Security Properties of Virtual Remotes and SPOOKing their violations

Joshua Majors
Purdue University
West Lafayette, IN, USA
jmajors@purdue.edu

Edgardo Barsallo Yi
Purdue University
West Lafayette, IN, USA
ebarsall@purdue.edu

Amiya Maji
Purdue University
West Lafayette, IN, USA
amaji@purdue.edu

Darren Wu
Purdue University
West Lafayette, IN, USA
wu1797@purdue.edu

Saurabh Bagchi
Purdue University
West Lafayette, IN, USA
sbagchi@purdue.edu

Aravind Machiry
Purdue University
West Lafayette, IN, USA
amachiry@purdue.edu

ABSTRACT

As Smart TV devices become more prevalent in our lives, it becomes increasingly important to evaluate the security of these devices. In addition to a smart and connected ecosystem through apps, Smart TV devices expose a WiFi remote protocol, that provides a virtual remote capability and allows a WiFi enabled device (e.g., a Smartphone) to control the Smart TV. The WiFi remote protocol might pose certain security risks that are not present in traditional TVs. In this paper, we assess the security of WiFi remote protocols by first identifying the desired security properties so that we achieve the same level of security as in traditional TVs. Our analysis of four popular Smart TV platforms, Android TV, Amazon FireOS, Roku OS, and WebOS (for LG TVs), revealed that *all these platforms violate one or more of the identified security properties*. To demonstrate the impact of these flaws, we develop Spook, which uses one of the commonly violated properties of a secure WiFi remote protocol to pair an Android mobile as a software remote to an Android TV. Subsequently, we hijack the Android TV device through the device debugger, enabling complete remote control of the device. All our findings have been communicated to the corresponding vendors. Google *acknowledged our findings* as a security vulnerability, assigned it a CVE, and released patches to the Android TV OS to partially mitigate the attack. We argue that these patches provide a stopgap solution without ensuring that WiFi remote protocol has all the desired security properties. We design and implement a WiFi remote protocol in the Android ecosystem using ARM TrustZone. Our evaluation shows that the proposed defense satisfies all the security properties and ensures that we have the flexibility of virtual remote without compromising security.

CCS CONCEPTS

• **Security and privacy** → *Security protocols*; **Malware and its mitigation**; **Mobile platform security**.

KEYWORDS

Android TV, privileges escalation, attack, virtual remote

ACM Reference Format:
Joshua Majors, Edgardo Barsallo Yi, Amiya Maji, Darren Wu, Saurabh Bagchi, and Aravind Machiry. 2023. Security Properties of Virtual Remotes and SPOOKing their violations. In *ACM ASIA Conference on Computer and Communications Security (ASIA CCS '23), July 10–14, 2023, Melbourne, VIC, Australia.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3579856.3582834

1 INTRODUCTION

There has been recent explosive growth of Over The Top (OTT) digital streaming platforms [56], reaching globally $101B in 2020 and expected to grow at a CAGR of 14% over the next 5 years [36]. In the US itself, in 2020 a household on average has 1.64 OTT subscriptions. The OTT platforms are dominated by the manufacturers Roku, Apple, Amazon, and Google and account for over 1.1B devices worldwide [27]. With the recent growth of these OTT platforms, purchases through streaming devices, especially Smart TVs, has also been growing rapidly at a CAGR of 19.8% and is expected to reach $340.8B in 2027. Specifically, Android TV, developed by Google, has grown incredibly fast, doubling the number of devices each year since 2016 [1]. Currently, there are over 80M Android TV (and its newer variant Google TV) devices [48] and more than 5K apps published in the Google Play Store for this platform [25, 27].

Smart TV devices store passwords to many different services a user may have, such as Google, Amazon Prime, and Netflix. In fact, in order to use the full suite of features in Android TV devices, the user is required to sign into a Google account. Smart TV devices are typically continually connected to the Internet. Further, they have few UI screens that will allow a user to query their status, like currently running processes. This makes them a ripe target for attacks meant for credential stealing or for using them as agents in a botnet [12]. In fact, in August 2019 WootCloud disclosed a vulnerability in Android TV devices that allowed them to be employed in a botnet [57]. Moghaddam *et al.* [41] showed the prevalence on OTT devices of pervasive tracking and collecting device identifiers, while [42] shows how to take over a smart TV running Android TV OS by abusing the infrared communication protocol when an attacker has direct access to the device. There have been a series of warnings and published exploits about smart TVs and set-top

ASIA CCS '23, July 10–14, 2023, Melbourne, VIC, Australia

ACM ISBN 979-8-4007-0098-9/23/07.
https://doi.org/10.1145/3579856.3582834

boxes [13, 15, 31, 54], including some from government sources like the FBI [54].

Despite this sense that OTTs provide an attractive attack surface, there is remarkably little in the research literature evaluating the security architectures and the resulting foundational vulnerabilities of these streaming platforms. Existing works, such as the recent work by Aafer *et al.* [2] assess Smart TV from a general software security perspective, i.e., traditional vulnerabilities in Smart TV devices.

This paper focuses on software remote capability in smart TV devices implemented through the WiFi remote protocol. The software remote protocol enables any WiFi enabled device to be used as a remote control to the Smart TV. For instance, TV vendors provide smartphone apps allowing the phone to be used as a virtual remote. However, in addition to increasing flexibility, the virtual remote also introduces a new attack vector that is missing in traditional TV remotes. Analyzing the root cause of the attack allows us to develop a foundational defense against it.

To systematically explore potential threats in virtual remote, we first identify the necessary security properties i.e., *Remote Control Device Security Properties (RCDSPs)*, that should hold so that we have the same level of security guarantees as in traditional TVs. We explore the potential threats that can arise because of the violation of the RCDSPs.

Second, we analyze four popular Smart TV platforms and show that none of these satisfy all RCDSPs. To demonstrate the impact of missing RCDSPs, we develop ***Spook***, an app that exploits Android TV's WiFi remote protocol, the underlying mechanism for virtual remotes. *Spook is completely automated and takes control of any Android TV it can communicate with.* We demonstrate this attack on real Android and Android TV OS devices. We have put videos of Spook at work on real devices on an anonymized page [51].

Responsible Disclosure: All our findings have been reported to the corresponding vendors. Google acknowledged our findings and marked it as a *severe* security vulnerability [29]. This vulnerability has been assigned CVE-2021-0889. In response, Google has started to roll out changes to the Android TV OS that address *some* issues with their WiFi remote protocol pairing policy[1]. We are waiting for responses from other vendors.

Finally, using ARM TrustZone, we design and implement modifications to WiFi remote protocol that prevents Spook and satisfies all our RCDSPss. We believe that our defense serves as a feasible proof-of-concept and encourages vendors to modify their WiFi remote protocol to prevent security issues because of the virtual remote.

In summary, this paper makes the following contributions:

1. We present a security analysis of the virtual remote capability of Smart TV platforms and identify the necessary security properties (RCDSPs) that guarantee the same level of security as traditional TVs. Our analysis of four major platforms shows that they all have security flaws due to not complying with one or more of the security properties.
2. We demonstrate the impact of missing any of the security properties by implementing an attack Spook[2], which exploits the Android TV's WiFi remote protocol and is shown to work on current Android devices.
3. The evaluation of Spook demonstrates that our attack is effective, practical, and remains stealthy (e.g., the attack runs in the background even with the TV monitor switched off).
4. We develop a defense technique, leveraging the ARM TrustZone on the mobile to prevent Spook and comply with all the RCDSPs. Further, we suggest architectural changes that vendors can use to protect Smart TVs from attacks targeting the virtual remote capability.

The rest of the paper is organized as follows. First, we present the properties that guarantee security in traditional TVs and relevant aspects of the *virtual remote* capability provided by Smart TV OSes in Section 2. Next, we delve into the details of an attack that violates these security properties in Section 3 We then subsequently discuss our proposed defense in Section 4. Then, in Section 5, we present the detailed experiments and results, and we discuss further design considerations in Section 6. Finally, we discuss related work and conclude the paper. We provide details of Trusted Execution Environments and the reverse engineering process of the WiFi remote protocol in the appendix.

2 BACKGROUND

2.1 Virtual Remote Guarantees

Traditionally, users control TVs mostly through a remote, which we call the Remote Control Device (RCD), which uses short-range wireless protocols to communicate with the TV. TVs come with a receiver and the RCD has a transmitter, thus facilitating one-way communication from the RCD to the TV. We take inspiration from traditional TV RCDs to design a security policy for modern virtual remotes. We identify four properties modern virtual remotes must satisfy in order to ensure their security.

- **Only users with a valid RCD can control the TV.** Analogous to capability-based permission systems, the RCD provides the capability to control the TV. All users with a valid or compatible RCD can control the TV. In principle, getting a valid RCD for a TV requires knowing the make and model of the TV [3]. Which requires physical (or visual) access to the TV as the make and model are usually written on the outer frame of the TV [53]. To summarize, getting a valid RCD requires visual access (i.e., the ability to see the TV). We call this the *Visual Access Property* (P_{va}).
- **Users are required to be within a certain distance to control the TV.** Most RCDs use short-range wireless protocols such as Infrared and Bluetooth [18]. This limits the maximum distance from which one can use an RCD. Hence, to control a TV, the user should be within a certain distance (~ 100 m) from the TV. We call this the *Distance Bounding Property* (P_{db}).
- **Human-triggered inputs.** Only a human can perform key presses on an RCD. In other words, a human is responsible for initiating all key events from an RCD. We call this the *Human Attestation Property* (P_{ha}). Although it is possible for someone to

[1] Specifically, the release increases the code length and enforces lockout after a specific number of unsuccessful attempts. But it does not provide the more foundational defense, how to verify that UI events are generated by a human user. We design and implement such a defense (§ 4).

[2] Source code is available at [50].

