Android SmartTVs Vulnerability Discovery via Log-Guided Fuzzing

Yousra Aafer
University of Waterloo

Wei You *
Renmin University of China

Yi Sun, Yu Shi, Xiangyu Zhang
Purdue University

Heng Yin
UC Riverside

Abstract

The recent rise of Smart IoT devices has opened new doors for cyber criminals to achieve damages unique to the ecosystem. SmartTVs, the most widely adopted home-based IoT devices, are no exception. Albeit their popularity, little has been done to evaluate their security and associated risks. To proactively address the problem, we propose a systematic evaluation of Android SmartTVs security. We overcome a number of prominent challenges such as most of the added TV related functionalities are (partially) implemented in the native layer and many security problems only manifest themselves on the physical aspect without causing any misbehaviors inside the OS. We develop a novel dynamic fuzzing approach, which features an on-the-fly log-based input specification derivation and feedback collection. Our solution further introduces a novel external observer that monitors the TV-related physical symptoms (i.e., visual and auditory) to detect potential physical anomalies. We leverage our technique to analyze 11 Android TV Boxes. Our analysis reveals 37 unique vulnerabilities, leading to high-impact cyber threats (e.g., corrupting critical boot environment settings and accessing highly-sensitive data), memory corruptions, and even visual and auditory disturbances (e.g., persistent display content corruption and audio muting).

1 Introduction

Recent years have witnessed a runaway rise in the adoption of IoT devices aiming to embed the *smart* digital world into our surrounding physical environment, thus creating opportunities for a *smarter* life. According to [18], the number of connected IoT devices in use is expected to hit 18 billion in 2022, covering a wide variety of consumer products such as SmartTVs, Smart Bulbs, and Smart Thermostats. SmartTVs, which are TVs that can be connected to the Internet to provide a consumer-tailored entertainment experience, are expected to achieve a market value of USD 253 billion by 2023 [17].

*Corresponding author.

While these IoT devices are introducing previously inconceivable experiences, they are also opening doors for criminals to carry in *cyber* and *physical* harm. For example, security researchers [14] have discovered that it is possible to reveal WiFi keys through ringing IoT doorbells and even to open the door to attackers by exploiting smart voice recognition technologies [32]. Given the large market share of SmartTVs, the situation could be aggravating. Cyber criminals can exploit various attack vectors, physical or cyber. Although existing research [21, 26, 29] has shown that physical channels, such as crafted broadcast signals, are quite effective, exploiting such channels may have a lot of constraints in practice. For instance, the attacker has to be in the vicinity of the target SmartTV. Thus, a more attractive attack vector is to spread malware (e.g, through social engineering), potentially achieving more damages unique to the SmartTV ecosystem.

To proactively address this attack model, we propose to perform an evaluation of SmartTV system security, which entails a set of challenges, including identifying the potential exploit targets specific to SmartTVs (i.e., APIs for SmartTV functionalities), which often lie in both the Java layer and the native layer; efficiently generating proper inputs to trigger/execute those targets, which demands inferring input specification (from native implementation); and assessing test results which may lie in both the cyber space (e.g., segfaults) and the physical space (e.g., corrupted display and sound). Note that traditional static analysis is insufficient due to the heavy involvement of native code, hence hindering potential static inference of input specifications.

To tackle these challenges, we propose a novel approach that combines both static analysis and dynamic fuzzing techniques in a unique manner. Through static analysis, we pinpoint interesting customization targets that should undergo extensive testing for vulnerability discovery. We focus on custom public APIs, which vendors integrate into the original OS to trigger the SmartTV's peculiar functionalities.

Our dynamic fuzzing module features a novel on-the-fly log analysis that allows inferring input specification of target APIs and collecting execution feedback (i.e., if an exception

has been induced). In the context of SmartTV testing, input specification is very difficult to acquire as many of the APIs are implemented in the native layer, which is very hard to analyze due to the lack of symbols. We hence leverage the input validation messages emitted by such APIs when ill-formed inputs are provided, to extract input specifications (e.g. keywords, format, and value ranges). Specifically, the log analyzer can determine which log messages are related to input validation of the API under testing (note that such messages are buried in a large volume of other non-deterministic messages), and further classifies them based on the validation types. This is achieved by training a number of classifiers: We leverage the observation that native layer log messages share a lot of similarity with Java layer messages in terms of semantics. Therefore, we use static analysis to extract a large corpus of logging statements *from the Java layer* and reconstruct the log messages through string analysis. We then label if they are input validation related and if so, the validation category, through an automatic static analysis. Training with the corpus via natural language embedding allows the classifiers to predict (native) input validation messages.

The fuzzer also features an external observer to monitor *physical* properties of SmartTVs while executing the target APIs, Such physical manifestations cannot be captured by monitoring the internal execution state of the APIs. Specifically, we rely on an assistant technology to redirect the HDMI audio and display output to our external observer and employ state-of-the-art image and audio comparison techniques to detect potential anomalies caused by a test case execution.

We applied our proposed solution to systematically evaluate 11 popular Android TVBoxes from different vendors. Our analysis led to the automatic discovery of 37 unique vulnerabilities, including 11 high-impact cyber threats, 10 new memory corruptions, and 16 visual and auditory anomalies. Our study shows that these flaws are quite extensive. Each analyzed device contained 1 to 9 vulnerabilities; spanning reputable TVBoxes such as NVidia Shield and Xiaomi MIBOX3 and MIBOX S. We have responsibly reported the attacks to the corresponding vendors. NVidia has acknowledged the flaws as *Critical* and released a corresponding patch in its latest versions. Xiaomi has already fixed the identified vulnerabilities in MIBOX3 [1] and MIBOX S.

To prove that our detected vulnerabilities can be exploited in the wild, we carried out targeted attacks. Specifically, our fuzzing uncovered that an unprotected API on one victim device allows corrupting critical boot environment variables, leading to a complete device failure. We exploited other flaws to access highly-sensitive data, overwrite certain system files, delete directories and create hidden files in the internal storage, all without any permissions or user consent.

We further leveraged the physical-anomalies enabling flaws to conduct new attacks. Specifically, we exploited the exposure of the TV's display color aspects to manipulate them maliciously (e.g., flicker the brightness in high frequency), possibly affecting the viewer's visual health silently [35]. We exploited another flaw allowing to interrupt the HDMI interface to put the target TVBox in a fake powered-off mode, potentially threatening the user's privacy [9]. More details about the discovered attacks can be found in Section 9.5.

[1] The vulnerabilities were known-internally and fixed prior to our report.

Contributions. The scientific contributions of the paper are:

- *New Technique.* We develop a novel technique to automatically detect cyber and physical anomalies using a combination of static analysis and log-guided dynamic fuzzing. It provides a solution when instrumentation and collecting fine-grained execution feedback is not feasible. Our technique proved to be effective through discovering 37 unique vulnerabilities in 11 TVboxes.

- *New Findings.* We systematically evaluate Android SmartTV API additions. Our evaluation reveals security-critical cyber threats, previously-unknown memory corruptions, as well as *visual* and *auditory* interruptions causing a disarm of the TVs' basic entertainment functionality.

2 Background and Motivation

Android SmartTVs typically run a heavily customized version of the Android Open Source Project (AOSP), with additional hardware and system components to support the TV's functionalities. To understand the extent of deployed customization, we extracted custom system services and pertaining APIs in popular Android SmartTVs (details can be found in Section 9. We found that the number of custom APIs is high, reaching up to 101 in H96Pro. Since these services execute in the context of highly-privileged processes, an inadequate protection can be exploited to achieve various damages: traditional *cyber* attacks as well as *physical* damages unique to SmartTV – e.g., breaking basic functionalities through corrupting the display content or interrupting the HDMI signal. Note that manipulating such physical aspects is a privileged operation in AOSP - i.e., requiring system permissions such as `permission.CONFIGURE_DISPLAY_COLOR_TRANSFORM` and `android.permission.HDMI_CEC`. An unintentional exposure of such functionalities can be misused to affect the viewer's health (e.g., impairing the eyes visual performance through configuring a non-healthy color aspect or flickering the display light in a human-unnoticeable frequency [11, 35]).

Example. Consider the (native) API in Xiaomi MIBOX3 `SystemControl.XYZ(int x, int y, int w, int h)`, enabling to setup the HDMI display at a position (`x, y`) and with size (`w, h`). Through our analysis, we found that the API does not enforce any protection, thus allowing any app to mess up the display under specific parameters. Figure 1 shows MIBOX3's home screen before and after invoking

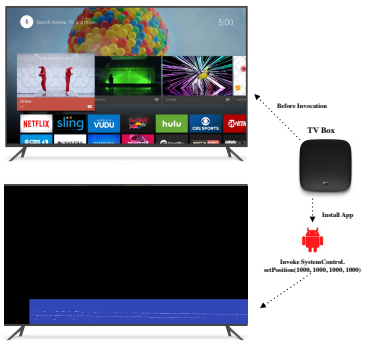

Figure 1: Display before and after invoking a vulnerable API

the API with the parameters `(1000, 1000, 1000, 1000)`. After the invocation, the projected display moves to the lower right corner of the screen with corrupt content as it cannot be rescaled to fit the provided size. In such scenario, the user will resort to rebooting the device to hopefully fix the display; however, it turns out that the malicious display parameters are persistent across reboots, making it impossible to fix the problem without a hard device reset. We envision such vulnerable scenario might be used even more maliciously by attackers. With the help of other side channels [31, 40], an attacker could mount a targeted DoS, where she corrupts the screen content each time a target app (e.g., Netflix) is playing on the top. The viewer will not be able to watch its content unless she pays some money. With the SmartTV ransomware already in-the-wild [15], it is reasonable to assume that the new APIs can be exploited for a similar purpose. A demo of the attack is available on the website [4].

The high privilege of the SmartTV services and the unique broad spectrum of attack consequences (e.g., cyber and physical) motivate the need for developing a specialized analysis framework to uncover hidden flaws. However, as vendors often do not provide access to their custom additions, often implemented as a hybrid form of multi-language (i.e., C/C++ and Java), the direct adoption of existing static analysis tools is infeasible. To address the limitation, we propose a fuzz-testing approach to detect potential anomalies.

A fuzzer's anatomy can be naturally broken into three major components: (1) an input generator responsible for generating test inputs to the program under test, (2) an executor that runs the target test case, and (3) a monitoring system that observes the target execution to detect potential vulnerabilities and provide feedback. Under the context of detecting SmartTV vulnerabilities, the design of an *effective* and *efficient* fuzzer is very challenging and entails obstacles in each of the components as follows.

2.1 Reverse Engineering Target Interfaces

Reverse Engineering Input Specification. The search space for valid test inputs to (complex) parameters is typically huge and random input generation can only explore limited (shallow) program paths. To tackle the problem, existing approaches resort to collecting information about the execution states to infer the program's feedback about a supplied input. Such information is usually collected through source code instrumentation or running the target program in an emulated environment. However, the lack of source code for custom SmartTV services and the inability of existing emulators to run proprietary services makes the task infeasible.

