Paranoid Android: Zero-Day Protection for Smartphones Using the Cloud

Georgios Portokalidis *, Philip Homburg *, Kostas Anagnostakis †, Herbert Bos *

*Vrije Universiteit Amsterdam
Amsterdam, The Netherlands
porto@few.vu.nl, philip, herbertb@cs.vu.nl

†Niometrics R&D
Singapore
kostas@niometrics.com

Abstract

Smartphones have come to resemble PCs in software complexity. Moreover, as they are often used for privacy-sensitive tasks, they are becoming attractive targets for attackers. Unfortunately, they are quite different from PCs in terms of resources, so that PC-oriented security solutions are not always applicable. Worse, common security solutions (such as on-access file scanners, system call profilers, etc.) protect against a very limited set of attacks. Comprehensive measures require a far wider and more expensive set of checks - some of which are much beyond the capacity of a phone.

We propose an alternative solution, where security checks are applied on remote *security servers* which host exact *replicas* of the phones in virtual environments. The servers are not subject to the same constraints, allowing us to apply *multiple* detection techniques *simultaneously* (including ones that are very heavy-weight). Moreover, as the full execution trace is preserved on the security server, attackers *cannot hide their traces*. This property ensures that attacks for which a detection method exists at the security server - even if the detection method is installed at a later time - are detectable eventually. This is a stronger guarantee than existing solutions can give. It allows administrators to trade server resources for security by specifying how far back in time they want to be able to start looking for intrusions with a new detection method. We implemented the security model for Android phones and show that it is both practical and scalable: we generate about 2KiB/s and 64B/s of trace data under high-load and idle operation respectively, and are able to support more than a hundred replicas on a single server.

1   Introduction

Smartphones have come to resemble general-purpose computers: in addition to traditional telephony stacks, calendars, games and address books, we use them for browsing, reading email, watching videos, and many other activities that we used to perform on PCs. A plethora of new applications, such as navigation and location sensitive information services, are becoming increasingly popular. As software complexity increases, so does the number of bugs and exploitable vulnerabilities [20, 36, 24, 35]. Vulnerabilities in the past have allowed attackers to exploit bugs in Bluetooth implementations to take over various mobile phones, such as the Nokia 6310, the Sony Ericsson T68, and the Motorola v80. More recently, Apple's iPhone and Google's Android platform have also shown to be susceptible to remote exploits [32, 27, 29].

Moreover, as phones are used more and more for privacy sensitive and commercial transactions, there is a growing incentive for attackers to target them. For instance, credit card numbers and passwords are entered in phone-based browsers, and many companies (including Apple, Google, and Microsoft) operate online stores selling software, music and videos. Phone-based payment for physical goods, services, mass

transit, and parking is also provided by various companies like Upaid Systems, Black Lab Mobile, Verrus Mobile Technologies, RingGo, and HKL/HST in Finland.

While many home users consider security of secondary importance, this is not the case for senior officials in industry, government, law enforcement, banks, health care, and the military[1]. In this paper, we address the problem of security for smartphones for organisations and individuals that care deeply about the detection of attacks. Administrators in such organisations want to find out that a phone is compromised as soon as possible. Our goal is to prevent situations where a device has been compromised for a long time without anybody noticing. To achieve this, we will accept a short delay between the time of attack and the time of detection, but the delay should be as short as possible. Assuming a detection method exists for the attack, we should know about the compromise in the order of minutes, rather than weeks or months, as is currently the case for popular antivirus (AV) products [33]. Additionally, we want to detect a wide range of attacks, including zero-days. To achieve this, we want to employ as many different detection techniques as possible, be they AV scanners, anomaly detectors, methods based on taint analysis, or others.

Rather than doing all of this on already severely resource-constrained smartphones, we propose a different security model that completely devolves attack detection from the phone. At a high level, we envision that security (in terms of attack detection) will be just another service hosted in a separate server in the Cloud, much like storage is hosted in a file server, email in a mail server, and so on. Whether this is feasible at the granularity needed for thwarting today's attacks has been an open research question, which we attempt to answer in this paper.

More specifically, we run a synchronised replica of the phone on a dedicated *security server*. As the server does not have the tight resource constraints of a phone, we can now perform security checks that would be too expensive to run on the phone itself. To achieve this, we record a minimal trace of the phone's execution (enough to permit replaying and no more) which we then transmit to the server, as illustrated in Figure 1. The implementation of our security model is known as *Paranoid Android* (*PA*).

Previous work on decoupling security checks from production PC systems, makes use of tailored virtual machines (VMs), and assumes ample and cheap communication bandwidth. In contrast, no such VMs are available on smartphones, and communication bandwidth is more limited and expensive in terms of battery consumption.

This paper makes the following contributions.

1. We fully implemented devolved attack detection for Android G1 phones and show that it is practical and capable of detecting zero-day attacks.

2. The architecture scales in detection methods: we can run many detection methods in parallel (including expensive ones), with *zero* overhead on the phone for each additional method.

3. We protect user data by means of transparent backup.

4. We ensure that attackers cannot hide their traces.

We also show that a single server can support many phones. The paper is organised as follows:

---

[1]A famous case in point was US president Obama's 2008 struggle to keep his Blackberry phone after being told this was not possible due to security concerns. Eventually, he was allowed to keep an extra-secure smartphone.

2 Threat and Security Model

In this section, we discuss the context of our work, its threat model, and assumptions, as well as the proposed security model and its properties.

2.1 Threat Model

We assume that all software on the phone, including the kernel, can be taken over completely by attackers. However, we assume that a compromise of the kernel takes place via a compromised userspace process (as is usually the case). We do not care about the attack vector itself. We expect that attackers will be able to compromise the applications on the phone by means of a variety of exploits (including buffer overflows, format string attacks, double frees, integer overflows, etc.). Nor do we care about the medium: attacks may arrive over WIFI, 3G, Bluetooth, infrared, or USB. In the absence of exploits, an attacker may also persuade users to install malicious software themselves by means of social engineering. Typical examples include trojans disguised as useful software.

In contrast, we assume that the security server is safe or can be hardened sufficiently to make compromises unlikely. While this is a strong assumption, the model is used by many Cloud services (e.g., GMail), and also by automated update services in Windows, Ubuntu, etc. We also assume that the security server can be trusted to not disclose user private data (as millions of users do trust GMail, Hotmail, and other services to safeguard the confidentiality of their email messages).

2.2 What is wrong with existing solutions?

With both opportunity *and* incentive for attacking smartphones so clearly on the rise, what about protective measures? One might think that this is a familiar problem: if phones are like computers, we should simply copy existing solutions from the PC and server domain. There are two problems with this way of reasoning: (1) existing measures can often not be applied on phones due to resource constraints, and (2) even if they can, they are not sufficient.

While smartphones are like small PCs in terms of processing capacity, range of applications, and vulnerability to attacks, there are differences in other respects, most notably power and physical location. These two aspects matter when it comes to security. Unlike normal PCs, smartphones run on battery power - an extremely scarce resource. For instance, one of the main points of criticism against Apple's iPhone 3G concerned its short battery life [28]. Vendors work hard to produce efficient code for such devices, because every cycle consumes power, and every Joule is precious.

