Inner Conflict: How Smart Device Components Can Cause Harm

Omer Shwartz, Amir Cohen, Asaf Shabtai, Yossi Oren

Ben-Gurion University of the Negev, P.O.B. 653, Beer-Sheva, 8410501 Israel.

Abstract

Mobile smart devices are built of smaller components that are often fabricated by third-parties, and not by the device manufacturers themselves. Components such as sensors, radio transceivers, and touchscreen controllers are generally supplied to the manufacturers, along with driver software that facilitates communication between the component and the host device. Driver software is normally embedded within the operating system kernel, where it is trusted to behave within defined parameters. Since the hardware of the smart device is expected not to change frequently, the device driver source code implicitly assumes that the hardware of the component is authentic and trustworthy. Such trust permits driver designs with a lax approach towards the integrity and security of data exchanged between the main processor and the hardware component.

In this paper, we question this trust in hardware components. Smart devices such as phones are often repaired with replacement components. Identifying and authenticating the source of these components is usually very difficult. We assume the threat consists of a malicious replacement touchscreen procured from an untrusted vendor. We construct two standalone attacks based on malicious touchscreen hardware. Our experiments demonstrate that these attacks allow an adversary to impersonate and eavesdrop on a user, exfiltrate data, and exploit the operating system, enabling the execution of privileged commands. For mitigating these and other similar attacks, we build and evaluate a machine-learning, hardware-based countermeasure capable of detecting abnormal communications with hardware components.

Keywords: Computer security, Privacy, Reverse engineering, Smart devices.

1. Introduction

Smart devices and gadgets can break or malfunction and often require repair. A common example is the smartphone which is easily dropped, resulting in a shattered touchscreen. According to a 2015 study, over 50% of global smartphone owners have damaged their phone screen at least once, and 21% of global smartphone owners are currently using a phone with a cracked or shattered screen [1]. While phones with broken touchscreens may be repaired at phone vendor-operated facilities such as Apple Stores, it is often more convenient and cost-effective for phone users to use third-party repair shops. Some technically savvy users may even purchase touchscreen replacement kits from online marketplaces such as eBay and perform the repair themselves. These types of unofficial repairs tend to include the cheapest possible components, and thus may introduce, knowingly or unknowingly, counterfeit components into the phone.

Replaceable components, including phone touchscreens, batteries or various sensors, are seldom produced by the phone manufacturers themselves. Instead, original equipment manufacturers (OEMs) such as Synaptics, MediaTek and Maxim Integrated provide these components, as well as the device driver source code, to phone manufacturers who integrate the components into their phones. The manufacturers then integrate this device driver source code into their own source code, making slight adjustments to account for minor differences between device models, such as memory locations, I/O bus identifiers, etc. These minor tweaks and modifications make the process of creating and deploying patches for such device drivers a very difficult endeavor, as we discuss further in Section 2. The example in Figure 1 illustrates this setting. The smartphone's main logic board runs a specific OEM code (the device driver) that communicates with the touchscreen over the internal bus using a common, simple interface. Even hardened, secure, or encrypted phones, such as those used by governmental and law enforcement agencies, often use commercial operating systems and a driver architecture that follows the same paradigm [2].

An important observation we make is that the device drivers (written by the OEMs and slightly modified by the phone manufacturers) exist inside the phone's trust boundary. In contrast to drivers for "pluggable" peripherals such as USB accessories, these OEM drivers assume that the internal components they communicate with are also inside the phone's trust boundary. We observe, however, that these internal components actually belong outside

Email address: omershv@post.bgu.ac.il, amir3@post.bgu.ac.il, shabtaia@bgu.ac.il, yos@bgu.ac.il. (Omer Shwartz, Amir Cohen, Asaf Shabtai, Yossi Oren)

of the smartphone's trust boundary. Indeed, there are some hundreds of millions of smartphones with untrusted replacement screens. This inherent dissonance underlies our research questions: *How might a malicious replacement peripheral abuse the trust given to it by its device driver? How can we defend against this form of abuse?*

Hardware replacement is traditionally considered a strong attack model, under which almost any attack is possible. However, in our case, we add an important restriction to this model: we assume that only a specific component, with an extremely limited hardware interface, is malicious. Furthermore, we assume that the repair technician installing the component is uninvolved. One can assume that these limitations make this attack vector weaker than complete hardware replacement; we show that it is not. Given the hundreds of millions of devices satisfying these assumptions in the wild [1], this form of abuse poses a significant danger.

While composing this paper, we have searched for allegations or reports of hardware implants in smartphones in public news sources. Despite our efforts could find none. This is unsurprising, and can be attributed to the highly directed and covert nature of these attacks, which are most likely to be mounted by nation-state attackers against targets who are either unaware of the attacks, or highly interested in suppressing reports about them.

In this work we highlight the risk of counterfeit or malicious components in the consumer setting, where the target is the user's privacy, personal assets, and trust. We show how a malicious touchscreen can record user activity, take control of the phone and install apps, direct the user to phishing websites and exploit vulnerabilities in the operating system kernel in order to gain privileged control of the device. Since the attack is carried out by malicious code running from the CPU's main code space, the result is a *fileless attack,* which cannot be detected by anti-virus software, leaves no lasting footprint and survives firmware updates and factory resets. We also propose a method for designing a flexible and generic countermeasure that may help protect against such attacks.

Our paper makes the following contributions:

1. We explore the risk of malicious peripheral attacks on consumer devices and argue that this avenue of attack is both practical and effective.
2. We detail recent events and analyses in regard to allegations of malicious hardware implants discovered in deployed severs and investigate the similarities to the attack model described in this paper.
3. We describe two attacks that form building blocks that can be used in a larger attack: a **touch injection** attack that allows the touchscreen to impersonate the user, and a **buffer overflow** attack that lets the attacker execute privileged operations. We implement both of these attacks on two Android devices assembled by different manufacturers.
4. Combining the two building blocks, we also present a series of **end-to-end** attacks that can severely compromise a stock Android phone with standard firmware. We implement and evaluate three different attacks, using an experimental setup based on a low-cost microcontroller embedded in-line with the touch controller communication bus. These attacks can: **Impersonate the user** - By injecting touch events into the communication bus, an attacker can perform any activity representing the user. This includes installing software, granting permissions and modifying the device configuration; **Compromise the user** - An attacker can log touch events related to sensitive operations such as lock screen patterns, credentials or passwords. An attacker can also cause the phone to enter a different state than the one expected by the user by injecting touch events. For example, we show an attack that waits until a user enters a URL for a website and then stealthily modifies the touch events to enter a URL of a phishing website, thereby causing the user surrender his or her private information**; Compromise the phone** - By sending crafted data to the phone over the touch controller interface, an attacker can exploit vulnerabilities within the device driver and gain kernel execution capabilities.
5. We make a case for a hardware-based countermeasure that listens to or intercepts communications over a hardware interface and monitors for anomalies. We implement such a countermeasure using machine learning techniques, and evaluate it against the attacks demonstrated. The evaluation shows a true-positive rate of 100% with a false-positive rate of 0% over a detection period of 0.2 seconds.

The rest of this paper is structured as follows. Section 2 reviews related work on smart device malware and hardware attacks. We summarize some of the current state of malicious hardware implants in Section 3. We explore the attack model in detail in Section 4, including a survey of the attack surface along with the reasoning behind the proposed countermeasure. In Sections 5, 6, and 7, we demonstrate the attacks described above. Section 8 describes the implemented countermeasure and its effectiveness against the attacks. Finally, our work is concluded in Section 9.

