IoTReplay: Troubleshooting COTS IoT Devices with Record and Replay

Kaiming Fang
Department of Computer Science
Binghamton University, State University of New York
kfang2@binghamton.edu

Guanhua Yan
Department of Computer Science
Binghamton University, State University of New York
ghyan@binghamton.edu

***Abstract*—Internet-of-Things (IoT) devices have been expanding at a blistering pace in recent years. Many of these devices have not been thoroughly tested for their security and dependability prior to shipment. These COTS (Commercial-Off-The-Shelf) IoT devices pose severe security threats to not only their users but also critical infrastructures, as evidenced by the infamous Mirai botnet attack. This work explores how to use the record and replay technique to troubleshoot COTS devices. To this end, we have developed an edge-assisted system called IoTReplay, which identifies contextual events in an IoT system that may affect the operations of the target IoT device. These contextual events are recorded when the IoT device is operating in the real world and then replayed in a test environment. We evaluate the performance of IoTReplay for troubleshooting four different types of COTS IoT devices. Our experiments demonstrate that IoTReplay is able to replay execution sequences of these devices with high fidelity while causing negligible interference with the operations of these IoT devices in the real world.**

I. Introduction

Internet-of-Things (IoT) devices have been expanding at a blistering pace in recent years. According to a new forecast by International Data Corporation, in 2025 there will be 41.6 billion connected IoT devices, generating almost 80 zettabytes of data [19]. Despite the great convenience, automation, and cost-effectiveness offered by IoT devices, they raise growing user concerns with their dependability, security, and privacy. Indeed, many consumers in a recent survey have had trust issues with IoT devices [9]: 28% of the people who never own smart devices are deterred from buying one due to security concerns and 53% of the people surveyed do not believe that their IoT devices have protected or respected their privacy effectively. Vulnerabilities of IoT devices not only pose great security risks to their users' information, but also can be exploited to cause infrastructure-level service disruption, as evidenced by the infamous Mirai botnet attacks against Dyn's DNS (Domain Name System) service [12].

Ideally, IoT device vendors should have thoroughly tested their products to fix all possible vulnerabilities before shipping them to the market. However, many of the vast IoT devices are produced by manufacturers that have only limited capabilities, such as budgets and expertise, for security testing [1]. Particularly, the laboratory settings in which IoT devices are tested prior to shipment can be drastically different from the real-world scenarios where these devices are operated by users with diverse knowledge levels of IoT and interact with myriad physical environments that are hard to emulate exhaustively. Therefore, numerous vulnerable COTS (Commercial-Off-Shelf) IoT devices have been placed into operation, which is confirmed by a recent finding by Palo Alto Networks that 57% of 1.2 million IoT devices deployed in thousands of physical locations in the United States are vulnerable to medium- or high-severity attacks [22].

Against this backdrop, this work explores how to troubleshoot COTS IoT devices using record and replay. Record and replay is an automated blackbox system testing technique with a wide range of applications, including system reverse engineering [10], malware analysis [26], intrusion analysis [11], network troubleshooting [30], Web application analysis [5], [21], Android application analysis [14], [17], [16], [23], reproduction of Windows application execution [15]. Even with many successful examples for record and replay, there are unprecedented challenges when applying this technique to troubleshoot COTS IoT devices, because they interact with not only other components in an IoT system, including the counterpart mobile app and the backend IoT cloud, but also their physical environments, which are unpredictable and hard to emulate perfectly.

In this work we develop a new edge-assisted system called IoTReplay to facilitate record and replay for troubleshooting COTS IoT devices. Its key idea is to identify a comprehensive set of external events that may affect the behavior of a COTS IoT device and truthfully replay them in a test environment. IoTReplay instruments the operational environment of a COTS IoT device to record direct or indirect contextual events while minimizing interference with its operation. The contextual events collected are replayed within test environments which run either in parallel with the operational one or in an offline fashion. A test environment uses IoT components of identical models or versions in hope of replaying the same problems seen in the operational environment. Moreover, extra tools can be deployed inside the test environment to help diagnose the COTS IoT devices.

In a nutshell, our key contributions in this work are summarized as follows. (1) We identify the key challenges in implementing record and replay for COTS IoT devices. To address these challenges, we identify various contextual events that may affect the operation of a COTS IoT device in an

IoT system and analyze whether or how they should be replayed. (2) We design a scalable edge-assisted architecture for IoTReplay to support both online and offline modes of replaying contextual events within a test environment. (3) We implement IoTReplay for COTS IoT devices managed by Android apps. We develop a combination of static analysis and dynamic instrumentation techniques to record and replay indirect contextual events affecting IoT applications, including UI operations, geolocation information, and sensor data. We also implement a method for translating Android UI operations across different mobile phones to ensure their replayability. (4) We perform extensive experiments to evaluate the effectiveness of IoTReplay in troubleshooting IoT devices, its usability due to UI performance degradation, and its scalability for troubleshooting multiple IoT devices simultaneously. Our experimental results suggest that the record and replay mechanism offered by IoTReplay complements existing techniques in enhancing the security and dependability of IoT devices.

The remainder of the paper is organized as follows. Section II presents the challenges for troubleshooting COTS IoT devices as well as the rationale of IoTReplay. Section III describes the design of IoTReplay. Section IV gives the implementation details of IoTReplay. Section V shows the performance evaluation results of IoTReplay with four types of IoT devices. Section VI surveys related work and Section VII draws concluding remarks.

II. Background

A typical IoT system works as follows. An *IoT device* operates in a certain *physical environment*, from which its sensors can collect environmental data, such as temperature, smell, and surrounding scene. Through the UI (User Interface) of an *IoT app* installed on a mobile phone, an *IoT user* can monitor, control, and manage the IoT device remotely. There can be direct communications between the IoT app and the IoT device using wireless technologies, such as WiFi, Bluetooth, and ZigBee, when they are in physical proximity. In case that no direct messaging between the IoT app and the IoT device is possible, the IoT cloud can act as a proxy to relay every communication message sent between them.

Between the IoT device and the IoT cloud, there can also be communication messages initiated in both directions. The IoT device can send its sensor data and other telemetry events to the IoT cloud. For example, a smart camera can automatically upload the video recorded to the cloud. On the other hand, the IoT cloud usually maintained by the IoT device vendor can actively push messages (e.g., notifications and fireware updates) to the individual IoT devices.

Challenges for troubleshooting COTS IoT devices. Troubleshooting COTS devices has the following challenges. First, although their functionalities are explained in their manuals, their implementation details are usually unavailable, making it difficult to perform whitebox or greybox tests on these devices. Second, many issues found with IoT devices are encountered by their users when they are operating in the real world. It is hard to reproduce these problems without repeating the same user operations in a test environment. Last but not least, IoT devices usually need to interact with their physical environments, which are impossible to reproduce exactly. Such non-determinism of the physical world, which is wisely characterized by Greek philosopher Heraclitus' quote, "No man ever steps in the same river twice, for it's not the same river and he's not the same man," makes it particularly difficult to diagnose issues previously observed in the operations of COTS IoT devices.

Rationale of IoTReplay. Our key idea for troubleshooting IoT devices is to record all relevant contextual events that may affect the operations of an IoT device in the real world and replay these contextual events truthfully in a test environment with additional diagnosis instruments. A *contextual event* is defined to be one that is triggered by the environment external to the operation of an IoT device. An example contextual event can be a user input to the IoT app or a network packet received by the IoT device. A self-triggered event by the IoT device is not deemed as a contextual one.

IoTReplay strives to record and replay all possible contextual events with minimal interference with the operation of the IoT device in the real world. Although not all contextual events contribute to the issue observed in the operational environment, the true factors are not known before troubleshooting. Therefore, having all possible contextual events replayed in the test environment maximizes the likelihood that the same problem should appear again.

