Understanding Device Integration Bugs in Smart Home System

Tao Wang*
State Key Lab of Computer Science at ISCAS, University of CAS, China
wangtao19@otcaix.iscas.ac.cn

Kangkang Zhang*
State Key Lab of Computer Science at ISCAS, University of CAS, China
zhangkangkang19@otcaix.iscas.ac.cn

Wei Chen†
State Key Lab of Computer Science at ISCAS, University of CAS, Nanjing Institute of Software Technology, China
wchen@otcaix.iscas.ac.cn

Wensheng Dou†
Jiaxin Zhu
State Key Lab of Computer Science at ISCAS, University of CAS, University of CAS Nanjing College, China
{wsdou,zhujiaxin}@otcaix.iscas.ac.cn

Jun Wei
State Key Lab of Computer Science at ISCAS, University of CAS, Nanjing Institute of Software Technology, China
wj@otcaix.iscas.ac.cn

Tao Huang
State Key Lab of Computer Science at ISCAS, University of CAS, China
tao@otcaix.iscas.ac.cn

ABSTRACT

Smart devices have been widely adopted in our daily life. A smart home system, e.g., Home Assistant and openHAB, can be equipped with hundreds and even thousands of smart devices. A smart home system communicates with smart devices through various device integrations, each of which is responsible for a specific kind of devices. Developing high-quality device integrations is a challenging task, in which developers have to properly handle the heterogeneity of different devices, unexpected exceptions, etc. We find that device integration bugs, i.e., *iBugs*, are prevalent and have caused various consequences, e.g., causing devices unavailable, unexpected device behaviors.

In this paper, we conduct the first empirical study on 330 iBugs in Home Assistant, the most popular open source smart home system. We investigate their root causes, trigger conditions, impacts, and fixes. From our study, we obtain many interesting findings and lessons that are helpful for device integration developers and smart home system designers. Our study can open up new research directions for combating iBugs in smart home systems.

CCS CONCEPTS

• **General and reference** → **Empirical studies**; • **Networks** → **Home networks**.

KEYWORDS

Empirical study, smart home system, integration bugs, Home Assistant

ACM Reference Format:
Tao Wang, Kangkang Zhang, Wei Chen, Wensheng Dou, Jiaxin Zhu, Jun Wei, and Tao Huang. 2022. Understanding Device Integration Bugs in Smart Home System. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '22), July 18–22, 2022, Virtual, South Korea.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3533767.3534365

1 INTRODUCTION

A smart home is equipped with software driven devices, namely smart devices, which can be automatically controlled remotely from anywhere via a terminal with internet connection, e.g., mobile phones [22]. The smart home system is used to manage smart devices. It usually provides services of scheduling tasks, e.g., turning on sprinklers to water flowers at the preset time, and handling events, e.g., turning lights on when people come over. More and more people begin to construct their own smart homes. The global smart home market size is expected to grow from 78.3 billion dollars in 2020 to 135.3 billion dollars by 2025 [23]. To have a more flourish smart home ecosystem and more reliable smart home systems is anticipated to meet the growing demand of smart home market.

There are more and more smart home systems in the market. Many smart home systems are closed source or commercial, e.g., Samsung SmartThings [24], Google Assistant [19], and Huawei HiLink [20]. Most of these systems only support the devices in their closed ecosystems. In contrast, open source smart home systems are crowdsourcing software and allow third parties to integrate all kinds of devices. openHAB [1] and Home Assistant [6] are two typical open source smart home systems. Both of them have powerful automation engines, friendly UI, and open interfaces to integrate devices. Various smart home devices can be integrated into the systems, and the number of supported devices keeps increasing. To make a device connectable and accessible to a smart home system, developers have to contribute an adaptive program for the device to the system, which is called a ***device integration***. Through device

*Tao Wang and Kangkang Zhang contribute equally. CAS is the abbreviation of Chinese Academy of Sciences. ISCAS is the abbreviation of Institute of Software, Chinese Academy of Sciences.
†Wei Chen and Wensheng Dou are the corresponding authors.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ISSTA '22, July 18–22, 2022, Virtual, South Korea
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9379-9/22/07…$15.00
https://doi.org/10.1145/3533767.3534365

integrations, end users can control the devices in the smart home systems.

Developing high-quality device integrations is a challenging task, in which developers need to properly handle the heterogeneity of different devices, possible situations and errors, master domain knowledge of target systems and devices. Therefore, device integration bugs often occur in smart home systems, e.g., no responses from devices, incorrect device states in the systems. These problems can potentially make a smart home inconvenient and even dangerous, e.g., serious accidents. In this paper, we call such a device integration bug as an ***iBug*** for brevity.

Substantial studies have investigated smart home systems. Some of them investigated bugs occurred in smart devices [34, 49, 53], and some of them looked into smart home applications to find bugs that could lead to safety and security problems [28, 31–33, 39, 60]. However, none of these studies has focused on iBugs in smart home systems. There is little knowledge about iBugs of smart home systems in the literature. Understanding iBugs is of significant interest to researchers and practitioners in the smart home community, which could help identify areas where we need better tool support and offer guidelines on high-quality device integration development.

In this paper, we conduct the first empirical study on iBug in an open source smart home system. We select Home Assistant [6], the most popular and active open source smart home system as our study object. We investigate 2767 issues committed from 2020.5 to 2020.12 in the Home Assistant repository. Among them, 330 device integration bugs are identified. Furthermore, we attempt to answer the following four research questions through analyzing these iBugs.

- **RQ1 (Root cause):** What are the root causes of iBugs?
- **RQ2 (Fix)**: How do developers fix iBugs, and are there some common fix strategies?
- **RQ3 (Trigger condition):** What are the trigger conditions of iBugs?
- **RQ4 (Bug impact):** What impacts do these iBugs have?

Through our in-depth analysis of these iBugs, we obtain 23 atomic categories of root causes. For example, developers may not know that the established device authentication can expire, making devices inaccessible, and developers often generate unique device ID based on the device's MAC address, but some devices do not expose their MAC addresses. For iBugs under the same root cause, we extract their fix patterns and trigger conditions. We observe that more than half of the categories have frequent fix patterns, which means that most iBugs under the same root cause can be resolved by its associated fix pattern. For example, all iBugs caused by generating incorrect device information can be resolved by generating unique device ID. Based on the findings, we also offer some useful lessons for researchers and practitioners of smart home systems. For example, device integration developers should keep in mind that some devices can become invalid or expired, and they should set up some polling tasks to maintain the authenticated state. Smart home system designers should standardize the device life cycle and provide frameworks to assist developers developing device integrations. We have made our studied iBugs publicly available at https://github.com/tcse-iscas/iBugs.

We summarize our contributions as follows.

- We present the first systematically empirical study on iBugs in the most popular open source smart home system, Home Assistant.
- We provide a large-scale benchmark of iBugs in a smart home system, which can be used to evaluate the effectiveness of tools in combating iBugs.
- We make some takeaways to help developers implement a high-quality device integration and a highly extensible smart home system.

The rest of this paper is organized as follows. Section 2 briefs Home Assistant basics and its integration model. Section 3 presents our study methodology. Section 4 presents our empirical study results and findings. Section 5 discusses lessons learned from our study. Section 6 discusses related work and finally Section 7 concludes this paper.

2 HOME ASSISTANT

Home Assistant (HA) is an open source home automation software designed to be the central control system for smart home devices with a focus on local control and privacy [25]. The heart of Home Assistant is the **Core** module. HA Core interacts with users, smart devices and services, and schedules and processes events within the system. With the help of HA Core, users can easily control their smart devices and invoke services, such as turning on a light and checking the weather. HA Core can be extended with modules, termed as **integrations** in HA. Each integration is responsible for a specific domain within HA, especially interacting with external devices and services.

More specifically, there are four types of integration, which can listen for or trigger events, offer services, and maintain states. In particular, most integrations are responsible for interacting with various external devices and services, making them available in Home Assistant. For example, the *Philips Hue* integration can be extended from *light* integration, and it can be used to control the physical device.