Intrusion detection techniques all consume processing power (and thus battery time). As a consequence, many security solutions that work for desktop PCs may not be portable to smartphones. AV workloads can be quite expensive [44], certainly if we consider the limited processing and memory resources available on phones [2]. And AV scans are still cheap compared to other, really expensive solutions like taint analysis which requires modified emulators and typically incurs slowdowns of orders of magnitude [31]. Such overheads place taint analysis and similarly expensive solutions well beyond the capabilities of smartphones.

Some solutions could be applied in principle, but not in practice. For instance, it is theoretically possible to write all software on the phone in a language immune to memory corruption attacks. However, due to market pressure, performance, and the need for code reuse, most (if not all) vendors have opted for unsafe languages for all core system services (including the browser, media players, etc.). If we have access to source code, we may still apply novel compiler extensions like write integrity testing (WIT) to transform unsafe code into hardened binaries that are robust against memory corruption [2]. Unfortunately, no such

---

[2]A typical smartphone runs at about 600MHz and contains 256MB of RAM, while a typical PC features a multicore CPU running at over 2GHz, and more than 2GB of RAM.

solutions are available for current phones. While it is possible that such toolchains become available in the future, we would rather not wait. More importantly, we will see that memory corruption is by no means the only problem.

In addition to resource constraints, phones differ from PCs by operating in varying and often hostile environments. Unlike traditional computers, they go everywhere we go, and attacks may come from sources that are extremely local. A person with a laptop or another smartphone in the same room could be the source of a Bluetooth or WiFi spoofing attack [3]. That means that traditional perimeter security in general is insufficient, and mobile phone security solutions that are based on "upstream" scanning of network traffic [8] will never even see the traffic involved in attacking the phone.

Most importantly, however, none of the existing solutions provides sufficient protection by itself. Safe languages, and compiler extensions protect against memory corruption, but do not detect Trojans installed by the user. AV software may detect Trojans, but only after the company has generated a signature for it. By that time, the attacker may have removed all traces of the attack, or even disabled the AV scanner. Moreover, AV programs typically do not detect memory-resident attacks at all. System call anomaly detection [38] is fairly cheap, but vulnerable to mimicry attacks [19].

High-grade security demands the application of a host of detection techniques on the phone. However, doing so exacerbates the power problem and may even incur unacceptable slowdowns. Since running even a single detection method (AV scanning) can be expensive, running many different detection mechanisms simultaneously does not seem a viable solution. As battery life sells phones, and consumers hate recharging [28], the likely result is that both vendors and consumers will trade security for battery life.

Worse, if we happen to miss an attack (e.g., because the AV database did not yet have a signature at the time of attack), the attacker has an opportunity to hide her traces. Missing attacks is quite likely for current AV solutions. Experiments show that the detection rate of 10 popular AV products ranges between 39% and 79%, even after a week has elapsed from the moment new malware was identified for the first time. [33]. Even after a year, the detection rate varies between 65% and 94%. Existing malware like 'Fantibag', 'MS Antivirus', and various other rootkits show that by the time a signature is finally added to the AV product, the attack may have disabled the AV product, removed the tell-tale files, or changed system programs and system calls to remain undetected.

2.3 Security Model

In the security model of *PA*, we push all security checks to a replica of the phone running on a dedicated security server in the Cloud. To achieve this, we record a minimal trace of the phone's execution and transmit this to the server (Figure 1). The server replays the execution and applies all attack detection methods.

Let $\mathbb{D} = \{d_0, ..., d_n\}$ be the set of all attack detection mechanisms that can be used in a system. $\mathbb{D}$ may change over time as new detection methods are added and old ones are removed. Each detection method d_i is capable of detecting a specific set of threats by means of data and events collected at the device. The events collected by all detection methods in $\mathbb{D}$ collectively will be referred to as the *trace*. The threat model T is defined as the set of all attacks described in Section 2.1, a subset T_D of which will be detectable by $\mathbb{D}$. In the ideal case, $T = T_D$. The proposed security model provides four desirable security properties, that we will now discuss in detail.

2.3.1 Attack visibility

We say that a security model guarantees attack visibility if it prevents attackers from hiding their traces. For attack visibility, it suffices to ensure 2 properties for all attacks in T:

- *Guaranteed tamper-evident attack trace preservation*: attackers cannot hide their traces to escape detection and traces remain available for a long time (potentially forever).

4

- *Guaranteed check execution*: attackers cannot disable the security checks.

Together, these properties ensure attack visibility. That is, if at any point after an attack, $\mathbb{D}$ contains a method d capable of detecting such attacks, we *will* be able to detect it, even if d was added after the attack.

PA guarantees a high-degree of attack visibility. The phone cannot disable detection techniques running on the server, and after trace data have been transmitted, they are safe. Moreover, *PA* uses tamper-evident storage to prevent attackers from modifying trace data that the phone has not yet transmitted (Section 3.2.2). The worst an attacker can do is to delete entries or block communication to the server. However, if entries are missing, the security server will notice this and conclude that the phone is compromised. If the phone does not communicate with the server, the server notifies the user out-of-band.

2.3.2   Scalability in detection techniques

A security model is scalable in the set of detection techniques $\mathbb{D}$, if the framework itself has limited overhead, and each additional attack detection method incurs zero overhead on the phone.

We will see that the overhead of *PA* in terms of trace size, CPU overhead, and battery consumption is sufficiently small to allow the system to be used on real phones. New attack detection methods are introduced only at the security server and do not interfere with the phone at all. To validate our approach, we implemented several real detection techniques including traditional AV scanning (based on ClamAV), and heavy-weight detection of memory corruption by means of taint-analysis.

2.3.3   Flexibility in detection techniques

We say that a security model is flexible in $\mathbb{D}$ if we can apply any detection method. The use of many detection methods is essential if we want T_D to approach T. Flexibility requires that the trace contains sufficient information to satisfy many detection methods. The flexibility in *PA* is high: since we replay the execution of the protected code exactly as it occurred on the phone, we can apply any method that does not interfere with the execution of the protected processes.

This consists a significant advantage over previous work in the field, which has focused on applying a single detection methodology, be it AV scanning [33] or behavioural analysis [8].

2.3.4   Protection of user data

We say that user data is protected if such data can be restored to a safe state regardless of attacks, misconfiguration, loss, or physical damage to the phone.

PA achieves partial protection of user data. Since the full execution trace is available on the security server, we can restore user data to any point in time. Since a small portion of the trace may still be on the phone, a small amount of data loss may still occur. *PA*'s model for data protection is similar to periodic backup systems.