Due to nondeterminism of the physical world, contextual events observed in the operational environment may not be reproducible inside the test one. For example, a scene captured by a smart security camera may not be exactly reproducible to another security camera, even if they are placed physically close to each other, or to the same security camera which attempts to capture the same scene at a different time. *Even with such irreproducible contextual events, record and replay can still be a powerful tool for troubleshooting IoT devices, because the bug bothering the IoT device in an operational environment can be activated in the test environment as long as its logical prerequisite is reproducible.* Following the previous security camera example, an ill-formed user command sent from the IoT app and causes the security camera to crash can be replayed to troubleshoot the device if its generation does not depend on a nondeterministic input.

Scope of work. IoTReplay aims to record and replay *external events* that can drive the IoT device under test into a faulty state. It is thus not effective in troubleshooting issues triggered by its internal events. For example, if the IoT device is infected by a malware, those problems caused by local malware activities may not be manifested through the record and replay mechanism offered by IoTReplay. However, IoTReplay can still be instrumental in revealing how the malware spread onto the IoT device from a remote machine.

III. System Design

In this section we first present the architecture of IoTReplay and then discuss the contextual events identified by IoTReplay.

A. IoTReplay architecture

The architecture of IoTReplay is illustrated in Figure 1. IoTReplay distinguishes two types of worlds, *operational world* and *shadow world*. The operational world is the real environment where an IoT device of interest is used. In the operational world, the operational IoT device can be managed by a human user from its counterpart IoT app installed on a real mobile phone (operational mobile phone). The operational IoT app is instrumented to record relevant contextual events that happen to the operational mobile phone. The communications between the IoT app and the IoT device can be be direct or relayed through an IoT cloud. A network gateway is placed in front of the operational IoT device to record relevant network packets destined to it. The operational dataflows, which are shown as black solid arrows in Figure 1, include all communication messages that exist in the real environment, including those among the IoT app, the IoT cloud, and the IoT device.

IoTReplay records different types of contextual events collected from the operational world and replays them in shadow worlds. A shadow world consists of a shadow IoT device, which is a duplicate of the operational IoT device, and its counterpart shadow IoT app, which is instrumented from the original IoT app to support replay of contextual events. The shadow IoT app can run on a virtual or real mobile phone (shadow mobile phone).

As illustrated in Figure 1, a shadow world is not fully isolated from the operational one or another shadow world, because it is impractical to create duplicate vendor-specific IoT clouds for testing purposes. When necessary, it is possible to execute multiple shadow worlds for the same operational one. Moreover, depending on how a shadow world is operated, we can have the following two modes:

- *Online mode*: An online shadow world is running *simultaneously* with the operational one. Hence, any new issue encountered in the operational world can be immediately replayed in the shadow one where additional diagnosis tools are deployed to troubleshoot the problem. In an online shadow world, however, the shadow IoT device must differ from the operational one, because two COTS devices even of the same brand and model have their unique product identifiers. If these product identifiers are used as inputs to encrypt or obfuscate their communication payloads, we cannot simply record network packets from the operational world and replay them within the shadow world to troubleshoot these IoT devices.
- *Offline mode*: An offline shadow world, which replays the contextual events recorded from the operational world during an earlier time period, allows to reuse the operational IoT device as its shadow one. Hence, the aforementioned problem can be avoided when unique product identifiers are used for traffic encryption or obfuscation. However, as the offline shadow world operates in a different time epoch as the operational one, the physical environment experienced by the same IoT device may still differ from the one in the operational world even if its location has not changed.

In the IoTReplay architecture, the *dispatcher* is responsible for coordinating record and replay activities between operational and shadow worlds. On the arrival of each contextual event recorded from the operational world, the dispatcher either forwards it to an online shadow world immediately, or stores it for a certain period of time before delivering it to an offline shadow world.

Benefits of edge computing. Deploying IoTReploy on edge computers can improve the reproducibility of the problems encountered by an operational IoT device. First, the IoT gateway in the operational world should be able to intercept any network packets of interest destined to the operational IoT device and then forward them to the shadow ones. Hence, deploying the gateway on an edge device close to the operational IoT one helps improve the coverage of network packets replayable in the shadow worlds. Second, we can split the logical functionalities of the dispatcher into two components: *dispatcher-app* and *dispatcher-gateway*. The dispatcher-app component can run on the same physical machine as the virtual mobile phones hosting the shadow IoT apps. This physical machine can be an edge computer which stays close to the operational phone so the latency in forwarding the recorded activities from the operational IoT app to the shadow ones can be minimized, which is particularly ideal for online deployment of IoTReplay. Similarly the dispatcher-gateway component can run on an edge computer close to the gateways for forwarding network packets between operational and shadow worlds to reduce the record and replay latency.

B. Contextual Events

At the heart of IoTReplay is how contextual events are identified. The behavior of an IoT device, assumed to be a *blackbox* for troubleshooting, can be modeled as a deterministic finite state machine (FSM) whose state transitions are driven by an input alphabet including all possible contextual events. In a typical IoT architecture, the contextual events that affect the operation of an IoT device can originate from the IoT app, the IoT cloud, and its physical environment. A shadow world can run its shadow device inside the same physical environment as the operational device or in an emulated environment. Our discussions in the rest of this section will focus on other types of contextual events.

The *direct* contextual events of the operational IoT device obviously include all network packets it receives. A naive approach, which we call the *original blackbox approach*, is to replay each of these packets to a shadow one in a shadow world. This scheme, however, does not work if a different IoT device is used in the shadow world and the in-band messages destined to the IoT devices are encrypted with device-specific identities. Even if the same IoT device is used in an offline shadow world, oft-used cryptographic nonces or time stamps in network security protocols can still make the naive packet replaying method futile.

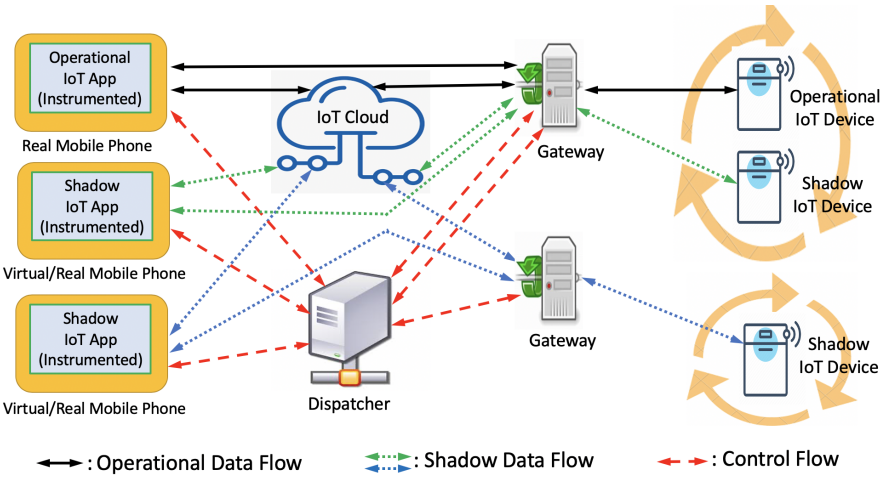


Fig. 1. Architecture of IoTReplay

To circumvent this problem, we consider *indirect* contextual events, which trigger internal state transitions of the operational IoT device through their effects on the other components in the IoT system. For example, the IoT app receives a user input and sends an encrypted command to the IoT device. Although it is hard to replay the encrypted command message directly to the shadow IoT device because it is encrypted by a different key, we can replay the user input to the shadow IoT app, which generates the same command message encrypted by the key shared with the shadow IoT device.

Based on indirect contextual events, we can extend the original blackbox, which includes only the COTS IoT device, to include the other components in an IoT system. The advantage of this approach is that, although we need to rerun the extended blackbox during the replay phase, we do not need to record and replay any communication messages, which may be encrypted with device-specific keys, among the components inside the blackbox. Instead, we only need to identify and replay the contextual events that are external to the extended blackbox. However, it is impractical to extend the blackbox to include the entire IoT system, because it is hard to create a duplicate copy of the IoT cloud running inside the shadow world or instrument the IoT cloud in the operational world to record its contextual events. Another challenge is that recording and replaying all possible contextual events external to the extended blackbox can be computationally prohibitive. It is also unnecessary to do so in some cases. For example, suppose that an IoT app reads the geolocation information of the mobile device. If this information is only shown locally to the mobile user but never used to update the messages sent to the IoT device, the contextual event of reading the geolocation does not have any effect on the behavior of the IoT device, making it unnecessary to replay this contextual event for troubleshooting the device.

