Technical Report IR-CS-54 (September 2009)

Protecting smart phones by means of execution replication

Georgios Portokalidis
Vrije Universiteit Amsterdam
porto@few.vu.nl

Philip Homburg
Vrije Universiteit Amsterdam
philip@cs.vu.nl

Nicholas FitzRoy-Dale
NICTA
nfd@cse.unsw.edu.au

Kostas Anagnostakis
Institute for Infocomm Research
kostas@i2r.a-star.edu.sg

Herbert Bos
Vrije Universiteit Amsterdam
herbertb@cs.vu.nl

Abstract

Smartphones have come to resemble PCs in software complexity, with complexity usually leading to bugs and vulnerabilities. Moreover, as smartphones are increasingly used for financial transactions and other privacy-sensitive tasks, they are becoming attractive targets for attackers. Unfortunately, smartphones are quite different from PCs in terms of resource constraints imposed on the design of protection mechanisms, as battery power is an extremely scarce resource. As a consequence, security solutions designed for PCs may not be directly applicable to smartphones, as they may reduce battery lifetime significantly. Worse, as no single protection mechanism offers 100% security, it may be desirable to tighten up security further by applying multiple security solutions at the same time, thus increasing attack coverage and accuracy. Doing so exacerbates the problem.

In this paper, we explore a design where the detection of attacks is almost completely decoupled from the phone. We propose an architecture where all detection mechanisms are placed on a separate, loosely-synchronised security server which hosts one or more exact replicas of the smartphone. As the security server is not subject to the battery power constraint, we can apply a host of expensive detection techniques that would otherwise be too heavy-weight to ever consider implementing on the actual phone. We evaluate an implementation of the architecture on the HTC Dream / Android G1 phone platform and show that the overhead, in terms of computation and power consumption, is acceptable, even though we are applying extremely heavy-weight attack detection techniques based on dynamic taint analysis.

Categories and Subject Descriptors D.4.6 [*Security and Protection*]: Invasive software

General Terms Security, Mobile phones

Keywords Android, Decoupled security

1. Introduction

Smartphones have come to resemble general-purpose computers: in addition to traditional telephony stacks, calendars, games and addressbooks, we now use them for browsing the web, reading email, watching online 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.

The Problem As software complexity increases, so does the number of bugs and exploitable vulnerabilities [18, 33, 21, 32]. Vulnerabilities in the past have allowed attackers to use Bluetooth to completely take over mobile phones of various vendors, such as the Nokia 6310, the Sony Ericsson T68, and the Motorola v80. The process, known as bluebugging, exploited a bug in Bluetooth implementations. While these are older phones, more recent models, such as the Apple iPhone have also shown to be susceptible to remote exploits [29, 25].

Moreover, as phones are used more and more for commercial transactions, there is a growing incentive for attackers to target them. Credit card numbers and passwords are entered in phone-based browsers, while Apple, Google, Microsoft and other companies oper-

ate online stores selling applications, music and videos. Furthermore, payment for goods and services via mobile phones is provided by Upaid Systems and Black Lab Mobile. Companies like Verrus Mobile Technologies, RingGo, Easy Park, NOW! Innovations, Park-Line, mPark and ParkMagic all use phones to pay for parking, and yet others focus on mass-transit, such as for instance, HKL/HST in Finland and mPay/City Handlowy in Poland that both allow travellers to pay for public transport by mobile phone over GSM.

We see that both opportunity *and* incentive for attacking smartphones are on the rise. What about protective measures? On the surface, one might think this is a familiar problem: if phones are becoming more like computers, there is existing technology and ongoing research in fighting off attacks to PCs and servers. Unfortunately, it may not be feasible to apply the same security mechanisms in their current form.

While smartphones are like small PCs in terms of processing capacity, range of applications, and vulnerability to attacks, there are significant differences in other respects, most notably power and physical location. These two aspects matter when it comes to security. First, unlike normal PCs, smartphones run on battery power, which is an extremely scarce resource. For instance, one of the main points of criticism against Apple's iPhone 3G concerned its short battery life [26]. Vendors work extremely hard to produce efficient code for such devices, because every cycle consumes power, and every Joule is precious.

As a consequence, many of the security solutions that work for desktop PCs may not be directly portable to smartphones. Anti-virus file scanners [24], reliable intrusion detection techniques [10], and other well-known techniques all consume battery power. While the occasional file scanning may be relatively cheap, more thorough security checks in light of the increasing software complexity and the threat of code injection attacks are pushing the likely security overhead upwards. Furthermore, for many organisations, such as law enforcement, banks, governments, and the military, the use of phones is both critical and sensitive, and cannot be subjected to the same aggressive security restrictions at the policy level that are common for their office intranets[1].

For high-grade security, it is desirable to run a host of attack detection methods simultaneously to increase coverage and accuracy. However, doing so exacerbates the power problem and may even incur some unacceptable slowdowns. Indeed, some of the most reliable security measures (like dynamic taint analysis [7, 28]) are so heavy-weight that they can probably *never* be used on battery-powered mobile devices, unless we make significant changes to the hardware. Battery life sells phones, and consumers hate recharging [26]. The likely result is that both vendors and consumers will trade security for battery life.

Second, phones are required to operate in unprotected or even hostile environments. Unlike traditional computers, phones 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 [2]. That means that traditional perimeter security in general is insufficient, and even mobile phone security solutions that are based exclusively on "upstream" scanning of network traffic [8] will never even see the traffic involved in attacking the phone.

Worse, phones are small devices, and we do not always keep an eye on them. We may leave them on the beach when we go for a swim, slip them in a coat or shopping bag, forget them on our desks, etc. Theft of a phone is much easier than theft of a desktop PC or even a laptop. Attackers could 'borrow' the phone, copy data from it, open it physically, install back-doors, etc. For instance, after the bluebugging vulnerability mentioned above was fixed, phones could still be compromised as long as the attacker was able to physically access the device [21]. This is an important difference with the PCs we have sitting on our desks.

In summary, the trends are not favourable. On the one hand, mobile phones are an increasingly attractive target for attackers. On the other hand, because of power limitations and increased exposure to hostile environments, phones are inherently more difficult to protect than traditional computers.