[3] We acknowledge the existence of universal remotes. But, in the general case, one still needs to know the make and model to get a valid remote.

design machinery to automatically press buttons on a RCD, any attack using this approach would have strong assumptions and little to no real world impact. For this reason, we ignore specialized hardware circuity that can initiate key events automatically.

- **Only an authenticated remote can control the TV.** While we take inspiration from traditional TVs for the desired security properties of a RCD, Smart TVs are granted access to users' sensitive information (e.g., credentials) and therefore require a stronger security policy. We propose that RCDs for Smart TVs should be explicitly authenticated to restrict access. We call this the *Authentication Property* (P_{au}). This is different from some traditional RCDs, especially IR protocols. IR protocols used by traditional TVs do *not* authenticate the source from which the signal originated. If a policy satisfies P_{au}, it must have a way of identifying which source a command came from and whether it has been authorized.

The properties P_{va}, P_{db}, P_{ha}, and P_{au} are the *necessary conditions* that should hold to control a TV from an RCD. We call these *Remote Control Device Security Properties (RCDSPs).*

Flexibility without increasing attack surface: As explained in Section 1, Smart TVs store passwords to many different services and unauthorized access could lead to loss of private user information, which could, in turn, cause financial loss. Hence, we should strive to *not increase the attack surface for Smart TVs* relative to traditional TVs. In traditional TVs, any attack through an RCD should satisfy RCDSPs. To ensure that we do not increase the attack surface through a virtual remote, it should also have RCDSPs as its necessary conditions along with additional flexibility that any communication-enabled device (e.g., Smartphone) can be used as a remote control.

2.2 Virtual Remotes in Smart TVs

Smart TVs run OSes that enable users to stream multimedia content and interact with their TVs. These OSes can be either built-in to the TV monitor or installed on a dongle that can be affixed to the TV to provide "smart" capability. These OSes provide a rich ecosystem in streaming content apps, such as Netflix, Hulu, or Amazon Prime Video. These systems offer the user multiple ways to interact with the TV, such as remote controllers (either physical or software remotes) or other technologies to cast content on the TV, like Google's Chromecast. Currently, the market is being dominated by Smart TVs that run Samsung Tizen OS, LG WebOS, Google's Android TV, Roku OS, and Amazon Fire OS, with a combined worldwide share of 60% [9, 52].

All these OSes provide *virtual remote* capability through WiFi remote protocols. This protocol is handled by a *remote service* that runs in the Smart TV OS . The remote service comes preinstalled with the OS on the Smart TV. It runs as a background service and has access to higher privileged functions such as injecting UI events and starting voice commands. The virtual remote is simply an unprivileged app from the Google Play Store that runs on the mobile that only needs network permissions. In theory, any network enabled device could behave as a virtual remote.

In most cases, Smart TVs are connected to the home network and sit behind a Network Address Translation (NAT) framework [55], and hence are not open to arbitrary connections from anywhere on the Internet. So, to use a device as a virtual remote, the device should be either in the same local area network or within the communicable range of supported short-range protocols such as Bluetooth.

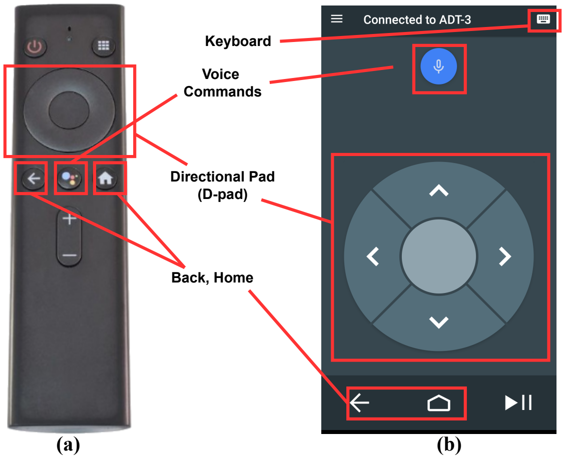


Figure 1: Android TV Remote Control. The figure shows a Hardware Remote (a) and a Software Remote (b). The hardware remote works by communicating over Bluetooth or IR, while the software remote uses Bluetooth or a network connection over a local WiFi network.

As per our analysis, the policies of WiFi remote protocols are the same across all Smart TV OSes. For a device to act as a virtual remote, it needs to be *paired* (except in Roku, which we discuss in Section 2.2.1). The pairing is a one-time task per device and, in principle, establishes a shared secret between the TV and remote device. The remote device should use this shared secret to send key events to the TV. The remote service on the TV verifies that the communicating device is indeed paired through the established secret before accepting key events.

2.2.1 Pairing. Figure 2 shows the overview of pairing process in WiFi remote protocol. The remote device initiates the pairing process by sending a particular message to the target Smart TV. If the remote device is not previously paired, the TV will display a one-time token, usually an alphanumeric code. The remote device is expected to send the displayed token back to the TV, which verifies the token, and if it matches, the corresponding device is paired. Other steps occur after pairing, such as setting up a shared secret and other identifiers to ensure persistence. We ignore these steps as they are not needed for our discussion.

2.2.2 Interaction. Once paired, the device acts as a virtual remote and can send key events to the TV. There are several apps on the Google Play Store that use WiFi remote protocol to allow users to connect their smartphone to their Android TV streaming devices as a virtual remote [5, 14, 30, 35, 47]. We call these Software Remote apps. These are quite popular, with the official apps for WebOS, Android TV, Fire OS, and RokuOS all having over 10M installs from the Google Play Store [5, 30, 39, 47]. One such app is shown in Figure 1.

As we can see the Software Remote enables any key event to be sent to the TV enabling the Smartphone to be used as a virtual remote device.

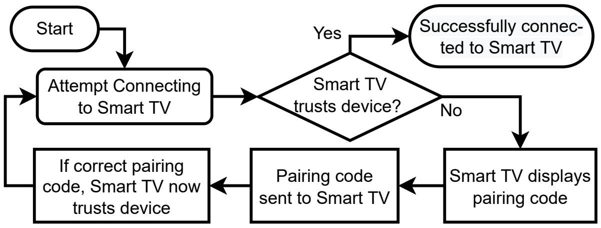

Start
Attempt Connecting to Smart TV
Smart TV trusts device?
Yes
No
Successfully connected to Smart TV
Smart TV displays pairing code
Pairing code sent to Smart TV
If correct pairing code, Smart TV now trusts device

Figure 2: Pairing process using the WiFi remote protocol. This is shown from the point of view of the mobile device as the source; the device at the other end is the Smart TV.

2.2.3 Verifying RCDSPs. As explained in 2.1, we want the virtual remote device to satisfy all four RCDSPs. Table 1 shows the summary of our analysis on the top 5 popular Smart TV OSes.
Visual Access Property (P_{va}): As explained in Section 2.2.1, all virtual remotes should be first paired (i.e. the token on the TV should be sent back to it). The P_{va} holds if the only way to get the token is by seeing (i.e., visual access) the TV screen. However, there could be other ways to get the token, specifically:

- **Guessing (G)**: The token could be guessed by the virtual remote if the OS uses cryptographically weak techniques (e.g., fixed string) to generate the token.
- **Bruteforcing (B)**: If invalid (i.e., with wrong pin) pairing attempts are silently discarded by the TV, then the virtual remote could brute force the token space until successfully paired.

Hence, to ensure that P_{va} holds, we need to make sure that the pairing process is not susceptible to at least the above two attacks. Unfortunately, our analysis of the popular TV platforms revealed that all but one prevent brute force attacks. Roku OS *does not even have any requirement for a pairing code*, and consequently, any device that can communicate with Roku TV can control it. Specifically, one can initiate the pairing process and send random pairing codes until the TV accepts the code. Furthermore, these pairing attempts can be made even when the TV screen is turned off (i.e., blank screen), making the attack stealthy.

We show in Section 3, how this small vulnerability can be used to have complete and persistent control of TVs by developing an end-to-end exploit. In response to our report, Google rolled out changes for Android TV in late September 2021, i.e., Android TV (New). The patch addresses the brute-forcing issue by introducing a damping and a soft ban mechanism. Our analysis is indicated by the third column of the Table 1.
Distance Bounding Property (P_{db}): Since most TVs are behind NAT and are not internet-facing, the virtual remote should be either in the same local area network or within the communicable range of supported short-range protocols such as Bluetooth. Hence, *virtual remotes satisfy P_{db}* as indicated by the fourth column of Table 1.
Human Attestation Property (P_{ha}): Once paired, as shown in Figure 1, the Software Remote (i.e., virtual remote device) can send any key events to the TV. Although the user interacts with Software Remote to make it send the keys events, the user interaction is not required. In other words, Software Remote (an app) can easily send events to the TV without human interaction. Furthermore, the *remote service* on the TV does not verify that a human indeed generates the key events through the corresponding virtual remote. Consequently, *the current TV OSes virtual remote does not satisfy our P_{ha} condition* as shown in the last column of Table 1.
Authentication Property (P_{au}): As we explain in Section 2.2.1, Smart TVs (except for Roku) use a pairing process that enables a device to be used as a virtual remote. The pairing process also establishes a shared secret and derives a cryptographic key, which is used to encrypt all the communications between the virtual remote device and the corresponding Smart TV. This key also serves to authenticate the corresponding device. Hence, the pairing process ensures P_{au}. Our analysis shows that most Smart TVs use a pairing process and thus satisfy P_{au}. However, Roku devices blindly accept any virtual remote that attempts to pair with them (without any authentication) and thus, do not satisfy P_{au}. The second column in Table 1 shows the summary of our analysis.