Due to these concerns, IoTReplay still treats the COTS IoT device as a blackbox but handles three types of contextual events based on their receivers. The actions taken by IoTReplay for these contextual events are summarized in Table I.

C. Replayability analysis of contextual events

We next analyze the replayability of different types of contextual events shown in Table I.

Indirect contextual events received from IoT app. IoTReplay records and replays UI operations, geolocation information, and sensor data received by the IoT app. The UI operations include a variety of mobile gestures performed by either the user's fingers (e.g., tap, swipe, drag, and slide) or hand movements (e.g., shaking, tilting, moving, and rotating the mobile device). An IoT app may read the location information of the mobile device. For example, the Google Nest IoT platform obtains the current location of a mobile device to arrange its user's schedule. Some IoT apps read data from sensors like accelerometer and linear acceleration on their mobile devices. These sensor data may also affect the communications between the IoT app and the IoT device.

IoTReplay does not record and replay any network packets received by the IoT app. To explain how this affects the precision of IoT device troubleshooting, we first define the following properties:

Definition 1. *We say that entity A's responses are time-insensitive if and only if for any two other entities, B and C, if B's request R_B and C's request R_C have identical contents, then A's responses to R_B and R_C must also have identical contents, regardless of the time and order of these two requests. The contents of a request or a response do not include how it is encrypted.*

Definition 2. *We say that entity A's unsolicited requests are undifferentiated to entities B and C if and only if for any unsolicited request that A sends to B (or C), there must be another one with the same contents sent from A to C (or B) at the same time.*

The network packets received by an IoT app fall into the following categories based on the entities involved. (1) *Network packets directly from the IoT device:* These packets should *not* be recorded and replayed to avoid the chicken-and-

TABLE I
ACTION TABLE FOR DIFFERENT TYPES OF CONTEXTUAL EVENTS

Event Receiver	Event Type	Shadow mode	IoTReplay Action
IoT App	UI operations	Online/Offline	Record & Replay
	Geolocation information	Online/Offline	Record & Replay
	Sensor data	Online/Offline	Record & Replay
	Network packets from IoT device	Online/Offline	None
	Network packets from IoT cloud	Online/Offline	None
	Exotic network packets	Online/Offline	None
	Timer events	Online	None
	Timer events	Offline	Start time alignment
IoT Device	UI operations	Online/Offline	Human Replay
	Sensor data	Online/Offline	Physical Replay
	Network packets from IoT app	Online/Offline	None
	Network packets from IoT cloud	Online/Offline	None
	Exotic network packets	Online/Offline	Record & Replay
IoT Cloud	Any	Online/Offline	None

egg problem as our goal is to examine the effects of indirect contextual events on the IoT device through the IoT app. (2) *Network packets from the IoT cloud:* If they are relayed packets from the IoT device, they should not be replayed for the same reason as those directly from the IoT device. If they are responses to app-triggered packets, they should be automatically replayed if the following two conditions are satisfied: all the contextual events triggering the IoT app to communicate with the IoT cloud (e.g., UI operations) are replayed and the IoT cloud's responses are time-insensitive to the requests from the IoT app (see Definition 1). If they are cloud-triggered packets, we have the following cases. When the shadow world works in an online mode, these indirect contextual events should *not* be replayed if the IoT cloud's unsolicited requests (i.e., cloud-triggered packets) are undifferentiated to all the IoT apps (see Definition 2). When the shadow world works in an offline mode, the situation becomes complicated because these cloud-triggered packets may or may not be received by the shadow IoT device. (3) *Exotic network packets*: these packets originate from sources other than the IoT devices and the IoT cloud. When the IoT app contacts an external network service, the responses received may taint its communications with the IoT device. Similarly, assuming that any network service contacted by the IoT app must have time-insensitive responses to the requests from the IoT app, exotic network packets should *not* be replayed in the shadow world.

An IoT app may use a timer to trigger some internal state changes. If the shadow world is operating in an online mode, timer events without using random delays should be identical between the two worlds so there is no need to replay such timer events. For an offline shadow world, sometimes the conditions for firing the time events can be reproduced by carefully aligning the world's start time. For example, if a timer event is scheduled by the IoT app exactly at the beginning of each hour, we can align the start time of an offline shadow world at the hour level to ensure that the same timer events are fired in the shadow world. However, if the IoT app uses irregular or random timer events, it would be difficult to replay them in the shadow world.

Network packets received from IoT cloud. IoTReplay does not record and replay any packets received from the IoT cloud, partly because they may be encrypted with device-specific keys. Such packets fall into two categories. For the IoT cloud's responses to device-triggered packets, it is unnecessary to record and replay them if the following conditions are satisfied: the contents of the deviced-triggered packets in the shadow world are identical to those in the operational world (although they may be encrypted with different keys) and the IoT cloud's responses are time-insensitive to the request packets from the IoT devices. However, the IoT cloud can also initiate communications with individual IoT devices, such as firmware upgrade, keepalive messages, and system notifications. For broadcast messages, they should be received by the shadow IoT devices in online shadow worlds, but can be missed by those running in offline shadow worlds due to time epoch difference. For unicast messages, they should be recorded and replayed, which is however hard to do because they are usually encrypted or obfuscated.

Exotic packets received by IoT devices. Without a firewall that allows only in-band messages (i.e., those from the IoT cloud or from the IoT app directly) to reach the IoT device, out-of-band network packets from network scanners or IoT device intruders can cause internal state changes to the IoT device. These unexpected packets are usually not encrypted with device-specific keys. As they can also cause trouble to the IoT device, IoTReplay records them in the operational world and replays them in the shadow ones.

IV. IMPLEMENTATIONS

In this section we present the implementation details of IoTReplay. As seen from Table I, record and replay are needed for UI operations, geolocation information, and sensor data received by the IoT app and exotic network packets destined to the IoT device.

A. Record and replay for IoT app

The current implementation of IoTReplay targets COTS IoT devices with companion IoT apps developed for Android systems, whose open nature allows for use of abundant freely available analysis and instrumentation tools. Figure 2 presents IoTReplay's workflow of recording indirect contextual events from the IoT app.

1) Indirect contextual events: To explain the workflow, we first discuss the three types of indirect contextual events affecting the IoT app, UI events, geolocation information, and sensor data, which need to be recorded at the end of the workflow (orange boxes in Figure 2).

UI events. In each Android device including AVD (Android Virtual Device), there is a device at */dev/input/* that logs the user's finger operations. When a user touches the screen, moves around, or even uses multiple fingers to do operations, this device generates a sequence of *input_event* data in tuple: "type code value", where *type* and *code* determine the category of an event, respectively, and *value* denotes its numeric value. An example of an event is given by Google Nexus 5: "0003 003a 000000ff", indicating an event has a type of an absolute value (0003 or EV_ABS), a code of pressure event (003a or ABS_MT_PRESSURE), and the value of the touch pressure is 000000ff, all represented in hexadecimal format.

Because the UI events, which are generated and processed from the Linux OS level, are shared by all running apps, we need to tease out those relevant to the IoT app. Intuitively, an app is affected by UI operations when it is running in foreground. Based on the life cycle of an Android app activity [6], an app runs in foreground after its activities invoke *onResume()* and goes into the background after *onPause()* is called. Although some apps, such as malicious apps, may generate UI events even running in the background, for usually simple IoT apps, it is sufficient to use these calls to determine which events logged at */dev/input* are relevant to the IoT app.