In this work, we focus on this type of integrations and refer to them as device integrations hereafter. As mentioned above, many device integrations are contributed by developers in the open source smart home system community, and they evolve along with IoT device updates. However, it is challenging for developers to implement a high-quality device integration. They should apply proper communication protocols, handle device lifecycle management mechanisms, deal with errors and exception situations. Consequently, various iBugs can occur in the implemented device integrations.

To guide the analysis of iBugs, we build a general integration model of Home Assistant by investigating the existing device integration code and official documents. As shown in Figure 1, Home Assistant consists of two parts: Home Assistant Core (Figure 1a) and integrations (Figure 1b). For better understanding, an integration can be thought of as a simplified digital twin of a physical device. Through the analysis of the existing device integration code, we find that the device integration development needs to focus on four stages, which we call the four lifecycle functions. We will cover each of these stages in turn.

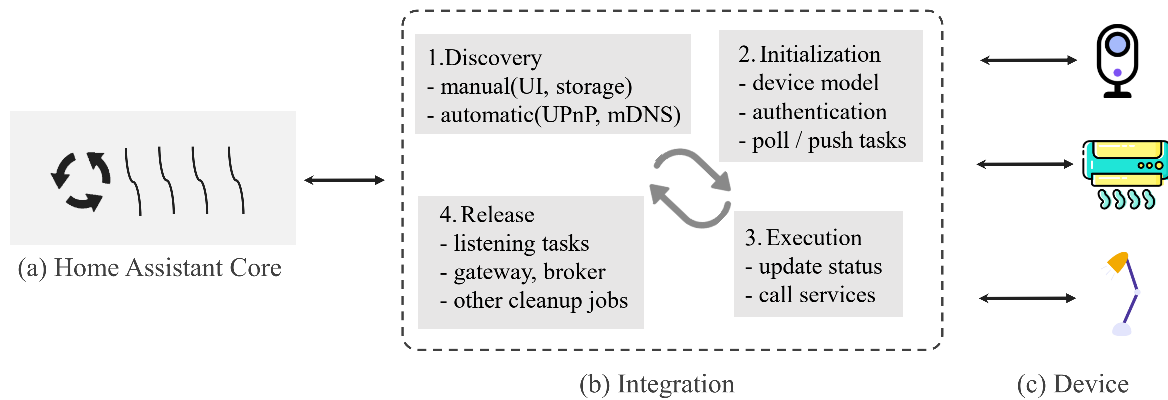

Figure 1: The integration model of a smart home device.

Discovery. In the discovery stage, the device integration is responsible for obtaining some device information, e.g., IP, port, username, password. The information, obtained manually or automatically, is used to connect or initialize the device in the system. In the manual way, developers provide the device information through the UI or configuration files. In the automatic way, some automatic discovery protocols, e.g., UPnP and mDNS, are provided by some third-parties, and developers can use them to implement the automatic discovery.

Initialization. The purpose of initialization is to make a device integration ready for managing the corresponding device. It usually includes three steps. First, devices are connected through device information like network addresses and device IDs. If required, the authentication and authorization are conducted. Second, the device integration establishes a mapping of devices from physical space to cyberspace for further operations. Third, the device integration sets polling or monitoring tasks to synchronize states between the physical device and the digital mapping in the system or some repeatable tasks.

Execution. A device integration is responsible for communicating with remote devices and external services via various protocols in execution. On the one hand, a device integration polls or monitors the target device's state and synchronizes the device's digital model to the perceived actual device state. On the other hand, a device integration schedules and conducts tasks that either change the remote device's properties or control the device to perform expected functions.

Release. When a device is no longer in use or malfunctions, the allocated resources for them should be released properly, such as terminating the monitoring process and cancel timers. For indirectly connected devices (e.g., through hardware gateways, and MQTT servers), the corresponding clean-up work should be required. There also exist some global shared states for inter-integration communication or used for reloading. These shared states should be correctly handled.

Device integrations should implement the full lifecycle functions and ensure each stage is carefully managed. Otherwise, integration bugs can occur.

3 METHODOLOGY

In this section, we present the methodology of our empirical study. In order to answer the four research questions, we conduct a comprehensive empirical investigation on 330 iBugs. In the following, we first explain why we choose Home Assistant as our target smart home system, and describe how we collect iBugs. We then present how these iBugs are analyzed. Finally, we discuss the threats to our study.

3.1 Target Smart Home System

We select the open source smart home system *Home Assistant* as the target system to conduct our empirical study considering its popularity and open source nature. Home Assistant is the most popular smart home project which has supported thousands of device integrations on GitHub. Meanwhile, it has an open source project in which issue reports and code repository are publicly available. This can greatly facilitate our study. We have also compared Home Assistant with another popular smart home systems, i.e., openHAB. The device integration model of these two typical systems are similar. We have summarized three common concepts as follows.

- *Device integration.* Developers can develop device integrations to extend the capability of systems.
- *Device monitor.* Both of the two systems provide device automation mechanism and unified interfaces for monitoring devices.
- *Device object.* Physical devices are mapped to virtual objects in the system, and the states and behaviors of the objects can be accessed and controlled within the system and reflect in the physical space.

The concerned difference is that Home Assistant has a special design, which separates and maintains the device SDKs from the device integrations. Each device SDK encapsulates the communication protocols and device capabilities. Developers can control the device through the device SDKs, e.g., turn on the light. Therefore, it is straightforward to analyze and locate iBugs in Home Assistant, and there is little noise from device related bugs in our study, such

as incorrect communication protocols and incorrect wrapper of device APIs.

3.2 Collecting iBugs

We manually went through issues and pull requests (PRs) of the Home Assistant repository [3] on GitHub to collect iBugs. First, we concentrated on the closed bug reports with the label "integration", the number of which is larger than ten thousand. Second, we selected issues in the second half of 2020 (2020.5-2020.12) . Next, we searched for each issue's associated merged pull request, which indicated that the issue was potentially a real bug. Then we manually read the issue report and discussions to screen out iBugs. Eventually, we have obtained 330 iBugs for further analysis.

3.3 Analyzing iBugs

The collected GitHub issues were manually analyzed by four authors following an open coding procedure [59]. We created a shared document containing five columns, i.e., issue ID, root cause, fix, impact, trigger. The authors need to label the last four columns. To explore the root causes of iBugs, we tried to answer the question "what caused this bug at the code level". Based on the stack traces and messages in the issue reports, we located the buggy code and figured out what caused the bugs. The fixes were obtained from the associated pull requests, and finally, we manually extracted the common patterns by abstracting the patches. The impacts and trigger conditions can be inferred by issue reports straightforwardly.

Each author independently labeled the document assigned to him by defining the range of issue IDs. The shared document can show all the created labels during the labeling process, and all authors can further use them. Such a choice can help us use consistent naming without introducing bias. Authors also discarded issues and PRs that can not represent a bug or be fully understood.

The labeling process is conducted iteratively. In each round, each author was assigned different issues from his previous round, and the authors discussed the problems they met. After each labeling iteration, all potential conflicts between authors were resolved. We continued this process until the labels reached a state of saturation where no new label appeared after three rounds.

Taxonomy Construction. We used a bottom-up approach [42] to construct the taxonomy. We first grouped the bugs according to the different dimensions based on the proposed integration model in Figure 1. Then we grouped labels that correspond to similar views into categories. After that, we constructed parent categories to extract common features further. Each sub-category should belong to its parent category. The whole process was discussed and decided by all authors through offline meetings.

The labeling and analysis of the 330 iBugs took about four months.

3.4 Threats to Validity

Similar to other bug studies [41, 46, 47, 51, 65], potential threats to our work are the representativeness of the studied system and bugs, the accuracy of our analysis methodology, and the replicability, reproducibility, generalizability of the empirical study.

Generalizability. Smart home systems aim to integrate diverse IoT physical devices into cyberspace for control. Both open-source (e.g., Home Assistant [6], openHAB [1], and KubeEdge [18]) and commercial systems (e.g., Samsung SmartThings [24], Google Assistant [19], and Huawei HiLink [20]) have many design choices in common. These systems provide integration components to control diverse IoT devices. To integrate devices into smart home systems, developers need to map the devices in the physical world into cyberspace. To manage device models, these systems adopt similar lifecycle management mechanisms, including device discovery, initialization, execution, and release. Therefore, these systems can suffer from similar kinds of bugs as Home Assistant.