2.4   Context

Our approach is consistent with the current trend to host activities in centralised servers (e..g, in the *Cloud*), including security-related functions. Oberheide *et al.* have explored AV file scanning in the Cloud with [33] and [34]. As file scans are not able to detect zero-days, remote exploits, or memory-resident attacks (all of which have targeted mobile phones in the past [24, 17, 35, 29]), we take a more aggressive approach and aim to prevent attacks on the phone software itself (e.g., exploits against client applications like the browser, media player, email program, and calendar), while also covering misbehaving code installed by the user
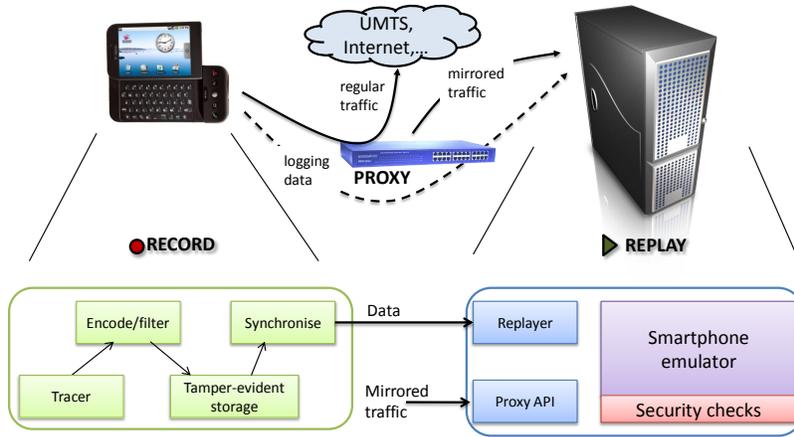
Figure 1: *Paranoid Android* architecture overview

Our solution builds on work on virtual machine (VM) recording and replaying [15, 47, 30, 9, 16, 23, 39, 40, 26]. *Rather than recording and replaying at the VM level, we record the trace of a set of processes (running everything in a VM on the phone is not realistic on any current phone). In addition, we tailor the solution to smartphones, and compress and transmit the trace in a way that minimises computational and battery overhead.* We consider this an important auxiliary contribution of this paper as it opens up a new application domain for replaying: resource-constrained devices that cannot provide comprehensive security measures themselves.

2.5 Assumptions

The *PA* security model is based on the following assumptions:

1. The server will not be compromised. This assumption also applies in services like GMail and Windows updates.

2. Attackers cannot break the encryption.

3. The device is able to contact the server safely, to create an initial replica, and setup the tracer.

4. The servers have out-of-band channels to notify users about problems and a way to restore the image.

The last assumption is strong, but essential. If a security server detects an attack, it needs to tell the user. As the phone may be under the control of the attacker, this is not a trivial matter. Some phone manufacturers (like Blackberry) add a switch kill to the phones that can be triggered remotely and forces the phone to boot with a safe recovery image. Our own notification and restoration measures are quite simple. First, we see if we can still reach the phone to notify the user and load a clean image with the latest user data. If that fails, we notify the users by means of an alert service running on their normal computers. Moreover, even if the phone is under the attacker's control, we are still able to flash the phone's recovery image by connecting it to a PC via USB, or plugging the phone's SD card into a laptop.

3   Architecture

A high-level overview of *PA*'s architecture is illustrated in Figure 1. In this section we discuss the basic idea, and sketch the various components of the system. On the phone, a tracer records all the information

needed to accurately replay the execution. The recorded *execution trace* is transmitted to the cloud over an encrypted SSL channel, where a replica of the phone is running on an emulator. On the cloud, a replayer receives the trace and faithfully replays the execution within the emulator. The system can apply security checks externally, as well as from within the emulator, as long as they do not interfere with the replayed applications (i.e., do not perform IPC with replayed processes, modify user files, etc). Provided we observe this rule of non-interference, we may even run additional processes or instrument the kernel.

We use a network proxy that temporarily stores traffic destined to the phone, to reduce the amount of data the tracer needs to store and transmit to the replica. The replayer can later access the proxy to retrieve the data needed for the replaying.

3.1 Recording and Replaying Execution

Recording and replaying a set of processes or even an entire system has been broadly investigated by previous work [15,47,30,9,16,23,39,40,26]. In this section we will only briefly discuss how execution replaying is performed, while implementation details and various optimisations are discussed in Section 4.1. Readers interested in recording and replaying in general are referred to the above cited papers.

A computer program is by nature deterministic, but it receives non-deterministic inputs and events that alter its execution flow. To replay any given execution, we need to record all the non-deterministic inputs and events that drive a program's flow. Such inputs come from the underlying hardware (e.g., time comes from the HW clock, network data from the WiFi adaptor, location from the GPS sensor, etc). From a process's perspective they are received mostly through system calls to the kernel. Thus, to replay execution the tracer records the data transferred from the kernel to userspace through system calls. The replayer then uses these recorded values when replaying system calls at the replica.

In addition to inputs, the operating system (OS) may deliver signals to a process. Signals are events that alter a process' control flow, and they can be synchronous or asynchronous. Synchronous signals are delivered when a serious error has occurred, usually by the instruction currently executing (e.g., a segmentation fault, or a floating point exception). Simply logging the delivery of such a signal is sufficient, as it will also be generated at the same instruction during replay. On the other hand, asynchronous signals can be delivered arbitrarily, and in fact most OSs (except real-time ones) do not even guarantee their delivery. To ensure that such signals are delivered at exactly the same time during replay, the tracer defers their delivery until the target process performs a system call.

Concurrency and inter-process communication (IPC) can also be a source of non-determinism. Two processes can exchange data using various mechanisms such as pipes, message queues, files, sockets, shared memory, and memory mapped files. Most of these mechanisms are implemented using system calls to send and receive data. For system call based IPC, it is sufficient to record the call and its result..

This is not the case for shared memory and memory mapped files, since a process can access them directly. When two processes use such objects to communicate, they may affect one another in unpredictable ways, producing non-reproducible behaviour. In the case of threads, almost all process memory is shared. A well-known solution is to adopt a concurrent-read-exclusive-write (CREW) protocol [12] that ensures a valid serialisation of all accesses to the shared memory object [23]. An alternative approach, valid on uniprocessor systems only, involves modifying the task scheduler of the OS to provide repeatable scheduling [40]. We have adopted the latter approach for *PA*.

3.2 Synchronisation

Smartphone users today enjoy plentiful wireless connectivity over 3G, WiFi, GPRS, etc. Following services like email and calendars, *PA* aims to take advantage of the available connectivity to offer security and data back-up services. However, wireless connectivity is costly in terms of energy consumption. Mobile device

vendors spend a great deal of effort getting the power management on their devices right, e.g., to ensure that the wireless adaptors do not stay turned on longer than needed.

3.2.1 Loose synchronisation

PA adopts a loose synchronisation strategy to minimise its effects on battery life. Particularly, it does not activate or keep any of the network adaptors from sleeping, but rather synchronises only when a connection is already available. That can be due to the user accessing the web, downloading email, etc. An extremely loose synchronisation model could even allow for synchronisation to be performed only when the phone recharges. Such a model may be suitable for users with more relaxed security requirements, where the user detects attacks only at the end of the day, but at least the attacks would not be lingering undiscovered on the phone for weeks.

3.2.2 Tamper-evident secure storage

Loose synchronisation with the server is ideal for preserving power, but if we wait too long, an attacker may compromise the phone and disable the synchronisation procedure. Even worse, the attacker could modify the execution trace to remove the entries that expose the attack (e.g., a specific read from the network), while keeping the system operational to make it appear as if everything is still running properly. Since the device may be compromised via Bluetooth or infrared while disconnected from the Internet, this is a viable threat that we cannot afford to ignore.

We protect against the latter scenario, by adopting a secure storage to hold the execution trace, when disconnected from the server [41]. Every block of data written to secure storage is associated with an HMAC code [6], that simultaneously verifies the block's authenticity and integrity. HMAC is a specific type of message authentication code (MAC) that involves a cryptographic hash function in combination with a secret key. Tamper-evident secure storage is based on 'key rolling'. After generating the HMAC for an entry written to secure storage, we generate a new key by applying a second cryptographic hash function on the old key (that is completely overwritten). This way an attacker compromising the device, cannot alter older entries in the execution trace to hide the attack. At worst, attackers can delete entries or block synchronisation, which both count as synchronisation errors and are described below.

$$STORE(message + HMAC(key, message))$$
$$key' = HASH(key)$$
$$key = key'$$

This method is more lightweight than digital signatures, demanding less processing cycles and less storage. It only requires that a secret key is initially shared between the device and the server. Such a key can be established when setting up the device for use with *PA*. The replayer can in turn authenticate the data by calculating an HMAC code for it, and comparing that with the one received by the tracer.