2. Related Work

Throughout the relatively short history of smartphones, both smartphone malware and smartphone protection mechanisms have evolved drastically as the smartphone platform grew popular. Android malware in particular has been shown to utilize privilege escalation, siphon private user data and enlist phones into botnets [3]. Bickford et al. [4] address the topic of smartphone rootkits, defining a rootkit as "a toolkit of techniques developed by attackers to conceal the presence of malicious software on a compromised system". A rootkit's malicious activities include

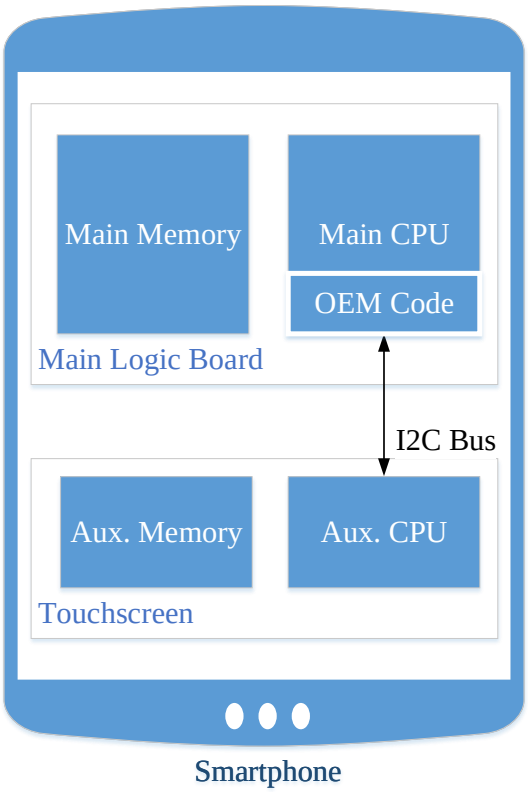

Figure 1: The smartphone, its touch screen, and its associated device driver software.

wiretapping into phone calls, exfiltration of positioning information and battery exhaustion denial of service. They can be detected with dynamic or static techniques [5].

Hardware interfaces have recently been the subject of concern for security researchers in the personal computer setting, due to their involvement in highly privileged processes [6]. Hardware components enjoying Direct Memory Access (DMA) such as the Graphics Processing Unit (GPU) can implant malware within the kernel memory [7]. Ladakis et al. [8] demonstrate a GPU-based keylogger where the GPU abuses its DMA capabilities for monitoring the keyboard buffer inside the kernel memory and saving keystroke information to the GPU memory. Brocker et al. [9] use the firmware update functionality of a MacBook iSight camera for installing malicious firmware on the camera. The authors showed how their own firmware could be used to take photos discretely or videos, without turning on the indicator light that informs the user that the camera is in use. Additionally, the authors use their firmware for enumerating the camera as a USB keyboard and present the ability of the enumerated device to escape virtual machine confinement.

When considering vulnerabilities emerging from hardware interfaces, some prior research has focused on external hardware interfaces such as USB. Wang et al. [10] performed an extensive study on the exploitation of smartphones via trusted profiles such as the Human Interface Device (HID), as well as through undocumented capabilities. Yang et. al [11] used power analysis techniques on a smartphone's USB connection to detect and classify touch screen activity, allowing them to infer browsing activity through this channel.

Most of the existing work dealing with internal hardware interfaces focused on hardware components that can either be updated by the user or easily replaced. Compared to PCs, smartphones are more monolithic by design, with a static hardware inventory and components that can only be replaced with matching substitutes. The smartphone operating system contains device firmwares that can only be updated alongside the operating system. Thus, there has been far less research emphasis on smartphone hardware, based on the assumption that it cannot be easily replaced or updated and is therefore less exposed to the threats discussed above. We challenge this assumption, noting that smartphone components are actually replaced quite frequently and often with non-genuine parts, as we show in Section 4.

The trouble that accompanies counterfeit components has not been completely ignored by the mobile industry (e.g., the "error 53" issue experienced by some iPhone users after replacing their fingerprint sensors with off-brand ones and failing validity checks [12]), however, it seems that such validity checks are not widely accepted, as counterfeit replacement components usually pass unnoticed. The risk of counterfeit components was also raised in the national security setting in a National Institute of Standards (NIST) draft, with an emphasis on supply chains [13]. Indeed, counterfeit assemblies or components are a key point of concern for the hardware industry [14].

A factor that hinders development of standard methods for counterfeit components is device customization. Android device customization security hazards have been systematically studied by Zhou et al. [15]. The authors designed a tool, ADDICTED, that detects customization hazards. The authors raised the concern that the customizations performed by manufacturers can potentially undermine Android security. In a previous work [16] we focused on driver customizations, reviewing the source code of 26 Android phones and mapping the device drivers that are embedded in the kernel of their operating system. Our survey found a great deal of diversity in OEMs and device drivers. Each phone contained different driver software, and there were few common device drivers among the phones tested. This landscape makes it difficult to patch, test, and distribute fixes for vulnerabilities found in driver code.

In the wider context of smartphone security, several works [17, 18, 19] have investigated the potential harm to user privacy by OEM and third party code, such as advertisement and analytics libraries. In contrast to these works, which address third-party code running as part of applications with user-level permission, we discuss here OEM code in drivers, which has kernel-level permissions and therefore carries a higher security risk.

3. Alleged Hardware Implants Reported

On October 4th, 2018, an article titled "The Big Hack: How China Used a Tiny Chip to Infiltrate U.S. Companies" was published in the Bloomberg Businessweek website [20]. The article described a malicious hardware implant of Chinese origin, as small as a grain of rice, that was found in electronic boards used in servers deployed in datacenters. According to Bloomberg, "The attack by Chinese spies reached almost 30 U.S. companies, including Amazon and Apple, by compromising America's technology supply chain, according to extensive interviews with government and corporate sources". Bloomberg also claimed that the affected companies discovered and reported these implants to U.S. authorities.

3.1. Responses

The report from Bloomberg was quickly met with skepticism and denial, a response article published by The Washington Post under the title "Your move, Bloomberg" [21] called for Bloomberg to provide more information while reaching out to the allegedly attacked companies and publishing their responses. The responses quoted of Amazon and Apple included a complete denial of any knowledge of hardware modification or implants in their datacenters. Later on, some of the companies even called for Bloomberg to retract the article [22].

While the equipment manufacturer Supermicro's stock lost over 40 percent of its value [23], the security community remained conflicted and confused about whether these attacks happened and if such an implant can actually be manufactured, installed and maintain effectiveness without being detected.

3.2. Properties of the Implant

The Bloomber article described the hardware implant as smaller than a grain of rice and also not being a part of the original board design. Additionally, Bloomberg provided a drawing showing the implant's location on the motherboard. Security researchers [24] proposed that based on the drawing, the implant was disguised as a passive component near the Flash memory chip containing the firmware for the BMC (Baseboard Management Controller).

The BMC is an integrated circuit (IC) that is responsible for the management of various resources of the electronic board and usually have direct access to other processors and their memory as well a network connection. Compromising the BMC can mean compromising the whole system.

A typical BMC, such as the one installed in the motherboards examined, makes use of an SPI (Serial Peripheral Interface) Flash chip to hold its firmware. Subverting the SPI Flash or it's communication lines is equal to modifying the BMC firmware.

3.3. Feasibility of an Implant

Security researcher Trammel Hudson investigated if such an implant can be manufactured and would it have any ability to disrupt and spy on the motherboard [25]. In his work, Hudson addresses several key issues contesting such an implant:

1. The ability to change the board behavior - by analyzing finding unencrypted portions in the BMC firmware Hudson showed how a simple modification to the firmware resulted in execution of arbitrary code in the BMC.
2. Initiating a change from a small physical footprint - by inserting his implant instead of a resistor on the motherboard, Hudson gained the ability to intercept data sent from the SPI Flash to the BMC and nullify bits on demand. Hudson also showed how these capabilities are sufficient for making useful changes to the firmware loaded to the BMC.
3. Manufacturing of a component small enough to avoid detection - Hudson showed how a modern microcontroller core is smaller than a standard surface mount resistor and therefore can be embedded within it.

3.4. Review and Limitations

A malicious implant in hardware, similar to the one reported by Bloomberg has the capacity for doing great damage. It benefits from the trust components have in embedded systems and also from simple interfaces such as SPI and Inter Integrated Circuit (I^2C) buses that facilitate communication between components.

The replacement of components is also demonstrated in projects such as the TPMGenie [26] where a malicious implant hijacks the I^2C bus between a CPU and a Trusted Platform Module (TPM) thus disabling a crucial level of security used in encryption and verification.

These attacks are limited by their requirement for access to the victim device during manufacturing or transport. In the rest of this paper we investigate how these attacks can be performed on an already deployed device by targeting replaceable units of hardware.