3 EXPLOITING VIRTUAL REMOTE

In this section, we aim to demonstrate the impact of violating RCDSPs by developing an end-to-end attack that exploits the violation of P_{va}, a RCDSP, to gain complete control of Smart TVs. As shown in Table 1, P_{va} can be violated in most of the analyzed Smart TV OSes. While we only implement Spook on Android TV, we do perform experiments in other Smart TV OSes to show the violation of RCDSPs, implying Spook (or a modified version of Spook) is possible on other platforms.

3.1 Threat model

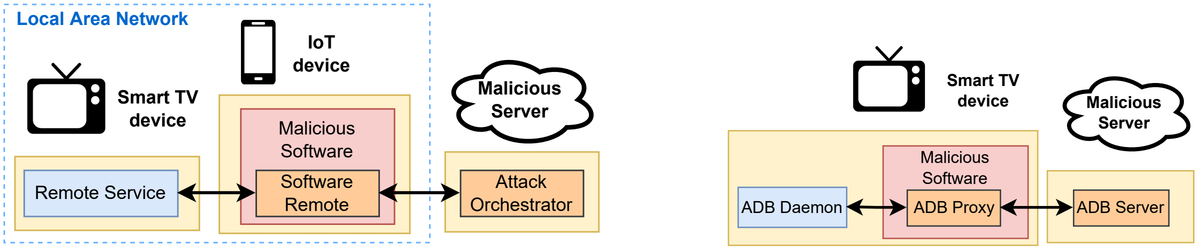
We assume that the attacker can communicate with the Smart TV directly over a network connection. Specifically, she can initiate a TCP connection directly to the remote service running on the target Smart TV. We also assume that the Smart TV and attacker are sitting inside the same local area network behind a NAT framework. Consequently, no connections can be *initiated* from a device/server outside the local network to a device inside the local network (although the opposite is possible). We do *not* assume that other devices on the local area network are trusted by the remote service on the Smart TV. All virtual remotes that wish to pair to the smart TV must be authenticated. We also do *not* assume that the TV screen is ON.

The objective of the attacker is to gain *complete control* of the target Smart TV without turning ON the TV display. Specifically, the attacker can install/uninstall apps or perform unlimited screen recording, unbeknownst to the user. These can be further used to compromise users' privacy by stealing valuable credentials or employing the user's device as an agent in a botnet.

3.2 Attack Roadmap

As explained in Section 2.2, sending any key events to Smart TV requires pairing. We also need a way to execute arbitrary commands on the TV without customizing for every TV's screen layout. We break our attack into three phases: first Pairing, then Escalation, and finally Attack. Details of the three phases can be seen in Figure 4.

3.2.1 Pairing. Our goal here is to pair with target Smart TV successfully, which requires the attacker to send the pairing pin (displayed on Smart TV) to the remote service (Section 2.2.1). We use the guessing strategy to get the correct pairing pin, as we do not assume visual access to the TV. We randomly generate a pairing



(a) Persistence using a Virtual Remote

(b) Persistence without Virtual Remote

Figure 3: Methods used by Spook to ensure persistence. In either scenario, the command and control can come from a malicious server located anywhere, not necessarily in the same network as the target Smart TV. In the scenario with the Virtual Remote, Spook achieves persistence leveraging the ADB Proxy packaged in the malicious app. In contrast, in the scenario without the Virtual Remote, the ADB Proxy is silently installed in the Smart TV and used by Spook to gain persistence.

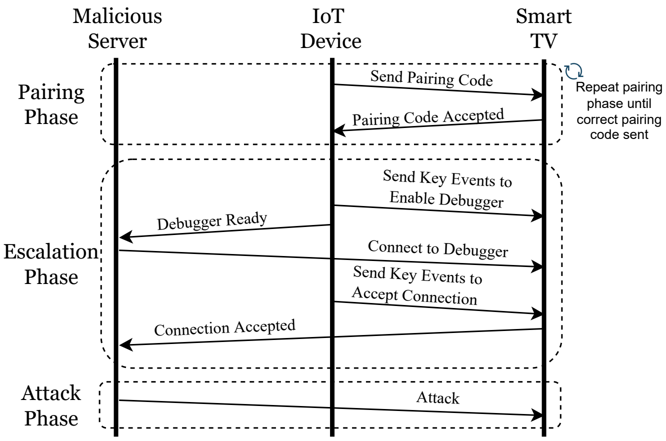


Figure 4: Diagram showing the attack phases. There are 3 devices involved in the attack: the Smart TV under attack, a malicious IoT device (a smartphone in the case of Spook) inside the same local network as the Smart TV, and a malicious server located anywhere in the world.

pin (r_p) and try it until the pairing succeeds. Specifically, we start the pairing process by sending a pairing request to the Smart TV. Next, we send r_p as the pairing pin. If the pairing fails, then we restart the pairing process until pairing succeeds with r_p, i.e., the displayed code on the Smart TV is r_p.

Consider the pairing pin four hexadecimal digits (most Smart TV OSes), the total number of unique pairing pins, N, is 16^4. Because a new code is generated with each trial (with the possibility of reuse), each trial is independent and has a probability of $p = 1/N$ to succeed. The probability of guessing the secret code by the n-th trial is given by Eq. 1.

$$P(S \leq n) = \sum_{k=1}^{n} P(S = k) = \sum_{k=1}^{n} (1 - p)^{k-1} \cdot p \qquad (1)$$

The probability of success increases with the number of trials. After 100K trials, the probability of success is greater than 75%. However, as we show in Section 5.3, in reality, the number of trails required for successful pairing is quite less. Furthermore, *the pairing process can be done stealthily, even if the display monitor is turned off, i.e., even after the user presses the power off button on the remote.* Pairing attempts, in fact, keep the Android TV from falling asleep. On other Smart TV platforms such as FireTV OS, a pairing attempt is even able to wake the device from a sleep state. Additionally, there is a plethora of evidence indicating that users do *not* power off (i.e., Remove the power supply) their Smart TVs or disconnect them from the network when not in use [34, 45, 46].

3.2.2 *Escalation.*

After pairing, we can send any key events to the Smart TV (Figure 1). Although sending arbitrary key events gives complete control of the device, it could be quite tedious to precisely perform any action on the Smart TV by just using key events. For instance, to install an app, we need to send directional (up/down/left/right) key events to select the install widget. Next, we need to select the search button (again through directional key events) and enter the app name (through keyboard events). Finally, we need to select directional key events to select the target app and then send key events to confirm. This key event-based mechanism needs to be specialized for layouts and dimensions of the Smart TV, making it tedious to have arbitrary control, rendering the attack near useless.

In order to make the attack practical, we need to find a layout-agnostic method to escalate privileges on the Smart TV. We found the steps to activate and connect to the device debugger are standardized across Smart TV devices and do not change frequently, even with different OS versions. While the steps to activate the device debugger are purposely convoluted to avoid accidental activation by everyday or even curious users, the steps taken to activate the debugger seldom change across major OS updates. Consequently, it is possible to script key events to send to the Smart TV that enable the debugger. In our study of Android TV, we enable the Android Debug Bridge on the Smart TV.

Android Debug Bridge (ADB). All Android-based devices, including Smart TVs, have in-built support for developer debugging tools. The Android Debug Bridge [23] is one of the most important components of these debugging tools. Because FireTV OS is based on Android (although not Android TV), it is also equipped with ADB. ADB allows a system to communicate seamlessly with an Android device through a daemon (ADB daemon), a background process running on the Smart TV. The ADB Daemon runs at elevated privileges to give developers the necessary tools to develop and debug Android apps, such as installing, and uninstalling packages, rebooting the device, capturing screenshots, and accessing the internal file system of the device, etc.

We use the ADB daemon on the Smart TV to send malicious commands. However, the ADB daemon needs to be activated on the Smart TV by first enabling developer mode, then enabling debugging. We inject the sequence of key events into the Smart TV device to mimic a user enabling the desired settings to activate this mode. Once the ADB daemon has been activated, an ADB server can now connect to the daemon. Upon receiving a new unrecognized connection, a popup is shown on the Smart TV asking the user to confirm the connection. Because we have a software remote paired, we can send the proper key events to accept this dialogue. We have posted videos demonstrating this process in practice on an anonymized web page [51]. Unlike app installation, the sequence of key events to activate does not vary with the screen layout but *can* vary with different OS versions. This limits the number of possible sequences of key events to at most the number of major Android OS Versions. More specifically the number of combinations is limited to the number of times the settings menus have been changed in the Android TV OS, which does not occur in every major OS update. Additionally, we can obtain the manufacturer and major OS version of the TV after pairing, reducing the number of sequences we must try to activate the ADB daemon.

3.2.3 Attack. With the ADB Daemon active on the Smart TV, an attacker can perform a long list of practically unbounded privileged actions. This section describes two attacks that can target individual or all apps to steal users' credentials.

Our first attack involves installing a malicious look-alike application. Using ADB, the attacker obtains a list of installed apps, and if it detects an app such as Amazon Prime Video or Netflix, it can uninstall the legitimate app and install a malicious look-alike in its place. The next time the user attempts to use that app, she will be directed to the malicious look-alike, enter her credentials as if she is on the trusted benign app. The stolen credentials are then sent back to the malicious server, giving the attacker access to the user's account. To state the obvious, stealing of credentials of several of these services (Google, Amazon, etc.) is serious and goes far beyond illegitimate access to entertainment options. These credentials are re-used for purchases and professional purposes (such as, shared documents and cloud computing).

The second attack is the generalization of the above attack. Again, Spook scans the device for installed apps that it wishes to steal credentials from and then wipes all of the user data in the targeted apps, including saved passwords. The next time the user launches the app, she will have to sign in to continue using the service. Because Android TV devices lack physical keyboards, the user must navigate through an on-screen keyboard with the directional pad on their remote. Spook starts recording the screen and analyzes which characters were selected on the on-screen keyboard, extracting the user's credential. In a sense, this attack is easier than the previously mentioned one since the attacker does not have to develop a look-alike malicious app. The victim Smart TVs can also be used in a botnet since they are typically connected to the Internet through high bandwidth pipes, e.g., the Ares botnet [38].