Our approach: attack detection on remote replicas At a high-level, we envision that security (in terms of detecting attacks) will be just another service to be devolved from the mobile device and hosted in a separate server, much like storage can be hosted in a separate file server, email in a mail server, and so on. Whether or not this is feasible at the granularity necessary for thwarting today's attacks has been an open research question, which we attempt to answer in this paper.

More specifically, we explore the feasibility of decoupling the security checks from the phone itself, and performing all security checks on a synchronised copy of the phone that runs on a dedicated security server. This server does not have the tight resource constraints of a phone, allowing us to perform security checks that would otherwise be too expensive to run on the phone itself. To achieve this, we record a minimal trace of the phone's execution (enough to allow the security server

[1] A high-profile case in point is US president Barack Obama's struggle to keep his Blackberry smartphone, after he was told this was not possible due to security concerns. Eventually, it was decided that he could keep an extra-secure smartphone and months of speculation followed about which phone, its additional security measures, and the tasks for which he is permitted to use it.

to replay the attack and no more), and subsequently transmit the trace to the server for further inspection. The implementation of the full system is known as *Marvin*. It is illustrated in Figure 1.

Our approach is consistent with the current trend to host activities in centralised servers, possibly consolidated in a *cloud*, including security-related functions. For instance, Oberheide *et al.* have explored anti-virus file scanning in the cloud [30], and have more recently highlighted the opportunity for doing the same for mobile devices [31].

Although we subscribe to this trend at a high-level, we take a more aggressive approach to protection, especially considering the underlying threat landscape. Software on smartphones itself has frequently shown to be vulnerable to attacks for which the file scanning model is insufficient [21, 15, 32]. We therefore aim to prevent attacks on the phone software itself, focusing on exploits against client applications (like the browser, media player, email program, and calendar), and also covering misbehaving code installed by the user, and even cross-site scripting attacks. This is much more involved than scanning files for signatures. Nevertheless, our design still enables all the security checks to be pushed to an external security server.

Our solution builds on work on virtual machine (VM) recording and replaying [13, 41, 27, 9, 14]. Similar to these approaches, we record the execution of software running on a mobile phone and replay the exact trace on the security server. Rather than recording and replaying at VM level, we record the trace of a set of processes. We tailor the solution to smartphones, and compress and transmit the trace in a way that minimises computational and battery overhead. In addition to the replaying technique itself and the steps we take towards minimising the trace size, an important contribution of this paper is therefore the new application domain for replaying: resource-constrained devices that cannot provide comprehensive security measures themselves.

With the recorded trace, we can apply any security measure we want (including very expensive ones) and we can run as many detection techniques as we desire. By allowing heavy-weight attack detection solutions, we are able to detect attacks that could not possibly be detected otherwise. Not only that, but we make it possible to study the attack in detail. We can replay attacks arbitrarily, possibly with more detailed instrumentation. And as not all phones are active at the same time, it is highly likely that replicas of multiple phones can share one physical machine.

An additional advantage is that loss or theft of a phone does not mean the loss of the data on it. All data, up to and including the last bytes transmitted by the phone, is still safely stored on the replica.

Contribution To the best of our knowledge, we are the first to use decoupled replaying to provide security for resource constrained mobile devices. More broadly, we claim that this is the first architecture capable of offering comprehensive security checks for devices that are increasingly important for accessing the network, often store or use sensitive information, exhibit a growing number of vulnerabilities (thus forming attractive targets), and cannot reasonably afford the security measures developed for less constrained systems.

Furthermore, we have fully implemented the security architecture on a popular smartphone (the HTC Dream/Android G1 [19]). The implementation demonstrates that the approach is feasible, while our experimental analysis suggests that the tracing and synchronisation cost is reasonable when compared to the kind of security offered on the server side. While the implementation is tied to Android, the architecture is not, as our dependencies on a specific phone or even operating system are very limited, and the *Marvin* design applies to other models also.

Paper Outline Implementing a project on the scale of *Marvin* involves several person years in programming effort, much of which is spent on solving low-level engineering problems. Rather than trying to cram these details into our paper, we limit ourselves to the most interesting aspects of the architecture and implementation, and only discuss details when they are essential for understanding the bigger picture. The remainder of this paper is organised as follows. Section 2 presents a brief overview of the threat model and the likely configuration in terms of function placement. Section 3 outlines the proposed system architecture and the key design decisions and trade-offs. Section 4 provides details on the tracing techniques and how they can be made efficient to mminimise synchronisation overhead, and Section 5 outlines the server-side environment for replicating phone state and performing security checks. Our experimental analysis of the Android-based implementation is presented in Section 6. Section 7 discusses related work that has influenced our design, and Section 8 summarises our research findings.

2. Threat model and example configuration

We assume that all software on the phone, including the kernel, can be taken over completely by attackers. In practice, a compromise of the kernel takes place via a compromised userspace process. 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 over-

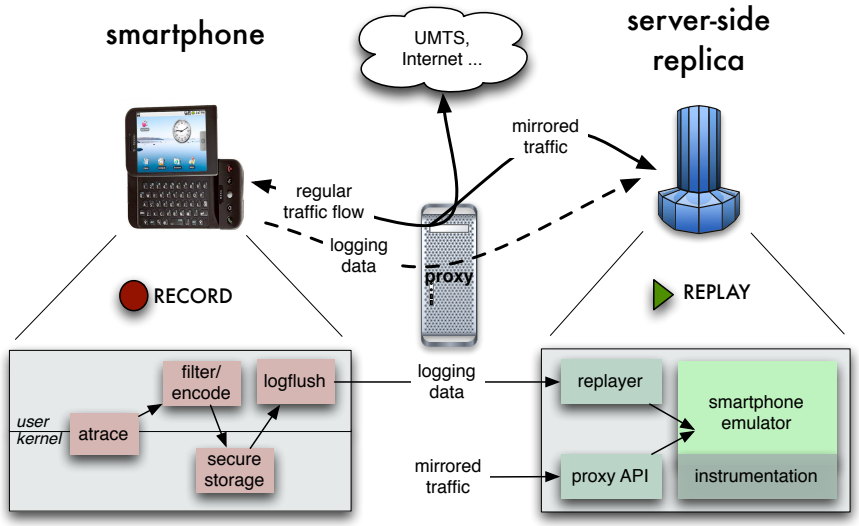

Figure 1. *Marvin* architecture

flows, etc.). Nor do we care about the medium: attacks may arrive over WIFI, 3G, Bluetooth, infra-red, 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.