4. Our Attack Model

Counterfeit components have been in existence ever since the dawn of the industrial age. When tampered with, their effectiveness as attack vectors is also well known. What, then, is unique about the particular setting of a smartphone? We argue that our specific attack model is a unique restriction the hardware replacement attack model: we assume that only a specific component, with an extremely limited hardware interface, is malicious, while the rest of the phone (both hardware and software) can still be trusted. Furthermore, we assume that the repair technician installing the component does not have malicious intent, and won't perform any operations beyond replacing the

original component with a malicious one. One can assume that these limitations make this attack vector weaker than complete hardware replacement, however, we show that this is not the case, demonstrating that the nature of the smartphone ecosystem makes this attack model both *practical* and *effective*. Our attack model is particularly dangerous, given that there are hundreds of millions of devices in the wild that satisfying these assumptions.

The pervasiveness of untrusted components in the smartphone supply chain was investigated in September 2016 by Underwriters Laboratories (UL) [28]. UL researchers obtained 400 iPhone charger adapters from multiple sources in eight different countries around the world, including the U.S., Canada, Colombia, China, Thailand, and Australia, and discovered that *nearly all of them* were counterfeit and contained sub-standard hardware. Similarly, in October 2016, Apple filed a lawsuit against Amazon supplier Mobile Star LLC, claiming that "Apple, as part of its ongoing brand protection efforts, has purchased well over 100 iPhone devices, Apple power products, and Lightning cables sold as genuine by sellers on Amazon.com [...and] revealed that *almost 90% of these products are counterfeit*." [29]. Considering the condition of the third-party marketplace, one can assume with high confidence that unless a phone has been repaired at a vendor-operated facility such as an Apple Store, it is likely to contain counterfeit components.

4.1. Malicious Peripherals

Let us next assume that a malicious peripheral, such as a touchscreen, has made it into a victim's smartphone. What sort of damage can it cause?

As stated in [16, 30], attacks based on malicious hardware can be divided into two different classes. *First-order attacks* use the standard interaction modes of the component, but do so without the user's knowledge or consent. In the specific case of a malicious touchscreen, the malicious peripheral may log the user's touch activity or impersonate user touch events in order to impersonate the user for malicious purposes. We demonstrate some of these attacks in Section 5. *Second-order attacks* go beyond exchanging properly-formed data, and attempt to cause a malfunction in the device driver and compromise the operating system kernel. Such an attack requires that the peripheral send malformed data to the CPU, causing the device driver to malfunction and thereby compromise the operating system kernel. Once the kernel is compromised, it is possible to disable detection and prevention of suspicious system activity, eavesdrop on sensors and on other applications, and most significantly, operate on systems where only a partial software stack has been loaded, such as a device in a charging, standby, or off state [31, 32, 15, 33, 34].

While *first-order attacks* don't rely on the presence of a software vulnerability and can be performed by any malicious peripheral contained in consumer electronics, the existence of a vulnerability that can be exploited is a prerequisite of *second-order attacks*. Similar attacks performed by malicious *pluggable* peripherals (e.g., USB peripherals), to compromise a smartphone have been demonstrated [35]. A review of 1077 Android CVEs (Common Vulnerabilities and Exposures) patched between August 2015 and April 2017 shows that at least 29.5% (318 items) take place in the device driver context [27]. Figure 2 shows the growth in driver related CVEs.

Driver vulnerabilities are often detected in the pluggable setting where the driver controls a peripheral that might be detached or replaced by the user. This leads to a general lack of attention to the internal component setting. We argue that it is very likely that internal components might be used to exploit vulnerabilities just like pluggable components are known to do. In this paper we describe two such vulnerabilities that we found in common touchscreen drivers (Synaptics S3718 and Atmel T641).

4.2. Case Study - Touch Controller Communications

In most smartphones, the touch controller communicates with the device driver residing on the host processor via a dedicated I^2C bus [36], a general purpose, low speed bus designed for cost effective communication between ICs. The I^2C bus behaves as a physical layer between master and slave devices, where master devices are allowed to read and write from and to registers in the slave device's memory. By manipulating these registers, the device driver, acting as master, can control the behavior of the touch controller, acting as slave. In the other direction, the touch controller can send events to the device driver by populating the appropriate registers and triggering an interrupt. On top of this low-level communication interface, the device driver typically defines a proprietary layer required for the instrumentation and operation of the touch controller.

In the Nexus 6P phone, the Synaptics S3718 touch controller daughter board has I^2C connections to the host processor. It has an additional contact for generating an interrupt signal notifying the host processor of touch-related events. The I^2C bus operates at the rate of 400 Kbps.

A basic mapping of the touch controller registers and functions was extracted from the open source device driver made available by Google. Additional reverse engineering and observation provided a fuller picture of the communication protocol.

4.2.1. Boot up process

During the boot up process, the device driver probes the touch controller memory and learns which functions the controller possesses. A controller function or capability is reported through a six byte function descriptor. The function descriptor contains four register addresses used for manipulating the function, along with an interrupt count that signifies the number of interrupt types the function generates. A mapping of several controller functions can be seen in Table 1. After probing and querying for the functions, the device driver checks the firmware installed

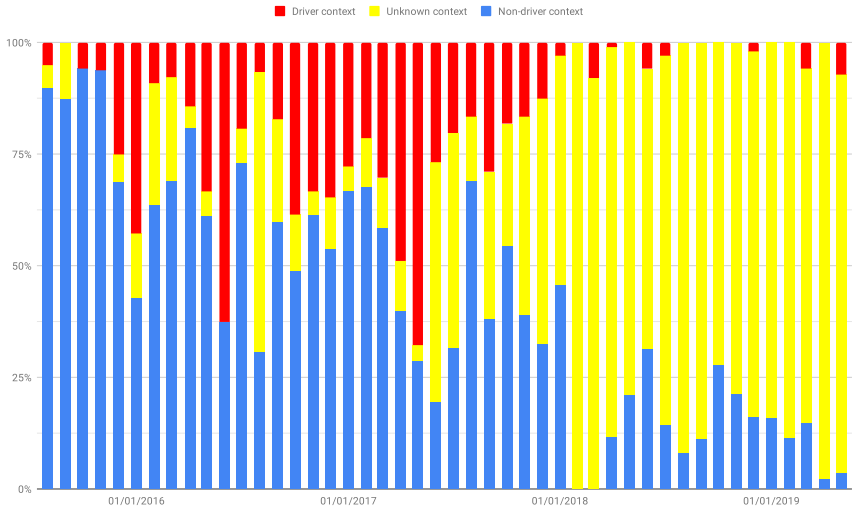

Figure 2: The percentage of patched Android CVEs that occur in driver context out of all patched Android CVEs. The figure was compiled using information from the Android Security Bulletins [27]. On May 2017, changes to the Android Security Bulletins website format decreased the amount of information given, rendering the context on many CVEs uncertain.

Table 1: Partial mapping of the main Synaptics S3718 functions and their purpose

Function ID	Query Address	Command Address	Control Address	Database Address	Function Purpose
0x01	0x3F	0x36	0x14	0x06	General control and status of the touch controller
0x12	0x5C	0x00	0x1B	0x08	Reporting of simple touch events, including multi-finger touches
0x51	0x04	0x00	0x00	0x00	Firmware update interface

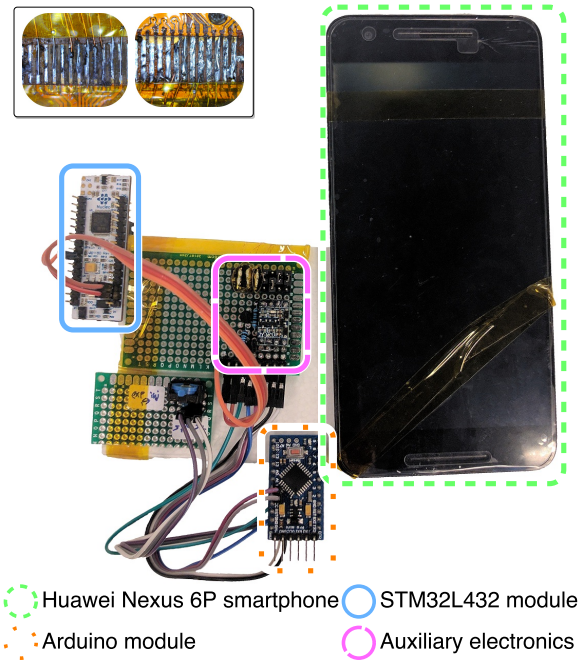

Figure 3: The complete attack setup. The figure shows an exposed touch controller interface wired to a prototyping board embedded with auxiliary electronics and connected to an Arduino microcontroller module. The prototyping board is also connected to an STM32L432 microcontroller module [37] which is used as a countermeasure. Inset: wires soldered onto the touch controller communication connection

against the firmware file embedded in the kernel memory and triggers a firmware update if necessary. Eventually, the device driver enables the appropriate handlers for all function specific interrupts and writes the configuration data to the relevant functions.