3.2.3 Synchronisation errors

An error during synchronisation can be the result of a software bug, or a failed attempt by an attacker to cover his tracks. It can manifest itself as a mismatch in the HMAC code, a corrupted execution trace, or failure to communicate for a long period of time. The true cause of such an error cannot be determined by the security server, and in any case we lose the ability to replay execution. Consequently, devices exhibiting such errors need to be treated as potentially compromised, and the user needs to be notified.

3.3 Attack Detection

The real power of *PA* lies in the scalability and flexibility in detection methods. By replicating smartphone execution in the cloud, we have ample resources for running a combination of security checks. Moreover, we can apply any detection method that obeys the rule of non-interference (including methods that add processes or instrument the kernel). For instance, all of the following detection methods are compatible with *PA*'s security model. As a proof of concept, we implemented the first two in the list (Section 4.4) and are currently working on the others.

1. Dynamic analysis in the emulator. We instrument the emulator to perform runtime analysis to detect certain types of zero-day attacks such as buffer-overflows and code-injection attacks [22, 46, 13, 11].

2. AV products in the emulators. We modified a popular open source AV to run in the emulator. This way, we are able to run periodical file scans out of the box. Additionally, on access file scanning can be applied with few modifications to the scanner. On access scanning AV intercept file handling system calls and scan the target file before allowing a process to access it. As the replayer already intercepts system calls, we only have to use this information to perform a scan before allowing replaying to proceed.

3. Memory scanners. We can scan emulator memory for patterns of malicious code directly. Memory scanners are able to detect memory-resident attacks that leave no files behind for AV scanners to detect.

4. System call anomaly detection. Detection methods based solely on the system calls [38, 18], can even be applied directly to the execution trace, without any need for replaying. As a result, system call detection methods are extremely fast.

While, all the techniques we have referred to in this section have been around for some time, execution replay offers great flexibility, even enabling future runtime security solutions to be applied retrospectively.

Finally, *PA*'s security model makes it possible to exploit all available cores on the security servers, by running multiple replicas and detection techniques in parallel. The number of security checks that can be applied in parallel is only limited by the amount of resources one is willing to devote for security.

3.4 Server Control and Location

Where to host the security server and who controls it is a policy decision beyond the scope of this paper. For instance, the security service may be offered by the provider who uses it to differentiate itself from other providers and to generate income by charging for the service. Privacy must be ensured, but many companies already rely on the Cloud to store their private data and in that sense the model does not change much. If traffic can be routed through an organisation's own servers (possible hosted in the Cloud), we could also keep the server under the organisation's control. As an extreme case, users with strong privacy considerations could run the replicas on their own PCs. Doing so gives them full control over their data, but implies a very loose synchronisation model. In this case, users would most likely synchronise only when connecting their phones to their PCs.

4 Implementation

In this section, we discuss the *PA* prototype implementation. *PA* runs on the HTC G1 Android developer phone. G1 is the first smartphone featuring Google's Android OS. It supports 3G, WiFi, and BlueTooth, and

comes with a host of pre-installed applications, such as a browser, mail client, media players, etc. Thousands more applications are available by download from Google's online marketplace.

At the security server, replicas run on the Qemu-based Android emulator [5], which is part of the official SDK. While emulation incurs a significant overhead, we will see later that this is more than compensated for by the more powerful resources on the security servers. In addition, we benefit from the fact that we often do not need to block or sleep on the security server, because the results of system calls are directly available in the execution trace.

It is possible to implement the tracer in different ways. The most efficient way is to intercept system calls and signals in the kernel. It is also the most convenient way to influence the scheduling. However, it is not very portable. As we target our solution at other smartphones also (notably the iPhone), and for many phones source code is not available, portability is important. For this reason, our first prototype is fully implemented in user space.

PA only requires tracing capabilities, comparable to Linux's *ptrace* system call. Ptrace allows one to attach to arbitrary processes, and intercept system calls and signals. Adopting such minimal requirements should allow us to port *PA* to other architectures supporting *ptrace*-like interfaces (Linux, BSD, and Windows support such interfaces for debugging purposes). Furthermore, it also allows us to port *PA* on other Linux based phones with minimal effort.

Using ptrace we are able to track a system's processes, and receive event notifications each time they interact with the kernel. Events received include system call entry and exit, creation and termination of threads, signal delivery, etc.

4.1 Recording and Replaying Android

In this section, we explain how we implemented recording and replaying of execution for Android phones.

4.1.1 Genesis - starting the tracer and everything else

In UNIX tradition, Android uses the `init` process to start all other processes, including the supporting framework, client applications, and the JVM. The tracer itself is also launched by init, before launching any of the processes we wish to trace. Init launches the processes that are to be traced using an execution stub. This process serves a twofold purpose: it allows the tracer to start tracing the target processes from the first instruction, and it enables us to run processes without tracing them (e.g., debugging and monitoring applications).

`Init` in *PA* brings up the tracer process first. The tracer initialises a FIFO to allow processes that need tracing to contact it. Next, `init` starts the other processes. Rather than starting them directly, we add a level of indirection, which we call the *exec stub*. So, instead of forking a new thread and using the `exec` system call directly to start the new binary, we fork and run a short stub. The stub writes its process identifier (pid) to the tracer's FIFO (effectively requesting the tracer to trace it) and then pauses. Upon reading the pid, the tracer attaches to the process to trace it. Finally, the tracer removes the pause in the traced process, making the stub resume execution. The stub immediately `execs` to start the appropriate binary with the corresponding parameters.

4.1.2 Scheduling and shared memory

In Section 3.1, we briefly mentioned that we serialise accesses to shared objects using a modified task scheduler that operates in a deterministic way. Modifying the scheduler requires that we modify the kernel. Again, since our goal with this prototype was to keep all implementation in userspace, we also used *ptrace* to control scheduling.

Unfortunately, we can only do so with a coarse granularity, as we can only pause a running thread when it is entering a system call. Our scheduling algorithm is quite simple and far from optimal, but sufficient for our purpose, as it is reproducible. Also, it does not require logging of any additional information in the execution trace. In essence, it makes sure that no two threads that share a memory object can ever run concurrently. Because the scheduler is triggered by system calls, it can be unfair, and it may theoretically deadlock in the presence of spinlocks. To avoid the latter, we created a spinlock detector that is activated when a task keeps running for more than a predefined period of time. In practice, however, Android does not use spinlocks as they are wasteful in terms of CPU cycles. Instead, locking is performed with mutexes, which perform a system call in case of contention, and are handled by *PA* in a straightforward way. While the spinlock detector provides the robustness that is required for a production system, so far we have only seen it triggered for contrived test cases.

