Paranoid Android: Versatile Protection For Smartphones

Georgios Portokalidis*
Network Security Lab
Dept. of Computer Science
Columbia University, NY, USA
porto@cs.columbia.edu

Philip Homburg
Dept. of Computer Science
Vrije Universiteit
Amsterdam, The Netherlands
philip@cs.vu.nl

Kostas Anagnostakis
Niometris R&D
Singapore
kostas@niometrics.com

Herbert Bos
Dept. of Computer Science
Vrije Universiteit
Amsterdam, The Netherlands
herbertb@cs.vu.nl

ABSTRACT

Smartphone usage has been continuously increasing in recent years. Moreover, smartphones are often used for privacy-sensitive tasks, becoming highly valuable targets for attackers. They are also quite different from PCs, so that PC-oriented solutions are not always applicable, or do not offer comprehensive security. We propose an alternative solution, where security checks are applied on remote *security servers* that 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*. We implemented a prototype of this security model for Android phones, and show that it is both practical and scalable: we generate no more than 2KiB/s and 64B/s of trace data for high-loads and idle operation respectively, and are able to support more than a hundred replicas running on a single server.

Categories and Subject Descriptors

D.2.0 [**General**]: Protection mechanisms

General Terms

Design, Security, Reliability

Keywords

Decoupled security; Smartphones; Android

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. As software complexity increases, so does the number of bugs and exploitable vulnerabilities [17, 32, 20, 31]. Vulnerabilities in the past have allowed attackers to exploit bugs in the Bluetooth network stack to take over various mobile phones. More recently, Apple's iPhone and Google's Android platform have also shown to be susceptible to remote exploits [28, 24, 25] .

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, smartphones can be used to perform online purchases, control bank accounts, store passwords and other sensitive information like social security numbers, *etc.* Phone-based payment for physical goods, services, mass transit, and parking is also provided by various companies like Upaid Systems, Black Lab Mobile, and others. Compromised smartphones can also be used to spy upon users, as they include a GPS sensor and a microphone that can be used to obtain a user's location or eavesdrop.

Smartphones will most likely become targets in the future, and while average users may not be willing – for the time being – to pay the cost (both in financial and performance terms) of securing their devices, *this is not the case for senior officials in industry, government, law enforcement, banks, health care, and the military*[1]. Smartphones are already an integral tool in many such organisations, but due to security and privacy concerns, and due to the lack of security mechanisms, administrators often resolve in limiting the functionality of employees' devices (like disabling WiFi connectivity and reception of SMS messages). *In this paper, we address the problem of security for smartphones for organisations and individuals that care deeply about the detection of attacks.* Our goal is to provide versatile security for smartphones, offering detection of a wide range of attacks including zero-day ones.

Deploying security mechanisms on already severely resource-constrained smartphones can be problematic. For instance, running a simple file scanner like ClamAV on the Android

*This work was done while the author was in Vrije Universiteit Amsterdam.

[1]A famous case in point was 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.

ACSAC '10 Dec. 6-10, 2010, Austin, Texas USA

HTC G1's data and application folders took approximately 30 minutes, and reduced battery capacity by 2%. Other work [6] has also shown that running a naive file scanning application on an HTC G1 is $11.8x$ slower than running it on single-core virtual machine (VM) running on a desktop PC. *We argue for 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 the cloud, much like storage and email. Whether this is practical, or even 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 propose running a synchronised replica of the phone on a *security server* in the cloud. As the server does not have the tight resource constraints of a phone, we can 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. The implementation of our security model is known as *Paranoid Android* (*PA*).

Our approach is consistent with the current trend to host activities in the cloud, including security-related functions. Oberheide *et al.* have explored AV file scanning in the cloud with [29] and [30], but 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 [20, 14, 31, 25]). One could argue that smartphone components are frequently coded in languages like Java that do no suffer from such attacks. But the runtime environments (JREs) used on smartphones are usually smaller, optimized versions of the original JRE (*e.g.,* Android uses the DEX Dalvik VM), which do not necessarily provide the same security and isolation guarantees , and can be themselves vulnerable to attacks. Furthermore, most platforms (including Android) offer native APIs for high performance applications that are vulnerable to a wider range of attacks.

Our solution builds on work on VM recording and replaying [11, 42, 26, 5, 12, 19, 37, 38, 23]. Previous work on PC systems, makes use of tailored VMs, and assumes ample and cheap communication bandwidth. 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 also ensure that an attacker compromising a device cannot bypass the security measures applied at the server, and elude detection.