4.2.2. Touch reporting

In order to generate a touch event, the touch controller electrically pulls the interrupt line to the ground and thus notifies the device driver of an incoming event. The device driver in turn reads the interrupt register 0x06 and deduces which of the touch controller functions generated the interrupt. In the case of a normal touch event this will be function 0x12. The device driver continues to read a bitmap of the fingers involved in this event from register 0x0C and eventually reads register 0x08 for a full inventory of the touch event.

4.3. Attack Setup

The attacks were demonstrated on a Huawei Nexus 6P smartphone running the Android 6.0.1 operating system, build MTC19X, and operating with stock manufacturer firmware; in addition, the phone has been restored to factory state with a memory wipe using the "factory data reset" feature in the settings menu.

The touchscreen assembly was separated from the rest of the phone and the touch controller daughter board was located. Using a hot air blower on the connection between the touch controller daughter board and the main assembly daughter board we were able to separate the boards and access the copper pads. The copper pads were soldered to thin enameled copper wire that was attached to a prototyping board. Using this setup, we were able to simulate a chip-in-the-middle scenario in which a benign touchscreen has been embedded with a malicious integrated chip that manipulates the communication bus. Fig 3 shows the entire attack setup.

Our attack used a commonly available Arduino platform [38] based on the ATmega328 microcontroller. A setup such as the one described above can easily be scaled down in size by a factory or skilled shop in order to fit on the touchscreen assembly daughter board. ATmega328, the programmable microcontroller used in our attacks, is available in packages as small as 4x4x1 mm [39]. Other, more powerful microcontrollers are available with smaller footprints of 1.47x1.58x0.4 mm or less [40]. Since the data sent by our attack fully conforms to layers 1 and 2 described in the I^2C specification, it can also be implemented in the firmware of the malicious peripheral's internal microcontroller, removing the need for additional hardware altogether.

4.4. Applicability to Other Phone Models and Operating Systems

An analysis of other recent Android phones: Google Pixel 3 and Essential PH-1, suggests that similar architectures and practices are still shared among the majority of phone models.

The two phones analyzed use either I^2C or SPI buses to communicate with a multitude of peripherals such as wireless charger, power management, NFC, speaker amplifier, battery management and touchscreen.

Additional analysis of the driver source code revealed usage of unsafe programming practices such as the use of unrestricted memory copy functions. A few potential bugs and one minor security flaw were also found and are being communicated to the vendors.

5. First-Order Attacks

First-order attacks take advantage of peripherals' capabilities and abuse the trust they are afforded by the user or other components of the system. In this section, we demonstrate and evaluate several such attacks.

5.1. Touch Logging and Injection Attacks

In this attack, the malicious microcontroller eavesdrops on user touch events (touch logging) and injects generated ones into the communication bus (touch injection). The microcontroller software behind the phishing attack shown in Table 2 is built of three components: two state machines, one maintaining a keyboard mode and the other maintaining a typing state, and a database that maps screen

Table 2: A summary of the first-order attacks demonstrated.

Attack	Time to Execute	Screen Blanked?	Video Demo
Malicious software installation	21 seconds	Yes	`https://youtu.be/83VMVrcEOCM`
Take picture and send via email	14 seconds	Yes	`https://youtu.be/WS4NChPjaaY`
Replace URL with phishing URL	<1 second	No	`https://youtu.be/XZujd42eYek`
Log and exfiltrate screen unlock pattern	16 seconds	Yes	`https://youtu.be/fY58zoadqMA`

regions to virtual keyboard buttons. The state machine that maintains the keyboard modes changes state when a keyboard mode switch key had been pressed. The basic Nexus 6P keyboard has four modes: English characters, symbols, numbers, and emoji. The typing state machine is used for tracking the typed characters and matching them to specified trigger events (such as typing in a URL). Complex context information, such as keyboard orientation, language, activity, and even user identity, has been shown to be detectable from low-level touch events by other authors [41, 42, 43, 44]. When the required trigger event is reached, touch injection begins and a set of generated touch events is sent on the communication line. Our current hardware is capable of generating touch events at a rate of approximately 60 taps per second.

5.2. User Impersonation and User Compromise

The touch logging and injection capabilities shown can be extended and used for a variety of malicious activities. Since our attack model assumes a malicious touchscreen assembly, the attacker can turn off power to the display panel while a malicious action is performed, allowing most attacks to be carried out stealthily.

The first attack we demonstrate is the **malicious software installation** attack. As illustrated in the video, this attack installs and starts an app from the Google Play Store. By using Android's internal search functionality, the attacker can type in the name of the Play Store app instead of searching for it onscreen, making our attack more resilient to users who customize their graphical home screens. It is important to note that the attack can install an app with arbitrary rights and permissions, since the malicious touchscreen can confirm any security prompt displayed by the operating system. This attack takes less than 30 seconds and can be performed when the phone is unattended and when the screen is powered off.

Next, we show how the malicious touchscreen can **take a picture of the phone's owner and send it** to the attacker via email. As seen in the video, this attack activates the camera and sends a 'selfie' to the attacker. This attack also takes less than 30 seconds and can be performed while the display is turned off, allowing the attack to be carried out without the user's knowledge.

Our third attack shows how the malicious screen can stealthily **replace a hand-typed URL with a phishing URL**. As the video shows, this attack waits for the user to type a URL, then quickly replaces it with a matching phishing URL. The confused user can then be enticed to type in his or her credentials, assuming that a hand-typed URL is always secure. This attack takes less than one second but requires that the screen is turned on and the user is present, thus risking discovery. We note that the demonstrated attack setup has a typing speed of over 60 characters per second.

Our fourth attack shows how the malicious screen can **log and exfiltrate the user's screen unlock pattern** to an online whiteboard website. The video demonstrates how the attack records the user's unlock pattern and draws it on an online whiteboard, which is shared via the Internet with the attacker's PC. This attack demonstrates both the collection and the infiltration abilities of the attack vector. This attack also takes less than 30 seconds, and its exfiltration step can also be performed while the screen is turned off.

Table 2 summarizes the attacks discussed above.

6. Second-Order Attacks

Second-order attacks are attacks that exploit weaknesses in the system that are unintentionally exposed to the peripherals. The examples presented in this section show how software vulnerabilities, such as buffer overflows, can be leveraged in this fashion.

6.1. Arbitrary Code Execution Attacks

This attack exploits vulnerabilities in the touch controller device driver embedded within the operating system kernel in order to gain arbitrary code execution within the privileged kernel context. A chain of data manipulations performed by the malicious microcontroller causes a heap overflow in the device driver that is further exploited to perform a buffer overflow.

6.1.1. Design

As a part of the boot procedure, the device driver queries the functionality of the touch controller. We discovered that by crafting additional functionality information

we can cause the device driver to discover more interrupts than its internal data structure can contain, causing a heap overflow. Using the heap overflow we were able to further increase the amount of available interrupts by overrunning the integer holding that value. Next, an interrupt was triggered causing the device driver to request an abnormally-large amount of data and cause a buffer overflow. The buffer overflow was exploited using a Return Oriented Programming (ROP) [45] chain designed for the ARM64 architecture.

6.1.2. Implementation

When triggered, the malicious microcontroller shuts down power to the touch controller and begins imitating normal touch controller behavior. During the boot sequence, the malicious microcontroller emulates the memory register image of the touch controller and responds in the same way to register writes using a state machine. When probed for function descriptors in addresses higher than 0x500 that normally do not exist within the touch controller, the microcontroller responds with a set of crafted function descriptors designed to cause the interrupt register map to exceed its boundaries. Within the device driver, a loop iterates over the interrupt register map and writes values outside the bounds of an interrupt enable map, causing the integer holding the number of interrupt sources to be overwritten. After waiting 20 seconds for the boot procedure to complete, the microcontroller initiates an interrupt by pulling the interrupt line to the ground. The device driver, which should then read up to four interrupt registers, instead reads 210 bytes, causing a buffer overflow, a ROP chain that calls the Linux kernel function `mem_text_write_kernel_word()` that writes over protected kernel memory with a chosen payload resides within the 210 bytes requested from the touch controller. Table 3 contains additional information about the ROP chain.