Representativeness. Home Assistant is the most popular open source smart home system on GitHub. It offers good flexibility, user interface, and overall performance and updates frequently [4, 7, 21]. Compared to other open source smart home systems, Home Assistant has a more flourishing community, and the reported issues in its repository contain richer information. We consider all the issues reported in the studied time period, i.e., from 2020.5 to 2020.12, and carefully collect all iBugs. Since then, the design and architecture of Home Assistant stay stable. Therefore, Home Assistant can still suffer from similar integration bugs.

Studied bugs and analysis methodology. We carefully read the description, discussions, related source code and bug fix patches for each bug. Once the label changed during the study, all bugs were re-analyzed. All studied bugs have been discussed and confirmed by at least four authors. For bugs that we still labeled in conflict after the discussion, we asked developers to help us reach a consensus. If a bug still cannot be understood, we did not take it into consideration. We are very confident that all studied bugs are valid and have been thoroughly studied.

Replicability and reproducibility. This paper is a qualitative exploration study, and the results depend on the collected data and involved researchers. The labeling method strictly follows the open coding procedure. Besides, the empirical study's design and planning follow the structure suggested by Wohlin [35]. The replicability and reproducibility of qualitative exploration studies are common and recognized limitations. To reduce the threat, We have made our studied objects and analysis results publicly available at https://github.com/tcse-iscas/iBugs. Thus, other researchers are able to validate our study results easily.

4 STUDY RESULTS

This section presents our empirical study results and the answers to the research questions.

4.1 Root Cause (RQ1)

We present the result of our manual taxonomy of root causes in Figure 2, a hierarchical classification tree of depth four constructed by categories (gray rounded rectangles), sub-categories (gray rectangles) and atomic categories (white rectangles). For example, *authentication error (C.2)* is a sub-category containing two atomic categories: *incorrect authentication establishment (C.2.1)* and *forget to keep authentication (C.2.2)*. The number in the upper right corner denotes the number of bugs in each (atomic, sub-) category.

Device (A). This category covers 47 (14.2%) iBugs relating to device adaptation and malfunction, and contains two atomic categories.

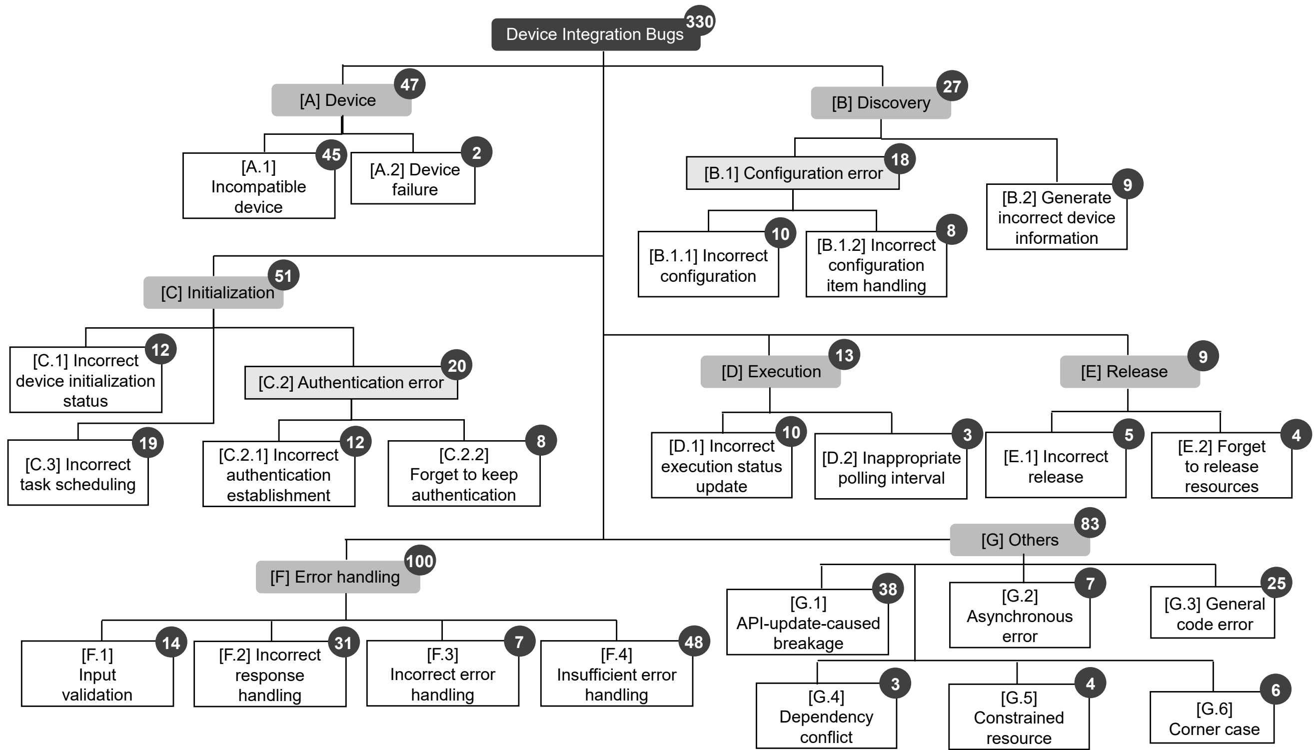

Figure 2: Taxonomy of root causes in device integration bugs

Incompatible device (A.1). This atomic category considers bugs caused by incompatible device adaptations and 45 iBugs belong to this category. Device integrations do not provide some abilities of the connected device or have not supported this device yet. Take issue #42921 [16] as an example, the latest version of Nest (SDM) for climate adds an "auto" mode. However, the user's device only supports "heat" and "off", and "auto" mode cannot cool down the room when the temperature is high. Since users do not know the device and the device integration well and the Home Assistant system usually does not give any clear error message, users are always confused by such bugs.

Device failure (A.2). Devices connecting to the system can malfunction and cause the corresponding device unavailable. However, these bugs in our study are rare and only have two cases, as users can easily perceive their device failures and will not report such bugs.

Finding 1: *13.6% (45/330) iBugs are caused by accessing incompatible devices without clear prompts.*
Implication 1: *The smart home system should give a clear prompt when a user encounters an incompatible device.*

Discovery (B). This category contains 27 bugs occurring during the device discovery stage. A device discovery requires device information for further device connection. However, developers can inadvertently provide incorrect information or configurations. In particular, this category is divided into three atomic categories further.

Configuration error (B.1). A device should be configured manually or automatically before its connection and use, and configuration errors can cause the device integration to fail. 18 iBugs belong to this category.

This sub-category has two atomic categories, i.e., incorrect configuration and incorrect configuration item handling.

- *Incorrect configuration (B.1.1).* Ten iBugs is caused by incorrect and incompatible configurations. Developers can provide incorrect parameters, redundant configurations and forget to set default configuration values.
- *Incorrect configuration item handling (B.1.2).* This category represents faults due to the negligence of developers. Developers do not take into account the configuration parameter information provided by the user when developing the integration, which means that all the configurations are hardcoded by the developer. More specifically, The user configures certain parameters such as the IP address of the device, but the developer does not use this configuration information. Eight iBugs belong to this category.

Generate incorrect device information (B.2). Nine iBugs are caused by the generation of incorrect device information. As mentioned above, the device information should be provided to the corresponding device integration. However, not all device information can be obtained from configurations directly, and some are obtained dynamically or after calculations. Generating wrong device information can cause iBugs. Take issue #39099 [11] as an example, the user wants to access surepetcare (The surepetcare allows people

to get information on Sure Petcare Connect Pet or Cat Flap) into the system and gets an error message: "`Platform surepetcare does not generate unique IDs`". Each device accessed into the smart home system should have its unique ID and the device integration is responsible for the ID generation. Therefore, the device integration should provide a unique ID for each device and avoid generating the duplicate ID. Interestingly, some developers use the MAC address as the unique ID, while some devices do not expose their MAC addresses through certain APIs. However, developers are not aware of this situation and can introduce iBugs.