Modern operating systems also allow processes to directly memory map HW memory. If such memory was to be used for directly reading data from hardware, neither repeatable scheduling nor a CREW [12] protocol could ensure proper serialisation of accesses to that memory. To the best of our knowledge, Android does not use memory in this way. However, it could be a problem in the future in a different hardware/software combination. In that case, we need to record all reads from such memory to keep execution deterministic. This can be accomplished by making the area inaccessible to the reader, and intercepting all read attempts using the generated page faults. Doing so would be expensive, especially if done from userspace. Fortunately, we have had no need for this in our implementation.

4.1.3 Ioctls

Finally, I/O control, usually performed using the *ioctl* system call, is part of the interface between user and kernel space. Programs typically use this to allow userland code to communicate with the kernel through device drivers. Each *ioctl* request uses a command number which identifies the operation to be performed and in certain cases the receiver. Attempts have been made to apply a formula on this number that would indicate the direction of an operation, as well as the size of the data being transferred [7]. Unfortunately, due to backward compatibility issues and programmer errors actual *ioctl* numbers do not always follow this convention. As a result, we had to check the *ioctl* numbers used by Android for correctness. Fortunately, Smartphones have fewer drivers than PCs, but the procedure is still a tedious one. For Android, we had to check about two hundred of them.

4.2 Execution Trace Compression

A great challenge for *PA* is to minimise transmission costs. All aspects of the execution that can be reconstructed at the security server should not be sent. In the next few paragraphs, we will discuss how we were able to trim the execution trace significantly. Each time, we will introduce a guiding principle by means of an example and then formulate the optimisation in a general rule.

Assuming that the phone and the replica are in sync, we are only interested in events that (a) introduce non-determinism, and (b) are not yet available on the replica. In principle, replica and phone will execute the same instruction stream, so there is no need to record a system call to open a file or socket, or to get the process identifier, as these calls do not change the stream of instructions being executed. Phrased differently, they do not introduce non-determinism. We summarise the above as follows.

Rule 1 *Record only system calls that introduce non-determinism.*

Similarly, even though the results of many system calls introduce non-determinism in principle, they can be still pruned from the trace, because their results are also available on the replica. For instance, the bytes returned by a `read` that reads from local storage probably influence the subsequent execution of the

program, but since local storage on the security server is the same as on the phone, we do not record the data. Instead, we simply execute the system call on the replica. The same holds for local IPC between processes that we trace. There is no need to transmit this data as the mirror processes at the security server will generate the same data. As data in IPCs and data returned by file system reads constitute a large share of the trace in the naive implementation, we save a lot by leaving them out of the trace. We now summarise this design decision.

Rule 2 *Record only data that is not available at the security server.*

In some cases, we can even prune data that is not immediately available at the security server. Data on network connections is not directly seen by the replica. However, it would be a serious waste to send data first from the network (e.g., a web server) to the phone, and then from the phone back to the network to make it available to the security server. Instead, we opted for a transparent proxy that logs all Internet traffic towards the phone and makes it available to the security server upon request (see also Figure 1). As a result, whenever the replica encounters a read from a network socket, it will obtain the corresponding data from the proxy, rather than from the phone. In general, we apply the following rule.

Rule 3 *Do not send the same data over the network more than once. Use a proxy for network traffic.*

Besides deciding *what* to record, we can further trim the trace by changing *how* we record it. By encoding the produced data to eliminate frequently repeating values, we can greatly reduce its size. An out of the box solution we employed was stream compression using the standard DEFLATE algorithm [14] which is also used by the tool gzip. Compression significantly reduces the size of the trace, but being a general purpose solution leaves room for improvement. We can further shrink the size of the trace by applying delta encoding on frequently occurring events of which successive samples exhibit only small change between adjacent values. We found an example of such behaviour when analysing the execution trace after applying guidelines 1-3. System calls such as `clock_gettime` and `gettimeofday` are called very frequently, and usually return monotonically increasing values. By logging only the difference between the values returned by two consecutive calls we can substantially cut down the volume of data they produce. Special provisions need to be made for `clock_gettime`, since the kernel frequently creates a separate virtual clock for each process. As a consequence we must calculate the delta amongst calls of the same process alone for higher reduction.

We use related, but slightly different optimisations when items in the trace are picked from a set of possible values, where some values are more likely to occur than others. Examples include system call numbers and return values, file descriptors, process identifiers, and so on. In that case we prefer Huffman encoding. For instance, we use a single bit to indicate whether the result of a system call is zero, and a couple of bits to specify whether one or two bytes are sufficient for a system call's return value, instead of the standard four. We summarise the principle as follows.

Rule 4 *Use delta encoding for frequent events that exhibit small variation between samples and Huffman encoding when values are picked from a set of possible values with varying popularity. Check whether the encoding yields real savings in practice.*

4.3 Local Data Generation

While we can save on data that is already available 'in the network' (at the security server or the proxy), no such optimisations hold for data that is generated locally. Examples include key presses, speech, downloads over Bluetooth (and other local connections), and pictures and videos taken with the built-in camera.

Keystroke data is typically limited in size. Speech is not very bulky either, but generates a constant stream. We will show in Section 5 that *PA* is able to cope with such data quite well.

Downloads over Bluetooth and other local connections fall into two categories: (a) bulk downloads (e.g., a play list of music files), typically from a user's PC, and (b) incremental downloads (exchange of smaller files, such as ringtones, often from other mobile devices). Incremental downloads are relatively easy to handle. For bulk downloads, we can save on transmitting the data if we duplicate the transmission from the PC such that it mirrors the data on the replica. However, this is an optimisation that we have not used in our project.

Pictures and videos taken using the device may incur significant overhead in transmission. In application domains where such activities are common, users will probably have to disconnect from the server, and only resynchronise when their device is recharging to avoid draining the battery. In the future, we could exploit the increasing trend of users uploading their content to the Internet directly from their devices, to proxy the uploaded data and make them available to the replica.

4.4 Attack Detection Mechanisms

We demonstrate the detection capabilities of the security server by developing two very different detection mechanisms: an anti-virus scanner, and an emulator-based detector that uses dynamic taint analysis.

4.4.1 Virus scanner

A simple security measure we apply at the server is to scan files in the replica's filesystem. We modified the popular open-source anti-virus ClamAV to do exactly that in the Android emulator. ClamAV contains more than 500000 signatures for viruses that a user would have to store locally on his phone and update daily. Using *PA*, we perform file scanning on the server where both storage and processing is much cheaper. Moreover, if we wish to further increase detection coverage we may employ multiple scanners at the same time, as suggested in the CloudAV project [34].

4.4.2 Dynamic taint analysis

To illustrate the sort of heavy-weight security checks that are possible with *PA*, we modified the Android emulator to perform dynamic taint analysis [13,31,11]. Taint analysis is a powerful, but expensive method to detect intrusions. The technique marks (in the emulator) all data that comes from a suspect source, like the network, with a taint tag. The tag is kept in separate (shadow) memory, inaccessible to the software. Taint is propagated through the system to all data derived from tainted values. Specifically, when tainted values are used as source operands in ALU operations, the destinations are also tainted; if they are copied, the destinations are also tainted, etc. Other instructions explicitly 'clean' the tag. An example is '`MOV R2,#0`' which zeroes the `R2` register on the ARM and cleans the tag. An alert is raised when a tainted value is used to affect a program's flow of control (e.g., when it is used as a jump target or as an instruction).