6.1.3. Evaluation

Four different payloads were demonstrated on top of the ROP chain described above and tested in attack scenarios on a phone with stock firmware and factory-restored settings and data.

Each of the four payloads succeeded in compromising the phone's security or data integrity. A list of the tested payloads is as follows:

- Disable all user capability checks in `setuid()` and `setgid()` system calls. This allows any user or app to achieve root privileges with a simple system call.
- Silently incapacitate the Security Enhanced Linux (SELinux) [46] module. While SELinux will still report blocking suspicious activity, such activity will not actually be blocked.
- Create a user exploitable system-wide vulnerability. The buffer check is disabled for all user buffers on system calls, resulting in many different vulnerabilities exploitable through many techniques.
- Create a hidden vulnerability within the kernel. A specific kernel vulnerability is generated, functioning as a backdoor for a knowledgeable attacker while remaining hidden.

6.1.4. Attacks on additional devices

While the main attack demonstrated here is crafted for the Nexus 6P phone, many other phones use similar device drivers [16]. A small scale review performed by the authors on three additional phones that contain a Synaptics touch controller (Samsung Galaxy S5, LG Nexus 5X, LG Nexus 5) show that these phones have similar vulnerabilities to the ones exploited in the attack described here.

To further demonstrate the generality of our attack method, we extended it to another target device with a different hardware architecture. The device we investigated was an LG G Pad 7.0 (model v400) tablet. This device runs the Android 5.0.2 operating system and contains a different touchscreen controller than the Nexus 6P phone.

An STM32L432 microcontroller module was connected to the communication lines belonging to the touch controller, and the original touch controller daughter board was disconnected. The microcontroller was programmed to replay previously recorded responses of a genuine touch controller. Inspection of the device driver revealed unsafe buffer handling in numerous locations. By falsely reporting an abnormally large entity, the malicious microcontroller was able to cause the device driver to read 2048 bytes from the bus into an 80-byte global array. The buffer overflow affected kernel memory and resulted in the overrun of various internal pointers and eventually a crash.

While the attack shown in this section is not complete, these preliminary results show how the complete attacks shown in Sections 5 and 7 can be implemented on additional devices with different peripherals.

In addition, the similarity in different peripheral implementations makes adapting existing attacks to new peripherals easier. For example, after reverse engineering the touch reporting mechanism of the Atmel touch controller, the Synaptics touch injection attack can be copied onto to devices with an Atmel touch controller, even without discovering any vulnerability in the Atmel device driver.

7. End-to-End Attacks

While each of the attacks described in Section 5 poses a threat on their own, a combination of these attacks along with second-order attacks can lead to an even more powerful outcome.

The final attack presented completely compromises the phone, disables SELinux, and opens a reverse shell connected to a remote attacker. This attack is unique in that it requires an exploitable bug in the third-party device driver code. We describe this attack in more detail in the

Table 3: ROP chain designed for the ARM64 architecture. This chain results in a call to a predefined function with two arguments.

Gadget Order	Gadget Code	Relevant Pseudocode
1	`ldp x19, x20, [sp, #0x10]; ldp x29, x30, [sp], #0x20; ret;`	`Load arguments from stack to registers X19 and X20`
2	`mov x2, x19; mov x0, x2; ldp x19, x20, [sp, #0x10]; ldp x29, x30, [sp], #0x30; ret;`	`Assign X2 := X19; load arguments from stack to registers X19 and X20`
3	`mov x0, x19; mov x1, x20; blr x2; ldp x19, x20, [sp, #0x10]; ldr x21, [sp, #0x20]; ldp x29, x30, [sp], #0x30; ret;`	`Assign X0 := X19; assign X1 := X20; call X2(X0, X1)`

Table 4: A demonstration of a complete end-to-end attack.

Attack	Time to execute	Screen Blanked?	Video Demo
Complete phone compromise	65 seconds	Yes	`https://youtu.be/sDfD5fJfiNc`

following subsection and provide additional information in Table 4.

7.1. Phone Compromise

To completely compromise the phone, we use a combination of touch events and driver exploits, as illustrated in Figure 4: First, the attacker uses **touch injection** to install an innocent-looking app from the Google Play app market. The next time the phone restarts, the malicious microcontroller initiates **kernel exploitation** during the boot sequence and creates a vulnerability in the kernel that is exploitable by app. Once the phone completes booting, the previously installed app uses the vulnerability created by the microcontroller to take control of the system and perform malicious activity. The malicious app then reboots the phone and the now-compromised phone resumes normal activity.

7.2. Attack Implementation

For this demonstration, a user app was created and uploaded to the Google Play app market. The app starts when the phone boots up, and performs a series of system calls by writing to the pesudo-file `"/prof/self/clear_refs"`. While the phone is in a normal state, these system calls cause no issues and shouldn't raise suspicion. During the exploitation of the kernel by the malicious microcontroller, the actions of the pesudo-file `"/prof/self/clear_refs"` are modified, and a vulnerability is introduced to it. This causes a change in the behavior of the app which is now able to exploit that vulnerability and execute code in the kernel context. We note that since the app is designed to exploit a vulnerability that is non-existent under normal conditions, it appears completely benign when a malicious screen is not present. This enabled our app to overcome malware filters and detectors, including Google Play's gatekeeper, Google Bouncer.

Once the app has gained the ability to execute commands with kernel permissions, it elevates privileges to root, deactivates the SELinux protection module, exfiltrates private application data and authentication tokens, submits the data to an online server, and finally, creates a root shell enabling an attacker to gain remote access. A video demonstration of the attack is available via the link in Table 4.

8. Countermeasure

A countermeasure that efficiently protects against the demonstrated attacks while not hindering development and production needs to have certain attributes. It needs to be able to detect threats quickly while remaining cheap, generic, and easy to implement. Additionally, it must not be affected greatly by mechanisms such as software updates which may damage its detection capabilities.

In order to protect the phone from a malicious replacement component, we propose implementing a low-cost, hardware-based solution in the form of an I^2C *interface proxy firewall*. Such a firewall can monitor the communication of the I^2C interfaces and protect the device from attacks originating from the malicious component.

The new hardware component can be physically located in-line, on the path between the component and the OEM code running on the CPU (as shown in Figure 5) and thus can modify or block malicious communication. Another alternative is that the component does not interrupt the path between the untrusted component and the OEM code running on the CPU, but rather passively monitors the traffic without modifying it. This may allow the use of lower-cost components with only a single monitoring port and lower clock rates, since the component does not have to operate at the line speed. In another possible implementation of the solution, the firewall is not implemented as an individual hardware component, but rather as independent software modules running on the primary CPU.

The use of a hardware countermeasure allows for protection against both inserted malicious components and

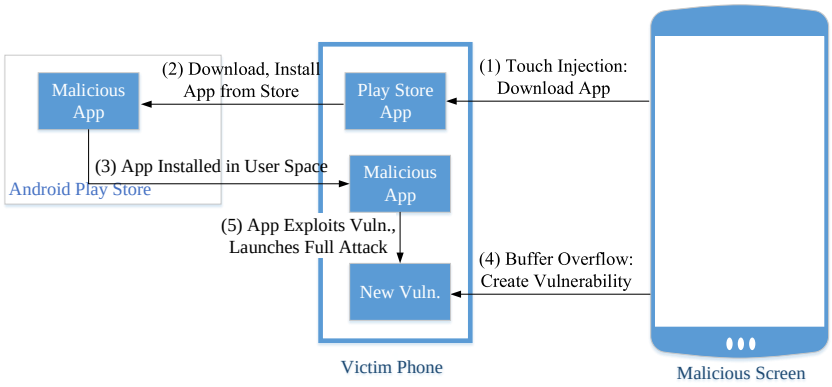

Figure 4: Fully compromising the phone using a malicious touchscreen.