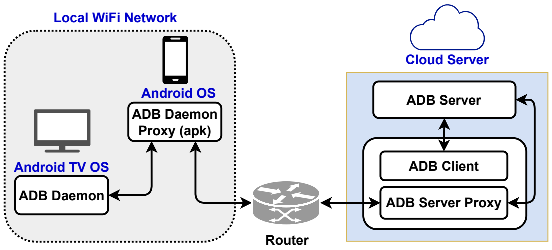


Figure 5: ADB Architecture using a NAT punchthrough via a series of proxies. The ADB Proxy allows the connection between the malicious ADB client (external network) and the ADB daemon located in the Android TV device.

3.3 Implementation

We implemented our attack in an Android app, which we call Spook. We need to understand the WiFi remote protocol semantics to implement Spook. While legacy code bases for older WiFi remote protocols can be found [20–22], to our knowledge there is *no available documentation or source code* for the version of the WiFi remote protocol we target. To handle this, we manually reverse-engineered the Android TV Remote Control app [30] from the Google Play Store[4]. Details from our reverse engineering process can be found in Appendix A.1.

Given the IP address of the target Smart TV, our Spook app uses the guessing strategy, as explained before in Section 3.2.1, to automatically pair with the Smart TV. We achieve persistence by using ADB daemon and use ADB Proxy, which allows a malicious server *anywhere in the world* to command and control the Smart TV directly. Figure 3a shows the command and control set up by Spook to the Smart TV, which provides the attacker direct access to the TV.

ADB Proxy. Figure 5 depicts the architecture of the ADB Proxy. The ADB Proxy employs two proxies to conduct a NAT punchthrough. This allows an ADB server anywhere in the world to connect to an Android device inside a local area network. The ADB Daemon Proxy is connected to the ADB Daemon on the Smart TV while the ADB Server Proxy is connected to the ADB Server on the cloud server.

The ADB Server needs a method for initiating a connection to the ADB Daemon. The proxy within Spook inside the local area network (the ADB Daemon Proxy) initiates a connection with the ADB Server Proxy in the cloud server. We will call this connection the Proxy Socket. When the Proxy Socket is opened, the ADB Server Proxy acts as an ADB Client and commands the ADB Server to connect to itself as if it was an ADB Daemon. This connection is the Server Socket. When the ADB Server opens the Server Socket, the ADB Server Starts sending ADB messages. These messages are then forwarded over the Proxy Socket to the ADB Daemon Proxy. When the ADB Daemon Proxy receives these messages, it will open a connection to the ADB Daemon on the Smart TV (the Daemon Socket) and forward the messages one last time. There is now an open TCP connection between the ADB Daemon and the ADB Server, and the two are able to send ADB messages over this TCP channel.

[4] Recall that this is the most popular software remote control app with over 10M installs.

3.3.1 Persistence without Virtual Remote. Instead of using Spook as the ADB Proxy, it is possible to install the ADB Proxy on the Smart TV device itself and have it run in the background. This installation passes successfully without any permission violation because the developer settings have already been enabled.

The ADB Proxy, now running on the Smart TV, can freely establish a connection between the ADB Daemon and a malicious server (again, possibly anywhere in the world). This mode of attack is more robust than having the proxy on the mobile because the attack persists even if the mobile leaves the network. Because Spook pairing happened silently (i.e., without user intervention) and the ADB Proxy installation on the Android TV also happens silently, the user stays blissfully unaware of the attack. Once installed, the ADB Proxy makes sure to relaunch itself upon system reboot (as a background process). Figure 3b shows the overview of this scenario. Note that with this persistent mode of attack, a traveling mobile device having Spook installed can infect as many Android TV devices as it comes in contact with. This significantly amplifies the effect of the attack.

3.4 Impact of missing RCDSPs

Although Spook starts by exploiting the violation of P_{va} property, it is important to note that the violation of P_{ha} is required to execute the other step of the attack, i.e., *Persistence* successfully. Consider the case where the property P_{ha} holds that means only humans can send the key events to the Smart TV. This prevents Spook from automatically performing any of its steps, not even pairing, reducing the effectiveness of the attack. In the presence of P_{ha}, Spook now needs to lure a human using Clickjacking techniques [32] to trigger the pairing request and subsequent key events, which decreases the effectiveness and practicality of the attack.

As shown in Table 1, all Smart TV OSes violate the P_{ha} property. An attacker can just use the violation of P_{ha} property to gain arbitrary control of the Smart TV. However, in this case, an attacker needs to first gain control of a virtual remote (or Software Remote) that is successfully paired with the target Smart TV. Then the attacker can use the exploited Software Remote to send key events to the Smart TV automatically.

4 DEFENSE MECHANISM

As shown in Table 1, *none* of the virtual remote protocols in Smart TV platforms has the *Human Attestation Property* (P_{ha}). This section presents modifications for adding P_{ha} to virtual remote protocol. Our modifications, along with the damping techniques (Android TV New), will ensure that the virtual remote protocol satisfies all the RCDSPs. Our solution relies on Trusted Execution Environments (TEE) [16], present in most smartphones through ARM TrustZone [44]. Details of TEE can be found in A.2. Furthermore, Smartphones with Software Remote is the most commonly used virtual remote device. Some background about TEEs is provided in the Appendix (Section A.2).

Ensuring P_{ha}. To ensure the pin is entered by a human, it must be signed by a TEE-protected key with a corresponding trusted public key certificate. Specifically, the pin p for pairing the virtual remote will be sent as $(Sign(p, T_{pk}), T_{cert})$, where T_{pk} is the private key and T_{cert} is the public key certificate corresponding to the hardware backed private key, T_{pk}. T_{pk} is embedded in the hardware of the device (hence the term, hardware key) and can only be read when the SoC is running in its most secure mode. Normal software cannot access this key. T_{cert} is made public on the Internet by vendors and can be verified through the appropriate certificate chain.

The remote service will first verify using T_{cert} that the message is correctly signed, and T_{cert} is not expired and attested by a known list of certificate authorities chain. The user needs to authorize the TEE every time it tries to access T_{pk} (i.e., sign a message). In order to sign any data with T_{pk}, the user must physically press a button on their device, *ensuring only a human can generate a signed message with T_{pk}, thus satisfying P_{ha}.*

Security analysis. We present possible attacks against our technique and how they will be prevented.

1) *Spoofing.* An attacker can try to spoof the signature by learning the private key. First, we use ephemeral keys (i.e., T_{pk} and T_{cert}) such that they are valid only for a very short time. Second, the root keys to generate T_{pk} and T_{cert} are stored in secure on-chip memory, which can only be accessed when the SoC is in the secure world, i.e., in TEE. Finally, these root keys are signed by device vendors (i.e., certificate authorities). So, it is impossible for the attacker to gain access to T_{pk} unless the attacker breaks the underlying cryptographic protocol or gets control of TEE (i.e., through software vulnerabilities [11]) or attacks the certificate authority infrastructure and steals the signing key.

2) *Replay attacks.* The attacker can store and replay previously signed messages to the remote service. We will use a nonce generated by remote service with periodic refreshes to reduce the replay window.

Implementation. We implemented our defense using AndroidKeystore [28], which follows TrustZone implementation for hardware-backed keys. Overall, our robust pairing process can be broken down as follows. First, the user initiates the pairing from Software Remote, in the mobile device. The Software Remote uses an action-bound key (T_{pk}) from AndroidKeystore, thereby forcing human interaction. When using T_{pk}, the user must press a physical button that does not allow software to fake the button press. In our case, the Google Pixel 3 ties the volume down button to the trusted execution environment, establishing a user is present every time the button is pressed. When the user wishes to pair the Software Remote to their TV, she is presented with a challenge response authentication. Similar to the systems in place, a 4 digit pin will be displayed to the TV that the user must enter. Once the pin has been entered, it is queued for signing with T_{pk}. The user then presses the button tied to the trusted-execution environment and the pin is signed with T_{pk}. The Software Remote then sends the signed pin as well as the public certificate to the TV for verification.

5 EVALUATION

We evaluate the effectiveness of both the proposed attack and defense. Specifically, we want our evaluation to answer three questions. (1) **Likelihood of Spook installation:** We want to evaluate the likelihood of a user installing our Spook app on their phone. (2) **Effectiveness of various phases of Spook:** How long does it take for Spook to gain complete control of a Smart TV? How much

Table 1: Summary of our analysis of the virtual remote support in popular TV OSes. Here ✓ and ✗ indicates whether the corresponding security property (Section 2) holds or not respectively. The letter G indicates that the property can be violated through guessing strategy (Section 2.2.3).

Smart TV OS	RCDSPs			
	Pairing Policy (P_{au})	P_{va}	P_{db}	P_{ha}
Android TV OLD	4-digit Hexadecimal (✓)	✗(G)	✓	✗
Android TV New	6-digit Hexadecimal PIN (✓)	✓*	✓	✗
Amazon FireOS	4-digit Decimal PIN (✓)	✗(G)	✓	✗
Roku OS	*None* (🐞)	✗(🐞)	✓	✓
Web OS	8-digit Decimal PIN (✓)	✗(G)	✓	✗

* After 5 failed attempts, the remote service does not allow a Software Remote to attempt pairing a 6th time until 30s have elapsed. After the 10th failed pairing attempt, the remote service requires the Software Remote to wait 30s after every subsequent failed pairing attempt.

🐞 The property never holds.