Taint analysis works against a host of exploits (including zero-days), and incurs practically no false positives. The overhead is quite high, typically orders of magnitude [11]. Alex Ho et al. show that it may be possible to bring the overhead down to a factor 2-3 [1], but this requires the presence of both a hypervisor and an emulator on the device, which is not very realistic on a smartphone. Moreover, even a factor 2-3 is probably too high for practical deployment on phones.

(a) Data generated by *PA* running for a day   (b) Average data generation rate, when performing various tasks
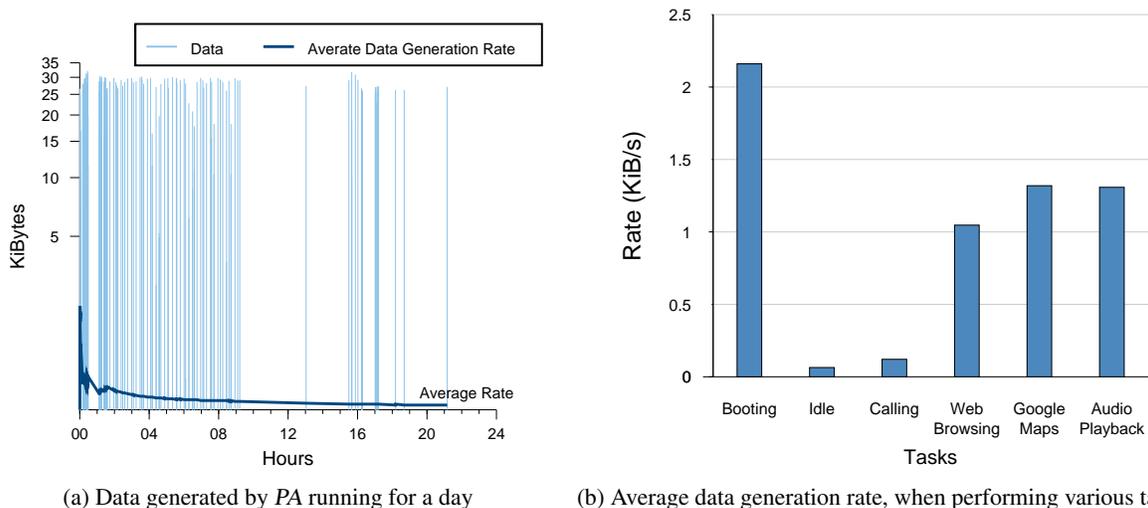
Figure 2: Data generation

5 Evaluation

We evaluated our implementation of *PA* along three axes: the amount of trace data generated during recording, the overhead imposed by the tracer on the device, and finally the performance and scalability of the server hosting the replicas. We ran the tracer on the HTC G1 Android developer phone, while the replayer was hosted on the Android emulator that comes with the official SDK.

5.1 Data Generation

The volume of data generated by the tracer constitutes an important metric, as it directly affects the amount of energy required to transmit the trace log to the server, and the storage space needed to store it on the device when disconnected from the server. Additionally, the upload bandwidth available to smartphone users (usually a few hundred Kbps) is a scarce resource, as it is frequently much less than the available download bandwidth [37].

Our traces with actual users show, not surprisingly perhaps, that mobile devices are mostly idle, or used for voice communications. A typical plot of the amount of data that is generated over time is shown in Figure 2a. Meanwhile, Figure 2b shows that the data generated when performing such tasks is negligible, with an average of 64B/s for idle operation, and 121B/s when performing a call. Even when performing more intensive tasks, such as browsing or listening to music, the tracer generates less than 2KiB/s. For instance, 5 hours[3] of audio playback would generate about 22.5MB of trace data. This shows that the trace is small enough to be stored locally on smartphones, which already offer relatively large amounts of storage (the iPhone 3GS comes with 32GB of storage).

5.2 Overhead

PA imposes two types of overhead on the phone. First, it consumes additional CPU cycles and thus incurs a performance overhead. Second, it consumes more power because of the increased CPU usage and the synchronisation with the server. To quantify these costs, we monitored the device's CPU load average, and

---

[3]Typical battery life when browsing or reproducing audio can range from 3 to 8 hours depending on the device.
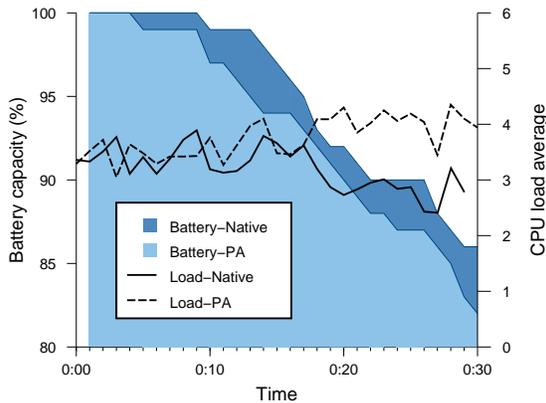
Figure 3: Battery consumption and CPU load average when browsing on the HTC G1 phone. We compare native execution, with execution under *PA*. Note that the y-axis on the left starts at 80%.
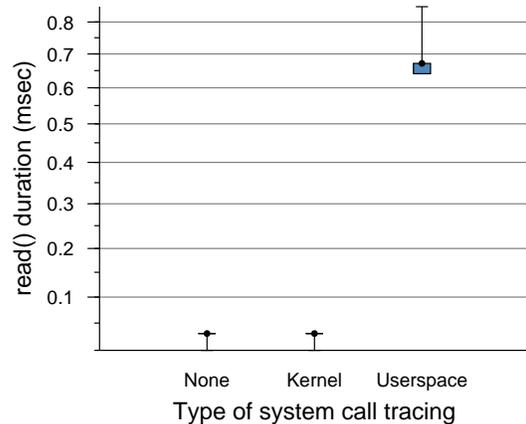
Figure 4: Time needed to read 4096 bytes from /dev/urandom with and without tracing system calls. The median (dot), 5%, 25%, 75%, and 95% percentiles are drawn.

Function	Time Spent %
ptrace()	% 33.63
waitpid()	% 32.68
deflate_slow()	% 7.62
pread64()	% 6.78
mcount_interval()	% 2.84
event_handler_run()	% 2.15

Table 1: Time consumed in various parts of the tracer.

Task	Utilisation %		
Games	90%	-	100%
Audio	20%	-	25%
Browsing	30%	-	100%
Idle	0%	-	5%

Table 2: CPU utilisation on the HTC G1 when performing various tasks.

battery capacity, while randomly browsing URLs from [4]. We performed this task natively as well as under *PA*, and show the results in Figure 3.

5.2.1 Performance

Figure 3 confirms that *PA* increases the CPU load on the device. In particular, the mean CPU load during this experiment was about 15% higher when using *PA*. The use of compression and encryption is somewhat costly in terms of processing, but the amount of data we generate does not seem to account for such a divergence. We investigated further to identify potential bottlenecks.

We profiled the tracer and report the results in Table 1. We see that indeed compression ($deflate\_slow()$) consumes only 7.62% of the tracer's execution time, and no cryptographic function is even reported in the top results. On the other hand, a bit more than 65% is spent in $ptrace()$ and $waitpid()$. The latter is called continuously to retrieve events from the kernel. Every time a traced process enters or exits a system call, it is blocked and such an event is delivered to the tracer through $waitpid$. Additionally, the tracer needs to use $ptrace()$ at least once for every event to retrieve additional information.