Finding 2: *8.2% (27/330) iBugs are caused by configuration errors and generating incorrect device information. The main cause of generating incorrect device information is that the accessed device does not have a unique ID.*
Implication 2: *Developers should not hard code the configuration and make sure their configurations are correct and customizable. Developers should make sure the device ID is unique in the system during the device discovery phase.*

Initialization (C). In this group fall all the aspects relevant to iBugs raised during the device initialization. The device integrations are responsible for setting the initial state, authentication, and scheduling task of the device during the initialization stage. 51 iBugs occurred during the device initialization.

Incorrect device initialization state (C.1). A device integration requires ensuring that the state of the device reflected in the device integration or smart home system is consistent with that of the physical device. Device integrations can have problems of handling initial device state settings, such as forgetting to configure the state or setting a wrong state. 12 iBugs belong to this category.

Authentication error (C.2). Some devices require authentication for accessing them, and the corresponding device integration should provide such an authentication process, including the authentication establishment and maintenance. Developers should make sure that their authentication parameters and implemented logic are correct. Otherwise, an access can be denied or return an incorrect response. 20 iBugs belong to this category. The following two atomic categories explain the main causes of such iBugs.

- *Incorrect authentication establishment (C.2.1).* The device authentication process is managed by the corresponding device integration. Home Assistant Core has already configured the standard auth providers, and developers need to provide the identity information. However, the identity information provided by some developers can be incorrect, which leads to authentication failures. Furthermore, some developers even implement wrong authentication logic in their device integrations, which can also lead to authentication failures. 12 iBugs are caused by incorrect authentication establishment.
- *Forget to keep authentication (C.2.2).* The authentications of some devices can become invalid or expire. Therefore, developers need to ensure that the device integrations they implemented always have valid authorizations. Take issue #42947 [17] as an example, it reports that Nest (SDM) Google Pubsub stops working after 60 minutes with observed non-recoverable stream error 401 request. Restarting Home Assistant with reloading the Nest integration temporarily resolves the issue in 60 minutes. After analysis, we find that the authentication of the device has expired after 60 minutes, making the device unavailable. Eight iBugs belong to this category.

Incorrect task scheduling (C.3). A device integration needs to schedule tasks for an initialized device. The main thread is unsuitable for carrying out some time-consuming tasks, such as database access, as it will be blocked. Instead, other threads should be delegated to execute such tasks asynchronously to avoid blocking the main thread and degrading the system performance. However, some device integrations do not follow such practical guidelines and set heavy tasks into the main thread, blocking the system. 19 iBugs belong to this category.

Finding 3: *5.1% (17) iBugs are caused by forgetting to maintain the establishment of device authentication and scheduling heavy tasks inappropriately.*
Implication 3: *Smart home systems should inform device integration developers of the existence of an expired device authentication.*

Execution (D). At runtime, the smart home system communicates with connected devices and controls them via the device integrations for updating their states. This category contains 13 iBugs related to device execution state update and polling interval settings.

Incorrect execution state update (D.1). During a device execution, the system triggers the device to change its state and meanwhile changes the properties/state of the device reflected in the system synchronously. However, device integrations can forget to update the reflected state or configure incorrect update information, which leads to the inconsistency between the actual device state and the reflected one in the system. Ten iBugs are caused by this.

Inappropriate polling interval (D.2). Device integrations are responsible for setting up polling tasks to synchronize system state and device state regularly. The polling interval can be inappropriate and make the state of a device in the system inconsistent with the real state. We find three iBugs are caused by inappropriate polling intervals.

Finding 4: *When the synchronization interval of device state is not appropriately set, there may be inconsistencies between the actual state and that shown in the system.*
Implication 4: *When an operation triggers a device or system state change, the state should be synchronized in time.*

Release (E). Device integrations are responsible for cleaning up when devices and services are released or unloaded, e.g., terminating state synchronization, stopping device discovery process. For some device integrations, there exist some shared variables for communication with other integrations. These shared variables also should be released. *Incorrect release (E.1)* will waste system resources or cause unexpected system behaviors. We have also found four device integrations *forgetting to release resources (E.2)*. Take issue #42781 [15] as an example, the device integration *RFXtrx*

remains in an automatic add state after removing the corresponding device. After analyzing the associated PR, we found that the dispatches were not cleaned up when unloading *RFXtrx*. As a result, without restarting Home Assistant, these residual dispatches will keep registering new devices into the system.

Finding 5: *Device integrations can forget to release resources or perform an incorrect release.*
Implication 5: *The system should provide some utilities to help the resource release process or introduce an automatic garbage collection mechanism.*

Error handling (F). Error handling is the largest category in the taxonomy, and it includes a wide range of bugs. Error handling refers to the response and recovery procedures from error conditions present in a software application [5]. Error handling helps keep the system operating normally and prompts error information. In fact, many applications face numerous design challenges when considering error-handling techniques [5]. 30.3% iBugs are of this category and can be further divided into four atomic categories.

Input validation (F.1). It is often assumed that the obtained data, especially some data returned from third-party requests, are correct and expected. Device integrations sometimes directly use the data without any validations. However, the data can be exceptional, e.g., `undefined`, `null`. 14 iBugs are caused by not applying the input validation.

Incorrect response handling (F.2). This category differs from the previous one because the obtained data is correct and legal but handled incorrectly. These iBugs are mainly related to the logic errors of data processing, and we have found 31 cases.

Incorrect error handling (F.3). Although some device integrations take errors into account, they do not handle them correctly. For the case where some services need to call a third-party API, the failure of the API call is considered, and a retry mechanism is implemented in the device integration to handle such exceptions. However, the device integration may repeatedly try without checking the correctness of the returned results and traps in an infinite retry loop.

Insufficient error handling (F.4). Bugs of this type differ from those in the prior atomic category as some errors are unknown and not be handled at all. It is often difficult for developers to comprehensively know the situations in the application, and corner cases are often not considered.

- *Network recovery.* After the network is restored from disconnection, the device integrations do not consider restoring the task schedule and device state. For example, the device integration needs to perform connection authentication again and continue the tasks which are not completed.
- *Non-existent device.* The device integration wants to control the device but does not consider whether the device is already connected to the system, which will throw an error like "`Device does not exist.`"
- *Incorrect retry & timeout setting.* A failure can occur whenever one device integration calls another service or integration. Various factors, such as target servers, networks, load balance, and even system errors, can cause failures. Some device integrations consider these exceptions and apply some methods, i.e., timeout and retry mechanism, to make them more robust. The system can handle failures when it tries to connect to a service or network resource by retrying a failed operation. Timeout allows for more efficient usage of limited resources without requiring additional resources. However, device integrations can apply incorrect retry and timeout settings. For example, issue #42687 [14] reported that the integration *RFXtrx* stopped working since upgrading the system. After upgrading the system, some resources are no longer cleaned up, and it will take more time to establish the *RFXtrx* connection. The fix strategy is to increase the timeout on connection from 5 to 30 seconds.

Finding 6: *Device integrations do not handle exceptions well, frequently causing iBugs. Developers may ignore some abnormal scenarios, or wrongly handle errors.*
Implication 6: *The handling of abnormal scenarios is necessary to improve the system's robustness, such as strengthening input validation and error handling.*

Others (G). This category contains the bugs whose root causes cannot be classified into the above categories. This is the second largest (83/330) category in the taxonomy, and it includes a wide range of bugs also appear in some other systems.

API-update-caused breakage (G.1). Updates and changes in software development are inevitable. As a consequence, clients are compelled to update, and thus, benefit from the available API improvements. However, some of these API changes can break contracts established previously, resulting in compilation errors and behavioral changes. 38 (11.5%) iBugs are caused by API updates.

Asynchronous error (G.2). Data are processed and accessed in an unexpected order due to the asynchronous loading of HA integrations. When the system is initialized, the loading sequence of HA integrations is asynchronous. When a HA integration is required to provide corresponding functions or information, the HA integration may not be loaded yet. Therefore, the normal function of the integration cannot be realized. Take issue #42188 [13] as an example, when the doorbird sends an HTTP(S) call to Home Assistant, Home Assistant's response is 500 internal service error. After analysis, we found that the corresponding device integration is not fully set up yet by the time doorbird sends the request.