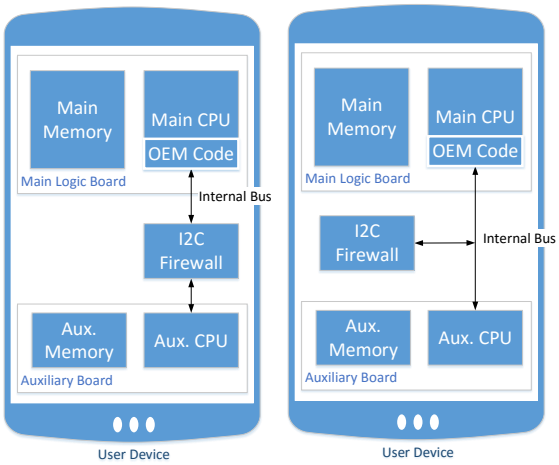

Figure 5: I^{2}C on path (left) and off path (right) firewall.

modified firmware attacks. It may also detect malicious behavior of firmware code that was modified by an insider and therefore still be signed or encrypted.

This proposed solution implements common firewall functionality such as signature matching, anomaly detection, filtering and rate limiting. Each data packet traveling between the component and the CPU, and vice versa, will be analyzed by the I^{2}C interface firewall. When the traffic traveling between the I^{2}C and CPU is detected as malicious, the firewall can block this traffic and thus protect the integrity of the CPU (if implemented inline as presented in Figure 5). In addition, the firewall can send a signal to the CPU or reset the CPU, thus returning it to a safe state.

8.1. Motivation

The unique attack model we discuss in our paper allows us to "fight hardware with hardware". While software countermeasures are capable of performing analysis of the contents of data packets, they are blind to variations in the implementation of the physical protocols. Hardware components have the ability to sense changes in the physical layer of the protocol, such as cadence, delays, voltages or even electronic interference. Many of these characteristics are unique for each family of microcontrollers, and a great deal of effort to is required to mimic or copy such characteristics in a different device.

Performing intrusion detection using hardware fingerprinting had previously been shown to be effective on more complex communication buses such as Controller Area Network (CAN bus) [47].

A hardware implementation allows for a transparent solution where the firewall component does not modify or influence the behavior of OEM-supplied code. Robustness is also achieved where when the firmware of the component or the OEM-supplied code has been updated, the firewall does not need to be modified or replaced. In addition, the hardware implementation can be developed to be generic at the interface level. An implementation of the firewall for the I^{2}C interface can be applied with little or no modifications to different devices and components communicating using the specific interface.

Such hardware implant can even be designed in a way that is agnostic and unaware of the type and function of the untrusted hardware peripheral. The implementation described and evaluated in the following subsections relies on data transmission statistics to create a model of normal communication and discover anomalies while being unaware of the type of peripheral it is guarding. Agnostic design also facilitates large scale implementation where a single design can be used in multiple hardware configurations.

8.2. Analysis

In our research, we focus on protecting against chip-in-the-middle or chip replacement attack scenarios where the main CPU interacts with a malicious microcontroller.

The design of our countermeasure was motivated by the anticipated slight variations in communication properties between different microcontrollers and implementations. While any microcontroller capable of I^{2}C should comply with the standard, the standard does allow some implementational freedom so as to maintain robustness. In chip-in-the-middle scenarios, the CPU communicates directly with the malicious chip and the activities on their

	# of samples	Max	Median	Mean	Std
SDA rise time (μs)	252296	2.8	1.6	1.4	0.5
Inter-frame delay (μs)	12042	26.4	24.4	24.7	0.4
Message length (bytes)	12019	74	3	5.7	10.2
Inter-message delay (μs)	12017	-	288.4	466441	4521765.2

(a) Data collected during phone boot (unmodified phone)

	# of samples	Max	Median	Mean	Std
SDA rise time (μs)	141370	2.4	1.6	1.4	0.5
Inter-frame delay (μs)	6935	25.4	25	24.9	0.3
Message length (bytes)	6926	8	2	4	2.8
Inter-message delay (μs)	6924	-	457.2	19145.5	140717.6

(b) Data collected while using the touchscreen (unmodified phone)

	# of samples	Max	Median	Mean	Std
SDA rise time (μs)	2038	1.8	1.6	1.6	0.2
Inter-frame delay (μs)	114	3494.6	24.2	955.6	1327.4
Message length (bytes)	80	74	3	6.7	12.5
Inter-message delay (μs)	79	-	3548.4	97373.3	595202.5

(c) Data collected during normal boot (chip-in-the-middle scenario)

	# of samples	Max	Median	Mean	Std
SDA rise time (μs)	7815	1.8	1.6	1.6	0.2
Inter-frame delay (μs)	527	2833.8	24.2	240.4	744.3
Message length (bytes)	505	8	2	3.9	2.8
Inter-message delay (μs)	504	-	3397.2	53320.6	322301

(d) Data collected during touch injection (chip-in-the-middle scenario)

Table 5: Statistics of the collected features from an unmodified phone operating under normal conditions (a,b) and from a phone with a malicious chip replacement to the touch screen controller (c,d).

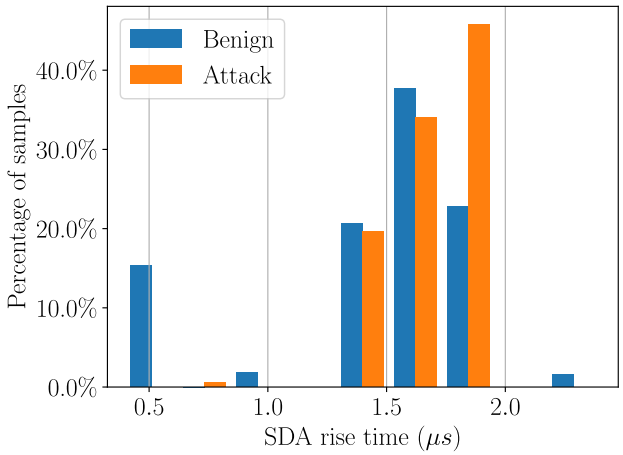

Figure 6: Differences in SDA rise time patterns for benign touch interactions (12294 samples) and chip-in-the-middle touch injections attack (7815 samples) scenarios. The differences originate from architectural differences between the malicious microcontroller and the original touch controller.

communication bus is expected to differ than when the malicious chip is absent.

In order to explore the differences in communication between the original controller and our malicious microcontroller, we connected a Saleae Logic Pro 8 logic analyzer and recorded the communication bus in both scenarios at a sampling rate of 5,000,000 samples per second.

Differences were observed and recorded between the communications in both scenarios. Notably, some of the differences were the result of the micro-architectural difference between the original controller and our micro-controller. The features observed are described below:

- SDA rise time after clock signal - this measures the elapsed time between a clock signal generated by the I^2C master device to the toggle of the SDA signal by the I^2C slave device. Since our bus operates at 400,000 bits-per-second, the SDA rise time can be as high as 2.5μs. Due to the sampling frequency, values slightly above 2.5μs were also recorded. This feature is determined by the micro-architecture of the slave. An example of the difference between this parameter on different micro-controllers can be seen in Fig. 6.

- Inter-frame delay - this measurement shows the elapsed time between consecutive frames in I^2C messages. While this parameter is usually determined by the I^2C master as the master controls the clock and rate of data transfer, the slave may employ clock stretching to delay its response, thus prolonging the inter-frame time. Usage of clock stretching is determined by the micro-architecture of the slave along with the software operating on it.

- Message length - the length of each transmitted message, in bytes.

- Inter-message delay - this measurement shows the

elapsed time between consecutive I²C messages. while the time difference between messages is governed by the I²C master, the master may be driven into sampling the slave at a certain rate by the slave, depending on the protocol.

Table 5 shows gathered statistics of these features. The samples were collected from 150 boot cycles of an unmodified phone along with several hundreds of touch events.

The table show how architectural features, such as the SDA rise time and inter-frame time, do not differ between operation modes while data-related features, such as message length and inter-message time, have small but significant differences.

8.3. Design

input : A sliding window W containing N I²C bytes
output : A classification $\in (benign, malicious)$ of the data
parameter : N - Size of the sliding window
parameter : t - Tolerance for benign classifications
parameter : O_{MSRT} - observed median of SDA rise time
parameter : O_{SSRT} - observed standard deviation of SDA rise time
parameter : O_{MIFD} - observed median of inter-frame delay
parameter : O_{SIFD} - observed standard deviation of inter-frame delay
parameter : O_{MML} - observed maximal message length

$MSRT \leftarrow$ median of SDA rise time in W;
$SSRT \leftarrow$ standard deviation of SDA rise time in W;
$MIFD \leftarrow$ median of inter-frame delay in W;
$SIFD \leftarrow$ standard deviation of inter-frame delay in W;
$MML \leftarrow$ maximal observed message length in W;
if $MML > O_{MML}$ **then**
  return *malicious*;
end
$values \leftarrow \{(MSRT, O_{MSRT}), \quad (SSRT, O_{SSRT}), (MIFD, O_{MIFD}), \quad (SIFD, O_{SIFD}))\}$;
foreach *(val, observed)* $\in$ *values* **do**
  if *val < observed * (1-t) or val > observed * (1+t)* **then**
    return *malicious*;
  end
end
return *benign*;

Algorithm 1: Detection of communications done by a malicious implant using micro-architectural and behavior differences.