These two calls cause a large number of context switches between the tracer, traced processes, and the kernel, thus incurring large part of the overhead we observe. Similarly, $pread64()$ is used to copy data from the memory of traced processes (such as data returned by a system call). Moving event reception

15
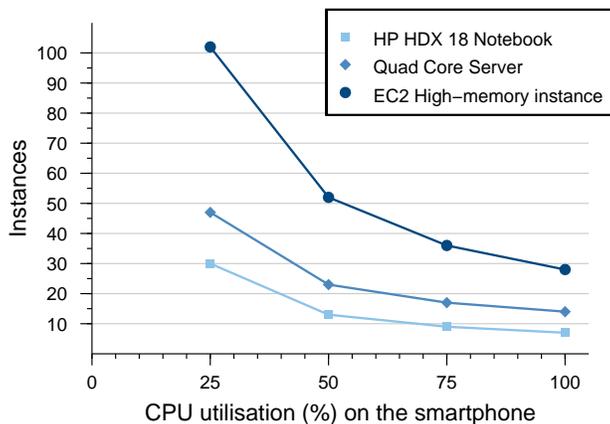
Figure 5: Number of replica instances that can be run on the server, concurrently and without any delay (in-sync with the phone). Increased CPU utilisation on the phone, means that the instances compete for more CPU cycles at the server.

and the initial part of event handling of *PA* in the kernel, would shed most of the overhead imposed by our prototype userspace implementation. Figure 4 shows that tracing the *read()* system call alone involves a huge overhead over native execution. On the other hand, tracing the call (including copying the results) in the kernel imposes no observable overhead. Future work on *PA* will focus on moving part of the implementation in the kernel.

5.2.2 Battery consumption

Figure 3 also shows how battery capacity drops in time while browsing. As expected power is consumed faster when using *PA*. Our prototype consumes about 30% additional energy when performing a costly task such as browsing. Almost all of this overhead is due to the additional CPU cycles consumed by the tracer. We confirmed this by way of an experiment where we (only) transmit the trace data corresponding to the browsing activity (using SSL as the tracer would do), and found that the device did not report *any* drop in battery capacity. As mentioned earlier, most of this overhead is due to the use of `ptrace` and will drop when we complete our implementation in the kernel.

When the device is idle, the system goes into power saving mode, where most software including the tracer is sleeping. Ergo, the overhead of the tracer is not even noticeable.

5.3 Server Scalability

Previous work has shown that simply moving computation from a smartphone to faster hardware such as a PC, even when running on the Qemu-based Android emulator, can increase performance up to 11.8 times [10]. Obviously, we cannot replay execution faster than it is recorded, but the significant difference in performance between smartphone devices and PCs, indicates that we can host multiple replicas on each security server.

We corroborated our assumption, by measuring the number of phone replicas that can be run concurrently on various hardware configurations. All the replicas hosted (at any given time) on the server were executing the same task, and were always in-sync with the replayed device, i.e. the replica had to wait for trace data from the device. Figure 5 shows the number of replicas that can be run under these conditions, on a powerful notebook computer, a quad-core small-office server, and an extra-large instance on Amazon's

Elastic Cloud (EC2) service. *When running in the EC2 cloud, we were able to concurrently run more than 100 replicas of devices exhibiting an average 25% CPU utilisation.* Utilisation is a key factor, determining the number of replicas that can be run without delays, as computation is relatively expensive when running under the emulator. Fortunately, this is a core competence of companies offering Cloud services.

Determining the average CPU utilisation of smartphones in a realistic scenario is not a trivial task, and we are not familiar of any preexisting studies on the subject. Nevertheless, we can look at the intensity of different tasks commonly performed on these devices. Table 2 shows how much CPU is used when running some common tasks on the HTC G1 phone. We can intuitively argue that smartphones remain idle or run non-interactive tasks like listening to music most of the time. In the opposite case, battery is drained quickly by the CPU (when running intensive tasks such as browsing or gaming), the display, and various device sensors (GPS, accelerometer, etc).

6 Related Work

The idea of decoupling security from execution has been explored previously in a different context. Malkhi and Reiter [25] explored the execution of Java applets on a remote server as a way to protect end hosts. The code is executed at the remote server instead of the end host, and the design focuses on transparently linking the end host browser to the remotely-executing applet. Although similar at the conceptual level, one major difference is that *PA* is *replicating* rather than *moving* the actual execution, and the interaction with the operating environment is more intense and requires significant additional engineering.

The Safe Execution Environment (SEE) [43] allows users to deploy and test untrusted software without fear of damaging their system. This is done by creating a virtual environment where the software has read access to the real data; all writes are local to this virtual environment. The user can inspect these changes and decide whether to commit them or not.

The application of the decoupling principle to the smartphone domain was first explored in Smart-Siren [8], albeit with a more traditional anti-virus file-scanning security model in mind. As such, synchronisation and replay is less of an issue compared to *PA*. Oberheide *et al.* [34] explore a design that is similar to SmartSiren, focusing more on the scale and complexity of the cloud back-end for supporting mobile phone file scanning, and sketching out some of the design challenges in terms of synchronisation. Some of these challenges are common in the design of *PA*, and we show that such a design is feasible and useful.

However, as smartphones are likely to be targeted through more advanced vectors than (just) viruses that rely mostly on social engineering, we argue that simple file scanning is not sufficient, and a deeper instrumentation approach is needed. Moreover, we also want to prevent attackers to hide their traces if they do manage to compromise the phone with a virus for which there was not yet a signature in the database. As argued in Section 2.3, *PA* provides scalability in number and variety of detection methods, and offers good attack visibility. These properties are needed for high-grade security on next-generation smartphones.

The *PA* architecture bears similarities to BugNet [30] which consists of a memory-backed FIFO queue effectively decoupled from the monitored applications, but with data periodically flushed to the replica rather than to disk. We store significantly less information than BugNet, as the identical replica contains most of the necessary state.

Schneier and Kelsey show how to provide secure logging given a trusted component much like our secure storage component [41, 42]. Besides guaranteeing the logs to be tamper free, their work also focuses on making it unreadable to attackers. We can achieve similar privacy if the secure storage encrypts the log entries. Currently, we encrypt trace data only when we transmit it to the security server.

Related to the high-level idea of centralising security services, in addition to the CloudAV work [33] which is most directly related to ours, other efforts include Collapsar, a system that provides a design for forwarding honeypot traffic for centralised analysis [21], and Potemkin, which provides a scalable frame-

work for hosting large honeyfarms [45].

7 Conclusion

In this paper, we have discussed a new model for protecting mobile phones. These devices are increasingly complex, increasingly vulnerable, and increasingly attractive targets for attackers because of their broad application domain. The need for strong protection is urgent, preferably using multiple and different attack detection measures. Unfortunately, battery life and other resource constraints make it unlikely that these measures will be applied on the phone itself. As an alternative, we presented a security model that performs attack detection on a remote server in the Cloud where the execution of the software on the phone is mirrored in a virtual machine. In principle, there is no limit on the number of attack detection techniques that we can apply in parallel. Rather than running the security measures on itself, the phone records a minimal execution trace, and transmits it to the security server, which replays the original execution in exactly the same way.

The evaluation of a userspace implementation of our architecture *Paranoid Android*, shows that transmission overhead can be kept well below 2.5KiBps after compression even during periods of high activity (browsing, audio playback) and to virtually nothing during idle periods. Battery life is reduced by about 30%, but we show that it can be significantly improved by implementing the tracer within the kernel. We conclude that the architecture is suitable for protection of mobile phones. Moreover, it allows for much more comprehensive security measures than possible with alternative models.