Geolocation information. An IoT app may use the current location of the mobile phone. For example, the Nest app developed by Google uses the current location of the mobile phone to arrange its user's schedule. The location information can be obtained by an app using Android's geolocation-related system APIs. There are typically three types of location information based on their sources:

- *WiFi:* The coarse-grained location of a mobile device can be found through the WiFi access point with which it is connected. The following APIs can be used by Android apps to obtain WiFi-related information: *WifiManager.getScanResults*, *WifiManager.getWifiState*, and *WifiInfo.getSSID*.
- *Cellular*: A mobile phone can acquire the TOA (time of arrival) and TDOA (time difference of arrival) of cellular signals from multiple base stations to estimate its location. APIs commonly used by Android apps to obtain cell locations are *TelephonyManager.getCellLocation* and *TelephonyManager.getAllCellInfo*.
- *GPS:* The geographic coordinate of a mobile device can be directly obtained from satellites through its GPS antenna. Related APIs used by Android apps include the following: *LocationManager.getGpsStatus*, *LocationManager.getCurrentLocation*, and *LocationManager.getLastKnownLocation*.

Sensor data. An app can acquire data collected by sensors such as accelerometer and gyroscope through an API called *onSensorChanged(SensorEvent se)*. This callback function is invoked whenever there is a new sensor event. The IoT app can process parameter *se* to extract new sensor events from it.

2) Static analysis: Although UI events can be directly obtained from */dev/input*, geolocation information and sensor data fed to an IoT app can only be acquired by intercepting the relevant APIs. A naive approach is to intercept all relevant API calls as discussed previously to record the geolocation information and sensor data read by the operational IoT app. This method, however, overestimates the effects of these indirect contextual events if they do not affect the messages sent from the app to the IoT device. To reduce the computational overhead in dynamic instrumentation, IoTRelay uses a backtracking technique to identify all the API calls in an IoT app that can taint the messages sent to the IoT device.

Data dependence graph. As illustrated in Figure 2, IoTReplay uses the Amandroid tool [27] to extract DDG (Data Dependence Graph) of the IoT app. Amandroid is a static analysis framework for Android apps capable of generating control flow graphs and data dependency graphs as well as performing taint analysis. Compared with other static analysis tools, Amandroid allows inter-component analysis of Android apps to improve the precision and completeness of static analysis. Using the Amandroid APIs, we develop a custom analysis tool to generate the data dependence graph of the IoT app, which is comprised of a set of points-to facts representing the relationships between the use sites and creation/definition sites of Dalvik objects or variables.

Backtracking. Let $G(V, E)$ be the DDG extracted by Amandroid for the IoT app. In this graph, each node $v \in V$ has a name attribute, $v.name$, and for each directed edge $(u, v) \in E$, the start node u is treated as a use site and the end node v its creation/definition site. In addition to $G(V, E)$, the inputs to the backtracking algorithm also include L, which is the list of system APIs that can be hooked to record contextual events, and F, which is a list of functions that can send messages to the IoT device. Each function in F is a node in the DDG after static analysis.

Given DDG $G(V, E)$ and any node $u \in V$, we define $reach(u, G)$ to include all the nodes that are reachable from node v in graph G. Similarly, for node set $U \subseteq V$, we define $reach(U, G)$ to be the union of $reach(u, G)$ for all $u \in U$. The output of the backtracking algorithm is the hooking set H, which is initialized to be empty and eventually includes all the nodes that need to be hooked to record contextual events. A simple graph traversal algorithm can be used to derive the hooking set H, which is defined as follows:

$$H = \{\forall v : v \in reach(F, G) \wedge v.name \in L\}$$

Informally, the hooking set H includes every system API call named in set L that obtains data (e.g., geolocation information or sensor data) capable of tainting the message-sending functions in list F.

Identification of message-sending functions. We use an example to show how to identify message-sending functions in an IoT app. For the IoT app of the Tycam LTE camera, we can find the following method in its smali code:

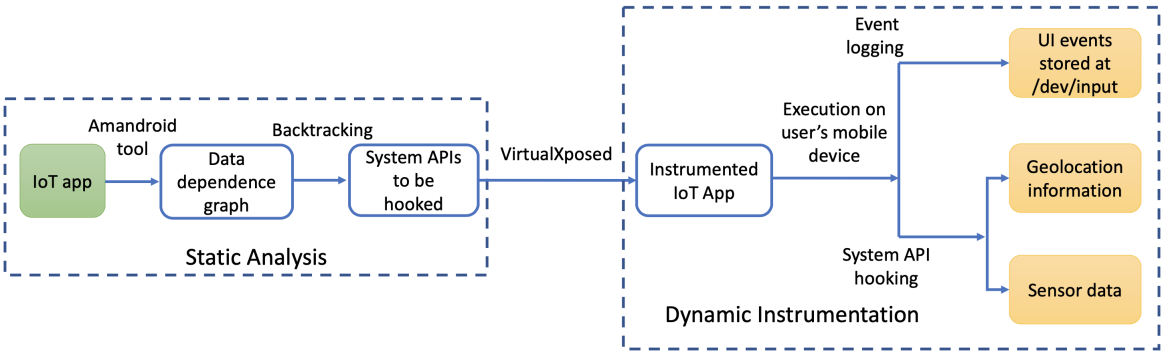


Fig. 2. Workflow of recording indirect contextual events from IoT app

```
// smali code
.method smali_classes7/com/tutk/IOTC/Camera$p;->run()V
```

This method is added to list F fed to the backtracking algorithm, because it invokes *avSendIOCtrl()*, where *avSendIOCtrl()* is a native function in its library *libAVAPIs.so* responsible for sending commands towards its camera device.

3) Dynamic instrumentation: IoTReplay uses VirtualXposed [3] to instrument the operational IoT app. VirtualXposed allows modification of Android app behaviors through the use of Xposed modules, but it does not need to root the mobile device or flash a custom system image on it. Therefore, VirtualXposed, which itself runs as a normal app on an Android device, offers a non-intrusive approach to hooking the API calls in the operational IoT app identified by the backtracking technique. More specifically, for each API call in the hooking set H, we write an Xposed module calling *param.getValue()* and *result.getResult()* to record its parameter and returned values in the operational IoT app, and a counterpart one calling *param.setValue()* and *result.setResult()* to replay them in the shadow IoT app. These parameter and result values are relayed by the dispatcher from the operational IoT app to the shadow one through TCP connections.

We use an approach similar to RERAN [14] for record and replay of UI events. On the operational mobile phone, we use the standard *getevent* tool to record the UI events from the */dev/input* directory. On the shadow mobile phone (either a real one or an AVD), the standard *sendevent* tool is not used due to its slow replay speed. Instead, we run a custom program on the shadow mobile phone to inject events recorded on the operational mobile phone into the event stream stored at */dev/input*.

The operational IoT app is also instrumented with an Xposed module to monitor whether the app's activity is running in foreground or background by hooking its two lifecycle methods, *onResume* and *onPause*. The module sends a notification to the dispatcher whenever one of the two methods is invoked by the app. The dispatcher uses these notifications to infer whether the IoT app is running in foreground or not. On the arrival of *onPause* notification, the dispatcher stops relaying the UI events from the operational mobile phone to the shadow one; on the arrival of *onResume* notification, the dispatcher resumes UI event relaying to the shadow phone.

B. Record and replay for exotic network packets

We use the classical ARP spoofing technique to insert the gateway between the router and the operational IoT device. The gateway sends one ARP packet with its source spoofed as the router to the IoT device, which updates its ARP table with the binding between the gateway's MAC address and the router's IP address. Similarly, the gateway sends another ARP packet with its source as the IoT device to the router, which adds the binding between the gateway's MAC address and the IoT device's IP address to its ARP table. This approach, which is also adopted by IoT Inspector [18], does not need configuration changes of the IoT device or extra dedicated devices for capturing packets. The gateway can forward intercepted packets to the dispatcher, who further decides which shadow worlds they should be delivered to.

C. Dispatcher

The dispatcher, which is implemented in Python 3, relays the following messages from an operational world to a shadow one: UI events, parameter and result values extracted by Xposed modules, and exotic network packets from the gateway of the operational IoT device. It also decides whether to relay UI events based on the *onResume* and *onPause* notification messages from the operational mobile phone. Another important function performed by the dispatcher is UI event translation, which is done from three different aspects.