Using the collected statistics of the features, a simple statistics-based classifier can be employed to detect anomalous behavior that points to a change in the I²C slave micro-architecture or behavior. The algorithm for such classification can be seen in Algorithm 1.

The parameters required for the calibration and operation of the detection algorithm can be determined and programmed during the production of the whole device, or learned by the countermeasure during a brief learning period. During the collection of data from 150 device boots and several hours of touch interactions, the recorded statistics did not deviate from a baseline established at the beginning of the experiment, suggesting that the statistics are very well constant under time, and that the learning period can be as short as several minutes.

8.4. Evaluation

An effective countermeasure is expected to discriminate between benign and malicious scenarios with a low rate of error. In the case of a hardware countermeasure that is integrated in critical hardware paths, error rates should be negligible.

We indicate an evaluation for such countermeasure to be successful if it provides a zero false-positive malicious identification rate along with safe and explainable margins for classification of new and unknown samples.

8.4.1. Implementation

A separate microcontroller module was attached parallel to the I²C bus and interrupt line of the touch controller. The microcontroller monitors the communication between the touch controller and the host processor, and issues alerts if it detects anomalous behavior.

An STM32L432 microcontroller module was connected to the SDA, SCL, and INTR lines of the touch controller. Patterns and frequencies of I²C communications were recorded under various usage scenarios by decoding the signals according to the I²C protocol. Due to the simple nature of the communication protocol between the host processor and the touch controller, the patterns had very little variance. Additionally, the maximum I²C message length represents the largest normal register read and never exceeds a certain value. Afterwards, the same setup was used for recording the attacks described in this paper.

The recorded data was then divided into two data sets as seen in Table 6. The training set was used to determine the baseline and threshold for detection and the test set was input to a script executing the algorithm described in Algorithm 1 for evaluation using a sliding window methodology.

8.4.2. Selection of detection parameters

The detection algorithm relies on the windows size to maximize TPR and minimize FPR. A window size too small increases noise and results in misdetections while a windows size too large can cause the algorithm to ignore small anomalies. The window size selection was done by analyzing the behavior of the device under normal settings (as described in the training set in Table 6) while inferring the minimal transaction size for an activity. In our case we notice that a full boot sequence involves the transfer

Data set name	Activities recorded	Size in bytes (benign)	Size in bytes (malicious)	Amount of overlapping windows (N=400)	Amount of distinct windows (N=400)
Training set	Phone boot sequence (benign), touch interactions (benign)	134714	0	134315	336
Test set	Phone boot sequence (benign), touch interactions (benign), phone boot with a chip-in-the-middle (malicious), exploitation through phone boot (malicious), touch injection (malicious)	128676	13127	141404	354

Table 6: Detection model evaluation data sets

of 1089 bytes and that a 0.2 seconds simple touch event is indicated by the transfer of 420 bytes. The window size chosen for this experiment was $N = 400$.

In order to select parameters for successful detection of threats, a baseline was established by observing the statistics generated on the training set using the window size previously determined.

Setting a window size of $N = 400$ and a threshold values of $SSRT = 0.22, t = 0.55$ yields an FP rate of $FPR = 0\%$ when evaluating the test set.

Other parameters can be determined similarly by observing statistics gathered from benign data. This method could also possibly be developed to be performed automatically during a learning period of the countermeasure device's life.

8.4.3. Detection of first-order attacks

When evaluated on the test set, the algorithm correctly detected all of the windows involving first-order attacks using the parameters of $N = 400$ and $t = 0.55$. In addition, no benign windows were mis-classified in the test set. Considering the minimal data transaction duration and size determined previously, we conclude that a successful detection of a chip-in-the-middle scenario can be done within 0.2 seconds while maintaining $TPR = 100\%$ and $FPR = 0\%$.

8.4.4. Detection of second-order attacks

This attack was performed using a malicious chip implant that replaced the original touch controller in the same way that was done in the demonstration of first-order attacks. The fact that the micro-architectural differences between the benign and test setup remains lead to the successful classification of the attack using the same parameters from the last Subsubsection.

In addition to micro-architectural based detection, it is worth mentioning that the second-order attack demonstrated in Section 6 injects an abnormally large amount of data using the I^2C bus, this allows for the simple classification done in line 6 of Algorithm 1. Using frame size for detection allows the countermeasure to maintain $FPR = 0\%$ over any windows size, but it relies on the exploitation to require an amount of data that exceeds normal transactions.

8.4.5. Detection of the complete end-to-end attacks

As shown in the previous sections, the countermeasure was capable of detecting both the first-order and second-order stages of this attack and generating an indication or interruption that would have completely stopped the attack.

8.5. Impact on Device Performance

The hardware countermeasure module performs an active collection and evaluation of data during its operation. As such, it requires power to operate.

The manufacturer datasheet of the microcontroller used for evaluating the countermeasure methods in this paper, STM32L432 [37], reports a power usage of 1.05mA when running on 8 MHz 1.8v which translates to 0.0019 Watt. While inactive and in shut-down state, the microcontroller consumes 63nA (1.1^{-7} Watt). Considering a typical smartphone battery capacity in the range of 8-13 Watt-hours [48] and active screen time of 10% on a 48 hour power budget, the microcontroller will be responsible for less than 0.1 percent of the overall power consumption.

As previously mentioned, the countermeasure may be placed in-line within the communication bus and belay messages in a manner that introduces a delay to messages. We observe that the duration of a single byte transmission by a full-speed, 400kbit/s I^2C device is equivalent to 1600 clock cycles of the 80MHz STM32L432 microcontroller. Therefore, we conclude that an efficient countermeasure program executed on a modern microcontroller should be sufficient for avoiding delays larger then the transmission duration of a single byte.

8.6. Limitation

While the protection of the proposed countermeasure against chip-in-the-middle scenarios and chip replacement

scenarios is hard to circumvent, the protection against malicious firmware is not as powerful. A malicious firmware operates on the same microcontroller as a benign one and may possess any of the properties of the latter. We expect that in the case of malicious firmware, data-based anomaly detection can be used to further enhance the countermeasure in a way that will improve detection.

9. Conclusions

The threat of a malicious peripheral existing inside consumer electronics should not be taken lightly. The conversation around the alleged hardware implants raises a serious concern about if and where these have been embedded, and what damage are they doing. As this paper shows, attacks by malicious implants and peripherals are feasible, scalable, and invisible to most detection techniques. A well-motivated adversary may be fully capable of mounting such attacks on a large-scale or against specific targets. System designers should consider replacement components to be *outside* the phone's trust boundary, and design their defenses accordingly.

Conservative estimates assume that there are about two billion smartphones in circulation today. Assuming that 20% of these smartphones have undergone screen replacement [1], there are on the order of 400 million smartphones with replacement screens in the world. An attack which compromises even a small fraction of these smartphones through malicious components will be comparable to that of the largest PC-based botnets.

The countermeasure described in our work was shown to be effective against the proposed attacks. As a low-cost and minimally disruptive solution, the hardware countermeasure can be further developed to protect against threats similar to those presented in this paper, as well as other threats that may be introduced to the system such as active fault attacks.

Acknowledgments

This research was supported by Israel Science Foundation grants 702/16 and 703/16.

References

[1] Motorola Mobility. Cracked screens and broken hearts - the 2015 motorola global shattered screen survey. `https://community.motorola.com/blog/cracked-screens-and-broken-hearts`.

[2] Defense Information Systems Agency. *The Department of Denfense Approved Products List.* `https://aplits.disa.mil/processAPList`.

[3] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *2012 IEEE Symposium on Security and Privacy*, pages 95–109. IEEE, 2012.