Depending on the attack detection solutions that we provide on the security server, *Marvin* allows us to detect any and all of these types of attacks. To illustrate the power of the design, we implemented a security server that implements detection of code injection attacks by way of dynamic taint analysis [11, 28, 10].

Dynamic taint analysis is a very powerful intrusion detection technique that is able to detect exploits (buffer overflows, format string attacks, double frees, and so on) that change the control flow of the program. For instance, it detects when the program tries to use a function pointer that has been manipulated by the attacker by means of a buffer overflow, or it will detect that the victim program starts executing instructions provided by an attacker. Dynamic taint analysis is very accurate and incurs practically no false positives. However, it is also extremely expensive. The overhead is typically orders of magnitude. For this reason, it is unlikely that taint analysis can *ever* be applied on the phone itself. In practice, taint analysis is only provided on honeypots.

3. Architecture

A high-level overview of the *Marvin* architecture is illustrated in Figure 1. We sketch the basic idea first and zoom in on various optimisations (such as the proxy and secure storage) in later sections. A *tracer* on the phone intercepts system calls of, and signals to, the set of processes that need protection. This set comprises all processes on the phone that may be attacked. It is typically a large set that includes the browser, media players, system processes, and so on). A *replayer* on the security server subsequently replays the execution trace, exactly as it occurred on the phone, while subjecting the execution to additional instrumentation. The transmission of the trace is over an encrypted channel.

3.1 A naive implementation: sketching the basic idea

A naive implementation would intercept and record *all* signals, all system calls, all the system calls' results, and all reads from and writes to shared memory. As soon as it records any of these events, it would transmit it *immediately* to the security server. The security server executes exactly the same processes on an exact replica of the system. Like the tracer, the replayer also intercepts all system calls and signals. Whenever it encounters a system call, it looks in the trace for the same call. At that point, it will not really execute the system call, but instead return the results that it finds in the trace.

Signals need special treatment. Because of their asynchronous delivery, they introduce non-determinism. More precisely, since we do not know the exact moment of delivery on the phone and on the replica, they may cause race conditions. To ensure that signals are delivered at the same point both in the phone and in the mirrored execution on the security server, we do not deliver signals until the target process performs a system call. When the system call returns, we post the signal for immediate delivery. As both sides handle signals in exactly the same way, we synchronise signals delivery on the phone and the security server.

This way, we are *almost* able to replay the execution faithfully. The remaining issue concerns thread scheduling. As the kernel-level schedulers in phone and replica operate independently, it may be that threads on the replica are scheduled in an order different from that on the phone. For unrelated processes this is not a problem, but for threads that share memory (e.g., multi-threaded applications), it is important that the scheduling order is preserved.

The simplest and fastest way to solve this problem is to modify the kernel scheduler. Of course, doing so limits portability and makes it difficult to apply our architecture to closed source systems. For now, we will assume that we have a kernel scheduler that schedules the threads exactly the same in both the phone and the security server. In Section 4, we will show an alternative way that enforces a schedule on the threads *over* the schedule generated by the kernel. Either method works as long as it satisfies the following two requirements: (1) two memory-sharing threads should never run at the same time, and (2) scheduling should be deterministic.

For now, we want to point out the flexibility that we have in terms of security measures. Given the trace, we can replay the execution as often as we like and employ any security measure we want, either one after another, or in parallel. For instance, we can look for anomalies in system call patterns [35, 16], while at the same time applying dynamic taint analysis [11, 28, 10], and n-version virus scanning [31, 30].

A possible drawback is that there is a lag between the attack and its discovery (and possibly analysis). However, if the alternative is that the attack would not be detected at all, detecting an attack a few seconds after it infected the device still seems quite valuable [9].

As long as we can keep the cost of recording and transmitting the execution trace within reasonable bounds, the design above yields a powerful model to detect, stop and analyse attacks. We will shortly discuss various techniques to bring down the costs. There are three more issues that we need to discuss first: (i) where to place the security server, (ii) when to transmit the trace data, and (iii) how to warn the user when an attack is detected.

3.2 Location of the security server

Where to host the security server is a policy decision beyond the scope of this paper. Rather than prescribing the right policy, we discuss three possible models. While the first of these models is the simplest and allows for most optimisation, we do not preclude the others. In practice, the optimal location of the security server is a trade-off between costs, privacy concerns, reliability, and performance.

In the most straightforward model, the security server is a service offered by the provider. The provider can use its security service to differentiate itself from other providers and to generate income by charging for the service. In addition, it is ideally suited for offering the service. Much of the data to and from the mobile phone is routed via the network provider. Routing the traffic via a security server is easy and cheap. While there may be concerns about privacy, we observe that even today many applications and data already reside in various 'clouds' and that much of the private data is already passing through the equipment of the providers. We trust the providers to respect the privacy of their clients.

However, alternative models are also possible. In a business environment where phones are provided by a company, the company may host its own security server. Sensitive business data will be stored only on company computers and the organisation can decide for itself what security measures to apply. The model requires that all phone communication is routed through the security servers in the company's server room. An extreme case would allow end users to run the replicas on their home machines. Doing so gives users full control over their data, at the cost of paying the provider to reroute the traffic plus the cost of the security server itself.

3.3 When to transmit trace data

So far, we have assumed that the phone transmits the trace data immediately. In practice, however, this is probably not necessary. As the transmission cost per byte in power is lower if multiple events can be batched, we should try to batch as much as we can. The key insight is that we only need to transmit if there is a chance that the phone is compromised [36]. This is the case when the phone receives data from the network (be it over 3G, WIFI, USB, or Bluetooth), but not when processes on the phone exchange messages or when users update their calendars. In other words, we may be able to batch the trace data until we receive data that could potentially lead to a compromise.