The only remaining channel that can be leveraged for the same purpose is Android logs. Logs often contain valuable information including input validation messages - e.g., stating legitimate input values or value ranges. The following log excerpt showcases an input validation log message:

```
1 BatteryChangedJob: Running battery changed worker
2 ImagePlayerService: max x scale up or y scale up is 16
3 DiskIntentProviderImpl: Successfully read intent from disk
4 MediaPlayer: not updating
```

As shown, executing a (native) target API `XYZ(int, int, float)` with random inputs `(20, 21, 20.2)` led to triggering the log message at line 2 - indicating the input is rejected because 2 argument values are > 16. While it is straightforward for a human analyst to extract this specific validation message, the automatic extraction is not: (1) The rejection does not correspond to a standard exception, but rather is represented by a message in free text form. (2) Identifying which parameters x and y refer to is not straightforward. (3) Automatically inferring the validation semantics of the message is difficult - i.e., integer range check, should match specific value, string of specific length, string with specific prefix, etc.

The log further depicts other challenges: deriving messages that are uniquely triggered by the target execution (lines 2 and 4) is not trivial. As shown, (1) log events triggered by the execution do not share a common identifier (e.g., Tag) since a target might trigger processes belonging to different components - i.e., `ImagePlayerService` and `MediaPlayer`. Besides, (2) target messages are often buried in a large number of unrelated messages; the logs often contain non-deterministic messages triggered by system events (e.g., lines 1 and 3) and hence it is difficult to draw a causal connection between a triggered API and related log messages.

Reverse-Engineering API Descriptors at Native Layer. In our analysis, we observe that SmartTV vendors add system services at both the Java and native (C/C++) layers (refer to Table 1 for a breakdown in the studied samples). Thus, it is essential to dissemble the framework binaries and correctly identify the native services' interfaces, i.e., the Binder transaction Ids corresponding to the remote functions, arguments types (including primitives and non-primitive), and order.

2.2 Assessing Execution Feedback

The monitoring system should observe cyber feedback and infer potential flaws. Besides, since the SmartTV functionalities

are often tied to *physical* components that need to be correctly working for a full-fledge experience, the monitoring system should detect changes in the physical feedback triggered by a target API. Note that corrupted physical state *may not* trigger any internal abnormal state – i.e., no exception or failure message is logged, hence would go undetected using existing testing tools. For example, executing the vulnerable API `SystemControl.XYZ()` with the malicious inputs does not trigger any failure messages although the display is corrupted.

3 Design Overview

Figures 2 illustrates a high level overview of our proposed system. Our design includes a Fuzzing Target Locator, a Log Analyzer, a Dynamic Fuzzer, and an External Observer. These components work cohesively to test Android SmartTV additions and detect potential anomalies. Given a SmartTV ROM, the Fuzzing Target Locator (A) identifies APIs of interest to be fed to the fuzzer. To ensure a comprehensive extraction of the SmartTV additions, it extracts custom APIs at the Java layer and recovers native APIs from the native layer. The Dynamic Fuzzer (C) generates test cases for each target API. Our system features a novel input generation approach to facilitate smart fuzzing. It leverages input-validation log messages to infer valid input specifications and drive the fuzzer towards exploring code regions guarded by these validation checks (i.e., without proper inputs, these regions cannot be explored). Specifically, the Log Analyzer (D) processes the log dumps of each target API's execution and looks for potential input validation messages using a set of classifiers - trained offline on a large corpus of Android logs (B). It then analyzes the input validation messages to extract input specs (e.g., value boundaries and constant values). The extracted specifications are in-turn fed back to guide the Dynamic Fuzzer in input mutation. This closed loop process – i.e., log-guided fuzzing – is carried out until no newer inputs can be recovered from the logs. During this loop, the Log Analyzer further analyzes the logs, looking for indications of newer state discovery (i.e., non-input related messages) or security related exceptions. (e.g., program crashes, faults).

Besides, to detect physical anomalies caused by the target API's executions, our design features an External Observer (E), responsible of monitoring the physical states. We trigger a standard visual and audio activity within the TVBox - through a custom MediaPlayer app that plays a short video clip before and after executing each case. If the execution outcome of the target API is normal, the video output should be the same. We then redirect the physical activity's output – i.e., HDMI signals – to the External Observer via an HDMI capture device. The Observer captures and compares the display and audio signals before and after each target execution using efficient image and audio comparison algorithms. It finally outputs alerts if discrepancies are detected. In the next sections, we explain the design details of each component.

4 Fuzzing Target Locator

To evaluate SmartTV additions, we focus on system services, customized or added by vendors. For this purpose, the goal of this section is to extract a list of APIs implemented at the Java or native system services. We compare the extracted APIs with those of reference AOSP models to locate custom ones.

4.1 Uncovering System Service APIs

Android native services expose their underlying functionality through dedicated APIs, invocable through the Binder IPC mechanism. To test these services, we have to go through the same interface. The Binder IPC mechanism allows apps to cross process boundaries and invoke exposed methods in the system service code. Upon system boot up, a system service publishes itself in the `ServiceManager` by supplying its service name and a service handle - i.e., an IBinder interface defining exposed methods. A client app process can invoke methods in the system service via binder transactions, which contain the method id to execute and raw buffer data (i.e., parameter and reply data). Specifically, each binder transaction follows a pattern depicted in Figure 3. After obtaining a service handle (IBinder interface) for a system service, the client process invokes a target function *A* within the service interface ①. The client proxy, which implements the service's IBinder interface, marshals parameter data through converting high-level application data structures into *parcels*, maps the specified method call to a raw *transaction id* and initiates the transaction call ②. Upon receiving the transaction, the service stub - also implementing the same IBinder interface - unmarshals parameter data, calls the actual server function ③, and marshals replies back to the client ④. The client proxy will then unmarshal the reply ⑤ and return the call ⑥.

To obtain available Java and native system service APIs, one can query the `ServiceManager` to list the registered services and retrieve the corresponding Service Interface Descriptors (i.e., string name of an IBinder Interface - native or Java). Once the interface names are identified, we need to locate the interface implementation in the Java and native layers. While the process of identifying Java level APIs is straightforward (i.e., extract methods in the bytecode of the corresponding service IBinder Interface), locating native layer methods is more challenging, since the binaries are largely stripped. To address the problem, we resort to extracting the native functions' interfaces at the low-level Binder IPC. That is, for each native API, we aim to recover the transaction id, parameter, reply data types from the native binaries; such that we can leverage the information to successfully invoke the API; basically replicating the system service's proxy transactions to invoke the service API (path ⑦ in Figure 3).

4.2 Extracting Native Function Interfaces

In this section, we explain how we recover the native function interfaces at low-level Binder IPC through binary analysis.

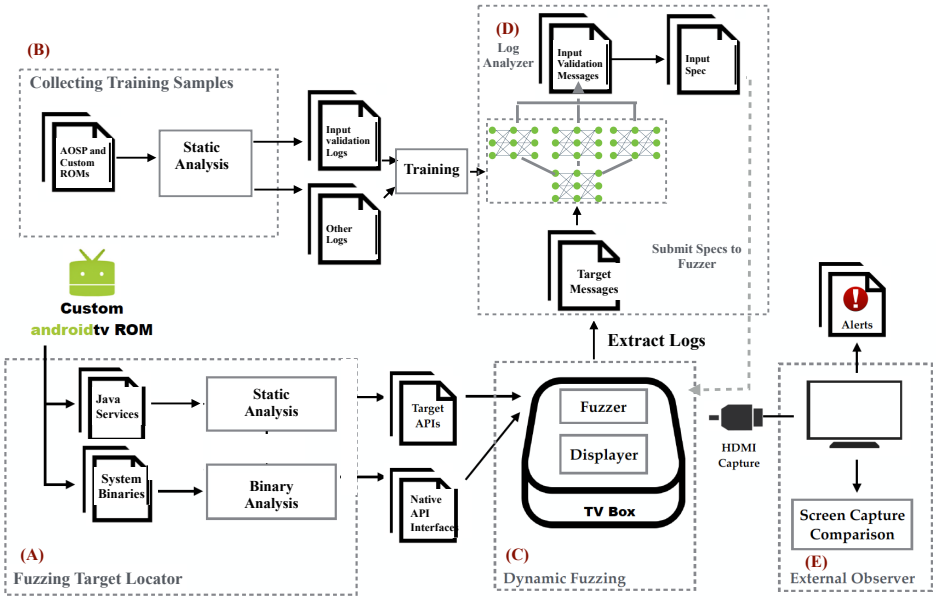

Figure 2: Approach Overview

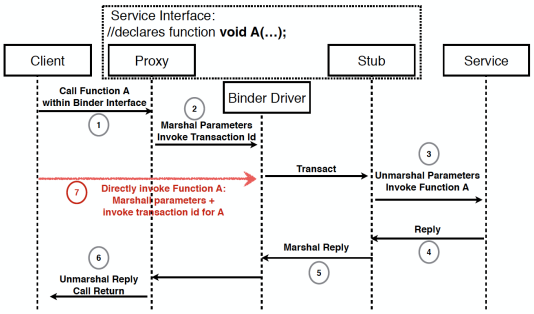

Figure 3: Remote Binder Transaction

To illustrate the process, we use a custom native service (*system_control*), whose recovered implementation is shown in Figure 4 (note that the recovered code is disassembled ARM code. The C++ code shown is only used for readability).

Available Information at the Binary Level. To facilitate discussion, we note that recovering native function interfaces at the low-level Binder IPC is possible thanks to certain available information. Throughout our analysis of system library binaries in the collected SmartTV samples (Section 9), we found that while most of the symbol information were stripped, certain basic symbols need to be preserved. Otherwise, the libraries cannot be properly used. Particularly, the names of interface classes (containing the API function implementations) were preserved in order to support runtime type check [13]. Besides, the Virtual Functions Table (VFT) were preserved due to a similar reason. We highlight in Figure 4 both available and absent symbols to ease discussion. Observe that the class name *BpSystemControl* is available. All the low level library APIs (e.g., *writeInt32* and *readInt32*) that are used not only by the proxy and the stub, but also by other classes, and methods inherited from public base classes (e.g., *onTransact*) need to retain their symbols. Methods specific to the (custom) service (e.g., *abc*) unfortunately do not have their symbols. In other words, given a native library, we know the list of function entries (from the VFT) but may not have their symbols or function call interface.

```
1  // Binder Proxy
2  class BpSystemControl : public BpInterface< SystemControl>{
3  ...
4    virtual status_t BpSystemControl::abc(...){
5      ...
6      Parcel data, reply;
7      data.writeInterfaceToken(SystemControl::getInterfaceDescriptor())
8      data.writeStrongBinder(c->asBinder());
9      data.writeInt32(a);
10     data.writeInt32(b);
11     remote()->transact(1, data, &reply);
12     status_t err = reply.readInt32();
```

```
1  // Binder Stub
2  class BnSystemControl: public BnInterface< SystemControl>{
3  ...
4    status_t BnSystemControl::onTransact(...) {
5      switch (code) {
6      case 1: //Transact ID
7        sp<ISystemControlClient> c =
               ISystemControlClient::asInterface(data.readStrongBinder());
8        int32_t a = data.readInt32();
9        int32_t b = data.readInt32();
10       err = mLocal->abc(c, a, b);
11       reply->writeInt32(err);
```

Figure 4: Snippets from `System_Control` Native Proxy and Stub (Blue boxes depict available symbols, while Red dashed boxes depict absent symbols)

Our Method. We propose the following methodology to reconstruct the function interfaces in the native layer: we begin by identifying the function bodies within the binder proxies (e.g. lines 5-12 of `BpSystemControl` in Figure 4), analyze them to extract the transaction id (1 in Figure 4) and parameter types (`Binder, int, int`) inferred through the proxy's marshaling methods `writeStrongBinder`, `writeInt32`, and `writeInt32`, respectively (in `BpSystemControl`). We finally

perform further analysis to handle the identification of custom data types (e.g., the recovered `Binder` is of sub-type `ISystemControlClient` as depicted in the return type of the stub's unmarshaling method `readStrongBinder`). In the following, we explain the individual steps.