General code error (G.3). 25 (6.6%) iBugs are similar to those in other software systems, including illegal argument, key not found, null reference, syntax error, undefined object. We no longer explain this type of iBug in detail.

Dependency conflict (G.4). The execution environment of Home Assistant is based on Python. Dependency conflicts occur when different Python packages have the same dependency but depend on different and incompatible versions of that shared package. Because only a single version of a dependency is permitted in the Python project's environment. Since the development of device integrations is often based on third-party libraries, such problems can also cause device integrations to fail to operate normally. We only found 3 such bugs because such bugs are often found during the testing phase, and developers will naturally solve them.

Constrained resource (G.5). Four bugs are caused by constrained resource. We have found several situations. First, there are too many concurrent requests at the same time, and the system cannot handle

Table 1: Frequent fix patterns for each root cause

Dimension	Root cause	Fix pattern description	#Issues
[A] Device (47)	[A.1] Incompatible device (45)	Improve the device integration and adapt to incompatible modules of the device.	24
		Prompt a warning to indicate that the device is unsupported contemporarily, waiting for the corresponding integration.	21
	[A.2] Device failure (2)	Use new device instead.	2
[B] Discovery (27)	[B.1.1] Incorrect configuration (10)	Add default configurations.	3
		Fix corresponding configurations.	6
	[B.1.2] Incorrect configuration handling (8)	Correctly handle conflicting parameters.	6
	[B.2] Generate incorrect device information (9)	Generate unique ID.	9
[C] Initialization (51)	[C.1] Incorrect device initialization state (12)	Fix corresponding state.	8
		After restart, load the original data from the cache.	4
	[C.2.1] Incorrect authentication establishment (12)	Update correct authentication information.	3
		Correct authentication logic.	5
	[C.2.2] Forget to keep authentication (8)	Add watchdog or set timer to maintain the authentication.	8
	[C.3] Incorrect task scheduling (19)	Remove heavy tasks from the event loop.	13
		Modify synchronous (asynchronous) tasks to asynchronous (synchronous) tasks.	6
[D] Execution (13)	[D.1] Incorrect execution state update (10)	Correct state processing logic.	5
	[D.2] Inappropriate polling interval (3)	Adjust polling interval.	3
[E] Release (9)	[E.1] Incorrect release (5)	Release the wrong part correctly.	5
	[E.2] Forget to release resources (4)	Release relevant resources.	4
[F] Error handling (100)	[F.1] Input validation (14)	Check before use.	14
	[F.3] Incorrect error handling (31)	Add retry & timeout setting.	9
	[F.4] Insufficient error handling (31)	Adjust retry & timeout setting.	9
[G] Others (97)	[G.1] API-update-caused breakage (38)	Fix API usage.	38
	[G.2] Asynchronous errors (7)	Waiting for synchronization.	7
	[G.3] General code error (25)	Fix corresponding errors, including illegal argument, null reference, syntax error, undefined object.	25
	[G.4] Dependency conflict (3)	Choose appropriate third-party library versions.	3
Total (330)			**240**

them. Second, the connection pool is exhausted, and subsequent requests cannot be processed. Third, there is a limit to the number of third-party API calls. These bugs are mainly caused by constrained system resource and cloud resource.

Finding 7: *Some common bugs are also raised in device integrations, i.e., API-update-caused breakage, asynchronous errors, general code errors, dependency conflicts and constrained resource.*
Implication 7: *Existing bug detection and fixing techniques can also be used on iBugs to improve the reliability and robustness of device integrations.*

4.2 Fix Pattern (RQ2)

Fixes of device integration bugs are highly related to their root causes. For each root cause, we summarize its fix patterns in Table 1, which depict the outlines to fix different types of iBugs. The columns "Dimension" and "Root cause" are derived from the results shown in Figure 2, and the number in parentheses is the number of iBugs in that category. The column "Fix pattern description" briefly presents the fix pattern and the last column "#Issues" is the number of issues fixed by that pattern. For example, out of the 47 bugs in *device*, 45 bugs are caused by *incompatible device*, and they can be fixed with two patterns, i.e., adapting an existing device integration to support the device (24 cases) or developing a new device integration (21 cases). We propose some general fix guidance for some root causes that are difficult to summarize their fix patterns. Note that the bugs which are not included in Table 1 need case-by-case fixes.

In general, we observe complicated crossovers between different fix patterns and root causes, confirming that fixing device integration bug is a challenging task.

Relationship between fix patterns and root causes. From Table 1, we observe that some pairs of fix patterns and root causes frequently co-occurring, which indicates that most iBugs caused by the same root cause can be resolved by its associated fix pattern.

We regard such combination of fix pattern and root cause as a pair. These pairs are useful for bug fixing. More specifically, when the root cause of an iBug is diagnosed, the corresponding paired fixes can be taken to fix it manually or automatically.

In Table 1, we find that for 15 atomic categories, more than half of its iBugs can be fixed by a specific pattern. For example, all iBugs caused by *generating incorrect device information (B.2)* are resolved by generating unique ID. 13 out of 19 class *incorrect task scheduling (C.3)* are fixed by removing the heavy task. All iBugs caused by *Forget to keep authentication (C.2.2)* can be fixed by adding watchdogs or set timer to maintain the authentication. Take issue #37134 [10] as an example, the session of the integration *tile* was expired after 6 hours and did not auto-renew, making the entities unavailable. The developers solved this issue by monitoring session expiration errors with the code below.

```
1  + except SessionExpiredError:
2  +     LOGGER.info("Tile session expired; creating a new
        one")
3  +     await client.async_init()
```

In summary, these frequent pairs can derive heuristic strategies for iBugs resolution.

Finding 8: *15 out of 23 atomic categories have frequent fix patterns, such as generating unique ID to resolve the incorrect device information and removing heavy tasks to fix the incorrect task scheduling.*
Implication 8: *Researchers can develop automated fixing tools for iBugs based on our fix patterns.*

Case-by-case fix. In spite of the categories and their co-occurring fix patterns, 90 iBugs do not have common fix patterns, and they need case-by-case fix strategies. Most (68) of them are related to the category *error handling (F)*. More specifically, iBugs caused by *incorrect response handling (F.2)* account for almost half (31). Device integrations use or process data incorrectly, causing the system behave abnormally. Fixes of these iBugs often require analysis of the specific scenarios. Another portion of iBugs is caused by incorrect or insufficient error handling. This observation indicates that device integration developers need comprehensive knowledge and experience for handling various errors and exceptions.

4.3 Trigger Condition (RQ3)

We have summarized six types of trigger conditions from 253 iBugs. We do not cover all iBugs as the information included in some issue reports and associated pull requests is insufficient to identify their trigger conditions.

Loading device integration. 93 iBugs occurred while loading device integrations without any external actions. These iBugs involve the discovery and initialization process. The category *incompatible device (A.1)* also accounted for a lot.

Reloading device integration. Reloading device integrations will trigger some lifecycle functions of the device integration, i.e., release and initialization. All iBugs caused by *incorrect release (E.1)* and 7 out of 12 iBugs of the category *incorrect device initialization state (C.1)* can be triggered by reloading device integrations. This trigger conditions totally involve 19 iBugs.

Invoking device service. 110 iBugs are raised while invoking device services. All iBugs caused by *incorrect task scheduling (C.3)* and *forgetting to keep authentication (C.2.2)* can be triggered under this condition. Most iBugs related to *error handling (F)* can also be found by invoking device services.

Frequent requests. Frequent requests can lead to iBugs related to *constrained resource(G.5)* and *device failure (A.2)*. It is challenging for device integration developers to recognize these iBugs. The main difficulty is which request needs to be replayed at a high frequency. Seven iBugs are triggered by frequent requests.

Network interruption and recovery. Ten iBugs can be triggered by network interruption and recovery. We find that few device integrations take into account the recovery of the network interruptions. When there is no response, users usually wait for a while or call the services again, and such iBugs are covered up. The actual number of them would be larger than that we count.