Event device. Different Android mobile phones may have different event devices in the Linux directory, */dev/input*. For example, the touchscreen device of Samsung Galaxy S5 resides at */dev/input/event2*, while AVD uses its touchscreen device as */dev/input/event5*. As each touchscreen device must have codes of *BTN_TOUCH* or *BTN_TOOL_FINGER* constants in its device information, the event devices can be mapped between different Android mobile phones.

Event code. Event devices in different mobile phones may be developed by different vendors. A vendor may use its own event codes and value definitions for its devices. Moreover, different phones use different event codes for touch, move/press, lift, pressure, and orientation operations. For example, some mobile phones use *ABS_MT_TRACKING_ID* value of -1 to indicate a finger release from the touchscreen, while others like Honor 8 use code *BTN_TOUCH*. Therefore, the dispatcher

relies on a map to translate these event codes and values received from the operational mobile phone before relaying them to the shadow one.

Event value. Event value translation can be accomplished by normalization based on the value ranges. For example, event *ABS_MT_PRESSURE* of Google Nexus 5 has a value range of [0, 65535], while Huawei Honor 8 has the same event code within [0, 255]. Hence, a value of 100 for the pressure event code in Google Nexus 5 can be mapped to $255 \times 100/65535$ in Huawei Honor 8.

V. Experiments

In this section we first discuss the setup of a testbed. Using this testbed, we perform various experiments to evaluate the effectiveness and execution performance of IoTReplay.

A. Experimental setup

We use four types of COTS IoT devices, Google Nest camera, D-Link smart plug, Roku TV, and Tycam LTE camera. The first two use their cloud servers as a relay to communicate with their IoT apps, while the latter two can directly talk to their IoT apps. The Tycam LTE camera system allows direct communications between its mobile app and its device (camera) even if they use different local networks, using UDP hole punching [29] assisted by a remote server; the relay mode through an intermediate server is enabled only when UDP hole punching fails. In our experiments the Tycam LTE camera is configured to have direct communications with its app.

For all the COTS IoT devices but the Tycam LTE camera, their operational IoT apps run on a Samsung Galaxy S5 phone and their shadow ones on an x86 AVD, which is a virtual mobile phone shown in Figure 1. For the Tycam LTE camera we run its operational IoT app on the Samsung Galaxy S5 phone and its shadow IoT app on a *real* Google Nexus 5 phone. This is because its IoT app uses native libraries for ARMv7 but the execution performance of an ARM AVD is too slow for record and replay.

In our experiments, all the operational and shadow phones run AOSP (Android Open Source Project) with Android version 7.1. The AVD used for the shadow phone uses the same hardware configuration (e.g., RAM and screen size) as the Samsung Galaxy S5 phone. Xposed modules are installed inside VirtualXposed along with IoT apps on real phones. The real Xposed is installed on each AVD to avoid virtualization overhead in our experiments.

In addition to the real mobile phones and the COTS IoT devices, the other hardware used include a workstation (CPU: I7-9700 3.00GHZ, RAM: 32GB) and a Dell Inspiron laptop (CPU: I5-6500U, RAM: 16GB). The workstation is used to run the dispatcher and AVDs (*i.e.*, virtual mobile phones in Figure 1), and the laptop to run the gateways.

B. Measurement results

For each type of COTS IoT devices under test, we perform an experiment where two separate systems with components of identical models are running in parallel. These experiments are done without any UI operations from their applications to avoid interference from the IoT apps. We monitor network traffic from and to the IoT devices and infer the type of each network message based on the traffic characteristics.

Table II summarizes our findings about the types of messages received by four IoT devices, along with their percentages. There are five different message types:

- *Application:* All four IoT devices receive messages from their counterpart IoT apps, either directly or through their corresponding IoT clouds. As discussed before, IoTReplay can record and replay messages triggered by the IoT apps through indirect contextual events (i.e., UI operations, geolocation information, and sensor data).
- *Cloud notification response:* The two camera devices receive responses from their IoT clouds after they report detected sounds or object movements. These messages are replayable if we can mimic the physical environment of the operational world.
- *Firmware upgrade:* We also observe firmware upgrade messages sent from the IoT cloud to only one of the D-Link smart plugs, as the two smart plugs were shipped with different versions of firmware. However, firmware upgrade does not take place frequently; we can also avoid such differentiated messages from the cloud by choosing shadow IoT devices with the same firmware version as the operational one.
- *Keep-alive:* The IoT cloud sends keep-alive messages to each individual Google Nest camera registered. This type of messages is inferred from the periodic occurrences (20 per second) of packets monitored between the device and its cloud server, even when there is no human operation from the IoT app side.

Table II also presents the different encryption/obfuscation techniques used by each COTS IoT device. Both Google Nest Camera and D-Link smart plug use TLS v1.0 to encrypt communication traffic. However, TLS v1.0 is not deemed as secure any more because it suffers both the BEAST and POODLE attacks [28]. The Tycam LTE camera use proprietary obfuscation techniques. The Roku TV transmits its messages in plaintext except authentication fields.

Timer events. As discussed in Section III-C, time events triggered by the counterpart IoT apps can affect the replayability of execution traces experienced by the COTS IoT devices. There are two Java classes that are often used for timer events in Android applications: *Timer* and *ScheduledThreadPoolExecutor*, both of which have two methods for scheduling future tasks: *schedule()* and *scheduleAtFixedRate()*, with minor differences in their parameters. Both methods have several overloads which can be classified into two types: with or without initial execution time specified by a starting date time (in java.util.Date). Without loss of generality, let *scheduleX* be either of these two scheduling methods. Although overloading the scheduleX method with an absolute date time can cause trouble to record and replay, particularly in an offline mode, our static analysis of the bytecode of the four IoT apps used in

TABLE II
NETWORK MESSAGES FOR EACH IOT DEVICE

IoT device	Message type	Encryption/Obfustation
Google Nest camera	Application (6.91%) Keep-alive (92.32%) Cloud notification response (0.77%)	TLS v1.0
D-Link smart plug	Application (93.55%) Firmware upgrade (6.45%)	TLS v1.0
Roku TV	Application (100%)	Plaintext (except authentication fields)
Tycam LTE camera	Application (89.41%) Cloud notification response (10.59%)	Proprietary obfuscation or encryption

our experiments (see Section V) shows that this never occurs.

All four IoT apps use only two overload types: scheduleX(task, delay) and scheduleX(task, delay, period), whose arguments fall into three types: *constant* (the arguments are fixed and hardcoded inside the bytecode), *random* (the arguments are randomized using relevant APIs), and *others* (the arguments are derived from other sources, such as user inputs). Our static analysis of the four IoT apps reveals that random arguments only exist in the D-Link smart plug application.

We use Xposed modules to hook methods invoking scheduleX to evaluate the fraction of each type parameter mentioned above. We run each IoT app for 10 minutes using the Monkey utility [4] to generate UI operations. Table III presents the average fraction of timer events per type over 10 runs for each experiment. It is noted that for the majority of time events, they are scheduled with constant delays; a small fraction of timer events depend on other inputs such as user inputs. Only for the D-Link smart plug, 7.89% of timer events are scheduled with random delay arguments.

TABLE III
FRACTION OF TIMER EVENTS SCHEDULED PER CATEGORY

IoT application	Constant	Random	Other
Google Nest camera	72.57%	0	27.43%
D-Link smart plug	77.64%	7.89%	14.47%
Roku TV	95.24%	0	4.76%
Tycam LTE camera	86.23%	0	13.77%

The results in Table III confirm our hypothesis that the majority of timer events can be truthfully replayed in a shadow world. Timer events with constant delays are replayable as long as the events that schedule these timer events can be replayed. The timer events with other inputs are replayable if these inputs themselves can be replayed. For instance, if the user, through UI operations, schedules a timer event with a delay with respect to the current time, this timer event is replayable as long as the UI operations can be replayed. In rase cases where the user schedules an timer event to be fired at an absolute time, such events may not be replayable in an offline shadow world because the position of the timer event fire time with respect to the start time of an offline experiment is different from that in the operational world. For timer events with random delays, they are typically not replayable unless the same seed is used to initialize the random number generator in both the operational and shadow worlds. Table III shows that timer events with random delays usually do not exist in IoT apps except the D-Link smart plug application.