(1) Identify function bodies of native binder transactions. As stated earlier, each system service has an interface descriptor. By performing a lookup for each descriptor, we can locate the library binary that contains the corresponding proxy and stub implementation. We rely on a naming convention of proxies and stubs, that is, **Bp**`InterfaceClassName` for proxies (e.g., `BpSystemControl`) and **Bn**`InterfaceClassName` for stubs (e.g., `BnSystemControl`). As such, based on the recovered interface descriptor (e.g., `SystemControl`) and the naming convention, we can infer the class names of proxy and stub. We then can look up all the function entries from the corresponding VFTs.

(2) Reconstruct function interfaces of low-level Binder IPC. We disassemble the binary and build a control flow graph for each function. As mentioned earlier, the parcel related function symbols are preserved in the disassembled code. For example, as shown in Figure 4 (blue boxes), `writeInterfaceToken`, `writeStrongBinder` and `writeInt32` were all preserved. To extract the arguments, we traverse the CFG starting from the argument parcel constructors till the destructor calls, and collect invocations to parcel read functions along the paths (e.g., `writeInt32`. To extract the binder transaction Id, we trace back the first parameter of `Binder()->transact(int, Parcel, Parcel, int)`. Observe that we do not rely on the symbol information for `transact()` to locate its invocation (since this particular function name is removed as illustrated in the dashed box in Figure 4), rather, we rely on its prototype `(int, Parcel, Parcel, int)` as depicted in line 11 of `BpSystemControl`.

5 Input Generation Through Log-Guidance

The fuzzer should be able to generate *smart* inputs to successfully trigger the target APIs and uncover potential vulnerabilities in deeper code regions, which may not be easily explored otherwise. Under the presence of input-validation checks, the API will terminate its execution if ill-formed inputs are supplied, without triggering its underlying functionality. To learn valid inputs, the existing approaches are mostly greybox or whitebox, meaning that they resort to collecting feedback about supplied inputs through source code (or binary) instrumentation or running the target program in an emulated environment. However, the lack of source code for native additions and the inability of existing emulator to run SmartTV native proprietary services make this approach infeasible. In other words, our fuzzer has to be blackbox.

To address this challenge, we resort to Android execution logs to derive valid input specifications. In fact, for debugging purposes, Android developers often accompany input-validation checks with logging statements, indicating specific details about the validation (e.g., reason for preventing the ill-formed data from entering, responsible parameter, and expected correct input value). As such, these log messages can be quite valuable in collecting feedback about the supplied inputs and inferring specifications.

To collect input specifications from a target API execution log, we propose to perform the following (Component (D) in Figure 2): we start by analyzing and processing the API's execution log to pinpoint *target messages* - i.e., those uniquely triggered by the API, since the execution log also contains a substantial number of other messages triggered by concurrent processes. We then process the filtered-out target messages to identify those that reflect an input-validation check. Since it is not trivial to distinguish input validations from other messages, we rely on a supervised learning based approach. Last, we analyze the selected input validations to extract input specs, which will be used to guide input generation for subsequent fuzzing. This latter may in-turn lead to new input validation messages and then another around of the aforementioned analysis. The log analysis (except the training process) is closely coupled with the fuzzing procedure.

More details about individual steps are discussed next.

5.1 Identifying Log Messages for Target API

Besides messages uniquely triggered by a target API, a target log dump contains other information. Concurrent processes may log messages to record program states, statistics, failures, etc. Such messages often substantially out-number a target API's messages. Furthermore, due to the non-deterministic nature of Android events, the target messages are often interleaving with messages from other processes.

A plausible approach to pinpoint an API's messages is to use the PID of the triggered process. However, an API's execution might span several processes with different identifiers. As such, we cannot rely on this approach. Similarly, we could plausibly rely on the `TAG` string - used by Android developers to identify the source of a log message - to group similar messages and reduce the search space. However, this approach is again infeasible since tags are non-unique across processes (consider the tag `DEBUG`, often used by different processes), and even vary within a process (e.g., tags often reflect class names while a process may span several classes).

To address this challenge, we resort to a statistical method. Intuitively, a message logged by a target API, should be contained in all log dumps obtained after its execution. In contrast, the target message should not occur in other log dumps, where the API was not executed. Thus, we could obtain the target messages through performing a set difference between log dumps obtained before and after invoking a target API. However, due to non-determinism, other unrelated messages might be fired during a target execution and thus would be

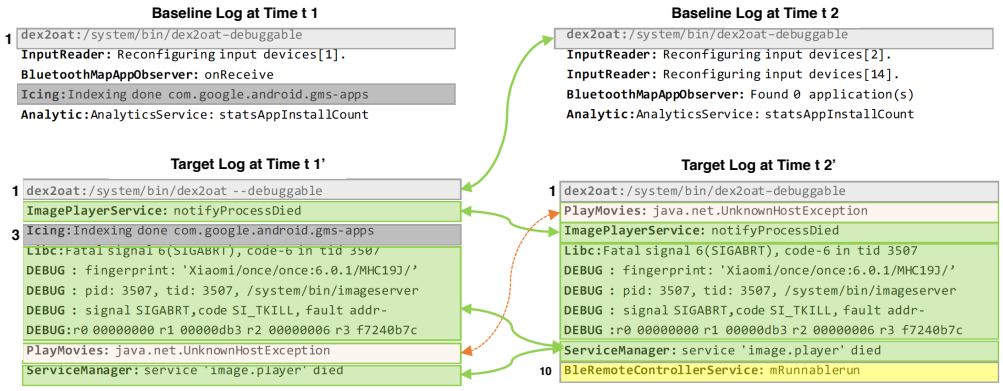

Figure 5: Log Excerpts before and after calling the (native) target API `ImagePlayer.XYZ()`

incorrectly predicted as a target message.

To illustrate this, consider the log dumps in Figure 5. The log dumps at the top correspond to log messages at two different timestamps without executing the target API (hereafter called *baseline logs*), while the excerpts at the bottom correspond to the log messages after triggering the target `XYZ()`, called hereafter *target logs*. Observe that *target logs* may contain both target messages and non-related messages.

As shown in the figure (highlighted in green), triggering this API leads to a libc fatal signal `SIGABRT` causing the image player service to die. To filter out the target messages, we perform set difference operations on all the target and baseline logs. We can remove the noisy messages: 1 and 3 from the target log at t'_1, and 1 from the target log at t'_2 since they were all observed in the baseline logs. However, the message *"PlayMovies: java.net.UnknownHostException"* spotted in the two target logs as well as the message "*BleRemoteControllerService: mRunnablerun*" dumped in the target log t'_2 would be misclassified as a target message since they were not observed in any baseline log.

Solution. To solve the problem, we rely on the empirical probability of a given message over a set of target logs and over another set of baseline logs to estimate its likelihood of being a target message. Intuitively, a message with a significantly higher empirical probability in the target logs and a lower one in the baseline logs indicates that it is likely a *target message*.

To calculate the probabilities, we begin by establishing a set of baseline logs, via capturing the log dumps at timestamps t_1 to t_{10} - while launching a dummy app in each execution. During fuzzing, we collect a set of target logs by executing the target API repeatedly over timestamps t'_1 to t'_{10}. Note that the design choice of launching the dummy app in the first scenario aims to avoid flagging messages triggered by Android app launching process as target messages. Afterwards, for each message i in a target log, we check whether it occurs in the other target and baseline logs to calculate the following score, reflecting its likelihood of being a target message:

$$score(i) = \begin{cases} 1 & \text{if } \frac{n_i^{target}}{N^{target}} \geq 0.9 \text{ and } \frac{n_i^{baseline}}{N^{baseline}} \leq 0.1 \\ 0, & \text{otherwise} \end{cases}$$

where n_i^{target} and $n_i^{baseline}$ are the frequency of the message i appearing in the target logs and the baseline logs, respectively. N^{target} and $N^{baseline}$ are the number of the target and baseline logs, respectively - (e.g., 10 each). The thresholds 0.9 and 0.1 are empirically selected to tolerate the inherent uncertainty. Performing a pairwise comparison over all messages to find whether a message appears in a specific log is quite expensive and would not scale. To tackle the problem, we utilize (1) optimization strategies to reduce logs through removing similar messages and (2) efficient string similarity measures to allow fast and scalable calculations. More details in Appendix.

5.2 Identifying Input-Validation Messages

Now that we have filtered log messages unique to the API execution, we aim to extract *input validation messages* (if any) since they can be helpful in inferring input specification for our fuzzing process. While it is easy for a human analyst to identify such messages, the automatic identification is not. Input-validation messages are quite diverse, featuring different syntactic structures, yet implying semantic similarities. As such, a simple whitelisting approach (e.g., relying on occurrences of specific keywords) would not suffice.

Consider the following input validation messages logged by the Java APIs `playSoundEffectVolume`, `dispatchFocusChange` and by the native API `AudioFlinger.createTrack` (extracted from AOSP):

```
1 public void playSoundEffectVolume(int t, ..) {
2   if (t >= 9||t < 0) Log.w(T, " Value" + t + "out of range"); ...
3 public int dispatchFocusChange(AudioFocusInfo a, ..) {
4   if(a == null) throw IllegalArgumentException("Illegal null Info")
```

```
1 sp<IAudioTrack> AudioFlinger::createTrack(int t, uint32_t r}
2   if (t >= AudioTrack::NUM_STREAM_TYPES) {
3     LOGE("invalid stream type"); goto Exit;}
4   if (r > MAX_SAMPLE_RATE || r > mSampleRate*2) {
5     LOGE("Sample rate out of range: %d", mSampleRate);
```

As shown in the snippets, the native and Java APIs log syntactically similar messages - i.e., containing the same keywords to indicate a range check *"out of range"*. More importantly, they also output *semantically* similar messages, which do not necessarily follow a similar syntactic structure - e.g., *"invalid type"* and *"illegal null"*. This indicates (1) the feasibility of learning from log messages in the Java implementation to classify log messages from the native implementation; and (2) the need of a sophisticated NLP technique as a simple syntax driven method (e.g., keyword lookup) is insufficient.