Network delay. 14 iBugs are triggered by network delay. The unstable network connection is common in many environments, and simulation of such a condition is important to reveal or reproduce such bugs.

Finding 9: *Most iBugs (253/330) can be triggered with six types of trigger conditions. We can perform systematically testing on these trigger conditions, e.g., reloading device integrations and injecting network interruptions.*

4.4 Impact (RQ4)

iBugs can directly affect user experience when using smart home systems and bring inconveniences and even dangers to users. We find five main impacts that both developers and users should be paid attention to.

Unavailable device. 174 iBugs make devices unavailable, and users may find that the expected devices or their properties/services are missing. Many aspects can cause a device to be unavailable, such as *incorrect response handling (F.2)* and *forgetting to keep authentication (C2.2)*.

Incorrect device state. 28 iBugs result in inconsistencies between the reflected device state by the system and the actual state. In issue#36767 [9], the developers set an inappropriate polling interval of synchronizing state, and the user observed an inconsistency between the actual state and the digital model within the system.

Unexpected device behavior. 12 iBugs lead to unexpected device behavior. In issue #42921 [16], the latest version of Nest (SDM) for climate adds the "auto" mode. However, the developers' device only supports "heat" and "off", and the "auto" mode cannot cool down the room when the temperature is high.

Unexpected system behavior. 12 iBugs result in unexpected system behavior, i.e., the devices are incorrectly managed. In issue #42781 [15], after the user unloads an integration, the system continues to add new devices belonging to this integration.

Slow response. 18 iBugs have abnormal CPU or memory usage and the system cannot respond to users' operations in time. This is very annoying when users expect the system to respond immediately. For example, developers do heavy IO operations in the main thread, which blocks other tasks. The infinite loop for retry also makes abnormal CPU and memory usage.

Finding 10: *iBugs can cause serious consequences, e.g., device unavailable, unexpected device and system behaviors, more than half of iBugs (174/330) make devices unavailable.*

5 LESSONS LEARNED

As shown in Section 4.4, iBugs can seriously affect the user experience of smart home systems. A high-quality device integration is of great significance for the reliability of smart home systems. In this section, we will discuss the pitfalls that device integration developers and smart home system designers should focus on, and the way to fight against iBugs.

5.1 Lessons for Device Integration Developers

Device integration developers should carefully handle all the four stages in device lifecycle management in Figure 1, including handling the various errors that may occur during system and device running.

Generate unique device ID. Each device connecting to the smart home system should have a unique ID. It is not recommended for developers directly using the MAC address as the device ID. We have found that some devices prevent the system from obtaining their MAC addresses, making the ID assignments fail (Finding 2). Developers also need to avoid generating duplicate IDs, e.g., maintain a list of already generated IDs and check new IDs against those in the list.

Handle configurations. Device information and configurations are necessary for device connections. The information is configured manually and automatically. Auto-discovery protocols can provide certain device information, e.g., IP, port. End-users can configure and change configurations through UIs or configuration files. Therefore, developers should not hard code configurations (Implication 2). There are differences between the developer environment and the user environment. Some configurations, e.g., timeout settings, password, should be configurable for users, and a device integration has to recognize the modifications and apply changes.

Maintain authentication. Finding 3 shows that the authentication of some devices can become invalid or expired. Therefore, developers should ensure each device is always in an authenticated state. They can set up some polling tasks to confirm the authentication state of each device and re-authenticate if expired.

Keep the main thread lightweight. The time-consuming tasks, such as some heavy IO operations, should not occupy the main thread, where runs the four stages of an integration lifecycle. Otherwise, the system performance may degrade significantly (Finding 3). Developers should not put lots of IO and computing tasks at the initialization stage. If certain tasks are necessary, developers can set these tasks asynchronously. In Home Assistant, they only need to add field *async* before the function.

Validate data. The responses of requests may not be the expected ones due to some errors (Finding 6), and developers need to validate the returned data before using them. More specifically, developers can check whether a response is empty and whether the response holds the desired fields for further processing.

Recover state automatically. Developers should be aware that there are various unexpected scenarios, e.g., system restart, network interruption and recovery. They should handle these cases to ensure the consistency and availability of the system. They can also design methods to recover from exceptions. For example, they can use the buffer pool to store states and scheduled tasks, and recover systems based on the buffer pool when exceptions occur.

5.2 Lessons for Smart Home System Designers

The development of a smart home system faces various challenges and a reliable smart home system face many challenges. We put forward two suggestions to strengthen the system reliability.

Standardize the device life cycle. Home Assistant does not give a unified standard and framework for device integration developers to implement their integrations. The integration model we proposed in Figure 1 can be used as the guideline. Smart home systems can apply this model and provide corresponding functions for developers to manage the development process, such as establishing device connections, processing requests by performing corresponding functions. A native implementation of the integration model can greatly reduce the difficulty of developing device integrations.

Reasonably allocate system resources. Home Assistant does not limit the resources requested by a device. When a device requests resources endlessly, the system may run out of memory, CPU times, and even crash. Take issue #41721 [12] as an example, the integration ceaselessly attempts to establish authentication. This occupies a thread and the CPU usage of the system increases continuously. In this case, having a resource management mechanism would be very helpful, e.g., aborting the task which repeats some operations and takes up too many resources.

5.3 Detecting iBugs

iBugs can make accessed devices unavailable and lead unexpected behaviors (Finding 10). Thus, resolving iBugs is of great significance for the reliability of the smart home system. Existing IoT testing approaches, e.g., device simulators [8] and emulators [30, 50], unit testing frameworks [2] mainly focus on model checking and testing and do not understand the root causes behind these bugs. None of them focuses on iBugs. Our study reveals that iBugs can be summarized into 23 atomic categories. These bug categories provide new light and guidance for iBug detection based on program analysis.

Some static and dynamic program analysis techniques can be used to detect iBugs. More specifically, some API testing methods [38, 40] can apply to detect the device ID generation functions to check whether the ID is unique. iBugs caused by class *forget to keep authentication (C.2.2)* and *incorrect task scheduling (C.3)* can be detected by analyzing code statically. Developers can locate the authentication code and then determine whether the code is assigned as a polling task. Developers can identify some code patterns of heavy tasks, e.g., accessing the database for large amounts of data. They can detect whether the initialization stage includes these patterns. Some model-based testing methods also can be used to test device integration. As mentioned in Section 3.1, Home Assistant separates the device SDKs from the device integration. Based on the device SDKs, we can conduct testing by invoking certain APIs. We do not list some solutions of common iBugs, e.g., breaking changes, general code errors and dependency conflict. In addition, we also propose some potential approaches for iBug detection.

State inconsistency guided detection. The inconsistent state between the physical device and the state present in the system indicates an iBug (Finding 4). These iBugs lie in the initialization and execution stages. State inconsistency guided detection approaches can be applied to detect these iBugs. To identify inconsistencies, we can compare the device state with system state after device initialization and executions. We can also generate device execution trace, and then judge whether the final state is consistent.

Error injection testing. If device integrations do not handle errors well, iBugs appear (Finding 6). Therefore, we can inject exceptions within the system to see whether device integrations can handle them well. Some exceptions that we can inject are summarized from *Error handling (F)*. (1) Incorrect response result. We can provide an incorrect response or empty result to check whether device integrations apply input validations. (2) Network delay. We can inject network delay to check whether device integrations apply retry or timeout settings. (3) Network recovery. After the network is restored from discussion, device integrations are responsible for restoring the device state and task schedule. We can inject network interruption and recovery to test this.

Non-deterministic analysis in device integrations. Home Assistant adopts an event-driven architecture and supports many asynchronous APIs. Home Assistant loads device integrations asynchronously, and the heavy tasks in some device integrations (Finding 3) can affect the loading sequence, causing non-deterministic iBugs. Little literature has focused on this aspect in IoT systems. To automatically detect non-deterministic iBugs, we need to build a clear model about the event-driven architecture, device lifecycle management, asynchronous APIs, heavy tasks in Home Assistant.

6 RELATED WORK

In this section, we discuss related works that are close to our work. We review the related work from three aspects, analyzing IoT bugs, testing IoT systems, and general bug studies.