Moreover, if the phone is fitted with secure storage that cannot be tampered with even if the phone is completely under the control of the attacker (e.g., storage protected by hardware or in a separate VM), we may batch data arbitrarily. In that case, we save all the trace data in secure storage and sync with the security server at a convenient time (for instance, every hour, or once a day). In the extreme case, we switch to offline checks, where the phone only synchronises when it is recharging and battery life is thus no longer an issue.

Secure storage means that the attacker may falsify the events sent to the trace since the attack, but not any of the events that lead to the attack [40]. Keeping the trace in storage for longer, also means that the attack can be

active for a longer time. However, we will eventually discover it. Moreover, it means minimal overhead in battery consumption.

We shall see in Section 4.3 that *Marvin* includes a secure storage implementation in the kernel (shown as a kernel module in Figure 1). We will also discuss our new, ongoing work on truly secure storage where a compromised kernel will not jeopardise the trustworthiness or even the availability of the trace data.

3.4 Informing the user about an attack

When *Marvin* detects an attack, it needs to warn the user, so that the user can start recovery procedures. This is not trivial. Sending an SMS or email message to say that the phone has been compromised may not work, as the phone is under the control of an attacker and the attacker may block such messages. We need a signalling channel beyond the control of the attacker.

The nature of this channel is not very important for this paper and various options exist. For instance, we could use what is known as a 'kill pill' on BlackBerry phones: hardware that allows administrators to thrash all data on a stolen or lost phone via a remote connection. A more advanced version could display a warning message on the display. Without hardware support, we could render the phone incommunicado on GSM, GPRS, UMTS and other networks under control of the operator. This will inform the user that something is wrong. Similarly, we may insert warning messages in all voice and data connections on these networks.

A pragmatic approach might be to start with SMS messages and take more draconian measures if the user does not initiate the appropriate recovery procedures within a certain time frame.

Users who know that their phones have been hacked will restore them to a safe state. We do not worry about this aspect in this paper, and simply restore to 'factory settings'. However, we envision that we can do much better than that. We can use the availability of the security servers (which have an exact copy of the file system on the devices) to restore the phone to a recent safe state, as long as we can determine a lower bound on the time of the attack.

4. Recording in practice

To make our architecture practical, we argued in Section 3 that the overhead of recording and transmitting the execution should be kept small both in computation and size. In this section, we will discuss how we achieved this in practice by means of various optimisations of the design. The implementation is known as *Marvin* and runs on an HTC Dream / Android G1, one of the latest 3G smartphones based on the open source Android software platform and operating system. The G1

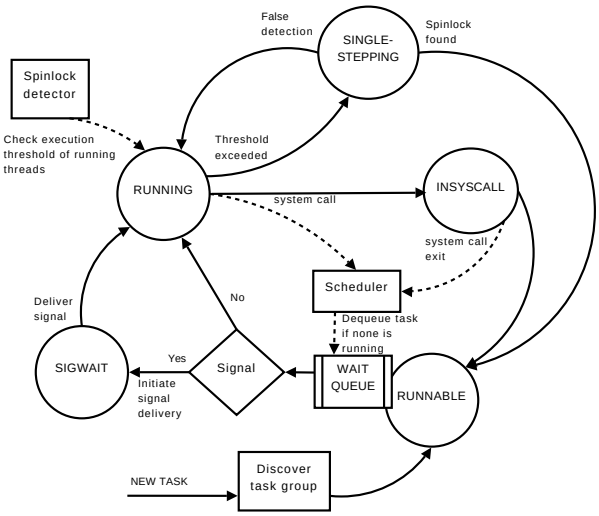

Figure 2. Scheduler FSM

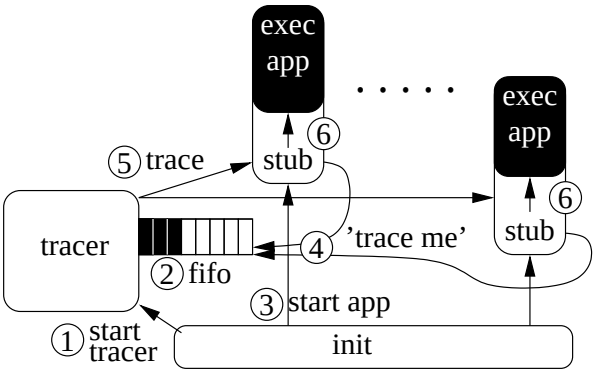

Figure 3. Tracing the processes from init

in its default configuration comes with a host of applications, including a browser, mail client, media player, and different types of messaging applications.

The security server runs one or more replicas of the phone on a Qemu-based Android emulator, which is part of the official SDK. Each replica runs a specific attack detection method, ranging from n-version virus scanning, to dynamic taint analysis.

4.1 Tracing on Android

For our implementation of Marvin on Android, we adopted a userspace approach based on the `ptrace` system call, which allows us to attach to arbitrary processes, and intercept both system calls and signals. By 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.

4.1.1 Tracing from the start

Marvin ensures that the tracing of relevant programs starts from the first instruction by means of a clean, two-step procedure that is illustrated in Figure 3. In UNIX tradition, Android uses the `init` process to start all other processes, including the client applications such as a browser and media players, but also the JVM, and so on). `Init` in *Marvin* 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 Issues

Several complicating factors demand further attention.

SIGKILL Ptrace cannot delay the delivery of SIGKILL and hence will not let the tracer intercept and defer it. To overcome this we intercept the signal at the source instead of the destination. Whenever a process sends a `SIGKILL` to another process, the tracer circumvents the kernel, and takes over the task of delivering the signal to the target, as well as recording the event.

Userspace scheduling A userspace implementation poses a challenge for the two scheduling requirements discussed in the previous section: (1) two threads that share memory should never run at the same time, and (2) scheduling should be deterministic. Rather than assuming that we can modify the kernel scheduler (or even receive scheduling events from the kernel), we opt for a userspace-only solution to allow our system to be ported to less open systems than Android.

Recall that the tracer intercepts both system call entry and exit, and signal delivery events. When the tracer receives such an event, it can decide to resume the thread, or delay its execution (e.g., by placing it on a waiting queue for later resumption). In other words, we have the functionality to determine which thread is run when. The aim is to enforce our own scheduling *over* the scheduling by the kernel.