Our method. We develop a novel method, leveraging the observation that a large number of logging statements can be statically extracted from the *bytecode* of Android ROMs. Through string analysis, we can reconstruct a large set of log message templates (i.e., parameterized strings), and through static taint analysis, we can determine and label if they are input-validation related. As such, we can use the labeled message templates to train a classifier to predict if log messages by *native APIs* (collected in the earlier step) are for input validation. Note that due to the lack of symbols and the difficulty of string and taint analysis on native code, we cannot directly determine if a native log message is for input validation. Additional classifiers can be constructed to determine fine-grained categories (e.g., range and parameter equality checks). It is worth noting that a more simplistic rule-based pattern matching approach might also work for our scenario. However, it may require comparing each target log message with the whole set of labeled samples to determine if it is validation related and its fine-grained category; and substantial manual efforts may be needed to construct the rules.

Figure 6 outlines our overall procedure for extracting input validation messages. Component (A) collects and automatically labels the training samples (∼57000 messages) from various Android frameworks. Component (B) uses the training corpus to train different classifiers: we leverage word2vec, the state-of-the-art predictive model for learning word embedding in raw text to build a feature vector for each message. We then train a CNN classifier with the feature vectors as the first layer of CNN. Details are discussed next.

5.2.1 Static Analysis for Training Samples Collection.

To build our training corpus, we perform an upfront static analysis of 6 Android ROMs – AOSP 7.0, NVidia Shield, Samsung Note10 (9.0), S9 (9.0), LG Q6 (8.1), and LG Vista (7.1). Note that we selected both AOSP and custom ROMs to take into account potential logging style differences introduced by custom ROM developers. As depicted in Figure 6 (A), for each ROM, we extract all entry points in the Java system services and manager classes - e.g., `buildRequestPermissionsIntent` and `request BugReportWithDescription`. We then build a CFG for each entry and traverse it starting from the root node to collect the following: (1) parameter related conditional nodes (e.g., `if (ArrayUtils.isEmpty(permissions))` and `if (ShareTitle.length() > 50)`), (2) log statements (e.g., `Log.d` and `Slog.d`), and (3) `IllegalArgumentException` related statements as they may indicate input validations. We then trace back the string argument in the logging statements to extract the messages, e.g., *"ShareTitle ... characters"*.

To differentiate input-related statements from other statements, the static analysis leverages the following definitions:

Definition 5.1: *A log statement is considered input-validation related if it is control dependent on an input-validation check.*

Definition 5.2: *An input-validation check n is a predicate that satisfies the following conditions: (1) at least one of the operands is a parameter; (2) a logging statement directly depends on it; (3) there exists a path from n to some exception (including return with error code) such that there is no other statement along the path except the logging statement (and its transitively data-dependent instructions).*

Intuitively, the termination must be solely caused by the parameter not conforming to the check in *n*. Back to Figure 6, both conditional statements are input validations since they are directly followed by a log statement or its transitive data-dependent instruction (e.g., `String err="shareTitle.."`), which is in-turn followed by a return. Consequently, the analyzer can determine that the first messages *"permission cannot be null"* and *"shareTitle should be less than 50 char"* are input validations, while *"Bugreport notification title.."* is not.

Handling String Operations. Log messages are often constructed by concatenating several sub-strings, including constants, parameters and return values of other functions. We use backward slicing and forward constant propagation to transitively resolve log arguments in log-related statements. Since parameters cannot be statically resolved (i.e., user-supplied as in *"Bugreport notification title "* + `shareTitle`), we use the place holder `$PARAM$` to denote their usage in the resolved log messages as shown in the figure.

Categorizing Input-Validation Messages. Input validations convey *different* parameter properties. We hence categorize input validation messages into sub-classes, each denoting a specific validation property of the message, depending on the preceding input-validation check - e.g, equality check, size check, non-empty string/buffer check. For example, the message *"permission cannot be null or empty"* can be classified into the category `StringNotEmpty` since the preceding predicate's first operand is a call to the method `ArrayUtils.isEmpty` on the supplied parameter. Similarly, the second message *"shareTitle should be less .."* is classified to the category `StringLength`. Note that here the goal is to label messages with their sub-category such that classifiers can be trained to classify log messages from native code.

The analysis yielded 56315 messages, with 6269 positive samples (with different categories) and 50046 negative samples.

(A) Training Data Collection (B) Log Classifier Training

Figure 6: Input Validation Classification

5.2.2 Log Classifier Training

As depicted in Figure 6 (B), the collected training data are used to build a number of classifiers. The first one determines if a (native) message is input-validation related. Additional classifiers were trained to determine a fine-grained class of input-validation messages. Here we use a CNN model with an embedding layer, two convolutional layers with max pooling layers and a fully-connected dense layer. Each convolutional layer is a one dimensional layer with 128 filters and a kernel size of 5. Each convolutional layer is followed by a max pooling layer with pool_size = 2. There is one dense layer following convolutional layers. We use the Adam optimizer, with 0.001 learning rate.

The figure further depicts an illustration of how we utilize the trained classifiers (in the fuzzing process). Given the message "`$PARAM$ size should be at least 9`" in a native API's execution log, the classifiers predicts its category `RangeCheck` and generated the spec value 9.

6 Dynamic Fuzzer.

To uncover potential vulnerabilities in the collected target APIs, the Dynamic Fuzzer (C) in Figure 2 generates test cases for each API and executes it within the corresponding SmartTV. During this process, it leverages the execution log to iteratively learn and generate valid inputs, and to assess the execution output - i.e., (1) a new log message indicates the fuzzer has reached a new execution state, and (2) the occurrence of certain keywords indicates anomalous states.

Specifically, given a target API, the fuzzer starts without any input specification. It invokes the API with a random input and performs the log analysis in Section 5 to identify, classify input-validation messages for the target API, and extract input specification, if any. We use an example depicted in Figure 7 to walk through the procedure. Here, the target API is native in the `image.player` service, identified by the recovered function interface: transaction Id 5 and parameters (`float, float, int`). At the beginning (iteration 1 in Figure 7), the fuzzer randomly generates a value for each input parameter (e.g., `100, 11`, and `102`). The resulted log is passed to the Log Analyzer, which recognizes the target messages (highlighted in the log dump in iteration 1). The target messages are fed to our trained classifiers to pinpoint input validations and extract possible input specs, namely, a valid range for x (which we do not know to which parameter it actually corresponds).

In the second iteration, the fuzzer speculates that x denotes the first parameter, hence generates a value 10 within the range, without changing the second and third parameters. The resulted log messages disclose a new input validation, indicating a parameter equality check: x and y should be the same. This implies (1) the speculation of x being the first parameter is likely correct; and (2) another parameter y should be identical to x. Note that a wrong speculation can be inferred by observing the same input validation failure message.

In the third round, the fuzzer speculates y denotes the second parameter and hence sets it to 10. Although the resulting messages did not yield new input validations, they still indicate valuable information; the fuzzer was able to reach a new execution state of the target API thanks to the inputs learnt during the previous iteration. At this stage - since no more inputs can be extracted, the fuzzer starts a random mutation procedure. Specifically, it randomly samples within the legitimate value ranges. In addition, it also samples beyond the legitimate value range of each variable while fixing the (legal) values of other variables. The detailed algorithm is elided.

To detect potential anomalies triggered by a test case, the Fuzzer leverages two channels. On one hand, it inspects the execution log to spot certain messages signaling cyber anomalies (e.g., segmentation faults using keyword lookup). On

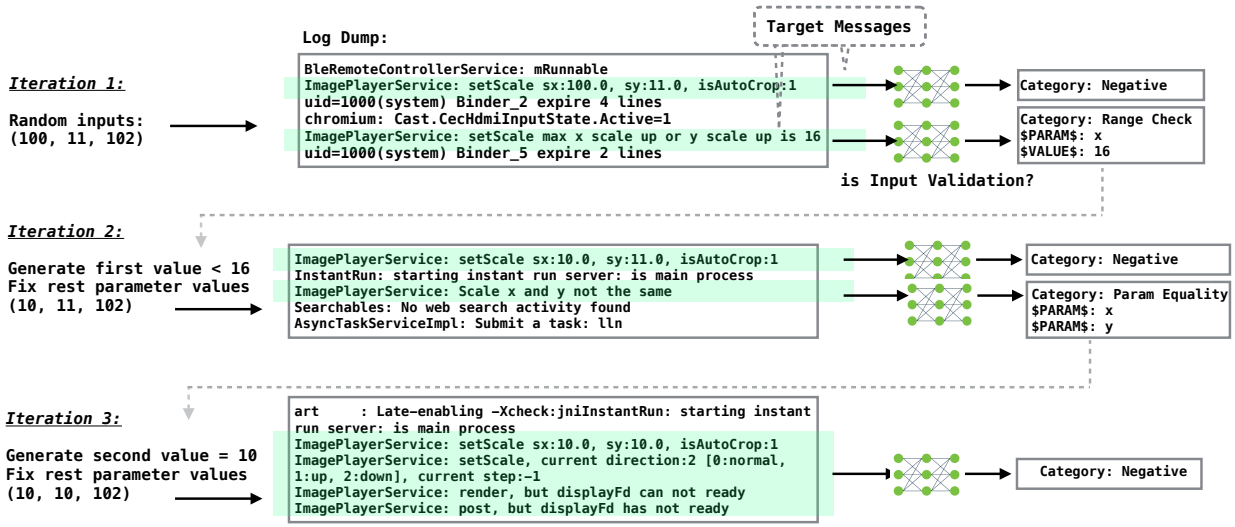

Figure 7: Extracting Seed Inputs from Log dumps for Native API -Transaction Id 5, (float, float, int)

the other hand, it employs an external observer to detect visual and audio anomalies that cannot be captured within the SmartTV. Next, we discuss the details of the external observer.

7 External Observer

To detect visual and auditory anomalies, our fuzzing framework features an external observer, responsible of monitoring the physical states. The fuzzer triggers a MediaPlayer to play visual/audio content before and after the test execution, and leverages an HDMI capture device to redirect the output signals to the Observer for comparison. We opted to capture the content before and after the test execution - rather than during the execution, to ensure capturing persistent effects. For the ease of comparison, the visual content is a still image video with a 1 sec audio clip. To mask other screen content (e.g., clock), the player plays in full screen. We suppress other non-deterministic factors (e.g., notifications) by disabling the corresponding components. While the fuzzer and MediaPlayer are running within the SmartTV as two independent apps, the Log Analyzer and Observer (Figure 2) are running out-of-box in a desktop computer. Detecting an abnormal physical state is performed by comparing captured states before and after executing the test case. We use standard image and sound waveform comparison algorithms to measure the observed visual and audio differences. Details are elided.