IoT bugs investigation. A few previous studies have inspected the bugs and design flaws in the IoT systems and provided some preliminary insights. Duc et al. [37] examine the main failure patterns of smart home systems. They focus on hardware-related failures, such as wireless link loss, battery damage, power outage. Chen et al. [34] model the IoT system with four layers: application, storage, communication, data. Based on the model, they analyze the fault symptoms and provide maintenance suggestions. Recently, a study [53] conducted by Makhshari et al. firstly provides a generalized and systematic overview of bugs in the IoT system. They present a taxonomy of the bugs in IoT system and the challenges of developing IoT systems. However, none of them look into the integration part of a smart home system. The literature lacks the knowledge to develop a good integration, which is essential to make the smart home system more reliable.

IoT system testing. A number of approaches have been proposed to test IoT systems. Security issues are frequently discussed. A number of papers raise the concerns related to security issues [29, 64]. A related topic, user's privacy and trust is also being frequently discussed. Sajid et al. [58] first discussed the data privacy in IoT systems and proposed the future research directions. Several system testing research also span to the IoT systems, e.g., model-based testing [27, 54], test cases generation [44, 45], and testing frameworks [57, 61]. However, these studies are conducted to find security issues. It is unknown whether they are helpful for detecting iBugs in smart home systems. We attempt to figure out the requirements in testing iBugs and help develop effective testing methods.

Empirical bug studies. Empirical bug studies play an important role in understanding the aspect of software reliability. These studies often can help to characterize bugs in the target systems and provide useful guidance for future works. For example, Lu et al. [52] conduct a comprehensive study on real world concurrency bug characteristics. Their work opens up a new light in combating concurrency bugs, such as concurrency bug detection [36, 63], bug fixing [48]. Gunawi et al. [43] study 3000+ issues in cloud systems. Their study brings new insights on some of the most vexing problems in cloud systems. Ocariza et al. [55] conduct an empirical study on the client-side JavaScript bugs and find that most bugs are DOM-related. This promotes some research [56, 62] on investigating DOM related JavaScript bugs. However, no empirical studies are performed on integration bugs in smart home systems.

7 CONCLUSION

Smart home systems allow developers to integrate diverse smart IoT devices. Community developers implement various device integrations, but few detailed documents are available to understand the bugs in them. To fill this gap, we conduct the first empirical study on 330 iBugs and analyze their root causes, trigger conditions, impacts, and fixes. We obtain many interesting findings and insights for researchers and practitioners of open source smart home systems. We believe our study can open up new research directions in combating iBugs.

8 DATA AVAILABILITY

The dataset of our paper is available at Zenodo [26].

ACKNOWLEDGEMENTS

This work was partially supported by National Natural Science Foundation of China (U20A6003, 62072444, 61732019), Frontier Science Project of Chinese Academy of Sciences (QYZDJ-SSW-JSC036), and Youth Innovation Promotion Association at Chinese Academy of Sciences (2018142).

REFERENCES

[1] 2012. *OpenHAB.* Retrieved June 23, 2021 from https://www.openhab.org/
[2] 2013. *Arduinounit.* Retrieved Jan 1, 2022 from https://github.com/mmurdoch/arduinounit
[3] 2013. *Github: Home Assistant.* Retrieved July 19, 2021 from https://github.com/home-assistant/core
[4] 2018. *Best of Open Source Smart Home: Home Assistant vs OpenHAB.* Retrieved June 23, 2021 from https://smarthome.university/your-smart-home-platform-home-assistant-vs-openhab/
[5] 2020. *Error Handling.* Retrieved June 23, 2021 from https://www.techopedia.com/definition/16626/error-handling
[6] 2020. *Home Assistant Homepage.* Retrieved June 23, 2021 from https://www.home-assistant.io/
[7] 2020. *Home Assistant vs OpenHAB – Which one is better?* Retrieved June 23, 2021 from https://everythingsmarthome.co.uk/home-assistant/home-assistant-vs-openhab-which-one-is-better/
[8] 2020. *IOTIFITY.* Retrieved Jan 10, 2022 from https://iotify.io/

[9] 2020. *Issue 36767: Inconsistency Status*. Retrieved June 23, 2021 from https://github.com/home-assistant/core/issues/36767

[10] 2020. *Issue 37134: Tile Integration: Session expiring and not renewing*. Retrieved Jun 26, 2020 from https://github.com/home-assistant/core/issues/34134

[11] 2020. *Issue 39099: Platform surepetcare does not generate unique IDs*. Retrieved June 23, 2021 from https://github.com/home-assistant/core/issues/39099

[12] 2020. *Issue 41721: Reconfigure Integration gets triggered multiple times*. Retrieved June 23, 2021 from https://github.com/home-assistant/core/issues/41721

[13] 2020. *Issue 42188: [Doorbird] API causing 500 Internal Service Error*. Retrieved June 23, 2021 from https://github.com/home-assistant/core/issues/42188

[14] 2020. *Issue 42687: RFXtrx serial error*. Retrieved June 23, 2021 from https://github.com/home-assistant/core/issues/42687

[15] 2020. *Issue 42781: RFXtrx integration remains in automatic add state*. Retrieved June 23, 2021 from https://github.com/home-assistant/core/issues/42781

[16] 2020. *Issue 42921: Nest (SDM) ECO Mode adds temperature range to Climate which is not supported by Thermostat*. Retrieved June 23, 2021 from https://github.com/home-assistant/core/issues/42921

[17] 2020. *Issue 42947: Nest (SDM) Google Pubsub stops working after 60 mins*. Retrieved June 23, 2021 from https://github.com/home-assistant/core/issues/42947

[18] 2020. *A Kubernetes Native Edge Computing Framework*. Retrieved Jan 10, 2022 from https://kubeedge.io/en/

[19] 2021. *Google Assistant*. Retrieved Aug 10, 2021 from https://assistant.google.com/smart-home/

[20] 2021. *HuaWei HiLink*. Retrieved Aug 10, 2021 from https://iot.hilink.huawei.com/

[21] 2021. *OpenHAB Vs Home Assistant: Detailed Comparison By An Expert in 2021*. Retrieved June 23, 2021 from https://purdylounge.com/openhab-vs-home-assistant/

[22] 2021. *Smart Home*. Retrieved July 16, 2021 from https://www.investopedia.com/terms/s/smart-home.asp

[23] 2021. *Smart Home Market*. Retrieved Aug 10, 2021 from https://www.marketsandmarkets.com/Market-Reports/smart-homes-and-assisted-living-advanced-technologie-and-global-market-121.html

[24] 2021. *SmartThings*. Retrieved Aug 10, 2021 from https://www.smartthings.com/

[25] 2021. *Wiki: Home Assistant*. Retrieved July 19, 2021 from https://en.wikipedia.org/wiki/Home_Assistant

[26] 2022. *ISSTA 22 Artifact for "Understanding Device Integration Bugs in Smart Home System"*. Retrieved June 28, 2022 from https://doi.org/10.5281/zenodo.6481927

[27] Abbas Ahmad, Fabrice Bouquet, Elizabeta Fourneret, Franck Le Gall, and Bruno Legeard. 2016. Model-based Testing as a Service for IoT Platforms. In *Proceedings of International Symposium on Leveraging Applications of Formal Methods (ISoLA)*. Springer, 727–742.

[28] Omar Alrawi, Chaz Lever, Manos Antonakakis, and Fabian Monrose. 2019. SoK: Security Evaluation of Home-Based IoT Deployments. In *Proceedings of IEEE Symposium on Security and Privacy (SP)*. 1362–1380.

[29] Elisa Bertino, Kim-Kwang Raymond Choo, Dimitrios Georgakopolous, and Surya Nepal. 2016. Internet of Things (IoT): Smart and Secure Service Delivery. *ACM Transaction on Internet Technology* 16, 4 (2016).

[30] Giacomo Brambilla, Marco Picone, Simone Cirani, Michele Amoretti, and Francesco Zanichelli. 2014. A Simulation Platform for Large-Scale Internet of Things Scenarios in Urban Environments. In *Proceedings of International Conference on IoT in Urban Space (URB-IOT)*. 50–55.