We organise threads that share memory in task groups and maintain a run queue of runnable threads that are waiting to be run. Each of the threads can be in one of five states (see Fig. 2 for the finite state machine that controls the transitions):

1. RUNNING: the thread is currently running;
2. RUNNABLE: the thread is ready to run but waiting in the run queue;
3. INSYSCALL: the thread has entered a syscall;
4. SIGWAIT: the thread is waiting for a signal;
5. SINGLESTEPPING: a special state that is used to detect spinlocked threads (discussed later).

To satisfy requirement (1) we ensure that only one thread per group is in the RUNNING or SIGWAIT state. These states comprise all threads that are 'active'. From the perspective of our scheduler, threads that enter the kernel on a system call are not active. They are not on the run queue either, since they are not ready to run. A thread waiting for a signal to be delivered (SIGWAIT) is 'pre-running' and makes a transition to the RUNNING state immediately upon signal delivery. A thread exiting from a system call is not resumed immediately, but is instead appended in RUNNABLE state to the back of the group's queue. The scheduler will decide on the scheduling and ensures that a thread can only make a transition to SIGWAIT or RUNNING if no other thread of the group is in either of these states. To satisfy requirement (2) we set the scheduler to run right after a deterministic event. Since system calls are deterministic, the scheduler is called after entry to or exit from a system call. When that happens it schedules the task at the head of the queue, and any pending signal is also delivered.

Sometimes processes share memory using mechanisms in the kernel. As in multi-threaded applications, such sharing memory may introduce non-determinism in the system. To cater to this issue, we have extended the scheduler to merge the run queue of processes that share a memory segment. Doing so ensures that all the threads sharing some memory are scheduled exclusively.

Spinlocks The scheduling solution above works well in practice. However, the scheduling that we force upon threads *may* lead to deadlocks when userspace threads use spinlocks. Spinlocks are considered to be bad programming practice for mobile device applications, because in terms of CPU cycles they are a wasteful way to perform locking. We have not encountered such deadlocks in Android, where other "sleeping" methods such as futexes are preferred (futexes perform a system call in case of contention). Nevertheless, we dealt with this potential issue by means of a spinlock detector which we tested on synthetic examples.

Marvin periodically activates a spinlock detector to look for tasks that are potentially within a spinlock. The detector marks tasks as 'possibly spinlocked' if they are in the RUNNING state (see Figure 2) and the time since their last system call exceeds a threshold. As the situation is so rare, we optimistically set the threshold to a few seconds so that spinlock detection creates minimal overhead. A possibly spinlocked thread is moved to the SINGLESTEPPING state. We then single-step the thread to check whether it really is within a spinlock (e.g., we check whether it is running in a tight loop and we may test other properties also). When the thread is not spinlocked, it returns to RUNNING. If it is, however, *Marvin* sets the thread's state to RUNNABLE, appends it to the back of the run queue and schedules another thread. As the thread that is holding the lock will eventually run, the deadlock is removed.

Memory mapped by hardware Some memory, like the frame buffer, is mapped by hardware. This is not a problem in practice, as the memory access is essentially write only (e.g., the hardware writes bits to the framebuffer, which are not read by applications). However, it could be a problem in the future in a different hardware/-software combination, if processes were to read the values produced by the hardware. In that case, we will probably have to record all reads to this memory area (to reproduce the same values at the replica). For instance, by mapping the area inaccessible to the reader, we could intercept all read attempts to this memory as page faults, but doing so would be expensive. Fortunately, we have had no need for this in our implementation.

I/O control Finally, I/O control, usually performed using the `ioctl` system call, is part of the interface between user and kernel space. Programs typically use `ioctl`s to allow userland code to communicate with the kernel through device drivers. The interface is very flexible and it allows the exchange of data of variable length with few restrictions. 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 [5]. Unfortunately, due to backward compatibility issues and programmer errors actual `ioctl` numbers do not always follow the convention. As a result, the tracer needs knowledge of each command number, so that it is able to identify and log the data being read into userspace. Obtaining this meta-data is a tedious procedure, since it requires referring to the kernel's source code. Luckily, a lot of meta-data for common `ioctl` commands are available in various userspace emulators which saved us a lot of time.

4.2 Pruning redundant data: trimming the trace

The design above allows us to trace any process and replay it to detect attacks at the mirrored execution on the security server. Using `ptrace` is attractive, as it allows us to implement the entire architecture in userspace, with the sole exception of secure storage (see Section 4.3 for details about secure storage). A userspace implementation facilitates portability and allows the *Marvin* architecture to be applied to other phones even if the software on these phones is closed by nature, as long as they provide a tracing facility comparable to `ptrace`. However, from a performance point of view, system call interception in userspace is not the most optimal solution, as context switching is computationally costly. Most of the overhead can be removed by implementing system call interception in the kernel.

The main challenge in either case 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 sections, 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 in a guiding principle:

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 still can be pruned from the trace, because the 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 `read`s constitute a large share of the trace in the naive implementation, we save a lot by leaving them out of the trace. Summarising 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 [12] 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.

In theory, delta encoding could be applied to all time related system calls with similar behaviour. However, doing so does not always reduce the trace size. For instance, we applied the technique to reads from `/proc/[pid]/stat` files, which also generated significant amounts of data. `/proc/[pid]/stat` files are files in Linux' `procfs` pseudo file system that are used to track process information (such as the identifier of the parent, group identifier, priority, nice values, and start time, but also the current value of the stack pointer and program counter). Typically, the entire file is read, but only a small fraction of the file actually changes between reads. As we will show in Section 6 the manual delta encoding of such reads may even lead to an increase in log size. The reason is manual encoding may replace high frequency data with less efficient encoding. In the final prototype, we therefore dropped this 'optimisation'.

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 in the following rule:

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 Secure storage

We argued in Section 3.3 that transmitting trace information to the security server as soon as it is generated uses a lot of power, reducing battery life. It also requires a continuous connection to the network, which is unlikely. Instead, data are batched on the mobile device until we receive data over the network (which could potentially lead to an intrusion).

Batching introduces two security concerns [36]: firstly, the attacker must be prevented from tampering with the trace information to hide the evidence of an attack; and secondly, the attacker must be prevented from erasing the trace data.