C. Effectiveness results

1) Execution similarity.: In a new set of experiments, we measure the execution similarity between the operational world and the corresponding shadow worlds for each COTS IoT device. The task is challenging because we cannot see the internal state of each IoT device *and*, as seen in Table II, communication messages sent or received by an IoT device can be encrypted or obfuscated. To measure execution similarity, we again resort to instrumentation of IoT apps, which records a sequence of methods that have been invoked for every message received by an IoT app. An example method sequence is ['a()Z', 'c(Lh/v/b$a;)V', 'a(Z)V', 'b()J', 'a(FF)V']. We assign a unique symbol to each distinct method to convert the sequence to a string, which we call *invoked method string*. For each run of an operational or shadow world, the IoT app can receive a sequence of messages. Assuming a unique symbol for each distinct invoked method string, their corresponding invoked method strings can be further transformed into an *execution state sequence*. For example, if in a run four messages are received by the IoT app, its execution state sequence should include four symbols, each representing an invoked method string for processing a received message.

We use the execution state sequence to approximate the execution trajectory of an experiment. Let E_1 and E_2 be two execution state sequences. Their *execution similarity score* $S(E_1, E_2)$ is calculated as follows:

$$S(E_1, E_2) = 1 - \frac{D(E_1, E_2)}{\max\{|E_1|, |E_2|\}}, \tag{1}$$

where $D(E_1, E_2)$ denotes the edit distance between E_1 and E_2, and $|X|$ the length of sequence X. If $E_1 = E_2$, then we have $S(E_1, E_2) = 1$.

In our experiments, we notice that two invoked method strings can be very close to each other, suggesting that they correspond to two similar or identical messages received. For example, we observe an invoked method string of length 17 from the operational world but a slightly different one of the same length from the shadow world. As their only difference is at the 16-th position ('b([BI)J' instead of 'E()Z'), we hypothesize that these two methods should deal with almost identical messages from the IoT devices. To characterize such situations, we define a tolerance level θ: if the edit distance between two invoked method strings is no greater than θ, they are treated as indistinguishable and thus mapped to the same

symbol. If $\theta = 0$, the two invoked method strings must be exactly the same to be assigned with the same symbol.

Figure 3 shows the execution similarity score between the operational and shadow world for each of the COTS IoT devices considered. We observe that even with $\theta = 0$ for exact matching between invoked method sequences, the execution similarity score is always higher than 90% for all four IoT devices; this means that less than 10% of the execution state sequence derived from the operational world need to be edited to get exactly the same one seen in the shadow world. If we increase the tolerance level θ, the execution similarity score gets closer to 1, suggesting that the IoT devices in the operational and shadow worlds go through similar internal state changes.

To gain insights into why execution traces may not be exactly the same, we examine the Google Nest camera application. It is found to use a UDP port to receive time information from a remote server. The receiving procedure converts the time information into strings to be used later. It is however possible that the IoT app fails to get the time information from the remote server, thus generating an error message"Failed to fetch time from server." Hence, the remote server's responses do not satisfy the time-insensitive property as discussed in Section III-C, which is necessary to achieve replayability.

2) Attacks due to exotic network packets.: In a new set of experiments we evaluate whether IoTReplay can successfully record and replay exotic network packets targeting the four COTS IoT devices. We consider four attack scenarios:

- *Reboot attack*: Through message fuzzing we have found a way to cause the Tycam LTE camera to reboot [1]. This attack does not apply to the other three IoT devices.
- *Nmap probing:* Nmap [2] is a popular network scanner supporting host discovery, port scanning, and OS detection with active probing packets.
- *IoTSeeker:* IoTSeeker [24] scans IoT devices in the local network through their HTTP services to check if they are using the factory-set credentials.
- *IoT Inspector:* IoT Inspector [18] is an open source tool for automatic penetration testing of IoT devices. This tool can be used to identify a variety of security, privacy, and performance problems for IoT devices.

In our experiments we consider both online and offline modes for the shadow world. For the two cameras, we record object movements and sounds into files and replay them in front of the lens of the shadow camera devices when they are tested in an offline mode.

Table IV summarizes the replayability of each attack for both online and offline modes. We observe that IoTReplay can successfully reproduce the attack effects from the Nmap scanner and IoTSeeker for all four IoT devices, regardless of the work mode of the shadow world. The reboot attack against the Tycam LTE camera can also be replayed for both modes. However, the penetration test results by IoT Inspector cannot be replayed by the current implementation of IoTReplay, because the tool generates extra attack packets (e.g., ARP spoofing) that are not destined to the operational IoT devices.

[1]The Tycam LTE camera was manufactured by Jimi with firmware IH21_-30_35V4_M5H_IR_V25 and mobile app of Version 4.12.0. The camera uses a closed-source library called TUTK from ThroughTek Co., Ltd. The reboot attack requires modification of a UDP packet sent to the camera. We have reported this issue to the vendor through its official website.

D. Execution performance results

1) UI performance degradation.: On an operational mobile phone the IoT app is instrumented by VirtualXposed to intercept system APIs of interest. As a human user interacts with the operational IoT app through its UI, the additional latency caused by VirtualXposed and its modules may affect the usability of IoTReplay. By default, an Android device refreshes its screen at the speed of 60 fps (frames per second). Hence, each frame should be generated within 16 milliseconds to prevent unsmooth motions which are called janks. To measure the UI performance impact of VirtualXposed and its modules, we use the new *framestats* command for *gfxinfo* introduced in Android 6.0.

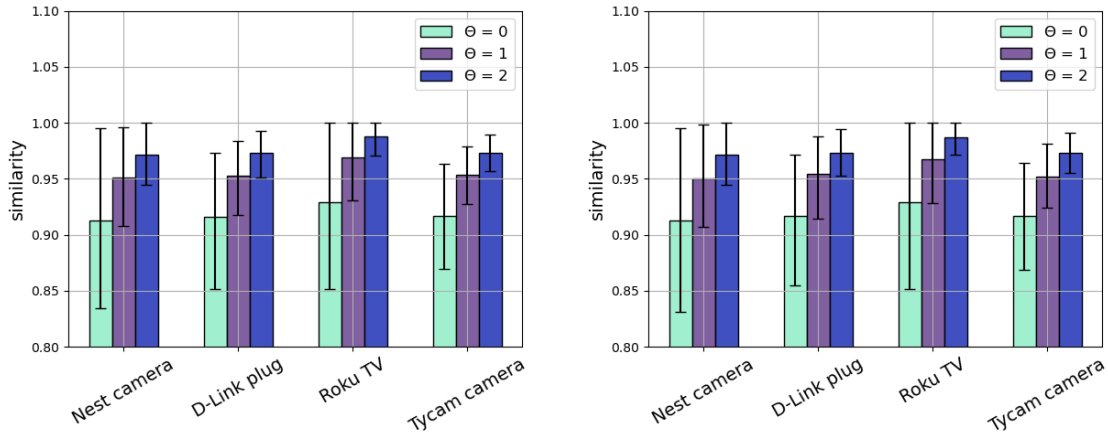
We run each IoT app on Samsung Galaxy S5, with and without VirtualXposed and its modules. Each application is operated by a user for about 5 minutes. The average frame latency over 10000 frames rendered is shown in Table V for each IoT app:

Table V tells us that the UI performance degrades only slightly because of VirtualXposed. The average increase in the frame generation time over the four IoT devices is only 2.19%. Even with VirtualXposed and its modules, the frame generation time is smaller than 16 milliseconds, which is needed to keep up with the pace of 60 frames per second on an Android device. The negligible performance degradation seen in Table V agrees well with our own experiences with operating the instrumented IoT apps.