[31] Z. Berkay Celik, Leonardo Babun, Amit K. Sikder, Hidayet Aksu, Gang Tan, Patrick McDaniel, and A. Selcuk Uluagac. 2018. Sensitive Information Tracking in Commodity IoT. In *Proceedings of USENIX Security Symposium (USENIX Security)*. 1687–1704.

[32] Z. Berkay Celik, Earlence Fernandes, Eric Pauley, Gang Tan, and Patrick McDaniel. 2019. Program Analysis of Commodity IoT Applications for Security and Privacy: Challenges and Opportunities. *Comput. Surveys* 52, 4 (2019), 30 pages.

[33] Z. Berkay Celik, Patrick McDaniel, and Gang Tan. 2018. SOTERIA: Automated IoT Safety and Security Analysis. In *Proceedings of USENIX Conference on Usenix Annual Technical Conference (USENIX ATC)*. 147–158.

[34] Yingyi Chen, Zhumi Zhen, Huihui Yu, and Jing Xu. 2017. Application of Fault Tree Analysis and Fuzzy Neural Networks to Fault Diagnosis in the Internet of Things (IoT) for Aquaculture. *Sensors* 17, 1 (2017), 153.

[35] Wohlin Claes, Per Runeson, Martin Höst, Ohlsson Björn Regnell, and Anders Wesslén. 2012. *Experimentation in Software Engineering*. Springer Science & Business Media.

[36] Dongdong Deng, Wei Zhang, and Shan Lu. 2013. Efficient Concurrency Bug Detection across Inputs. *Acm Sigplan Notices* 48, 10 (2013), 785–802.

[37] Anh Nguyen Duc, Ronald Jabangwe, Pangkaj Paul, and Pekka Abrahamsson. 2017. Security Challenges in IoT Development: A Software Engineering Perspective. In *Proceedings of XP Scientific Workshops (XP)*. 1–5.

[38] Hamza Ed-Douibi, Javier Luis Cánovas Izquierdo, and Jordi Cabot. 2018. Automatic Generation of Test Cases for REST APIs: A Specification-Based Approach. In *Proceedings of International Enterprise Distributed Object Computing Conference (EDOC)*. 181–190.

[39] Earlence Fernandes, Jaeyeon Jung, and Atul Prakash. 2019. Security Analysis of Emerging Smart Home Applications. In *Proceedings of IEEE Symposium on Security and Privacy (SP)*. 636–654.

[40] Tobias Fertig and Peter Braun. 2015. Model-Driven Testing of RESTful APIs. In *Proceedings of International Conference on World Wide Web (WWW)*. 1497–1502.

[41] Yu Gao, Wensheng Dou, Feng Qin, Chushu Gao, Dong Wang, Jun Wei, Ruirui Huang, Li Zhou, and Yongming Wu. 2018. An Empirical Study on Crash Recovery Bugs in Large-Scale Distributed Systems. In *Proceedings of ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 539–550.

[42] Vijayaraghavan Giri and Cem Kaner. 2003. Bug Taxonomies: Use them to Generate Better Tests. In *Software Testing Analysis and Review*. 1–40.

[43] Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, Agung Laksono, Jeffrey F. Lukman, Vincentius Martin, and Anang D. Satria. 2014. What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. In *Proceedings of ACM Symposium on Cloud Computing (SOCC)*. 1–14.

[44] Lorena Gutiérrez-Madroñal, Antonio García-Domínguez, and Inmaculada Medina-Bulo. 2019. Evolutionary Mutation Testing for IoT with Recorded and Generated Events. *Software: Practice and Experience* 49, 4 (2019), 640–672.

[45] Lorena Gutiérrez-Madroñal, Inmaculada Medina-Bulo, and Juan José Domínguez-Jiménez. 2018. IoT–TEG: Test Event Generator System. *Journal of Systems and Software (JSS)* 137 (2018), 784–803.

[46] Huaxun Huang, Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2018. Understanding and Detecting Callback Compatibility Issues for Android Applications. In *Proceedings of ACM/IEEE International Conference on Automated Software Engineering (ASE)*. 532–542.

[47] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. Understanding and Detecting Real-World Performance Bugs. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 77–88.

[48] Guoliang Jin, Wei Zhang, Dongdong Deng, Ben Liblit, and Shan Lu. 2012. Automated Concurrency-Bug Fixing. In *Proceedings of USENIX Conference on Operating Systems Design and Implementation (OSDI)*. 221–236.

[49] Hongliang Liang, Qian Zhao, Yuying Wang, and Haifeng Liu. 2016. Understanding and Detecting Performance and Security Bugs in IOT OSes. In *Proceedings of International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*. 413–418.

[50] Vilen Looga, Zhonghong Ou, Yang Deng, and Antti Ylä-Jääski. [n. d.]. Mammoth: A Massive-Scale Emulation Platform for Internet of Things. In *Proceedings of International Conference on Cloud Computing and Intelligence Systems*, Vol. 3. 1235–1239.

[51] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 329–339.

[52] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 329–339.

[53] Amir Makhshari and Ali Mesbah. 2021. IoT Bugs and Development Challenges. In *Proceedings of International Conference on Software Engineering (ICSE)*. 460–472.

[54] Julie L. Newcomb, Satish Chandra, Jean-Baptiste Jeannin, Cole Schlesinger, and Manu Sridharan. 2017. IOTA: A Calculus for Internet of Things Automation. In *Proceedings of ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward)*. 119–133.

[55] Frolin Ocariza, Kartik Bajaj, Karthik Pattabiraman, and Ali Mesbah. 2013. An Empirical Study of Client-Side JavaScript Bugs. In *Proceedings of ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 55–64.

[56] Jinkun Pan and Xiaoguang Mao. 2017. Detecting DOM-Sourced Cross-Site Scripting in Browser Extensions. In *Proceedings of IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 24–34.

[57] Philipp Rosenkranz, Matthias Wählisch, Emmanuel Baccelli, and Ludwig Ortmann. 2015. A Distributed Test System Architecture for Open-Source IoT Software. In *Proceedings of Workshop on IoT Challenges in Mobile and Industrial Systems (IoT-Sys)*. 43–48.

[58] Anam Sajid and Haider Abbas. 2016. Data Privacy in Cloud-Assisted Healthcare Systems: State of the Art and Future Challenges. *Journal of Media Systems* 40, 6 (2016), 1–16.

[59] Carolyn B. Seaman. 1999. Qualitative Methods in Empirical Studies of Software Engineering. *IEEE Transactions Software Engineering* 25, 4 (1999), 557–572.

[60] Rahmadi Trimananda, Seyed Amir Hossein Aqajari, Jason Chuang, Brian Demsky, Guoqing Harry Xu, and Shan Lu. 2020. Understanding and Automatically Detecting Conflicting Interactions between Smart Home IoT Applications. In *Proceedings of ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 1215–1227.

[61] Jeff Voas, Rick Kuhn, and Phil Laplante. 2018. Testing IoT Systems. In *Proceedings of IEEE Symposium on Service-Oriented System Engineering (SOSE)*. 48–52.

[62] Ran Wang, Guangquan Xu, Xianjiao Zeng, Xiaohong Li, and Zhiyong Feng. 2018. TT-XSS: A Novel Taint Tracking based Dynamic Detection Framework for DOM

Cross-Site Scripting. *J. Parallel and Distrib. Comput.* 118 (2018), 100–106.

[63] Zhendong Wu, Kai Lu, Xiaoping Wang, and Xu Zhou. 2015. Collaborative Technique for Concurrency Bug Detection. *International Journal of Parallel Programming* 43, 2 (2015), 260–285.

[64] Teng Xu, James B Wendt, and Miodrag Potkonjak. 2014. Security of IoT Systems: Design Challenges and Opportunities. In *Proceedings of International Conference on Computer-Aided Design (ICCAD)*. 417–423.

[65] Ru Zhang, Wencong Xiao, Hongyu Zhang, Yu Liu, Haoxiang Lin, and Mao Yang. 2020. An Empirical Study on Program Failures of Deep Learning Jobs. In *Proceedings of International Conference on Software Engineering (ICSE)*. 1159–1170.