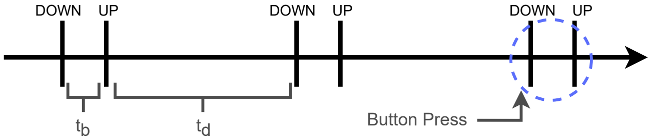


Figure 6: Diagram depicting the two types of delay required to create a UI event.

time is taken by each phase of our attack? (3) **Overhead of our defense:** What is the time overhead of our defense?

5.1 Likelihood of Installation

As we argue in Section 3.1, Spook only requires the INTERNET permission in the device. We want to verify how common this permission is. Hence, we utilize static analysis to determine how likely apps are to request this permission. To this end, we use the Androzoo [3] dataset, which has over 17M Android apps from different app stores (e.g., Google Play Store, F-droid)[5]. Our static analysis started with a random sample of 65,382 apps from the Google Play Store, with at least one release in 2021. Then, we filtered out apps no longer in the Play Store and prior versions of the same app. The final dataset had 24,837 apps. Next, we obtained the permissions required by each app from each manifest file. From the dataset, we find that most apps request the INTERNET permission (98.87%). Moreover, this is a *normal* permission and is granted when the app is installed, *without prompting the user for authorization*. Hence, we argue that it is safe to assume Spook can be disguised as a benign app and trick the user to install it without raising any suspicion.

5.2 Overhead

For these experiments, we measure the time it takes to initiate the ADB connection *after* the pairing process has completed. In this section we first discuss the artificial delay we must inject between key events such that the events are not dropped and then we discuss how long it takes to establish a connection to the ADB Daemon. We would like to minimize this time because if the user sees the Android TV reacting to UI events without her touching the remote, she may become suspicious. Because the UI events are scripted to precisely carry out the attack, even a single missed UI event will cause the end-to-end malicious action to fail.

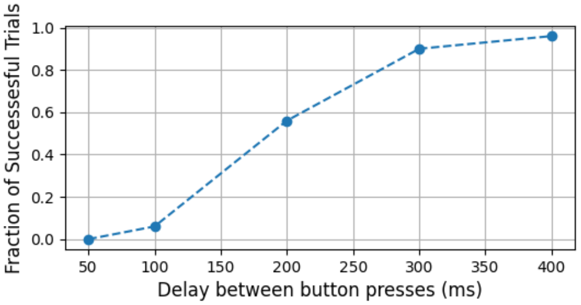


Figure 7: Plot showing the percent of trials that were successful in allowing ADB connections with different delays between button presses (t_d).

Delay between Key Events for Robust Attack. For every button press, there are two key events. The first signifies the button being pressed (ACTION_DOWN) and the second signifies the button being released (ACTION_UP). In order to simulate a button press, we send an ACTION_DOWN key event followed by an ACTION_UP key event. If the key events are sent too quickly, some are discarded. Therefore we must introduce some delay between the key events. This is shown in Figure 6 as t_b, 'b' because it simulates a button press. We empirically determine that at the value of $t_b = 10ms$, the button press registers reliably (and not at any lower value). Additionally, we needed to put some delay (t_d) between the simulated button presses, which would be the delay in the movement of the cursor. We experimented with several different delays between consecutive button presses to minimize the total duration of the attack while keeping it robust. We find that reducing the delay, t_d, too much would cause the OS to drop the key events, thus thwarting Spook from moving through the UI. In Figure 7 we evaluate the likelihood of success of the attack (to be exact, the attack steps after the pairing) as a function of the delay (t_d). Each delay value had 50 trials. Thus, we find that at the delay value of 400ms, the attack becomes robust, i.e., succeeds with high likelihood.

The most common reasons for key events to fail was the result of animations and the OS taking time to load different menus. Typically when the user clicks a button to navigate, there is a very short animation that takes place. The cursor does not simply disappear on one element and reappear on the next; usually there is a fading or scrolling animation to give the UI a smoother feel. These animations take time. If key events are sent too quickly, the cursor will not have had enough time to move and the key event will be discarded. As a side note, it is even possible for legitimate user UI events to be rapid enough that they are discarded by the OS [7].

We observe that an identical delay value is not needed between all button presses. Key events for navigating the settings menu can be sent very quickly, while key events to open menus or navigate the home screen typically require more delay. We fine tune the delay between the key events and rather than using the same homogeneous delay between all events, we empirically converge on the minimum amount of delay for each specific event. The delay value, reported in Table 2, is such that the end-to-end attack is successful 100% of the time, as measured over 50 trials for each delay value.

[5] This dataset is quasi-public. We had to apply for its use, and that triggered verification that we were legitimate security researchers, after which we were granted access.

Table 2: Required steps (along with the time) to establish an ADB connection from our cloud server to the ADB Daemon on the target Android TV device (Figure 5). Note, number of Button Presses and consequently key events for Enabling developer settings and ADB connections may change between manufacturers and API levels.

Required Step	Number of UI Actions needed on Android TV		Average Time (ms)
	Button Presses (B_p)	Key events ($B_p \times 2$)	
Enable developer settings & ADB connections	50	100	12,559
Initiate ADB connection on server	-	-	238
Allow incoming ADB connection from server	4	8	1,434
Verify the ADB connection	-	-	14
Total	54	108	14,244

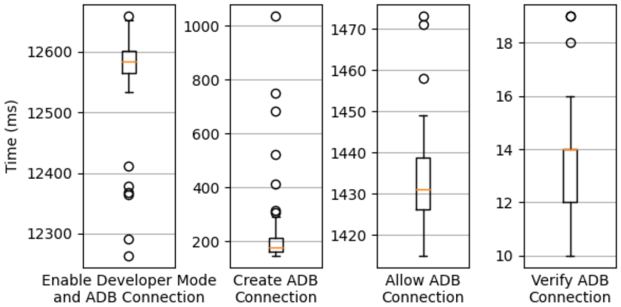


Figure 8: Plots showing the times it takes to complete the 4 steps in establishing an ADB connection over 50 trials.

Initiating an ADB Connection. This experiment measures the time it takes to initiate an ADB connection from the malicious server to the ADB daemon on the Android TV device (Figure 8). All the times are measured at the server and the server, the mobile, and the Android TV device are all on the same LAN for this experiment. The attack, after the pairing has been established, has four major steps.

(1) Inject key events to enable developer settings and enable ADB connections. **(2)** Open an ADB connection from the server to the TV. **(3)** Inject key events to allow the ADB connection. **(4)** Verify an ADB connection.

We show the individual times for each of these steps in Figure 8. Note the different scales on the Y axis for the different steps. Over 50 trials, we found the time it took to establish an ADB connection was quite consistent. The times ranged from 13.9s to 15.1s, with an average of 14.2s and standard deviation of 200ms. While this is not instantaneous, it is quick enough so that if the user is not paying careful attention to UI events on the TV monitor, she will not be suspicious. Looking at the individual steps, the first insight is that steps 1 and 3 which need many key events take longer than steps 2 and 4 which are mainly network activities. Since the amount of data sent in the handshake process is minimal and all entities are on the same LAN, the network times are low. As the number of key events to accomplish an activity goes up, the delay that needs to be interposed between the events adds up leading to a large value e.g., the "Enable Developer Mode and ADB Connection" step has many UI actions as shown in Table 2.

5.3 Cracking the Secret Code

For this experiment, we measure the time it takes to test a single pairing code. By knowing how long it takes to guess a single pairing code, we can estimate how long it will take to guess the pairing code correctly. For our tests, we used a Google Pixel 3 running Android 11 (API 30) attempting to pair with a Xiaomi Mi Box S running Android TV 9 (API 28). For each trial, we had the Software Remote start the pairing process. Spook would then generate a 4-digit hexadecimal code at random and attempt to pair to the Android TV device. Most of the time, the pairing (expectedly) failed. Upon failure, Spook would start a new trial. Each time a trial was started, we printed a timestamped message to the internal device log. Using this information we could find the time it takes to conduct each individual trial. After removing outliers from our data set, 1.65% of data points were removed and we were left with 79,455 trials. A

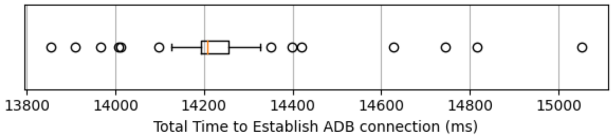


Figure 9: Plot showing the total time it takes to establish an ADB connection over 50 trials (summation of the 4 plots in Figure 8).

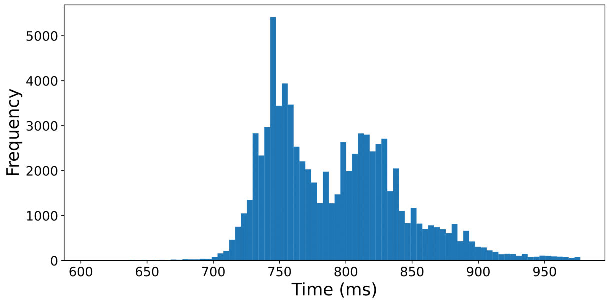


Figure 10: Histogram showing the distribution of the time it takes for one pairing attempt.

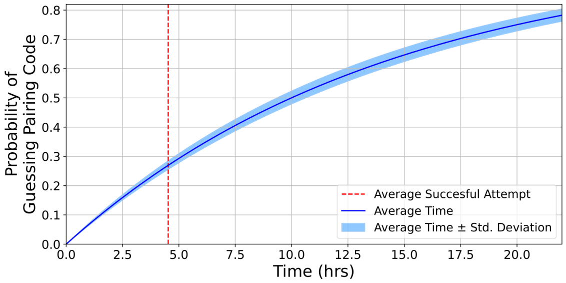


Figure 11: Probability of guessing the pairing code after a given amount of time analytically calculated using Equation 1. We also mark, with a vertical dotted red line, the average of the experimentally observed times in which Spook was able to successfully pair.

histogram showing the distribution of the times for a *single* attempt is shown in Figure 10. The amount of time it takes to attempt to pair will be dependent on the speed of the network connection, as well as the hardware being used. The variability in our result Figure 10 captures the variability in the network conditions over different times and different experimental locations. On average, it took 792ms to test a single pairing code. Using Equation 1, we calculate that after 10 hours there is over a 0.5 chance Spook has guessed the code correctly and after 20 hours this increases to over 0.75. The probability of guessing the pairing code after a given amount of time can be visualized in Figure 11. Recall again that the Spook does not need the UI to be in the foreground and can attempt to pair with the Android TV in a background service unbeknownst to the user. Note also that the TV monitor can be switched off and Spook's pairing component will still be executing. These contribute to the stealthiness of the attack. When we executed our automated attack script against an Android TV OS 11.0 device, it was able to successfully pair after an average time of 4hr 31m 30s.