The main contributions of this paper are:

- A scalable smartphone security architecture that is able to apply multiple security checks simultaneously without overburdening the device.
- A prototype implementation of an execution recording and replaying framework for Android.
- Transparent backup of all user data in the cloud.
- A replication mechanism that guarantees the detection of an attack.
- Application transparent recording and replaying.

The remainder of the paper is organised as follows. The architecture of *PA* is discussed in Section 2, while implementation details of our prototype are given in Section 3. We evaluate the system in Section 4, and review related work in Section 5. Conclusions are in Section 6.

2. PARANOID ANDROID ARCHITECTURE

A high-level overview of *PA*'s architecture is illustrated in Figure 1. On the phone, a *tracer* records all information needed to accurately replay its execution. The recorded *execution trace* is transmitted to the cloud over an encrypted 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. We 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.,* they 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. Furthermore, we use a network proxy to connect to the Internet, which allows us to intercept and temporarily store inbound traffic. The *replayer* can access the proxy to retrieve the data needed for replaying. This way the *tracer* does not have to retransmit the data received over the network to the replica.

2.1 Recording And Replaying

Recording and replaying a set of processes and entire systems has been broadly investigated by previous work [11, 42, 26, 5, 12, 19, 37, 38, 23, 16]. We will only briefly discuss how execution replaying is performed, while implementation specifics and various optimisations are discussed in Section 3.1. Readers interested in recording and replaying in general are referred to the above cited papers, and our technical report on *PA* [34].

A computer program is by nature deterministic, but it receives nondeterministic inputs and events that influence its execution flow. To replay a program, we need to record all these nondeterministic inputs and events. Such inputs usually come from the underlying hardware (*e.g.,* time comes from the HW clock, network data from the WiFi adaptor, location data from the GPS sensor, *etc.*), which a process receives mostly through system calls to the kernel. Thus, to replay execution the *tracer* records all data transferred from kernel to user space through system calls. The *replayer* then uses the recorded values when replaying the system calls on the replica. Note that we only replay process and not kernel execution. While this implies that *PA* may not be able to detect an attack against the kernel, most kernel vulnerabilities are only exploitable locally, which would require that the attacker first compromises a user process.

Beside system calls, operating systems (OSs) can also alter a process' control flow by using synchronous and asynchronous notification mechanisms such as signals. For instance, a signal may be sent to a process when a certain event occurs (*e.g.,* a timer expires). Signals that notify of serious errors (*e.g.,* a segmentation fault, or a floating point exception) are delivered synchronously, when the instruction that caused the error is executed. Consequently, they will be also generated by the OS on the replica. 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

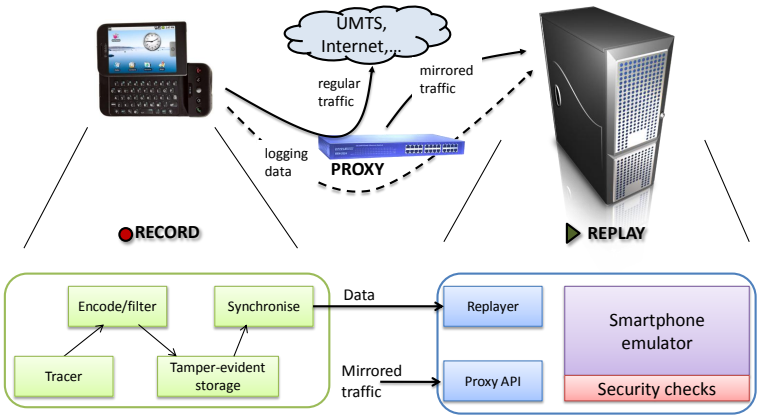

Figure 1: *Paranoid Android* **architecture overview**

at exactly the same time during replay, we defer their delivery until the target process performs a system call.

Concurrency and inter-process communication (IPC) can also be a source of nondeterminism. 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, therefore we implicitly support them by accurately replaying system calls.

This is not the case for shared memory and memory mapped files, since they can be accessed directly. When two or more processes use such objects to exchange data, they may affect one another in unpredictable ways, producing non-reproducible behaviour. In the case of threads, almost all process memory is shared. In the presence of shared objects, accesses on these objects need to be serialised to enable deterministic replay [19]. In past work, Courtois *et al.* [9] solve the serialisation problem using a concurrent-read-exclusive-write (CREW) protocol for shared objects, while Russinovich *et al.* [38] propose a repeatable deterministic task scheduler. He have adopted the latter for *PA*, as it outperforms CREW protocols on uniprocessor architectures.