2) Latency.: Latency measurements between the operational and shadow worlds can be classified as follows:

- **UI operations**: For each UI operation, we measure the difference between the time when a UI operation is sent by the operational mobile phone, and the time when its corresponding translated UI operation is received by the shadow mobile phone. The latency thus includes the processing time of a UI operation by the dispatcher and the network transmission delays.
- **Geolocation and sensor data:** The latency is measured by Xposed modules as the difference in time when the same system API hooked is invoked in the operational and shadow IoT apps.
- **Exotic messages:** This latency is measured as the time interval between the two gateways of the operational and shadow IoT devices.

The average latency over 10 runs, as well as its observed value range, is shown in Figure 4 for each type of IoT devices tested. We observe that the average latency measures are around 400$\sim$500 milliseconds, irrespective of the type of contextual events. Their latency measures are similar to each other because all of them are relayed through the same dispatcher.



(1) Online replay (2) Offline replay

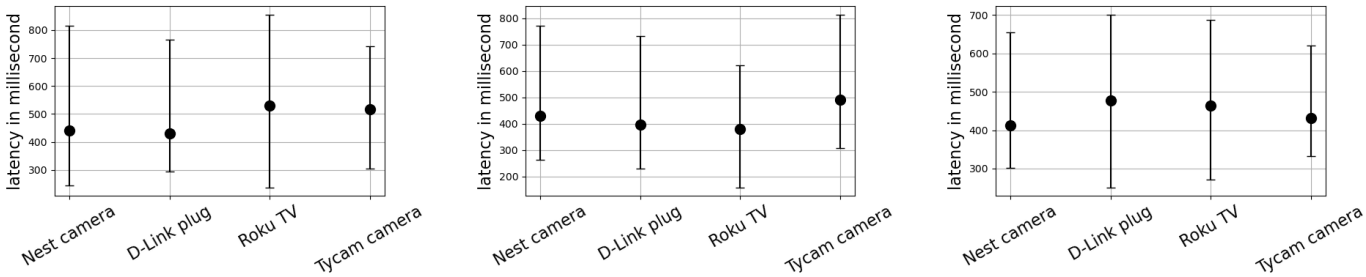
Fig. 3. Execution similarity. Each bar shows the average similarity score along with the minimum and maximum observed.

TABLE IV
REPLAYABILITY OF EXTERNAL ATTACKS. IN EACH ENTRY "a/b", a AND b GIVE THE TEST RESULT OF ONLINE AND OFFLINE MODES, RESPECTIVELY.

IoT Device	Nmap scanner	IoTSeeker	IoT Inspector	Reboot attack
Google Nest camera	Yes/Yes	Yes/Yes	No/No	-
D-Link smart plug	Yes/Yes	Yes/Yes	No/No	-
Roku TV	Yes/Yes	Yes/Yes	No/No	-
Tycam LTE camera	Yes/Yes	Yes/Yes	No/No	Yes/Yes

TABLE V
AVERAGE FRAME GENERATION TIME

IoT App	Average frame latency (w/o VirtualXposed)	Average frame latency (with VirtualXposed)	Overhead
Google Nest camera	11.43 ms	11.62 ms	1.66%
D-Link smart plug	10.67 ms	10.83 ms	1.50%
Roku TV	10.25 ms	10.31 ms	0.59%
Tycam LTE camera	12.58 ms	13.21 ms	5.01%

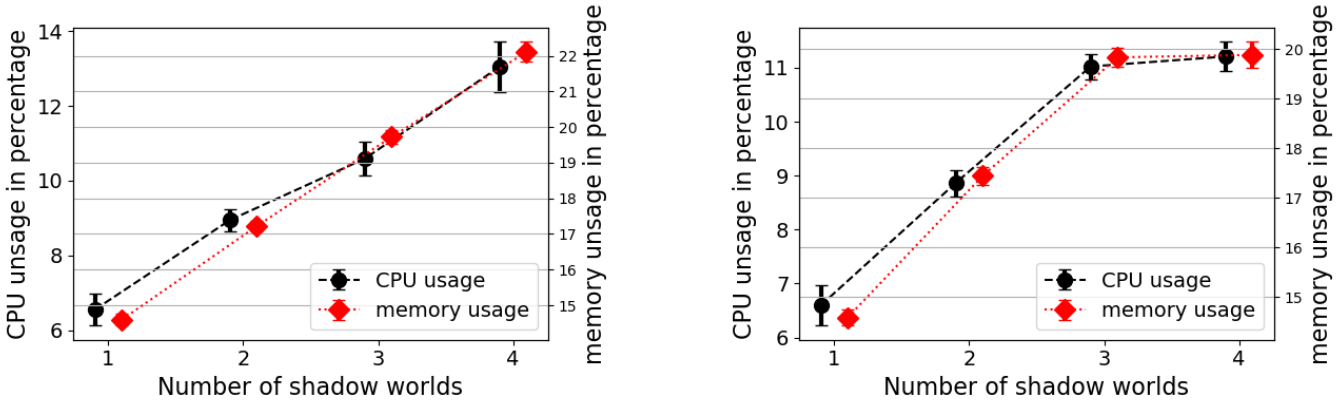


(1) UI operations (2) Geolocation & Sensor (3) Exotic messages

Fig. 4. Latency of different contextual events

3) Scalability.: We measure the resource usage of the workstation which hosts both the dispatcher and the AVDs under a vary number of shadow worlds. We perform two sets of experiments. The first set (*one-to-multiple*) includes parallel execution of one operational world and multiple shadow worlds for the Google Nest camera, and the second (*multiple one-to-one*) has one parallel execution of one operational world and one shadow world for multiple types of IoT devices. In the second set of experiments, we first order the IoT devices as follows: Nest Camera, D-Link smart plug, Roku TV, and Tycam LTE camera; in an experiment with k shadow worlds, we choose the first k IoT devices for testing.

Figure 5 gives the CPU and memory usage for both sets of experiments. In the one-to-multiple case, we observe that both the CPU and memory usages grow almost linealy with the number of shadow worlds: a new shadow world increases the CPU usage by about 1.7% and the memory usage by about 2.5%. In the multiple one-to-one case, when the number of



(1) One-to-multiple

(2) Multiple one-to-one

Fig. 5. CPU and memory usage of the workstation

shadow worlds increases from one to three, there is also a linear increase in CPU and memory usage: a new shadow world introduces an increase of about 2.4% in CPU usage and 2.0% in memory usage. However, when the number of shadow worlds increases from three to four, the CPU and memory usages both increase only slightly. The reason for this phenomenon is two-fold. First, the last IoT device added to the experiment is Tycam LTE camera, for which its IoT app runs on a real mobile phone instead of an AVD. Second, the computational overhead of running an AVD is much higher than that of the dispatcher.

VI. Related Work

The risks posed by vulnerable IoT devices have inspired many researchers to develop new methods to understand and enhance their security. Fernandes *et al.* performed thorough security analysis of Samsung's SmartThings IoT platform and found severe flaws that can lead to overprivilege and information leakage [13]. Jia *et al.* proposed a system called ContextIoT that can identify fine-grained contexts for sensitive actions and generate run-time prompts to assist IoT users with permission control [20]. Chen *et al.* developed a system called IoTFuzzer that can identify program-specific logic in IoT apps such as encryption functions and fuzz specific fields to expose memory corruption vulnerabilities in IoT devices [8]. Huang *et al.* developed an open source tool called IoTInspector to crowdsource labeled network traffic collected from smart IoT devices deployed within real-world home networks [18]. Celik *et al.* developed a system called IoTGuard that can monitor an IoT app's runtime behavior with a dynamic model and enforce safety and security for individual apps or sets of interacting apps [7]. Schuster *et al.* proposed environmental situation oracles to enhance access control for IoT platforms [25]. Our work on IoTReplay, which applies the record and replay technique to troubleshoot COTS IoT devices, is complementary to the large body of existing works in understanding and improving the pressing IoT security problem.