[4] Jeffrey Bickford, Ryan O'Hare, Arati Baliga, Vinod Ganapathy, and Liviu Iftode. Rootkits on smart phones: attacks, implications and opportunities. In *Proceedings of the eleventh workshop on mobile computing systems & applications*, pages 49–54. ACM, 2010.

[5] Seyyedeh Atefeh Musavi and Mehdi Kharrazi. Back to static analysis for kernel-level rootkit detection. *IEEE Transactions on Information Forensics and Security*, 9(9):1465–1476, 2014.

[6] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers: Where the Kernel Meets the Hardware.* " O'Reilly Media, Inc.", 2005.

[7] Janis Danisevskis, Marta Piekarska, and Jean-Pierre Seifert. Dark side of the shader: Mobile gpu-aided malware delivery. In *International Conference on Information Security and Cryptology*, pages 483–495. Springer, 2013.

[8] Evangelos Ladakis, Lazaros Koromilas, Giorgos Vasiliadis, Michalis Polychronakis, and Sotiris Ioannidis. You can type, but you can't hide: A stealthy gpu-based keylogger. In *Proceedings of the 6th European Workshop on System Security (EuroSec)*, 2013.

[9] Matthew Brocker and Stephen Checkoway. iseeyou: Disabling the macbook webcam indicator led. In *USENIX Security*, pages 337–352, 2014.

[10] Zhaohui Wang and Angelos Stavrou. Exploiting smart-phone usb connectivity for fun and profit. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 357–366. ACM, 2010.

[11] Qing Yang, Paolo Gasti, Gang Zhou, Aydin Farajidavar, and Kiran S Balagani. On inferring browsing activity on smartphones via usb power analysis side-channel. *IEEE Transactions on Information Forensics and Security*, 12(5):1056–1066, 2017.

[12] Apple Inc. *Error 53 support page.* `https://support.apple.com/en-il/HT205628`.

[13] National Institute of Standards and Technology. *Cybersecurity Framework v1.1 - Draft.* `https://www.nist.gov/cyberframework/draft-version-11`.

[14] Inez Miyamoto, Thomas H Holzer, and Shahryar Sarkani. Why a counterfeit risk avoidance strategy fails. *Computers & Security*, 66:81–96, 2017.

[15] Xiaoyong Zhou, Yeonjoon Lee, Nan Zhang, Muhammad Naveed, and XiaoFeng Wang. The peril of fragmentation: Security hazards in android device driver customizations. In *2014 IEEE Symposium on Security and Privacy*, pages 409–423. IEEE, 2014.

[16] Omer Shwartz, Guy Shitrit, Asaf Shabtai, and Yossi Oren. From smashed screens to smashed stacks: Attacking mobile phones using malicious aftermarket parts. In *Workshop on Security for Embedded and Mobile Systems (SEMS 2017)*, 2017.

[17] Xing Liu, Jiqiang Liu, Sencun Zhu, Wei Wang, and Xiangliang Zhang. Privacy risk analysis and mitigation of analytics libraries in the android ecosystem. *IEEE Transactions on Mobile Computing*, 2019.

[18] Yongzhong He, Xuejun Yang, Binghui Hu, and Wei Wang. Dynamic privacy leakage analysis of android third-party libraries. *Journal of Information Security and Applications*, 46:259–270, 2019.

[19] Jacob Leon Kröger and Philip Raschke. Is my phone listening in? on the feasibility and detectability of mobile eavesdropping. In *IFIP Annual Conference on Data and Applications Security and Privacy*, pages 102–120. Springer, 2019.

[20] Bloomberg Businessweek. The big hack: How china used a tiny chip to infiltrate u.s. companies. *bloomberg.com*, 2018.

[21] Eric Wemple. Your move, bloomberg. *washingtonpost.com*, 2018.

[22] Duncan Riley. Apple, amazon and super micro call on bloomberg to retract china spy chip story. *siliconangle.com*, 2018.

[23] Supermicro. Supermicro historic stock price. *supermicro.com*, 2019.

[24] Theo Markettos. Making sense of the supermicro motherboard attack. *lightbluetouchpaper.org*, 2018.

[25] Trammell Hudson. Modchips of the state. 2018.

[26] NCC Group Plc. *TPM Genie.* `https://github.com/nccgroup/TPMGenie`.

[27] Google. *Android Security Bulletin.*

[28] UL. Counterfeit iphone adapters.

[29] Amit Chowdhry. Apple: Nearly 90% of 'genuine' iphone chargers on amazon are counterfeit. *Forbes.com*, 2016.

[30] Omer Shwartz, Amir Cohen, Asaf Shabtai, and Yossi Oren.

Shattered trust: when replacement smartphone components attack. In *11th {USENIX} Workshop on Offensive Technologies ({WOOT} 17)*, 2017.

[31] Shakeel Butt, Vinod Ganapathy, Michael M Swift, and Chih-Cheng Chang. Protecting commodity operating system kernels from vulnerable device drivers. In *Computer Security Applications Conference, 2009. ACSAC'09. Annual*, pages 301–310. IEEE, 2009.

[32] CC Okolie, FA Oladeji, BC Benjamin, HA Alakiri, and O Olisa. Penetration testing for android smartphones. 2013.

[33] Mordechai Guri, Yuri Poliak, Bracha Shapira, and Yuval Elovici. Joker: Trusted detection of kernel rootkits in android devices via jtag interface. In *Trustcom/BigDataSE/ISPA, 2015 IEEE*, volume 1, pages 65–73. IEEE, 2015.

[34] Guillermo Suarez-Tangil, Juan E Tapiador, Pedro Peris-Lopez, and Arturo Ribagorda. Evolution, detection and analysis of malware for smart devices. *IEEE Communications Surveys & Tutorials*, 16(2):961–987, 2014.

[35] Nir Nissim, Ran Yahalom, and Yuval Elovici. Usb-based attacks. *Computers & Security*, 70:675–688, 2017.

[36] NXP. *I2C-bus specification and user manual*, April 2014. `http://www.nxp.com/documents/user_manual/UM10204.pdf`.

[37] STMICROELECTRONICS. *STM32L432 Datasheet*, May 2018. `https://www.st.com/resource/en/datasheet/stm321432kc.pdf`.

[38] Arduino. *Arduino Home Page*. `https://www.arduino.cc`.

[39] Atmel Corporation. *ATmega Datasheet*, 2018. `https://ww1.microchip.com/downloads/en/DeviceDoc/ATmega48A-PA-88A-PA-168A-PA-328-P-DS-DS40002061A.pdf`.

[40] Cypress. *PSoC 4000 Family Datasheet*, November 2017. `http://www.cypress.com/file/138646/download`.

[41] Mario Frank, Ralf Biedert, Eugene Ma, Ivan Martinovic, and Dawn Song. Touchalytics: On the applicability of touchscreen input as a behavioral biometric for continuous authentication. *IEEE Trans. Information Forensics and Security*, 8(1):136–148, 2013.

[42] Julian Fierrez, Ada Pozo, Marcos Martinez-Diaz, Javier Galbally, and Aythami Morales. Benchmarking touchscreen biometrics for mobile authentication. *IEEE Transactions on Information Forensics and Security*, 13(11):2720–2733, 2018.

[43] Chao Shen, Yong Zhang, Xiaohong Guan, and Roy A Maxion. Performance analysis of touch-interaction behavior for active smartphone authentication. *IEEE Transactions on Information Forensics and Security*, 11(3):498–513, 2016.

[44] Moran Azran, Niv Ben Shabat, Tal Shkolnik, and Yossi Oren. Brief announcement: Deriving context for touch events. In *International Symposium on Cyber Security Cryptography and Machine Learning*, pages 283–286. Springer, 2018.

[45] Ralf Hund, Thorsten Holz, and Felix C Freiling. Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In *USENIX Security Symposium*, pages 383–398, 2009.

[46] Stephen Smalley, Chris Vance, and Wayne Salamon. Implementing selinux as a linux security module. *NAI Labs Report*, 1(43):139, 2001.

[47] Kyong-Tak Cho and Kang G Shin. Fingerprinting electronic control units for vehicle intrusion detection. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*, pages 911–927, 2016.

[48] Kyung Mo Kim, Yeong Shin Jeong, and In Cheol Bang. Thermal analysis of lithium ion battery-equipped smartphone explosions. *Engineering Science and Technology, an International Journal*, 22(2):610–617, 2019.