We solve the problem of modified trace data by using digital signatures. Trace data in the secure storage is signed with a secret key when it is stored. The data and signature is transmitted to the replica, where the signature is checked for validity. To be effective, the secret key must be stored in a location inaccessible to the attacker. Trace data is only removed on the explicit and signed request of the security server, preventing attackers from wiping the log. Note that transmission of the trace is eventually performed by unprivileged user space processes. However, since the trace is signed by the kernel, there is no way for user space to tamper with the contents. If the phone fails to deliver the trace data even when it is connected for recharging, the security server assumes that the phone has been hacked.

Secure storage alleviates the problem of having to transmit data constantly. Current smartphones like the Apple Iphone 3G and the Android G1 already support

microSDHC cards with 16GB of storage. We shall see in Section 6 that with this amount of storage we are able to store an entire day's worth of activity locally and only synchronise with the security server when we recharge the phone at the end of the day.

There are several implementation options for secure storage. Our current implementation offers secure storage on the phone by way of a kernel module which stores and signs trace data. The key is stored in memory only accessible to the kernel module. By changing the key by means of a one-way hash each time a record is stored in the log, even a compromised kernel cannot tamper with an existing log. After all, an attacker has no way of recovering the keys that were used to sign existing entries. A kernel implementation protects log availability against attacks which gain access to user space, but is ineffective against attacks which compromise the kernel. In other words, the log can be deleted. However, a missing log is a clear indication of an attack.

Work on a more complete solution for secure storage which also protects against log deletion is ongoing. The design is straightforward. Truly secure storage requires keys and logs to be inaccessible even to the kernel. This can be achieved with custom hardware or, more realistically, by making use of a microkernel-based hypervisor in an arrangement similar to that of L4Linux [17] or Wombat [22]. If secure storage resides in a domain protected by hardware or by a hypervisor, we assume that it is a tamper-proof implementation of Schneier and Kelsey's trusted machine [36]: attackers can falsify the new data that is sent to secure storage, but they cannot tamper with existing logs.

We have opted for a hypervisor-based solution [4]. We currently run the security storage in a separate domain that communicates with a tracer on Linux on the appropriate ARM processor, but we still need to port the hypervisor to the full G1 hardware and paravirtualise the specific kernel used by Android. Using a hypervisor for logging combines the security benefits of dedicated hardware, such as well-defined separation from the guest and a narrow interface, with the cost-effectiveness and malleability of software [6]. The solution is a bit more portable than our kernel module as it can be also be used on other hypervisor-based systems.

4.4 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 6 that *Marvin* 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 incur significant overhead in transmission. In application domains where such activities are common, users will probably switch to offline checks, storing the data in secure storage and synchronising only when recharging the phone. Video conferencing is not possible at all on most smartphones, including the Android and Apple Iphone models, as the cameras are mounted on the back.

5. The security server

The security server decrypts and decompresses the trace it receives from the phone, and writes it to a file. In case the phone batched the trace events in secure storage, the security server will check the trace's signature and acknowledge the reception of a set of events with a signed request to the phone, to remove these events from the log.

The replicas run exact mirrors of the execution of the code on the phone in the Android emulator. The emulator is a QEMU-based application that provides a virtual ARM mobile device on which one can run real Android applications. It provides a full Android system stack, down to the kernel level. On top, we run the exact same set of applications as on the phone. Each replica implements one or more security checks and uses the trace file to remove potential non-determinism in the execution (as described previously). Initial execution starts with the same processor, memory and file system state.

A simple security measure is to scan files in the replica's file system using traditional virus scanners. To increase accuracy and coverage we may employ multiple scanners at the same time, as suggested in the CloudAV project [31]. Even more interesting is the application of more heavy-weight protection measures that could not realistically be applied on the phone itself.

To illustrate the sort of heavy-weight security checks that are possible with *Marvin*, we modified the Android emulator to include dynamic taint analysis [11, 28, 10]. 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

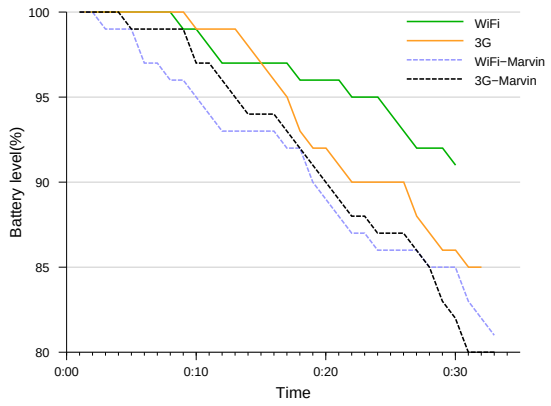

Figure 5. Battery Consumption

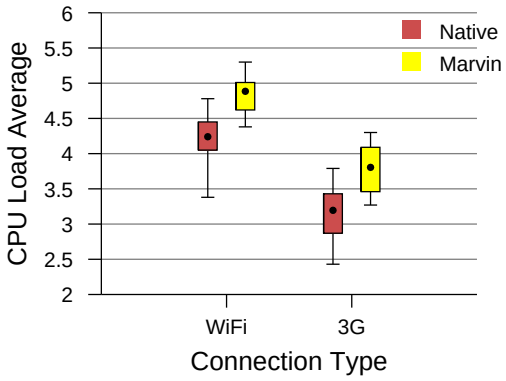

Figure 6. CPU Load Average

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

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 [10]. 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.

6. Results

In this section we evaluate our userspace implementation of *Marvin*. The recording side (tracer) was ran on an Android G1 developer phone, while the security server side (replica) ran on the Qemu-based Android emulator, which is part of the official SDK. We will attempt to quantify various aspects of the overhead imposed by the tracer on the device, and also evaluate the various optimisations we described in earlier sections.

6.1 Data generation rate

We have frequently mentioned that data transmission is costly in terms of energy consumption and consequently battery life. As such, the amount of data that our implementation generates and transmits to the replica consists a critical overhead metric. We calculated the rate the tracer generates data under different usage scenarios. Figure 4 shows the average rate measured in KiB/s. The tasks evaluated are from (left to right): booting the device, idle operation, performing and receiving a call, browsing the WWW using randomly selected URLs from a list of popular links [3], browsing random locations using the Google Maps application, and finally audio playback.