Record and replay is a classical technique for automatic testing of blackbox systems. PANDA is a reverse engineering tool that adds record and replay capabilities at the instruction level to the QEMU whole-system emulator for shareable and repeatable system diagnosis [10]. Malrec is malware sandbox system that takes advantage of PANDA's system-wide record and replay capability for fine-grained malware analysis with low computational overhead [26]. ReVirt is a system that applies virtual-machine logging and replay to improve the integrity and completeness of audit loggers in environments with non-deterministic attacks and executions [11]. OFRewind, which is built upon the split forwarding architecture of OpenFlow, enables record and replay debugging for large operational networks [30]. Targeting web applications, Warr is a high-fidelity tool that extends WebKit to record a user's actions such as mouse clicks and keystrokes and replays the interactions between the user and a web application [5]. Mahimahi is a set of composable shells that enables accurate record and replay for HTTP by emulating the multi-server nature of Web applications and using traffic isolation to minimize mutual interference across different instances [21].

More relevant to IoTReplay are those targeting Android applications. RERAN records and replays low-level events available as the */dev/input/event** device files on an Android device but is unable to deal with events available only through system APIs such as GPS locations and sensor data [14]. MobiPlay allows record and replay of not only UI events but also sensor data inputs by forwarding them to a server through a high-speed network connection, where the app is actually executed [23]. Motivated by fragmentation of the Android ecosystem, Mosaic is a system designed for cross-platform user-interaction record and replay through a human-readable platform-neutral intermediate representation [16]. VALERA is a stream-oriented record and replay system focused on comprehensive events on a smart phone, such as sensor and network inputs, event schedules, and inter-app communications based on intents [17]. Although our implementations of IoTReplay have been inspired by many previous record and replay techniques for Android systems, to the best of our knowledge, IoTReplay is the first of its kind in applying record and replay to troubleshoot COTS IoT devices.

VII. Conclusions

Numerous IoT devices shipped with weak security protection call for effective yet efficient methods for troubleshooting their problems encountered in the real world. This work presents the rationale, design, and implementation details of an edge-assisted system called IoTReplay, which uses record and replay to debug COTS IoT devices. Using four types of COTS IoT devices, we perform extensive experiments to show that IoTReplay is able to record and replay the execution sequences and behavior of the IoT devices with high fidelity. Our experimental results have also demonstrated that IoTReplay incurs negligible UI performance degradation to the IoT users and can be scaled to support parallel diagnosis of multiple COTS IoT devices.

Acknowledgments

We acknowledge the support of Critical Infrastructure Resilience Institute, a US Department of Homeland Security Center of Excellence, for this work. We also thank anonymous reviewers for their valuable comments.

References

[1] What Makes IoT so Vulnerable to Attack? *https://outpost24.com/blog/what-makes-the-iot-so-vulnerable-to-attack*, 2020.

[2] Nmap: the network mapper - free security scanner. *https://nmap.org*, Accessed in April 2020.

[3] Virtualxposed. In *https://github.com/android-hacker/VirtualXposed*, Accessed in April 2020.

[4] UI/Application Exerciser Monkey. *https://developer.android.com/studio/test/monkey*, Accessed in June 2020.

[5] S. Andrica and G. Candea. WaRR: A tool for high-fidelity web application record and replay. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*, pages 403–410. IEEE, 2011.

[6] Android developers. Understand the activity lifecycle. *https://developer.android.com/guide/components/activities/activity-lifecycle*, Accessed in April 2020.

[7] Z. B. Celik, G. Tan, and P. McDaniel. Iotguard: Dynamic enforcement of security and safety policy in commodity iot. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'19)*, 2019.

[8] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. C. Lau, M. Sun, R. Yang, and K. Zhang. IoTFuzzer: Discovering memory corruptions in iot through app-based fuzzing. In *Network and Distributed System Security Symposium*, 2018.

[9] Consumers International and Internet Society. The Trust Opportunity: Exploring Consumer Attitudes to the Internet of Things. *https://www.internetsociety.org/wp-content/uploads/2019/05/CI_IS_-Joint_Report-EN.pdf*, 2019.

[10] B. Dolan-Gavitt, J. Hodosh, P. Hulin, T. Leek, and R. Whelan. Repeatable reverse engineering with PANDA. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*, pages 1–11, 2015.

[11] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. *ACM SIGOPS Operating Systems Review*, 36(SI):211–224, 2002.

[14] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein. RERAN: Timing- and touch-sensitive record and replay for Android. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 72–81. IEEE, 2013.

[12] D. Etherington and K. Conger. Large DDoS attacks cause outages at Twitter, Spotify, and other sites. *https://techcrunch.com/2016/10/21/many-sites-including-twitter-and-spotify-suffering-outage/*, 2016.

[13] E. Fernandes, J. Jung, and A. Prakash. Security analysis of emerging smart home applications. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 636–654. IEEE, 2016.

[15] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek, and Z. Zhang. R2: An application-level kernel for record and replay. In *USENIX Symposium on Operating Systems Design and Implementation*, volume 8, pages 193–208, 2008.

[16] M. Halpern, Y. Zhu, R. Peri, and V. J. Reddi. Mosaic: cross-platform user-interaction record and replay for the fragmented Android ecosystem. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS'15)*. IEEE, 2015.

[17] Y. Hu, T. Azim, and I. Neamtiu. Versatile yet lightweight record-and-replay for Android. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2015.

[18] D. Y. Huang, N. Apthorpe, G. Acar, F. Li, and N. Feamster. IoT inspector: Crowdsourcing labeled network traffic from smart home devices at scale. *arXiv preprint arXiv:1909.09848*, 2019.

[19] International Data Corporation. The Growth in Connected IoT Devices Is Expected to Generate 79.4ZB of Data in 2025, According to a New IDC Forecast. *https://www.idc.com/getdoc.jsp?containerId=prUS45213219*, 2019.

[20] Y. J. Jia, Q. A. Chen, S. Wang, A. Rahmati, E. Fernandes, Z. M. Mao, A. Prakash, and S. Unviersity. Contexlot: towards providing contextual integrity to appified iot platforms. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'17)*, 2017.

[21] R. Netravali, A. Sivaraman, S. Das, A. Goyal, K. Winstein, J. Mickens, and H. Balakrishnan. Mahimahi: Accurate record-and-replay for HTTP. In *2015 USENIX Annual Technical Conference (ATC'15)*, pages 417–429, 2015.

[22] Palo Alto Networks. 2020 Unit 42 IoT Threat Report. *https://unit42.paloaltonetworks.com/iot-threat-report-2020/*, 2020.

[23] Z. Qin, Y. Tang, E. Novak, and Q. Li. Mobiplay: A remote execution based record-and-replay tool for mobile applications. In *Proceedings of the 38th International Conference on Software Engineering*, pages 571–582, 2016.

[24] Rapid7. IoTSeeker: Find IoT devices, check for default passwords. *https://information.rapid7.com/iotseeker.html*, Accessed in April 2020.

[25] R. Schuster, V. Shmatikov, and E. Tromer. Situational access control in the internet of things. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 1056–1073, 2018.

[26] G. Severi, T. Leek, and B. Dolan-Gavitt. Malrec: compact full-trace malware recording for retrospective deep analysis. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2018.

[27] F. Wei, S. Roy, and X. Ou. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *Proceedings of ACM Conference on Computer and Communications Security*, 2014.

[28] Wikipedia. Transport layer security. *https://en.wikipedia.org/wiki/Transport_Layer_Security*, Accessed in April 2020.

[29] Wikipedia. UDP hole punching — Wikipedia, the free encyclopedia. In *https://en.wikipedia.org/wiki/UDP_hole_punching*, Accessed in April 2020.

[30] A. Wundsam, D. Levin, S. Seetharaman, and A. Feldmann. OFRewind: Enabling record and replay troubleshooting for networks. In *Proceedings of the USENIX Annual Technical Conference*, pages 327–340. USENIX Association, 2011.