8 Implementation

Our static analysis is implemented on top of WALA [10], which provides comprehensive analysis support for Dalvik code, including call graph and control flow graph construction, and dependence analysis. Our binary analyzer is build on top of Radare2 binary analysis framework [6]. We build the neural network models for classification using the open-source neural-network library Keras [5]. Our fuzzer is implemented on top of Randoop [7], a unit testing tool for Java that randomly generates sequences of method invocations for the classes under test and uses the results of the execution to create assertions capturing the behavior of tested classes. As it is not directly suitable for our testing goal, we customize Randoop in the following four aspects. First, we ported it as an Android app that executes the target APIs within a background service. Second, we modified the test generation process to leverage the output gained from the Log Analyzer. Specifically, if the output reflects an input specification for a parameter, the test generation process constructs inputs conforming to the specs. If the output reflects a new log state (not input related), the test generation process fixes the current input and moves to fuzzing other parameters. Third, we further modified the test generation process to resort to random fuzzing when no insights can be gained from the Log Analyzer. Last, we extended the error detection module of Randoop with our cyber / physical anomalies detection logic.

9 Evaluation

We run our log-guided fuzzing on 11 Android TVBoxes. We discovered 37 security-critical flaws leading to various cyber attacks (11), physical disturbances (16) and memory corruptions (10). We reported the vulnerabilities to the responsible vendors: NVidia ranked the cases as Critical and has already patched them. Xiaomi has fixed the flaws. Here we report the evaluation results of our proposed technique.

9.1 SmartTV Device Collection

Our samples include 11 popular Android TVBoxes, ranking high according to different buyers' guides in North America

Table 1: Target TVBox Devices.

Device	Vendor	OS	Services	Recovered APIs	API Breakdown		
					Native	Java	Hybrid
MIBOX3	Xiaomi	6.0.1	5	49	49	0	0
X96	Ebox	6.0.1	7	101	91	6	4
RK MAX	RockChip	6.0.1	6	76	0	34	42
SHIELD	NVidia	7.0	17	73	33	13	27
X3	ZXIC	7.1	1	12	0	0	12
H96 Pro +	Ebox	7.1	7	95	85	6	4
V88	RockChip	7.1.2	7	75	0	19	56
MIBOX S	Xiaomi	8.1	2	31	0	0	31
RK3318	RockChip	9.0	1	29	0	0	29
Q+	CAT95S1	9.0	4	25	17	0	8
GT King	Beelink	9.0	1	37	0	0	37

Table 2: Log Dumps Statistics per API.

	Baseline Log		Target Log		Target Messages	FP	FN
	Raw	Reduced	Raw	Reduced			
Avg size	332 KB 2900 Lines	81 KB 376 Lines	21 KB 141 Lines	13 KB 79 Lines	7 Avg 139 Max	16%	5%

and Europe [1, 16, 19] from Dec 2018 to May 2020. Due to geographical restrictions (i.e., certain TVBoxes are not sold in our region), some of the samples are *variants* of the models listed in [1, 16, 19]. Note that we could have expanded our testing to include TVs with built-in monitors (e.g., Sony Bravia). However, due to their cost difference, we limited our testing to the TVBoxes. As shown in Table 1, our samples cover 8 vendors and operate Android versions 6.0.1 to 9.

Breakdown of Vendor Additions. Columns 4-5 of Table 1 show the number of custom system services and APIs. On average, the SmartTVs include ∼6 custom services, with NVidia Shield having the highest number (17). These services introduce all together 603 new APIs. Columns 6-9 further depict the implementation style of these APIs: the Java ones account for 13% on average. This clearly justifies the need for our proposed testing, as it is very difficult to infer any implementation details of the rest APIs with the current state-of-the-art Android binary analysis. Note that Java and hybrid APIs are recovered through static bytecode analysis.

9.2 Recovered Native Functions Interfaces

Column 6 in Table 1 shows the number of native APIs recovered through our proposed binary analysis. In total, we recovered a total of 275 native API interfaces, spanning 5 devices. As shown, these native APIs account for the majority of overall recovered APIs (44%). We note that that certain vendors (e.g, ZXIC, RockChip, Beelink) did not introduce custom system services at the native layer. Rather, their newly introduced services are all defined at the Java layer. Hence, there were no recovered native APIs in those devices.

Table 3: Trained Classifiers Accuracy

Classifier	# Positive Samples	Accuracy (%)	Precision (%)	Recall (%)	F1
InputValidation	6269	95.02	94.19	91.65	92.9
Numerical const Equality	131	99.64	76.78	82.17	79.22
Range Check	372	99.82	95.25	98.11	96.62
Param Not 0 / NULL	3684	97.59	92.68	96.42	94.5
String Not Empty	189	99.78	88.4	96.58	92.23
String const Equality	259	99.71	83.22	93.23	87.86
String Prefix Equality	186	97.66	87.17	94.48	80.43
String Length Equality	232	95.13	79.78	83.07	81.63
String is File	107	98.77	77.65	84.12	78.98

Due to the lack of ground truth (no symbols), we cannot explicitly validate the completeness of the recovered interfaces (i.e., number of recovered native functions), nor their correctness (e.g, parameter types and count). To approximate the completeness and correctness of our recovery approach, we rely on the intuition that custom native APIs *might be used* by other components in the system. Even if the registration site is at the native layer, other system components and apps at the Java layer can still retrieve an instance of the native services' Binder proxies (`IBinder` instance), using `ServiceManager.getService("service_name")` (where "`service_name`" is the native service name) and invoke corresponding native APIs through `IBinder.transact()`. Hence, by statically retrieving invocation sites to these native APIs and cross comparing them with our recovered interfaces, we can approximate the validity of our approach. Note that this approximate solution only samples the entire space as certain native APIs might be exclusively used by native components. Our solution works as follow: we start by dissembling framework and preloaded apps and extract corresponding entry points - i.e., public APIs in framework classes, public methods in app components, such as `onCreate` in Activities and `onReceive` in Broadcast Receivers. We then build a CFG for each entry, traverse it to identify invocations to `ServiceManager.getService`, and perform lightweight data-flow analysis to extract the supplied `service_name`. If it matches a native service, we use def-use analysis to locate invocations to `transact()` on the returned IBinder instance and extract the parameters: transaction code, data and reply parcels. We perform further def-use analysis on the parcels to extract interface parameter types (see Section 4).

Table 4 shows the comparison results for the static analysis and the interface recovery module (Section 4) for the ROMs defining native service(s). With the exception of Q+ in which our static analysis did not find any used native APIs, it located the usage of ∼73% of recovered APIs in the rest 4 ROMs. Specifically, it identifies 9 native system services (Column 2), and 184 native APIs (Columns 3 and 4) used within the Java framework and preloaded apps (we note that APIs used in the apps may also be used within the framework). As shown in Column 5, each one of these used APIs had a perfect match in our recovered API set - hence validating the correctness of our interface recovery approach, and highlighting its high coverage (no API has been missed).

Note that although we cannot corroborate these results for the rest unused APIs, the fact the fuzzer can execute them successfully implies that the interfaces are likely correct.

Table 4: Recovered Native Interfaces Validation Results

Device	# Used Native Services	# Used Native APIs		% APIs matching Recovered Interfaces	% Used APIs / Recovered
		Framework classes	Preloaded apps		
MIBOX3	2	43	7	100%	95.5%
X96	3	70	38	100%	76.9%
SHIELD	1	13	13	100%	39.3%
H96 Pro+	3	71	36	100%	83.5%
Q+	0	0	0	NA	0%

9.3 Evaluation of Log Analysis

Here we evaluate the effectiveness of our proposed strategies to generate input specifications from the analyzed logs.

Identifying Target API Log Messages. Table 2 reports statistics of the analyzed log dumps. Columns 2 and 4 show the average size of a raw baseline log (i.e., before API invocation) and a raw target log (i.e., after invocation), respectively. As shown, the size of target logs is smaller, since we purposely clear the log buffer before each API execution. Columns 3 and 5 report the average reduced size of baseline and target logs, respectively. The reduction is performed through log normalization (e.g., normalizing concrete values to a common symbolic space holder) and redundant message removal. As shown, the reduction yields an average 75%, 38% decrease of the raw baseline and target logs size, respectively, allowing efficient analysis. Column 6 presents the average count of log lines that our statistical analysis flagged as target messages. As shown, the APIs triggered 7 messages (avg). Observe that some APIs triggered no target messages at all, while others triggered up to 139 messages. To measure the FP and FN of our statistical method, we manually inspected the logs of 150 APIs. As shown, 16% of the cases were incorrectly flagged as target messages, meanwhile, 5% were missed by our method.

Identifying and Classifying Input-validation Messages. Table 3 reports the performance of a few classifiers. Column 2 reports the positive sample size for each classifier. Note that the first classifier - for predicting input validations - was trained on the whole dataset (∼57000), while the rest classifiers - for predicting input validation categories - were trained only on input validation messages (i.e., 6269). To evaluate the constructed models. We used the standard 10-fold cross-validation. We also ran it 10 times. As shown in Table 3, most of the classifiers achieve very good accuracy, precision and recall. Certain classifiers exhibit relatively lower precision and recall due to the smaller positive sample size.

9.4 Testing Evaluation.

Testing Setup. We conduct our static and dynamic analysis, log analysis, state capture and comparison in a 4-cores computer (Intel®Core™ i7-2600 CPU @ 3.40GHz). To redirect the HDMI signals to the external observer, we use an HDMI Video Capture Device (USB 3.0 1080P 60 FPS Video and Audio Grabber). We use adb for testing orchestration. Our conducted fuzzing takes on average ∼16 sec per test case.

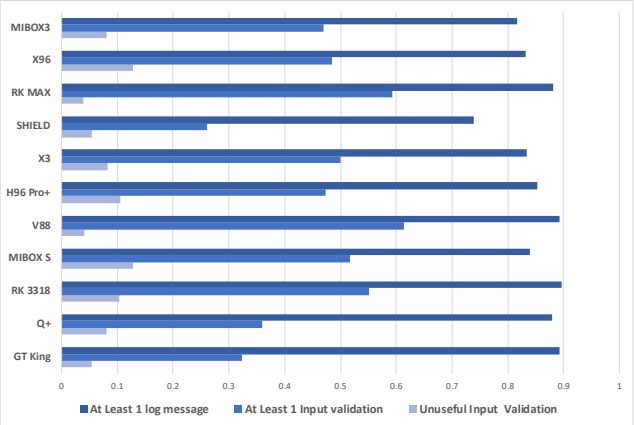

Figure 8: Availability and Breakdown of Observed Logs

More details can be found in Appendix.

