Blinded and confused: uncovering systemic flaws in device telemetry for smart-home internet of things. In: Proceedings of the 12th Conference on Security and Privacy in Wireless and Mobile Networks. pp. 140–150 (2019)
22. Shoshitaishvili, Y., Wang, R., Hauser, C., Kruegel, C., Vigna, G.: Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware. In: NDSS. vol. 1, pp. 1–1 (2015)
23. Sumner, W.N., Zheng, Y., Weeratunge, D., Zhang, X.: Precise calling context encoding. IEEE Transactions on Software Engineering **38**(5), 1160–1177 (2011)
24. Sun, Z., Feng, B., Lu, L., Jha, S.: Oat: Attesting operation integrity of embedded devices. In: SP. pp. 1433–1449. IEEE (2020)
25. Szekeres, L., Payer, M., Wei, T., Song, D.: Sok: Eternal war in memory. In: 2013 IEEE Symposium on Security and Privacy. pp. 48–62. IEEE (2013)
26. Tian, Y., Zhang, N., Lin, Y.H., Wang, X., Ur, B., Guo, X., Tague, P.: Smartauth: User-centered authorization for the internet of things. In: USENIX Security 2017. pp. 8–2 (2017)
27. Van der Veen, V., Andriesse, D., Göktaş, E., Gras, B., Sambuc, L., Slowinska, A., Bos, H., Giuffrida, C.: Practical context-sensitive cfi. In: CCS. pp. 927–940 (2015)
28. Wagner, D., Dean, R.: Intrusion detection via static analysis. In: Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001. pp. 156–168. IEEE (2000)
29. Xu, J., Ning, P., Kil, C., Zhai, Y., Bookholt, C.: Automatic diagnosis and response to memory corruption vulnerabilities. In: CCS. pp. 223–234 (2005)
30. Yao, Y., Zhou, W., Jia, Y., Zhu, L., Liu, P., Zhang, Y.: Identifying privilege separation vulnerabilities in iot firmware with symbolic execution. In: European Symposium on Research in Computer Security. pp. 638–657. Springer (2019)
31. Yuan, B., Jia, Y., Xing, L., Zhao, D., Wang, X., Zou, D., Jin, H., Zhang, Y.: Shattered chain of trust: Understanding security risks in cross-cloud iot access delegation. In: USENIX Security 2020. pp. 1183–1200 (2020)
32. Zeng, Q., Rhee, J., Zhang, H., Arora, N., Jiang, G., Liu, P.: Deltapath: Precise and scalable calling context encoding. In: Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization. pp. 109–119 (2014)
33. Zeng, Q., Zhao, M., Liu, P.: Heaptherapy: An efficient end-to-end solution against heap buffer overflows. In: DSN 2015. pp. 485–496. IEEE (2015)
34. Zhou, J., Du, Y., Shen, Z., Ma, L., Criswell, J., Walls, R.J.: Silhouette: Efficient protected shadow stacks for embedded systems. In: USENIX. pp. 1219–1236 (2020)
35. Zhou, W., Cao, C., Huo, D., Cheng, K., Zhang, L., Guan, L., Liu, T., Jia, Y., Zheng, Y., Zhang, Y., et al.: Reviewing iot security via logic bugs in iot platforms and systems. IEEE Internet of Things Journal **8**(14), 11621–11639 (2021)

36. Zhou, W., Jia, Y., Yao, Y., Zhu, L., Guan, L., Mao, Y., Liu, P., Zhang, Y.: Discovering and understanding the security hazards in the interactions between iot devices, mobile apps, and clouds on smart home platforms. In: USENIX. pp. 1133–1150 (2019)