5.4 Generality of Spook

Although we have created Spook primarily for Android TV, the attack and corresponding exploit are feasible for all other Smart TV devices that do not satisfy P_{ha} and P_{va}. Specifically, we should be able to send key events automatically, and the paring code can be guessed. To check the violations of P_{ha}, we designed an experiment with Android mobile and ADB. ADB enables us to inject tap events to Android mobile using the shell command `input tap <x> <y>`, where (x, y) is the coordinate on the screen where we wish to inject a tap event (i.e., the device behaves as if a user has touched the screen at (x, y) without actually having a user touch the screen). We wrote a script that enters the correct PIN on the device by sending `input` commands to the mobile using ADB. If the device pairs correctly, we know there are no sound checks to make sure a human is performing the pairing, verifying that P_{ha} has been violated. We also tested with random pairing codes and verified that there is no limit on the number of attempts, and the pairing code could be guessed without visual access to the Smart TV, thus violating P_{va}. This shows that Spook is feasible on other Smart TVs.

5.5 Defense Overhead

For this experiment, we evaluate the overhead added by our proposed solution. For this experiment, a Google Pixel 3 (Android 12) and a Google Chromecast TV (Android 10) were used. We compare the time it takes for two different methods to verify a virtual remote: the "Normal" insecure method and the "Secure" method that uses our defense. The normal method simply checks to see if the pin matches the one displayed on the TV. The secure method requires the virtual remote to send the signed pin to the remote service on the TV as well as the certificate used to create the signature. The TV then verifies this signature. We exclude the human time to press the button to access the hardware keys to sign the pin. We repeated the experiment 100 times for each of the control and the experimental setting. Figure 12 summarizes the results. On average, our defense mechanism added 113 ms to the pairing process. This overhead can be considered negligible and imperceptible to the human user considering the large times involved in the manual activities.

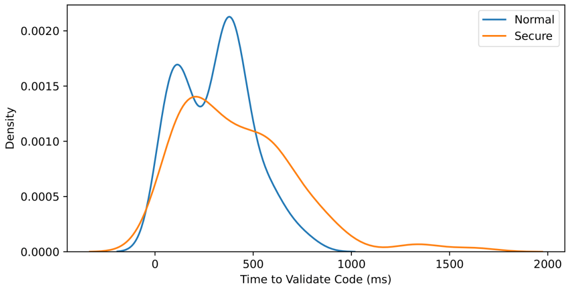


Figure 12: Distribution of running times of the pairing process with our defense vs without.

6 DISCUSSION

6.1 Root Causes

We have identified the root cause of the security vulnerability presented by Spook lies in the difficulty in verifying events that happen on a separate device. The problem is exacerbated when the only information channel is a network connection in which the protocol can be reverse engineered. The current systems in place do not verify a human user is attempting to pair the Virtual Remote, breaking the P_{ha}.

A root cause of the vulnerabilities relates to the Android ecosystem. Android provides different tools to interface with the Android-based platforms as part of the SDK. One of these tools is ADB, which provides a command-line interface useful for developers for debugging purposes. Prior researches [33, 40] have already noted security issues and vulnerabilities on ADB. As we discussed in Section 3 the main damage of our attack is done by connecting to the ADB interface. Spook leverages the use of an ADB Proxy to allow a connection between the ADB daemon in the Android TV device and an external ADB server. Unfortunately, ADB is an essential component of Android devices and cannot be removed as it prevents open access to the device. However, no verification that a human is present happens before activating the ADB daemon, a particularly dangerous policy considering that the daemon executes with elevated privileges. Further, the daemon accepts commands from an ADB server without regard to whether the server is in the same network as the device.

6.2 Traveling Attack

Because Spook is on a mobile device, it can travel between multiple networks. Consider that a visitor to the home has Spook installed on their phone. Spook can pair a Software Remote, activate the device debugger, and install the ADB Proxy on the Smart TV. Then, when the mobile that Spook is installed on leaves the local network the Smart TV remains under attack. Spook can then repeat this process when it travels to a new local network. An adversary anywhere in the world can build up an army of Smart TVs under their control with one instance of Spook installed on a mobile.

It is also noteworthy that the communication between the software remote and the remote service on the Smart TV uses standard Internet Protocol messaging. Therefore, *any* Internet enabled device on the network that is compromised can be weaponized to attack Smart TVs through their software remote service. Spook could even be reconfigured to be installed on the Smart TV itself and make a connection over localhost.

6.3 Alternative Defenses

Now we discuss other potential defenses and their shortcomings.
Increasing pairing code length. While increasing the pairing code length is a simple and quick fix, issues arise that make Arm TrustZone/TEE a much more elegant and permanent solution. Increasing the pairing code length may work temporarily, however as technology advances and connections become quicker, there may be a need in the future to increase the code length again, becoming an annoyance for end users. Additionally, there is a chance weak random number generation could be used, leading to the code being easily guessed. Hardware keys further protect against this possibility. Using hardware keys eliminates the current need and *any future* need to increase the code length because for each pairing attempt the user is required to press a physical button on their virtual remote to pair, satisfying P_{ha}.
Hardware Remote Verification. While Spook takes advantage of the WiFi-based pairing, one may think the same flavor of attack would be possible with Bluetooth-based pairing. For Android TV, that turns out to *not* be the case. Bluetooth-based pairing on Android TV has a fundamental distinction from WiFi-based pairing which satisfies P_{ha}, making this attack mode impossible over Bluetooth. A Bluetooth pairing process needs a physical button to be pressed. This by design eliminates the possibility of non-human, unsupervised pairing. However, the Bluetooth-based pairing lacks several features of a Software Remote, e.g., it still needs a hardware remote to be paired with the TV. One use case for Software Remotes is when a user is no longer able to use the hardware remote (eg. lost, broken, dead batteries, etc). Requiring the original hardware remote to pair a Software Remote removes these features from users proving a defense based on a TEE is much more effective and sensible solution.

Another potential solution requires a user to press a physical hardware button the smart TV. This approach is applicable for smart TVs that have the OS built into the monitor as they typically have several buttons. However, for smart TVs that come as a streaming stick and attach to a monitor, many do not have *any* buttons. The lack of hardware buttons on many smart TV device would make hardware remote verification an incomplete solution as it would leave many devices vulnerable.

7 RELATED WORK

Android Debug Tools. Security vulnerabilities and exploits on core components of Android OS (e.g., IPC model, multitasking mechanism, shared memory channels, etc.) have been widely explored. However, research focused on the security risks exposed by development and debugging tools available on the platform (such as, ADB) is limited. Lin *et al.* exposed vulnerabilities on ADB that allow an app to leak sensitive information of the device by capturing the GUI screenshots without elevated privileges [40]. Hwang *et al.* demonstrated privacy attacks by exploiting ADB capabilities [33]. All these attacks mainly focus on the power of ADB and the perils of enabling it and assume that ADB is enabled on the victim devices. But this assumption is invalid, as ADB is now disabled by default on all devices rendering these works not directly applicable. Although Spook uses ADB to carry out some phases of its attack, it first *enables* ADB. It exploits vulnerabilities in WiFi remote protocol, one of the essential components of Android TV, to enable ADB.
IoT and Streaming Device Platforms. Research focused on security threats of OTT platforms and streaming platforms is starting to gain momentum with the explosion of streaming platforms and services, like Netflix, Amazon Prime Video, and Hulu. These threats are commonly related to user privacy and the risk of DDoS attacks. In early work, Enev *et al.* demonstrated the possibility of inferring TV content by using TV's EMI (electromagnetic inferences) [17]. Johnson *et al.* presented a DoS attack targeting Android-based Smart TV device [37]. Mohajeri *et al.* presented a study focused on the security and privacy vulnerabilities on OTT streaming devices [41]. Their study was focused on Roku and Amazon Fire TV. Their study shed light on possible privacy leakage on streaming platforms due to bad security practices or API abuse from third-party apps. Spook can also leak sensitive information from the Android TV, but its scope is broader. Also we created an exploit that is related to a communication protocol while this prior work was a data collection and analysis study that shed light on the practice of using trackers and fingerprinting device IDs, both of which damage privacy. Recently, Aafer *et al.* found 37 unique vulnerabilities across 11 different Android TV devices by using a log-guided dynamic fuzzing approach [2]. In contrast, our work uncovers a serious security vulnerability in the OS itself that encompasses all vendor and device-specific features. Recently, Zhang *et al.* presented the EvilScreen attack [58], which, although similar in the end goal, requires multiple input channels such as Bluetooth, IR, and Ethernet and exploits inconsistencies across these channels. However, Spook has no such requirement and exploits the design flaw in the protocol itself independent of the communication channel.