Availability of Logs. A basic requirement for the success of our proposed log-guided fuzzing is the availability of log messages, particularly those related to input validations. Besides, we observe during our testing that not all input validations are *equally useful* for instructing the fuzzer to generate valid inputs; messages such as *"invalid value"* or *"illegal input"* do not contain specifications about the inputs. Thus, another important requirement for the success of our strategy is that log messages should be useful enough to guide the fuzzer to generate valid inputs and subsequently discover newer states. We report in Figure 8, a detailed breakdown of these log criteria observed during our testing of the APIs per SmartTV. As shown, on average 87% APIs triggered at least 1 log message and 46% triggered at least 1 input validation; meaning that 54% of the APIs do not have any input validation. Besides the reason that developers may not wish to log failed validations, this is also due to the fact that certain tested APIs had void parameters[2]. As shown in the same figure, 6.5% APIs triggered a non-useful input validation, meaning that our technique is most beneficial in the rest 39% cases.

Efficacy of Log-Guidance in Testing the APIs. To showcase the significance of our log-guidance, we count the number of input validations observed over the testing of an API and its effectiveness in uncovering new log states. We define a new log state as a unique set of target messages dumped during an API's execution, not seen in any previous testing iteration. The results are shown in Figure 9. The plot reads as follows: the x-axis depicts the # of unique log states and the y-axis shows the # of observed input validations. The bubbles depict the percentage of the tested API exhibiting a unique x-y combination. For instance, the 15% bubble at (2,1) means that 15% of the APIs have one (useful) validation message, which is leveraged by our technique to discover 2 unique log states. All the bubbles above the x-axis, which sum up to 46%, denote our technique can yield at least one new state.

[2]We test APIs with void parameters once, since they might also contain vulnerabilities.

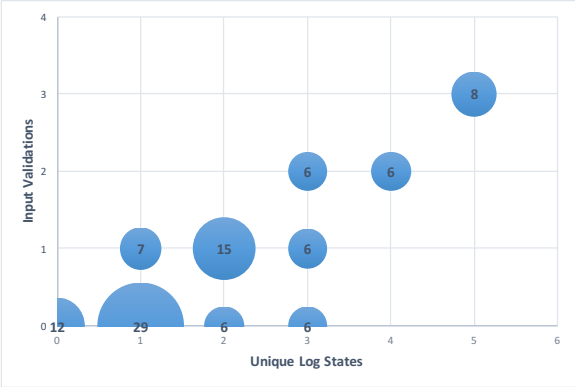

Figure 9: Significance of Log-Guidance over the Tested APIs

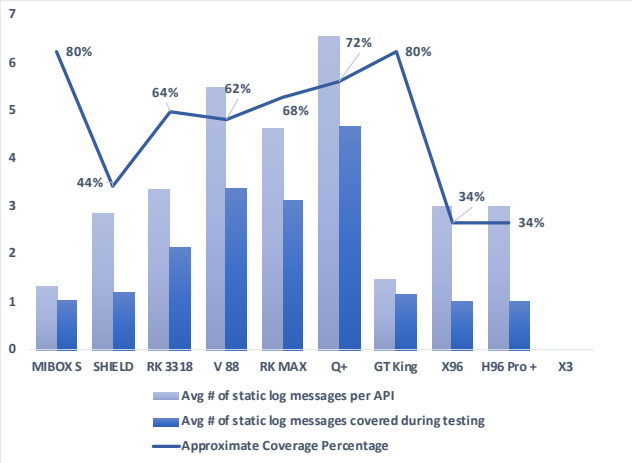

Figure 10: Coverage Approximation

Code Coverage Approximation. Due to the lack of SmartTV additions' source code, leveraging static code instrumentation to trace exercised code regions is not feasible. Leveraging dynamic binary instrumentation for the same purpose is not feasible either since we cannot run the instrumented binaries in the (unrooted) smartTVs nor use existing emulators because of hardware dependencies. Nonetheless, to gauge the code coverage of our approach, we propose a simple approximation for the Java-level APIs (pure and hybrid implementations). Since we can only observe the execution log during testing, we propose to statically extract all messages logged by a target API's implementation and compare them with those observed during the execution. Ideally, the static extraction of the log statements should be performed in a path-sensitive fashion such that each code path is mapped to an ordered sequence of (potentially) logged messages. A sequence matching the log of a particular execution of the target API indicates that the code path has been covered during execution. However, path-sensitive analysis is quite expensive and would not scale to tackle deep code paths. Thus, we propose to further simplify our approximation by statically extracting the log messages through path-insensitive analysis and looking for their occurrences in the target execution logs. Note that this approach is inherently limited since (a) not all code regions include a log statement and (b) more than one log message might reside within a code region.

Figure 10 depicts the achieved results per ROM. As shown, the approximate code coverage varies significantly among the ROMs, ranging from 34% in H96 Pro+ and X96 to 80% in GT King and MIBOX S. To understand the root causes behind the missed code regions (i.e., a statically extracted message was not identified in any execution log), we randomly selected 50 missed log messages and inspected their Java (bytecode) implementations. The majority of the missed log statements were guarded by conditions reflecting system-wide persistent settings (e.g, device model, build info) or environment-specific properties (like debug mode, network state). Since these conditions were not satisfied and remained unaffected during fuzzing, it is justifiable to miss those branch paths.

9.5 Findings

In this section, we verify the validity of our log-guided fuzzing in discovering security flaws in the tested SmartTV APIs.

Vulnerabilities Breakdown. By performing log-guided fuzz testing – alongside our specialized feedback monitoring, we uncovered 37 vulnerabilities in the collected TVBoxes. Table 5 reports a breakdown of the vulnerabilities. As shown, the flaws can be exploited to cause consequences of disparate security levels; ranging from high-impact cyber threats (denoted by CT), such as corrupting critical boot environment settings, overwriting system files and accessing highly-sensitive data, to memory corruptions (MC), and to significant physical disturbances (PD), affecting the overall SmartTV experience. As further shown in the Table, the vulnerabilities are pretty prevalent, affecting each tested device (1 to 9 unique flaws per device) and spanning over 11 distinct services and 30 unique APIs (17% of recovered APIs). Observe that some of the APIs led to different attacks depending on the supplied parameters.

A further dive into these vulnerabilities, particularly cyber threats and physical disturbances (see representative cases in Section 10) indicates that they are a consequence of weak and missing access control enforcement by the SmartTV vendor developers. The underlying functionality of the victim APIs should not be available to third party and unprivileged apps. In fact, similar functionalities (e.g., reading and overwriting private files, manipulating system-wide display settings) are all protected with high-privilege requirement in AOSP, hence cannot be accessed by third-party apps. The memory corruptions though, were caused by improper mediation of inputs (e.g., supplied integer reflects an out-of-bound index value).

Vulnerabilities in Native APIs. Column 3 in Table 5 lists the recovered names of the victim APIs. As shown, for APIs whose definition lies within the native layer, we report the corresponding binder transaction Id instead - since no symbols are present. As depicted, 17 flaws (46%) were caused by purely native APIs, clearly justifying the need for our binary analysis for recovering native functions interfaces.

Significance of Log-Guided Fuzzing in Discovering the Vulnerabilities. To showcase the significance of our log-guided input inference, we report in Column 6 the number of victim APIs that triggered at least one (useful) input validation message – which we leveraged to extract input specifications and accordingly drive the next fuzzing iteration towards discovering the vulnerable code path. As shown, 22 cases (59%) triggered at least one input validation message indicating various semantics about the supplied inputs, including string format types (e.g., file paths as in `systemmix.Transaction2`, `gpio.Transaction1`), typical keywords in `systemmix.Transaction16777215` and `Display_manager.enableInterface`, and valid input ranges in `Display_manager.setScreenScale`, etc.

As mentioned in Section 6, our fuzzer further leverages non-input related log messages - triggered by a target API, to derive whether a new state has been uncovered thanks to the current inputs and accordingly drive the next fuzzing iteration (e.g., fix most recent log-inferred inputs and mutate others). To showcase the prevalence and significance of non-input related log messages in *contributing* to the vulnerabilities discovery, we show in Column 7 the victim APIs which triggered at least one non-input related log message after the first fuzzing iteration. We observe that for all the 20 cases, non-input related messages *complemented* the role of input validation messages in discovering the vulnerabilities.

To further demonstrate the overall significance of our log-guided fuzzing in discovering the vulnerabilities, we report the time required to expose each vulnerability using a random (Column 10) and log-guided approach (Column 11). Specifically, we run our testing of the APIs (each API is tested for a maximum 24 hours) with randomly generated and log-inferred specifications (if any) and report the following: (A) Our log-guided fuzzing outperforms random fuzzing in the cases where the victim APIs logged an input specification: (1) for the cases whose triggers are difficult to generate (72% of the cyber threats, 59% cases with input validations, and 35% overall cases), random testing has timed-out without any success. In contrast, the vulnerabilities were quickly exposed using the log-generated specs. (2) In the rest cases - i.e., inputs can be generated using random approach, our approach detects the vulnerabilities faster than random fuzzer. (B) Our log-guided fuzzing performs comparably to random fuzzing if no log-guided specs can be inferred. This is intuitive as our approach falls back to random fuzzing in such cases.

Significance of Feedback Monitoring Channels in Detecting the Vulnerabilities. Our proposed testing leverages two channels - log feedback and external observer - to assess the outcome of an executed test case. Columns 8-9 report the significance of each channel in detecting the flaws. Observe that most of the cyber threats (CT) and all the memory corruptions (MC) can be detected by monitoring the logs. However, our external observer is more suitable for detecting the physical anomalies: 15/16 of the reported disturbances did not trigger any crashes at the log level and thus would go undetected without accounting for physical manifestations.

10 Case Studies

We discuss here 3 out of our discovered attacks. A description of another selected attack is in Appendix.

Cyber Attack I: Complete Device Breakdown. Our fuzzer uncovered that a custom API allows appending user-supplied inputs to a critical file (`"/dev/block/env"`), which contains important boot environment variables. If executed repeatedly, the API leads to corrupting this critical file, subsequently leading to a complete device breakdown as it cannot reboot due to the corrupt boot variables - even under safe-mode.

Our proposed approach has enabled us to uncover this vulnerability as follows: Initially, our fuzzer generated random inputs (`String ="ABC", String = "DEF"`) according to the recovered API's function interface. The subsequent executions led to several target log messages, which were fed to our trained classifiers. The *InputValidation* classifier flagged the message `"[ubootenv] ubootenv variable prefix is: ubootenv.var"` as an input validation. The sub-classifier *StringPrefix* (see Classifier 7 in Table 3 in Section 9.3) further predicted a fine grained category of the validation; namely string prefix validation. The fuzzer then extracted the specification "String Prefix value is `"ubootenv.var"`". Note that such extraction is guided by models learned from the training samples (Section 5.2.1). With the guidance, in the next iteration, our fuzzer speculates that `ubootenv` denotes the first string parameter and generates a new input (`String = "ubootenv.var", String = "DEF"`). The following execution then triggered new target messages: `[ubootenv] update_bootenv_varible name [ubootenv.var._deepcolor]: value [DEF]` and `[ubootenv] Save ubootenv to /dev/block/env succeed!`, indicating that new states were explored. Observe that without log-guidance, a random approach is unlikely to discover such new states. As the last identified messages did not indicate any input validations according to our classifiers (but rather just a new state), our fuzzer continues to mutate the variables while respecting the previously inferred specifications - e.g, by appending a random string `"ABC"` to the prefix `String = "ubootenv.var"`. The subsequent execution led to similar log messages (i.e., `update_bootenv_varible name [ubootenv.var.ABC_deepcolor]: value [DEF]`). After 0.11h, we detected a SIGSEGV fault and an overall system shutdown. Attempting to reboot the device afterwards was not successful due to the corrupt boot variables.