We also evaluate the effectiveness of the optimisations described in Section 4.2. Six different configurations (c1-c6) were tested in total. Each configuration introduces another data trimming optimisation, starting from *c1* were no optimisations are used. *C2* adds huffman-like encoding for event headers. Event headers comprise of common information logged for all system calls. Such information include the call number, the *pid* of the sender, return value, etc. *C3* and *c4* add delta encoding for `clock_gettime`, and reads from `/proc/pid/stat` respectively. Finally, *c5* uses a proxy to cache network data, and *c6* performs delta encoding for `gettimeofday`.

Compression was tested with all configurations, since it significantly reduces data volume. Fig. 4 shows that DEFLATE is the most efficient step in finding and eliminating repetition than any of our optimisations when network data are not involved, but the other opti-

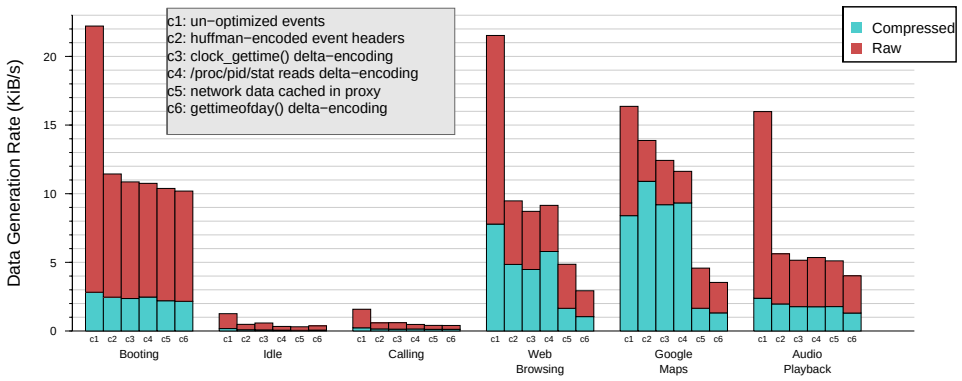

Figure 4. Data Generation Rate

misation also reduce the already small trace even more. Network access scenarios show that caching data using a proxy is necessary to keep overhead reasonable. Also, as mentioned earlier, the `/proc/pid/stat` delta encoding is counter-productive, since the encoding substitutes high frequency data with less compressable data. We do not use it in the prototype,

A mobile device usually spends most of its time idle, or it is used for voice communication. Fig. 4 shows that the data generation for these two scenarios is really negligible, with and average of just 64 B/s and 121 B/s for idle and calling respectively. These rates also show that employing secure storage (sec. 4.3) to store even an entire days of execution trace locally is feasible using devices such as the G1 (and also the Iphone).

6.2 Battery consumption

Transmission and reception of network data, along with the CPU, and the display are the largest energy consumers on mobile devices. *Marvin* directly affects two of these components, since it requires to transmit generated data and uses CPU cycles to operate. We evaluated the effect of the tracer on battery consumption, by using the device to browse the web in a similar fashion as earlier. Additionally, SSL encryption was employed to protect the data being transmitted. Encryption itself is probably decremental to battery life, but it is necessary. In the future, specialised hardware performing encryption cheaply could be included in mobile devices, allowing for broader adoption of encryption.

Figure 5 shows how battery levels drop in time while browsing. As expected battery levels drop faster when using *Marvin* than running Android natively. We also used both 3G and WiFi to evaluate their impact on battery life as well. When not using the tracer it is clear that WiFi is more conservative energy-wise. On the other hand, when using *Marvin* it is unclear if one is better than the other. On the positive side, our implementation incurs only a minor overhead on battery life, which, even for a costly operation such as browsing, does not exceed 7%.

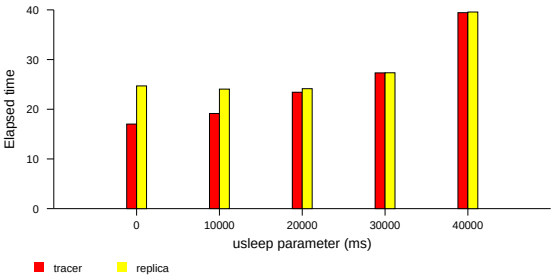

Figure 7. Security server lag

6.3 Performance

The tracer also incurs a performance overhead. Figure 6 shows the mean CPU load average during the experiment described in sec. 6.2. In both cases CPU load was higher when using the tracer. Using profiling tools we analysed the tracer to identify bottlenecks. Table 1 shows the top calls where time was spent in the tracer.

A bit more than 65% is spent in system calls that are responsible for controlling or waiting for events concerning the traced processes. On the other hand, DEFLATE only takes up 7.62%. A more optimised, and platform dependent kernel-space implementation could shed most of the overhead we see here by eliminating context switching and notification costs, as well as data copying between address spaces.

6.4 Security server lag

For particularly CPU intensive analysis, we expect that the security server will sometimes lag behind the tracer. Figure 7 shows how varying the average load will affect the ability of the replayer to keep up with the tracer. The experiment consists of running a test program that re-

peatedly compresses and decompresses 64 KByte worth of data. To vary the load, we insert a sleep between the compression and decompressing step with the parameter shown on the x-axis of the graph (useconds). The y-axis shows the elapsed time result of running the compression/usleep/decompression test 500 times. The tracer runs of the G1 hardware, and the replayer runs on a modified Qemu emulator that implements full dynamic taint analysis on an AMD Athlon64 3200+. The emulator is essentially the open source Argos attack detector [34] ported to the Android emulator.

The replayer cannot keep up with the real hardware when it comes to compression and decompression. However, the replayer does not have to execute the blocking system call (usleep). Therefore, increasing the time spend in blocking system calls reduces the lag of the replayer. The results for 0, 10000, and 20000 microseconds delay show that replayer is essentially unaffected by the delays and takes constant time. With the delays of 30000 and 40000 microseconds the replayer has to wait for results from the tracer and will therefore run at the same speed as the tracer.