There have been a few practical demonstrations of attacks against Smart TVs. Some exploited vulnerability in the Hybrid Broadcast Broadband (HbbTV) TV standard, such as, Flash and JavaScript on it [19], or the low-level wireless RF or infrared protocol between the hardware remote and the TV [42, 43], or a combination of channel hijacking and URL injection attack [10]. Our work is at a higher level in the protocol stack and uses a vulnerability in the pairing protocol. It is also not dependent on the TV standard that the device is compliant with, while these prior works are.
Security Analysis by Reverse Engineering. The vulnerabilities exploited by Spook were discovered by reverse engineering the Android TV WiFi remote protocol. Other recent works have demonstrated vulnerabilities in IoT devices by reverse engineering proprietary protocols. Antonioli *et al.* [8] exploit Android's Nearby Connections API by reverse engineering the protocol. Almon *et al.* [4] show attacks on Neighbor Awareness Networking (NAN) systems by reverse engineering NAN protocols present in Android. They then present an open source version of the NAN protocol and are able to infer issues with Apple's similar system, Wireless Direct Link. This is similar to how Spook only targets Android TV, but its principals can be applied to target many Smart TV OSes.

8 CONCLUSION

In this paper we investigated the properties (RCDSPs) that make traditional remote control devices secure. Upon our investigation, 3 unique properties became apparent that lead to a secure system in traditional remote control devices: the Visual Access Property (P_{va}), the Distance Bounding Property (P_{db}), and the Human Attestation Property (P_{ha}). Applying these properties to modern virtual remotes, we found several of these properties are violated in popular OTT streaming platforms. Taking advantage of the violated properties we formulated an attack for four of the most popular OTT platforms. In practice, we developed and showed the feasibility of Spook, a piece of malware packaged in an Android smartphone to take over an Android TV device.

We design and implement a defense using ARM TrustZone, that guarantees that a human is initiating the pairing. This defense thwarts Spook as well as any UI injection-based attack against Smart TVs, providing a secure version of WiFi remote protocol. The overhead in terms of time to pair is insignificant. All our findings have been communicated to the corresponding vendors. Google acknowledged our findings as a security vulnerability, assigned it a CVE, and released a new version of Android TV OS in September 2021 to *partially* mitigate the attack.

REFERENCES

[1] 9to5Google. 2019. Android TV has added over 130 operator partners since 2016, doubles unit growth every year, more. https://9to5google.com/2019/03/28/android-tv-operator-growth/.

[2] Yousra Aafer, Wei You, Yi Sun, Yu Shi, Xiangyu Zhang, and Heng Yin. 2021. Android SmartTVs Vulnerability Discovery via Log-Guided Fuzzing. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association. https://www.usenix.org/conference/usenixsecurity21/presentation/aafer

[3] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2016. AndroZoo: Collecting Millions of Android Apps for the Research Community. In *Proceedings of the 13th International Conference on Mining Software Repositories* (Austin, Texas) *(MSR '16)*. Association for Computing Machinery, New York, NY, USA, 468â€“471. https://doi.org/10.1145/2901739.2903508

[4] Lars Almon, Arno Manfred Krause, Oliver Fietze, and Matthias Hollick. 2021. Desynchronization and MitM Attacks Against Neighbor Awareness Networking Using OpenNAN. In *Proceedings of the 19th ACM International Symposium on Mobility Management and Wireless Access* (Alicante, Spain) *(MobiWac '21)*. Association for Computing Machinery, New York, NY, USA, 97â€“105. https://doi.org/10.1145/3479241.3486689

[5] Amazon Mobile LLC. [n. d.]. Amazon Fire TV. https://play.google.com/store/apps/details?id=com.amazon.storm.lightning.client.aosp.

[6] Ole Andre. 2021. Frida. https://www.frida.re.

[7] Android. 2019. Android: Input events overview. https://developer.android.com/guide/topics/ui/ui-events.

[8] Daniele Antonioli, Nils Ole Tippenhauer, and Kasper Rasmussen. 2019. Nearby Threats: Reversing, Analyzing, and Attacking Google's "Nearby Connections" on Android. In *Network and Distributed System Security Symposium (NDSS)*.

[9] Ben Schoon. 2020. Android TV makes up 1 of every 10 Smart TVs globally, trails behind Samsung and Roku. https://www.fiercevideo.com/video/samsung-tizen-expands-global-connected-tv-share.

[10] Pedro Cabrera. 2019. SDR Against Smart TVs; URL and channel injection attacks. In *Defcon*.

[11] David Cerdeira, Nuno Santos, Pedro Fonseca, and Sandro Pinto. 2020. Sok: Understanding the prevailing security vulnerabilities in trustzone-assisted tee systems. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1416–1432.

[12] Catalin Cimpanu. [n. d.]. A new IOT botnet is infecting Android-based set-top boxes. ([n. d.]). https://www.zdnet.com/article/a-new-iot-botnet-is-infecting-android-based-set-top-boxes/

[13] Catalin Cimpanu. 2019. A new IOT botnet is infecting Android-based set-top boxes. https://www.zdnet.com/article/a-new-iot-botnet-is-infecting-android-based-set-top-boxes/.

[14] CodeMatics Media Solutions. [n. d.]. Remote for Android TV's / Devices: CodeMatics. https://play.google.com/store/apps/details?id=codematics.android.smarttv.wifi.remote.tvremote.

[15] Zak Doffman. 2019. Samsung's Warning To Owners Of QLED Smart TVs Is Quickly Deleted. https://www.forbes.com/sites/zakdoffman/2019/06/18/samsung-issues-then-deletes-warning-to-check-smart-tvs-for-malicious-software/.

[16] Jan-Erik Ekberg, Kari Kostiainen, and N Asokan. 2013. Trusted execution environments on mobile devices. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 1497–1498.

[17] Miro Enev, Sidhant Gupta, Tadayoshi Kohno, and Shwetak N Patel. 2011. Televisions, video privacy, and powerline electromagnetic interference. In *Proceedings of the 18th ACM conference on Computer and communications security*. 537–550.

[18] Alois Ferscha, Simon Vogl, Bernadette Emsenhuber, and Bernhard Wally. 2008. Physical shortcuts for media remote controls. In *Proceedings of the 2nd international conference on INtelligent TEchnologies for interactive enterTAINment*.

[19] D Goodin. 2017. Smart TV hack embeds attack code into broadcast signalâ€”no access required. https://arstechnica.com/information-technology/2017/03/smart-tv-hack-embeds-attack-code-into-broadcast-signal-no-access-required/.

[20] Google. 2011. anymote-protocol. https://code.google.com/archive/p/anymote-protocol/.

[21] Google. 2011. google-tv-pairing-protocol. https://code.google.com/archive/p/google-tv-pairing-protocol/.

[22] Google. 2011. google-tv-remote. https://code.google.com/archive/p/google-tv-remote/.

[23] Google. 2017. Android Debug Bridge. https://developer.android.com/studio/command-line/adb

[24] Google. 2017. Google TV Pairing Protocol. https://android.googlesource.com/platform/external/google-tv-pairing-protocol/.

[25] Google. 2019. Google I/O 2019: Best Practice for Developing on Android TV. https://youtu.be/Vo-UQDVykIs.

[26] Google. 2020. Android Developers. KeyEvent. https://developer.android.com/reference/android/view/KeyEvent

[27] Google. 2020. Strategy Analytics: Samsung Leads As Global TV Streaming Device Population Reaches 1.1 Billion. https://www.businesswire.com/news/home/20200902005816/en/.

[28] Google. 2021. Android Developers. Hardware-backed Keystore. https://source.android.com/security/keystore. https://source.android.com/security/keystore

[29] Google. 2021. Android Security Bulletin. https://source.android.com/docs/security/bulletin/2021-11-01#android-tv

[30] Google LLC. [n. d.]. Android TV Remote Control. https://play.google.com/store/apps/details?id=com.google.android.tv.remote.

[31] Andy Greenberg. 2019. Watch a Drone Take Over a Nearby Smart TV. https://www.wired.com/story/smart-tv-drone-hack/.

[32] Lin-Shung Huang, Alex Moshchuk, Helen J Wang, Stuart Schecter, and Collin Jackson. 2012. Clickjacking: Attacks and defenses. In *21st {USENIX} Security Symposium ({USENIX} Security 12)*. 413–428.

[33] Sungjae Hwang, Sungho Lee, Yongdae Kim, and Sukyoung Ryu. 2015. Bittersweet ADB: Attacks and Defenses. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security* (Singapore, Republic of Singapore) *(ASIA CCS '15)*. Association for Computing Machinery, New York, NY, USA, 579â€“584. https://doi.org/10.1145/2714576.2714638

[34] ZDNet Inc. 2019. FBI warns about snoopy smart TVs spying on you. https://www.zdnet.com/article/fbi-warns-about-snoopy-smart-tvs-spying-on-you/.

[35] Innovation Lab. [n. d.]. Remote Android TV. https://play.google.com/store/apps/details?id=fr.bouyguestelecom.remote.

[36] Mordor Intelligence. 2021. Global Over the Top (OTT) Market - Growth, Trends, COVID-19 Impact, and Forecasts (2021 - 2026). https://www.mordorintelligence.com/industry-reports/over-the-top-market.

[37] Ryan Johnson, Mohamed Elsabagh, and Angelos Stavrou. 2016. Why Software DoS Is Hard to Fix: Denying Access in Embedded Android Platforms. In *Applied Cryptography and Network Security*, Mark Manulis, Ahmad-Reza Sadeghi, and Steve Schneider (Eds.). Springer International Publishing, Cham, 193–211. https://doi.org/10.1007/978-3-319-39555-5_11.

[38] WootCloud Threat Research Labs. 2019. WootCloud Discovers ARES ADB IOT Botnet Targeting Android Devices especially STBs/ TVs. https://wootcloud.com/wp-content/uploads/2019/10/WootCloud-Discovers-ARES-ADB-IOT-Botnet-Targeting-Android-Devices-especially-STBs_-TVs-1.pdf. (August 2019).

[39] LG Electronics, Inc. [n. d.]. LG ThinQ. https://play.google.com/store/apps/details?id=com.lgeha.nuts.