Cyber Attack II: Read sensitive system files. Our technique uncovers another flaw on one of the victim devices, enabling unprivileged callers to access highly-sensitive data stored anywhere on the device. Our log-guidance facilitated the discovery of this flaw as follows: Our fuzzer started by generating random inputs (`String ="ABC",`

Table 5: Details about Discovered Attacks

Flaw Type	Service	API	Description	Victim Devices (s)	Log-Guided			External Feedback	Exposing Time	
					Input Inference	New state Inference	Feedback Inference		Random	Guided
CT	system_control	Transaction Id 47	Corrupt boot environment variables	H96 Pro	✓	✓	✓	✓	Timed out	0.11h
CT	mount	createRemoteDisk	Overwrite System Directories	Nvidia Shield	✓	✓	✓	✓	Timed out	4.71h
CT	mount	destroyRemoteDisk	Delete Files in internal memory	Nvidia Shield	✓	✓	✓	✓	Timed out	2.14h
CT	window_manager	dispatchMouse	inject mouse coordinates	V88, Max	✗	✗	✗	✓	0.03h	0.04h
CT	window_manager	dispatchMouseByCF	inject mouse coordinates	V88, Max	✗	✗	✗	✓	0.03h	0.03h
CT	systemmix	Transaction Id 16777215	Change persistent system properties	Q+	✓	✓	✓	✗	Timed out	0.14h
CT	systemmix	Transaction Id 2	read highly-sensitive data	Q+	✓	✓	✓	✗	Timed out	0.14h
CT	gpio	Transaction Id 1	overwrite certain system files	Q+	✓	✓	✓	✗	Timed out	0.19h
CT	gpio	Transaction Id 16777215	read highly-sensitive data	Q+	✓	✓	✓	✗	Timed out	0.15h
CT	SubTitleService	load	create hidden files under /sdcard/	GT King	✓	✗	✓	✗	Time out	0.05h
CT	CecService	Transaction Id 1	reboot device into recovery mode	MIBOX4	✗	✗	✓	✓	0.03h	0.03h
MC	Imageplayer	Transaction Id 2	Memory Corruption	MIBOX3, X96, H96	✗	✗	✓	✓	0.15h	0.17h
MC	Imageplayer	Transaction Id 20	Memory Corruption	MIBOX3, X96, H96	✗	✗	✓	✓	0.11h	0.10h
MC	Imageplayer	Transaction Id 15	Memory Corruption	MIBOX3, X96, H96	✗	✗	✓	✓	0.45h	0.38h
MC	Imageplayer	Transaction Id 14	Memory Corruption	MIBOX3, X96, H96	✗	✗	✓	✓	0.47h	0.53h
MC	system_control	Transaction Id 17	Memory Corruption	H96	✓	✗	✓	✗	Timed out	0.07h
MC	Display_manager	getCurrentInterface	Memory Corruption	RK MAX	✓	✓	✓	✗	1.45h	0.11h
MC	Display_manager	enableInterface	Memory Corruption	RK MAX	✓	✓	✓	✗	Timed out	0.07h
MC	Display_manager	switchNextDisplayInterface	Memory Corruption	RK MAX	✓	✓	✓	✗	0.57h	0.23h
MC	systemmix	Transaction Id 16777215	Memory Corruption	Q+	✓	✗	✓	✗	Timed out	0.13h
MC	drm	setGamma	Memory Corruption	RK MAX	✓	✗	✓	✗	0.33h	0.11h
PD	Display_manager	switchNextDisplayInterface	Drop HDMI signal	V88, Max	✓	✓	✗	✓	0.05h	0.02h
PD	Display_manager	switchNextDisplayInterface	Corrupt display	Max	✓	✓	✗	✓	0.05h	0.03h
PD	Display_manager	getCurrentInterface	Corrupt display	Max	✓	✓	✗	✓	0.08h	0.02h
PD	Display_manager	setContrast	Blackout display	V88, Max	✗	✗	✗	✓	0.07h	0.1h
PD	Display_manager	setScreenScale	Rescale display	V88, Max	✓	✓	✗	✓	0.03h	0.02h
PD	Display_manager	enableInterface	Drop HDMI Signal	V88, Max	✓	✓	✗	✓	Timed out	0.02h
PD	Display_manager	setHue	Manipulate color aspects	V88, Max	✗	✓	✗	✓	0.38h	0.29h
PD	Display_manager	setSaturation	Manipulate color aspects	V88, Max	✗	✓	✗	✓	0.17h	0.19h
PD	Display_manager	setBrightness	Manipulate color aspects	V88, Max	✗	✓	✗	✓	0.18h	0.22h
PD	system_control	Transaction Id 13	Blackout Display	X96, H96	✓	✓	✗	✓	0.11h	0.03h
PD	system_control	Transaction Id 16	Rescale display	X96, H96, MIBOX3	✗	✗	✗	✓	0.54h	0.37h
PD	system_control	Transaction Id 16	Corrupt display	X96, H96, MIBOX3	✗	✗	✗	✓	0.46h	0.33h
PD	system_control	Transaction Id 15	Disable mouse pointer	X96, H96, MIBOX3	✗	✗	✗	✓	0.05h	0.03h
PD	system_control	Transaction Id 23	Mute Sound System	X96, H96, MIBOX3	✓	✓	✗	✓	Timed out	0.02h
PD	tvout	setPosition	Rescaling the display	X3	✗	✗	✗	✓	0.14h	0.15h
PD	tvout	setNewSdf	Stop streaming services	X3	✓	✗	✗	✓	0.06h	0.02h

`int = 3`) according to the responsible API's function interface. The execution triggered the following messages: `SystemMixService::putFileData() filepath=ABC count=3`, `SystemMixService: cannot open file ABC to read` and `read data is .` Our trained classifiers flagged the second message as an input validation and as a *StringIsFile* (by classifier 9 in Table 3). It accordingly extracted the following specifications for the first parameter: "a string parameter denotes a file". Since there is only one string parameter, in the following mutation, our fuzzer generated the value "/data/system/passwd" - referring to a valid file on the system. The fuzzer then uncovered a new log state `SystemMixService::putFileData() filepath=/data/system/passwd count=3` and `read data is <co`. Similar to the previous case study, our fuzzer continued with random mutations to uncover other potential new states - since no more specs were identified. Observe that for this case, we log the returned value of the target API and instrument our log analyzer to compare the value against that of the supplied file content (known before hand). Note that this intervention is done only for APIs with non-empty return values and carried out automatically.

Physical Disturbance: Dropping HDMI Signal for a Fake-Off Mode. Dropping the HDMI signal is a privileged operation initiated by the system when turning off the TVBox or when switching display interfaces. We found that this functionality is accessible to non-privileged apps in a few TVBoxes. Since a broken HDMI signal indicates a powered-off source, it can be maliciously used to trick the user into believing that the system is off (while it is still running). With the help of other SmartTV peripherals, such as the remote controller's built-in mic (as reported in WikiLeaks' *Weeping Angel* case [9]), an attacker can exploit this functionality to fake an off mode and spy on the SmartTV users.

11 Threats to Validity and Limitations

In this section, we discuss various factors that may affect the validity of achieved results, and significance of log-guided fuzzing, along with its limitations.

One validity threat lies in the selected SmartTV samples in our study: The low-cost and limited number of SmartTVs may not represent all respective Android based SmartTVs; specially those from high-end vendors (e.g., Sharp, Sony). To tackle this threat, we made sure to cover a reasonably diverse set of devices (including those from popular vendors such as Xiaomi and NVidia). Our dataset size is smaller than many used in static analysis, nonetheless, for a dynamic analysis, our set is aligned with the literature.

Our fuzzer is highly dependent on the availability of log messages, particularly those related to input validations and those signaling important feedback; which poses another threat to our results. The amount and usefulness of execution logs obtained during our testing may not well represent all logs of other SmartTVs' additions. Besides, our approach is reliant on the accuracy of target log messages identification.

We attempted to manage this threat through proposing a statistical method and classification models. Another threat to validity lies in the lack of precise measurement of code coverage, due to the black-box nature of SmartTV native additions. To mitigate the threat, we leverage the best resources available and show that our technique is effective in finding vulnerabilities and achieving good approximate coverage (measured by the number of covered logging statements in the Java portion).

12 Related Work

Grammar Inference. During fuzzing, it is essential to generate inputs with valid formats. For the programs whose input format (or grammar) is unknown, grammar inference can be a viable approach. Most of the existing grammar inference techniques are either whitebox or greybox. For instance, AUTOGRAM [27] and REINAM [36] conduct dynamic tainting and symbolic execution respectively to infer input grammars. Grimoire [23] infers the input grammar by instrumenting the program and observing the code coverage for each input mutation. REDQUEEN [20] does not directly infer grammars. By observing branch conditions during fuzzing, it can detect keywords and magic numbers in the inputs. Unfortunately, these techniques require code instrumentation and collecting fine-grained feedback such as code coverage or execution trace, which is not feasible in the SmartTV fuzzing scenario. Some grammar inference techniques are blackbox. For instance, GLADE [22] can automatically synthesize program input grammars from a set of program inputs, which are either accepted or rejected by the target program. While this approach is more practical than the whitebox or greybox approaches, it still requires a set of high-quality inputs (especially the accepted inputs). Unfortunately, for SmartTV fuzzing, we do not have any inputs to begin with. This is why we have to perform execution log analysis to infer potential specifications and generate possibly good inputs, and rely on log-guided fuzzing to increase code coverage.

Coverage-Guided Fuzzing. Evolutionary fuzzing that is guided by a variety of coverage information (e.g., edge coverage, calling contexts, and memory access patterns) has approved to be very effective in finding vulnerabilities in realworld programs. For instance, AFL [3] relies on code coverage information to select and mutate seeds. AFL-sensitive [33] further studies the impact of different coverage metrics. Some state-of-the-art fuzzers [37, 38] leverage the code coverage to infer the input field format to improve fuzzing performance. However, in SmartTV fuzzing, we cannot obtain these kinds of coverage information. Since log messages are available, in this work, we develop a novel fuzzing technique that is guided by the coverage of log messages.

Observing Anomalies through Fuzzing. Most state-of-the-art fuzzers [3, 33] can detect potential vulnerabilities through checking whether the target crashes. A few approaches though (e.g., address sanitizer [2], thread sanitizer [8]) propose enhancements to the fuzzer's ability to capture anomalous behaviors. As these fuzzers observe anomalies in-box, they cannot detect physical anomalies, triggered by SmartTV APIs.