References

[1] A. Ho, M. Fetterman, C. Clark, A. Warfield, and S. Hand. Practical taint-based protection using demand emulation. In *Proc. of EuroSys'06*, April 2006.

[2] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with WIT. In *Proc. of SP 2008*, pages 263–277, May 2008.

[3] P. Akritidis, W. Y. Chin, V. T. Lam, S. Sidiroglou, and K. G. Anagnostakis. Proximity breeds danger: Emerging threads in metro-area wireless networks. In *Proc. of the 16th USENIX Security Symposium*, August 2007.

[4] Alexa Internet Inc. Global top 500. `http://www.alexa.com/topsites`.

[5] F. Bellard. QEMU, a fast and portable dynamic translator. In *Proc. of USENIX'05*, April 2005.

[6] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. In *Proc. of Crypto'96*, pages 1–15, August 1996.

[7] M. E. Chastain. Ioctl numbers. Linux Kernel Documentation – `ioctl-number.txt`, October 1999.

[8] J. Cheng, S. H. Wong, H. Yang, and S. Lu. SmartSiren: virus detection and alert for smartphones. In *Proc. of MobiSys'07*, pages 258–271, June 2007.

[9] J. Chow, T. Garfinkel, and P. M. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *Proc. of the USENIX'08*, pages 1–14, June 2008.

[10] B.-G. Chun and P. Maniatis. Augmented smartphone applications through clone cloud execution. In *Proc. of HotOS XII*, May 2009.

[11] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-end containment of internet worm epidemics. In *Proc. of SOSP'05*, October 2005.

[12] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with "readers" and "writers". *Commun. ACM*, 14(10):667–668, 1971.

[13] D. E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, 1976.

[14] P. Deutsch. DEFLATE compressed data format specification version 1.3. RFC 1951, May 1996.

[15] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proc. of OSDI'02*, pages 211–224, December 2002.

[16] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen. Execution replay of multiprocessor virtual machines. In *Proc. of VEE '08*, pages 121–130, March 2008.

[17] F-Secure. "sexy view" trojan on symbian s60 3rd edition. `http://www.f-secure.com/weblog/archives/00001609.html`, February 2008.

[18] J. Giffin, S. Jha, and B. Miller. Efficient context-sensitive intrusion detection. In *Proc of NDSS'04*, February 2004.

[19] J. T. Giffin, S. Jha, and B. P. Miller. Automated discovery of mimicry attacks. In *Proc. of RAID'06*, pages 41–60, September 2006.

[20] L. Hatton. Reexamining the fault density component size connection. *Software, IEEE*, 14(2):89–97, 1997.

[21] X. Jiang and D. Xu. Collapsar: a VM-based architecture for network attack detention center. In *Proc. of the 13th USENIX Security Symposium*, pages 15–28, August 2004.

[22] V. Kiriansky, D. Bruening, and S. P. Amarasinghe. Secure execution via program shepherding. In *Proc. of the 11th USENIX Security Symposium*, pages 191–206, August 2002.

[23] T. Leblanc and J. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, 36(4):471–482, 1987.

[24] G. Legg. The bluejacking, bluesnarfing, bluebugging blues: Bluetooth faces perception of vulnerability. `http://www.wirelessnetdesignline.com/192200279?printableArticle=true`, August 2005.

[25] D. Malkhi and M. K. Reiter. Secure execution of java applets using a remote playground. *IEEE Trans. Softw. Eng.*, 26(12):1197–1209, 2000.

[26] P. Montesinos, M. Hicks, S. T. King, and J. Torrellas. Capo: a software-hardware interface for practical deterministic multiprocessor replay. In *Proc. of ASPLOS '09*, pages 73–84, March 2009.

[27] H. Moore. Cracking the iPhone (part 1). `http://blog.metasploit.com/2007/10/cracking-iphone-part-1.html`, October 2007.

[28] W. Mossberg. Newer, faster, cheaper iPhone 3G. Wall Street Journal, July 2008.

[29] R. Naraine. Google Android vulnerable to drive-by browser exploit. *http://blogs.zdnet.com/security/?p=2067*, October 2008.

[30] S. Narayanasamy, G. Pokam, and B. Calder. BugNet: Continuously recording program execution for deterministic replay debugging. *SIGARCH Comput. Archit. News*, 33(2):284–295, 2005.

[31] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proc. of NDSS'05*, February 2005.

[32] Niacin and Dre. The iPhone/iTouch tif exploit is now officially released. Available at `http://toc2rta.com/?q=node/23`, October 2007.

[33] J. Oberheide, E. Cooke, and F. Jahanian. CloudAV: N-version antivirus in the network cloud. In *Proc. of the 17th USENIX Security Symposium*, San Jose, CA, July 2008.

[34] J. Oberheide, K. Veeraraghavan, E. Cooke, J. Flinn, and F. Jahanian. Virtualized in-cloud security services for mobile devices. In *Proc. of MobiVirt '08*, pages 31–35, June 2008.

[35] oCERT. CVE-2009-0475: #2009-002 OpenCORE insufficient boundary checking during MP3 decoding. `http://www.ocert.org/advisories/ocert-2009-002.html`, January 2009.

[36] A. Ozment and S. E. Schechter. Milk or wine: Does software security improve with age? In *Proc. of the 15th USENIX Security Symposium*, July 2006.

[37] PCWorld. A day in the life of 3G. `http://www.pcworld.com/article/167391/a_day_in_the_life_of_3g.html`, June 2009.

[38] N. Provos. Improving host security with system call policies. In *Proc. of the 12th USENIX Security Symposium*, August 2003.

[39] M. Ronsse and K. De Bosschere. RecPlay: a fully integrated practical record/replay system. *ACM Trans. Comput. Syst.*, 17(2):133–152, 1999.

[40] M. Russinovich and B. Cogswell. Replay for concurrent non-deterministic shared-memory applications. In *Proc. of PLDI '96*, pages 258–266, May 1996.

[41] B. Schneier and J. Kelsey. Cryptographic support for secure logs on untrusted machines. In *Proc. of the 7th USENIX Security Symposium*, January 1998.

[42] B. Schneier and J. Kelsey. Secure audit logs to support computer forensics. *ACM TISSEC*, 2(2):159–176, 1999.

[43] W. Sun, Z. Liang, R. Sekar, and V. N. Venkatakrishnan. One-way isolation: An effective approach for realizing safe execution environments. In *Proc. of NDSS'05*, pages 265–278, February 2005.

[44] D. Uluski, M. Moffie, and D. Kaeli. Characterizing antivirus workload execution. *SIGARCH Comput. Archit. News*, 33(1):90–98, 2005.

[45] M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft, A. C. Snoeren, G. M. Voelker, and S. Savage. Scalability, fidelity, and containment in the potemkin virtual honeyfarm. In *Proc. of SOSP '05*, pages 148–162, 2005.

[46] J. Xu and N. Nakka. Defeating memory corruption attacks via pointer taintedness detection. In *Proc. of DSN '05*, pages 378–387, June 2005.

[47] M. Xu, R. Bodik, and M. D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. *SIGARCH Comput. Archit. News*, 31(2):122–135, 2003.