7. Related work

The idea of decoupling security from execution has been explored previously in a different context. Malkhi and Reiter [23] 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 *Marvin* 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) [38] 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 SmartSiren [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 *Marvin*. However, as smartphones are likely to be targeted through more advanced vectors compared to viruses that rely mostly on social engineering, we argue that simple file scanning is not sufficient, and a deeper instrumentation approach, as demonstrated in *Marvin*, is necessary for protecting current and next generation smartphones. Oberheide *et al.* [31] explore a design that is similar to SmartSiren, focusing more on the scale and complexity of the cloud backend 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 *Marvin*, and we show that such a design is feasible and useful.

The *Marvin* architecture bears similarities to BugNet [27] 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 [36, 37]. 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 [30] which is most directly related to ours, other efforts include Collapsar, a system that provides a design for forwarding honeypot traffic for centralised analysis[20], and Potemkin, which provides a scalable framework for hosting large honeyfarms[39].

8. 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, and the need for strong protection is urgent, preferably using multiple different attack detection measures. Unfortunately, battery life and other resource constraints make it unlikely that these measures will be applied on the phone itself. Instead, we presented an architecture that performs attack detection on a remote security server 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, the phone records a minimal execution trace. The trace is transmitted to the security server to allow it to replay the original execution in exactly the same way. The architecture is flexible and allows for different policies about placement of the security server and frequency of transmissions.

The evaluation of an implementation of the architecture in userspace, known as *Marvin*, shows that transmission overhead can be kept well below 2.5 KiBps

after compression even during periods of high activity (browsing, audio playback) and to virtually nothing during idle periods. Battery life is reduced by 7%. 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 *EuroSys, Leuven, Belgium*, April 2006.

[2] 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 *USENIX Security Symposium*, Boston, MA, August 2007.

[3] Alexa Internet Inc. Global top 500. `http://www.alexa.com/site/ds/top_500`, 2006.

[4] Anonymous. Anonymized for double blind reviewing.

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

[6] P. M. Chen and B. D. Noble. When virtual is better than real. In *HOTOS '01*, page 133, Washington, DC, USA, 2001. IEEE Computer Society.

[7] S. Chen, J. Xu, N. Nakka, Z. Kalbarczyk, and C. Verbowski. Defeating Memory Corruption Attacks via Pointer Taintedness Detection. In *DSN'05*, pages 378–387, June 2005.

[8] J. Cheng, S. H. Wong, H. Yang, and S. Lu. Smartsiren: virus detection and alert for smartphones. In *MobiSys '07*, pages 258–271, New York, NY, USA, 2007. ACM.

[9] J. Chow, T. Garfinkel, and P. M. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *ATC'08: USENIX 2008*, pages 1–14, Berkeley, CA, USA, 2008. USENIX Association.

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

[11] D. Denning. A lattice model of secure information flow. *ACM Trans. on Communications*, 19(5):236–243, 1976.

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

[13] 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 *OSDI '02*, pages 211–224, New York, NY, USA, 2002. ACM.

[14] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen. Execution replay of multiprocessor virtual machines. In *VEE '08*, pages 121–130, New York, NY, USA, 2008. ACM.

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

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

[17] H. Härtig, M. Hohmuth, J. Liedtke, S. Scönberg, and J. Wolter. The performance of -kernel-based systems. In *SOSP'97*, pages 66–77, 1997.

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

[19] HTC. T-Mobile G1 - Technical Specification. `http://www.htc.com/www/product/g1/specification.html`, 2009.

[20] X. Jiang and D. Xu. Collapsar: A VM-Based Architecture for Network Attack Detention Center. In 13^{th} *USENIX Security Sumposium*, pages 15–28, August 2004.

[21] G. Legg. The bluejacking, bluesnarfing, bluebugging blues: Bluetooth faces perception of vulnerability. TechOnline `http://www.wirelessnetdesignline.com/showArticle.jhtml?articleID=192200279`, April 2005.

[22] B. Leslie, C. V. Schaik, and G. Heiser. Wombat: A portable user-mode linux for embedded systems. In *Proceedings of the 6th Linux.Conf.Au*, 2005.

[23] D. Malkhi and M. K. Reiter. Secure Execution of Java Applets Using a Remote Playground. *IEEE Trans. Softw. Eng.*, 26(12):1197–1209, 2000.

[24] Y. Miretskiy, A. Das, C. P. Wright, and E. Zadok. Avfs: an on-access anti-virus file system. In *13th USENIX Security*, pages 6–6, Berkeley, CA, USA, 2004. USENIX Association.

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

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

[27] S. Narayanasamy, G. Pokam, and B. Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. In *ISCA '05*, pages 284–295, Washington, DC, USA, 2005. IEEE Computer Society.

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

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

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

[31] J. Oberheide, K. Veeraraghavan, E. Cooke, J. Flinn, and F. Jahania. Virtualized in-cloud security services for mobile devices. In *Proc. of MobiVirt*, Breckenridge, CO, June 2008.

[32] oCERT. CVE-2009-0475: #2009-002 opencore insufficient boundary checking during mp3 decoding. `http://www.ocert.org/advisories/ocert-2009-002.html`, January 2009.

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

[34] G. Portokalidis, A. Slowinska, and H. Bos. Argos: an emulator for fingerprinting zero-day attacks. In *ACM SIGOPS EuroSys '06*, 2006.

[35] N. Provos. Improving host security with system call policies. In *12th USENIX Security Symposium*, 2003.

[36] B. Schneier and J. Kelsey. Cryptographic support for secure logs on untrusted machines. In *7th USENIX Security Symposium*, pages 4–4, Berkeley, CA, USA, 1998. USENIX Association.

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

[38] W. Sun, Z. Liang, R. Sekar, and V. N. Venkatakrishnan. One-way Isolation: An Effective Approach for Realizing Safe Execution Environments. In *SNDSS*, pages 265–278, February 2005.

[39] 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 *SOSP'05*, pages 148–162, 2005.

[40] B. R. Waters, D. Balfanz, G. Durfee, and D. K. Smetters. Building an encrypted and searchable audit log. In *NDSS'04*, 2004.

[41] M. Xu, R. Bodik, and M. D. Hill. A flight data recorder for enabling full-system multiprocessor deterministic replay. In *ISCA '03*, pages 122–135, New York, NY, USA, 2003. ACM.