SmartTV Security. Oren et al. [29] describe attacks on SmartTVs causing a large-scale compromise on the Internet, due to flaws in combing broadband and broadcast systems in HbbTV. Closely related to our research is [28], which evaluates the SmartTV apps. The evaluation reveals erroneous practices in protecting critical data. In contrast, our work focuses on flaws in the framework.

IoT Security. The IoT market has attracted the attention of researchers. Zhang et al. [39] summarized security problems of IoT devices, (e.g, LAN mistrust, implementation flaws). To analyze IoT firmware, a few studies apply static analysis at source code [25] or at binary level [30]. Other studies [34, 41] conduct black-box testing. Our work features log-guided fuzzing. IOTFuzzer [24] employs fuzzing to discover memory corruptions in IoT devices. In contrast, our work is more general, detecting cyber and physical flaws.

13 Conclusion

To assess the security implications of SmartTV customization, we develop a novel log-guided dynamic fuzzing technique. Our approach provides a viable solution when instrumentation and collecting fine-grained execution feedback is not feasible. To detect SmartTV-specific anomalies (i.e., visual and auditory disturbances), we further propose a novel external observer which can detect potential physical anomalies triggered during fuzzing – which may not be detected in-box. Our technique proved to be effective through discovering 37 vulnerabilities in 11 Android TVBoxes.

Acknowledgments

This research was supported, in part by NSERC under grants RGPIN-07017, DGECR-00319, by NSF 1901242 and 1910300, and by ONR N000141712045, N000141410468, N000141712947, and N00014-17-1-2893. The RUC author was supported in part by NSFC under grant 62002361 and U1836209, and the Fundamental Research Funds for the Central Universities and the Research Funds of Renmin University of China under grant 20XNLG03. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

References

[1] 25 Best Android TV Boxes For 2020. https://androidpcreview.com/best-android-tv-box/.

[2] Address sanitizer. https://github.com/google/sanitizers/wiki/AddressSanitizer.

[3] Afl. http://lcamtuf.coredump.cx/afl/.

[4] Demos. https://sites.google.com/site/smarttvdemos/.

[5] Keras: The deep learning library. https://keras.io.

[6] Radare2. https://www.radare.org/r/.

[7] Randoop. https://randoop.github.io/randoop/.

[8] Thread sanitizer. https://github.com/google/sanitizers/wiki/ThreadSanitizerCppManual.

[9] Vault 7: Cia hacking tools revealed. https://wikileaks.org/ciav7p1/cms/page_12353643.html.

[10] Wala. https://github.com/wala/WALA.

[11] What is the Best Way to Stare at Screens All Day? http://time.com/4789208/screens-computer-eye-strain.

[12] Japanese cartoon triggers seizures in hundreds of children. http://www.cnn.com/WORLD/9712/17/video.seizures.update, 1997.

[13] The secret life of c++: Runtime type information and casting. http://web.mit.edu/tibbetts/Public/inside-c/www/rtti.html, 2015.

[14] Ring's smart doorbell can leave your house vulnerable to hacks. https://www.cnet.com/news/rings-smart-doorbell-can-leave-your-house-vulnerable-to-hacks, 2016.

[15] Ransomware on smart tvs is here and removing it can be a pain. https://www.pcworld.com/article/3154226/security/ransomware-on-smart-tvs-is-here-and-removing-it-can-be-a-pain.html, 2017.

[16] Best Android TV Box 2019. https://www.144hzmonitors.com/best-android-tv-box, 2019.

[17] Smart TV Market: Global Industry Trends. https://www.researchandmarkets.com/research/zrxx5w/250_billion?w=4, 2019.

[18] The Connected Future. https://www.ericsson.com/en/mobility-report/internet-of-things-forecast, 2019.

[19] The 10 Best Android TV Boxes. https://wiki.ezvid.com/best-android-tv-boxes, 2020.

[20] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. Redqueen: Fuzzing with input-to-state correspondence. In *NDSS*, 2019.

[21] Yann Bachy, Vincent Nicomette, Mohamed Kaâniche, and Eric Alata. Smart-tv security: risk analysis and experiments on smart-tv communication channels. *Journal of Computer Virology and Hacking Techniques*, 15:61–76, 2018.

[22] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. Synthesizing program input grammars. In *PLDI*, 2017.

[23] Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. GRIMOIRE: Synthesizing structure while fuzzing. In *USENIX Security*, 2019.

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

[25] Drew Davidson, Benjamin Moench, Thomas Ristenpart, and Somesh Jha. on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In *USENIX Security*, 2013.

[26] Miro Enev, Sidhant Gupta, Tadayoshi Kohno, and Shwetak N. Patel. Televisions, video privacy, and powerline electromagnetic interference. In *CCS*, 2011.

[27] Matthias Höschele and Andreas Zeller. Mining input grammars from dynamic taints. In *ASE*, 2016.

[28] Marcus Niemietz, Juraj Somorovsky, Christian Mainka, and Jörg Schwenk. Not so smart: On smart tv apps. In *SIoT*, 2015.

[29] Yossef Oren and Angelos D. Keromytis. From the aether to the ethernet—attacking the internet using broadcast digital television. In *USENIX Security*, 2014.

[30] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware. In *NDSS*, 2015.

[31] Raphael Spreitzer, Felix Kirchengast, Daniel Gruss, and Stefan Mangard. Procharvester: Fully automated analysis of procfs side-channel leaks on android. In *ASIACCS*, 2018.

[32] Takeshi Sugawara, Benjamin Cyr, Sara Rampazzi, Daniel Genkin, and Kevin Fu. Light commands: Laser-based audio injection on voice-controllable systems. In *USENIX Security*, 2019.

[33] Jinghan Wang, Yue Duan, Wei Song, Heng Yin, and Chengyu Song. Be sensitive and collaborative: Analyzing impact of coverage metrics in greybox fuzzing. In *RAID*, 2019.

[34] Zhiqiang Wang, Yuqing Zhang, and Qixu Liu. Rpfuzzer: A framework for discovering router protocols vulnerabilities based on fuzzing. *TIIS*, 7(8):1989–2009, 2013.

[35] Arnold Wilkins, Jennifer Veitch, and Brad Lehman. Led lighting flicker and potential health concerns: Ieee standard par1789 update. In *ECCE*, 2010.

[36] Zhengkai Wu, Evan Johnson, Wei Yang, Osbert Bastani, Dawn Song, Jian Peng, and Tao Xie. Reinam: Rein-

forcement learning for input-grammar inference. In *ESEC/FSE*, 2019.

[37] Wei You, Xuwei Liu, Shiqing Ma, David Mitchel Perry, Xiangyu Zhang, and Bin Liang. SLF: fuzzing without valid seed inputs. In *ICSE*, 2019.

[38] Wei You, Xueqiang Wang, Shiqing Ma, Jianjun Huang, Xiangyu Zhang, XiaoFeng Wang, and Bin Liang. Profuzzer: On-the-fly input type probing for better zero-day vulnerability discovery. In *S&P*, 2019.

[39] Nan Zhang, Soteris Demetriou, Xianghang Mi, Wenrui Diao, Kan Yuan, Peiyuan Zong, Feng Qian, XiaoFeng Wang, Kai Chen, Yuan Tian, Carl A. Gunter, Kehuan Zhang, Patrick Tague, and Yue-Hsun Lin. Understanding iot security through the data crystal ball: Where we are now and where we are going to be. *CoRR*, abs/1703.09809, 2017.

[40] Nan Zhang, Kan Yuan, Muhammad Naveed, Xiao-yong Zhou, and XiaoFeng Wang. Leave me alone: App-level protection against runtime information gathering on android. In *S&P*, 2015.

[41] Jixuan Zhou, Dan Feng, and Bo Li. A fuzzing method based on dual variation strategy for cisco ios. In *ICCC*, 2017.

APPENDIX

A Optimization Strategies

Log Normalization. We normalize messages sharing a certain template to facilitate removing duplicate entries. As log messages are generated by a print statement, similar messages with slight variations appear quite frequently. We tolerate the slight differences through normalizing numerical strings and other string formats reflecting common entities (i.e., package names, URLs, filenames, file paths, etc) to predefined values.
Efficient Similarity Measure To measure the similarity of two messages, we abstract the messages into N-gram sequences (N=2 here) and calculate the number of common N-grams. If this latter exceeds a threshold, we consider the two to be similar. Specifically, given two messages, we calculate the DICE Coefficient, a widely used lexicography for measuring lexical associations. DICE is defined as the ratio of the number of bigrams that are shared by the two messages and the total number of bigrams in both events:

$$DICE(X,Y) = \frac{2 * |bigrams(X)| \cap |bigrams(Y)|}{|bigrams(X)| + |Bigrams(Y)|}$$

where X and Y are two messages. A DICE result of 1 indicates identical messages and a 0 equals orthogonal ones. Note that we consider two messages as similar if DICE > 0.8

Table 6: Time Consumption in Different Testing Phases.

Device	Time Consumption of 10,000 Tests (seconds)			
	Generate & Execute	Analyze Log	Compare Image	Compare Audio
Min	99	87,454	54,506	49,843
Max	322	164,089	70,973	83,355
Average	157	144,137	77,903	78,647
Avg per 1 test case	**0.0233**	**15.78**	**6.9375**	**7.3409**

B Testing Efficiency

We evaluate here achieved testing efficiency. We measure the time incurred to generate and execute a test case, to analyze the log output, and to compare the content before and after executing each case. To give accurate estimates of the time incurred, we ran the fuzzer over 10,000 test cases. As shown in Table 6, the analysis is quite fast. Each test case takes on average ∼16 seconds (The image and audio comparisons are run concurrently on different cores). The log analysis incurs the largest time overhead; however, it is still acceptable.

C Physical Disturbances

Manipulating Color Aspects. We detected a non-protected API that allows manipulating color aspects (privileged on AOSP). We implemented an app that leverages this to perform the following: First, we manipulate the hue to shift the system wide color scheme towards the blue spectrum. This shifting can irritate the blue-light sensitive molecules in the retina, negatively affecting the body's circadian rhythms and the ability to sleep [11]. Second, we control the brightness to achieve other consequences. Specifically, we build an app by learning from the notorious *Pokemon Shock* incident [12] where more than 600 children suffered from convulsions, seizures and vomiting after watching a Pokemon episode featuring a 5 sec flashing red light. According to epilepsy experts [12], *television epilepsies* can be triggered when the viewer is immersed on a scene displaying flashing and colorful lights. Our POC exploits television epilepsy to perform the following: The malicious app sends phishing emails to SmartTV users, containing a link to a popular video with a lot of actions. The link is largely genuine except that it is enhanced: once the link is clicked, the app gets informed. In addition, the time periods of action scenes (e.g., fighting) are pre-determined. Therefore, when the app notices that the user starts playing the video, it starts a timer simultaneously. When those scenes are reached, it substantially flicks the display's brightness, piggybacking the rapidly changing movie contents.