[40] Chia-Chi Lin, Hongyang Li, Xiao-yong Zhou, and XiaoFeng Wang. 2014. Screenmilker: How to Milk Your Android Screen for Secrets.. In *NDSS*.

[41] Hooman Mohajeri Moghaddam, Gunes Acar, Ben Burgess, Arunesh Mathur, Danny Yuxing Huang, Nick Feamster, Edward W. Felten, Prateek Mittal, and Arvind Narayanan. 2019. Watching You Watch: The Tracking Ecosystem of Over-the-Top TV Streaming Devices. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 131â€“147. https://doi.org/10.1145/3319535.3354198

[42] Valerio Mulas. 2019. Hacking an Android TV in 2 minutes. https://medium.com/@drakkars/hacking-an-android-tv-in-2-minutes-7b6f29518ff3.

[43] Yossef Oren and Angelos D Keromytis. 2015. Attacking the internet using broadcast digital television. *ACM Transactions on Information and System Security (TISSEC)* 17, 4 (2015), 1–27.

[44] Sandro Pinto and Nuno Santos. 2019. Demystifying arm trustzone: A comprehensive survey. *ACM Computing Surveys (CSUR)* 51, 6 (2019), 1–36.

[45] Consumer Reports. 2018. Samsung and Roku Smart TVs Vulnerable to Hacking. https://www.consumerreports.org/televisions/samsung-roku-smart-tvs-

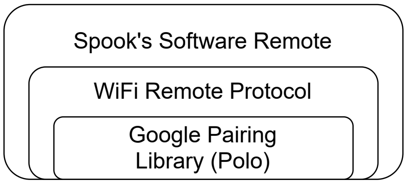


Figure 13: Protocol layering showing how our contribution (Spook's Software Remote) is built on top of two standard protocol layers provided by Google.

vulnerable-to-hacking-consumer-reports-finds/.

[46] Consumer Reports. 2021. How to Turn Off Smart TV Snooping Features. https://www.consumerreports.org/privacy/how-to-turn-off-smart-tv-snooping-features/.

[47] Roku Inc. [n. d.]. Roku - Official Remote Control. https://play.google.com/store/apps/details?id=com.roku.remote.

[48] Ben Schoon. 2021. Android TV and Google TV surpass 80 million active devices w/ strong US growth. https://9to5google.com/2021/05/18/android-tv-google-user-numbers-2021/.

[49] skylot. 2022. JADX: Dex to Java decompiler. https://github.com/skylot/jadx.

[50] Spook. Shared folder. 2021. https://drive.google.com/drive/folders/11Nh5XcMi_VkWm6h7brEJiGZDocpPkX7j?usp=sharing.

[51] Spook: The Unfriendly Ghost Remote. 2022. https://sites.google.com/view/friendly-ghost-remote.

[52] Statista. 2019. Share of smart TV shipments worldwide in 2018, by operating system. https://www.statista.com/statistics/257778/number-of-smart-tvs-installed-worldwide/.

[53] Enplug Support. 2020. Where can I find my TV's model number?

[54] Zack Whittaker. 2019. Now even the FBI is warning about your smart TVâ€™s security. https://techcrunch.com/2019/12/01/fbi-smart-tv-security/.

[55] Dan Wing. 2010. Network address translation: Extending the internet address space. *IEEE internet computing* 14, 4 (2010), 66–70.

[56] Alan Wolk. 2015. *Over the top: how the internet is (slowly but surely) changing the television industry.* Createspace Independent Publishing Platform.

[57] WootCloud. 2019. WootCloud Discovers ARES ADB IOT Botnet Targeting Android Devices especially STBs/TVs. https://wootcloud.com/wp-content/uploads/2019/10/WootCloud-Discovers-ARES-ADB-IOT-Botnet-Targeting-Android-Devices-especially-STBs_-TVs-1.pdf.

[58] Yiwei Zhang, Siqi Ma, Tiancheng Chen, Juanru Li, Robert H. Deng, and Elisa Bertino. 2022. EvilScreen Attack: Smart TV Hijacking via Multi-channel Remote Control Mimicry. https://doi.org/10.48550/ARXIV.2210.03014

A APPENDICES

A.1 Reverse Engineering a Software Remote

To implement Spook, we reverse engineered the software remote app developed by Google [22]. At the time of our work, although a few third-party apps exist [14, 35], there was no publicly available content detailing the Android TV WiFi remote protocol. We are the first to detail and publish inner workings of the Android TV WiFi remote protocol.

A.1.1 Reverse Engineering the Android TV Software Remote App. Our goal was to produce an executable that would attempt pairing a software remote to a given IP over a given port. In order to accomplish this, we first used the JADX Java decompiler [49] to decompile software remote apps for Android TV. While we decompiled third party Android TV software remote remote apps [14, 35], we ultimately used the Android TV software remote app developed by Google because the code was less obfuscated. After decompiling, we placed all of the code from the decompiled app into an Android Studio project. By viewing the decompiled code, we were able to look at external dependencies for the software remote app. We learned the software remote app uses Google's Pairing Protocol Library (Polo) [24]. Polo was developed by Google to establish pairing sessions between a client and server in the same local area network. We also learned that Android TV's WiFi remote protocol uses Polo heavily. This relationship is depicted in Figure 13. Upon analyzing the 3rd party software remote apps, we found they also used Google's Polo library and did not reverse engineer the Android TV's WiFi remote protocol from scratch.

Because we now had an Android Studio project with a large amount of code already written, we naturally decided to package Spook as an APK (Android Package Kit). However, JADX did not decompile all parts of the app, and there was manual effort required in order to make the app build. For example, there were some parts of the Java bytecode that JADX was unable to decompile, we went through manually and decompiled these sections. Additionally, there were many resource files for different activities within the app that did not decompile correctly and were preventing the Spook app from building. We went through manually and removed these references. This was not an issue, because our goal was to create an executable that when given an IP and port, it would attempt pairing to that device; we did not need any of the functionality from the activities. This also meant we can package Spook as a background service, adding to Spook's stealthiness. Keeping our goals for Spook in mind, we were able to remove other code for device discovery and the implementation for the Bluetooth software remote. This allowed us to better focus on the relevant parts of the decompiled code.

We used Frida [6], an instrumentation toolkit, to better understand the control flow of Googles software remote app, as well as the WiFi remote protocol and the Polo library it utilizes. Using Frida helped tremendously with debugging our own WiFi remote protocol implementation. For example, a function that would send key events to the TV accepted an integer, however we were unsure what integers input to the function to trigger different key events. Using Frida we were able to look at the arguments and map integers to different arguments. This led us to discover that the integer arguments followed a mapping specified in Android Developer documentation [26].

A.1.2 Details of the Android TV WiFi Remote Protocol. We also used Frida to learn details about the WiFi remote protocol used for Android TV. We were able to instrument a method with the signature `sendMessage(byte[] bArr)`. This function is called every time the software remote wants to send a message to the Android TV. `bArr` represents the bytes that are to be sent to the Android TV.

During the pairing process, there are three messages that are sent to the Android TV. Before the first message is sent, the software remote must create a channel to send the messages over. The software remote uses the Polo library to set up a TLS connection to the Android TV over the WiFi remote port (typically port 6466). The first message that is sent is a ping message. After analyzing the decompiled code, we found this ping message is typically sent after a time interval of no messages in order to keep the connection alive. The Android TV remote service is expected to send a message in return. This return message is called the pong message. While this message typically is sent after an interval without any messages that are sent, it is also used to initiate the pairing process. We infer that this ping message readies the Android TV remote service for pairing of a software remote.

After the first message is sent, the software remote sets up another TLS connection using Polo. This time, the software remote initiates the connection to the Android TV, but over the pairing port. The pairing port is one above the WiFi remote port. For example, if the WiFi remote port was 6466, the pairing port is 6467. The software remote maintains both TLS connections simultaneously. Upon receiving the connection on the pairing port, the Android TV remote service displays a 4 digit pairing code on the Android TV. In the intended usage, the user would then type the pairing code into the software remote and then the software remote would start generating the second message. The second message always contains the same header, followed by a series of bytes that create a message in ASCII. The message in ASCII represents a version of the pairing code that has been hashed or manipulated in some way. The third message that is sent is always the same, it is most likely some acknowledgement that the software remote is ready to Key Events to the Android TV remote service.

When the TV is paired, all messages to send Key Events to the Android TV are sent over the WiFi remote port (typically port 6466). These messages all follow the same format, with a 5 byte header, followed by 3 fields: sequence field, action up/down field, and the Key Event index. The sequence field is an 8 byte integer that increases with subsequent Key Events. Each time a Key Event message is sent, the sequence field increases by one. The action up/down field is a 4 byte integer and only appears to take on 2 values: either zero (`0x00000000`) or one (`0x00000001`). Zero represents `ACTION_DOWN`, when the button is pressed. One represents `ACTION_UP`, when the button is released. The final field is 4 bytes and represents the Key Event the software remote wishes to inject on the Android TV. The mapping of Key Events to integers can be found in the Android Developer Key Event documentation [26].

A.2 Trusted Execution Environment

A Trusted Execution Environment (TEE) provides an isolated and privileged execution environment. Specifically, it enables the existence of two separate worlds on the same system on a chip (SoC), called the secure world (i.e., the world inside the TEE, e.g., Qualcomm's QSEE) and the non-secure world (i.e., the world containing the main OS, e.g., Android). The TEE also guarantees that sensitive data, such as hardware keys, are stored and processed in an isolated and trusted environment. ARM implements TEE through ARM TrustZone, which offers hardware-enforced isolation built into the CPU providing a secure world separated from the OS (non-secure world). ARM TrustZone is an integral part of modern mobile devices allowing Android OS to provide hardware-backed, key attestation, among other robust security services.