2.2 Synchronisation

Smartphone users enjoy plentiful wireless connectivity over 3G, WiFi, GPRS, *etc.PA* can use any of these networks to synchronise with the replica by transmitting the execution trace. However, wireless connectivity can be costly in terms of energy consumption, and detrimental to battery life. Therefore, we assume that network connectivity may not be always available (*e.g.,* because the device is low on power), and safeguard the execution trace to ensure that attacks which occured while disconnected are eventually discovered.

2.2.1 Loose Synchronisation

We adopt a loose synchronisation strategy between the phone and the cloud to minimise its effects on battery life. Particularly, we do not activate or keep any of the network adaptors from sleeping, but rather attempt to transmit the trace only when the device is awake and connected to the Internet. This can be due to the user performing an action like reading email or surfing the web, when he is also most likely to be attacked (*e.g.,* by receiving a malicious email, or accessing a malicious web site). Alternatively, we also support an extremely loose synchronisation model, where the device synchronises with its replica only when it is recharging. Such a model may be suitable for users with more relaxed security requirements, as attacks can only be detected after synchronising with the server.

2.2.2 Tamper-Evident Secure Storage

Loose synchronisation with the server is ideal for preserving power, but unless we protect the execution trace, an attacker may compromise the phone and disable the synchronisation procedure. Even worse, a capable 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.

We defend against such attacks by employing a *secure storage* to detect if someone has tampered with the execution trace. Every block of data written to secure storage is associated with an HMAC code [2], 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. We achieve *tamper-evidency* by continuously "rolling" the key used with the HMAC, as we explain below.

Each time an entry along with its HMAC code is written to secure storage, we generate a new key by applying a second cryptographic hash function on the old key (which is completely overwritten). This way an attacker compromising the device, cannot alter old entries already in the execution trace to hide an attack. At worst, attackers can delete entries or block synchronisation, which both count as synchronisation errors.

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

Writes to secure storage occur regularly during the operation of *PA*, or can be triggered by a specific event. While the system is running, the data produced by the *tracer* are initially buffered and compressed in the manner described in Section 3.2. When data can no longer be buffered (*e.g.,* because the buffer has been exhausted), or when it is determined that they cannot be further compressed, they are

written to secure storage and a new key is generated. Alternatively, writes to secure storage may be "forced" when certain events occurs, even if additional buffering is possible. For instance, when a network read occurs that could potentially introduce malicious data, we request that the entry describing the network read (as well as the previously buffered entries) are written into secure storage. Different algorithms and strategies that determine the frequency of writes to secure storage can be explored in future work.

Using HMAC is more lightweight than digital signatures, as it requires less processing cycles (and consequently power) and storage. The only requisite is 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* authenticates the received data by calculating their HMAC code, and comparing it with the one received.

2.2.3 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 with confidence by the security server, and in any case we lose the ability to further replay execution. Consequently, devices exhibiting such errors are treated as potentially compromised, and the user needs to be notified and his device restored to a clean state (Section 2.5).

2.3 Security Methods

The real power of *PA* lies in the scalability and flexibility in security methods. By replicating smartphone execution in the cloud, we have ample resources for running a combination of security tasks. Moreover, we can apply any detection method that obeys the rule of noninterference 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 3.3) 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 [18, 41, 10, 8].
2. AV products in the cloud. We modified a popular open source AV to run in the emulator, and perform periodical file scans. Additionally, on access file scanning can be applied with few modifications to the *replayer*. On access scanning AV intercept file handling system calls and scan the target file before allowing a process to access it. As we already intercept system calls, the *replayer* could be transformed to an on access AV scanner.
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 [36, 15], 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. Furthermore, the execution trace can be retained and used for auditing purposes.

2.4 Proxy And Server Location

The location of the security server and the proxy, and who controls them is a policy decision beyond the scope of this paper. For instance, institutions running their own cloud could deploy the proxy and replica in-house. Alternatively, *PA* could be offered as a service by wireless providers, hosting the server on their own cloud. While privacy is important both for companies and individuals, smaller companies and individuals frequently place their data on cloud services offered by providers such as Amazon and Google.

In an extreme scenario, users with strong privacy considerations could run their own replicas on their desktop or notebook, and not use a proxy at all. Doing so gives them full control over their data, but implies a very loose synchronisation model, where the device synchronises with the server only when the device is plugged to the computer, or when they are on the same network (*e.g.,* similarly to Apple's *Time Capsule*).

2.5 User Notification And Recovery

When an attack is detected, *PA* needs to warn the user, so that recovery procedures can be initiated. This is not trivial. Sending an SMS or email message may not work, as a skilled attacker could block such messages. As such, a signalling channel beyond the control of the attacker is needed. The nature of this channel is not very important for this paper, but various options are already available. For instance, we could use special hardware on the phone to have it destroy all data, when it receives a privileged message by the owner or provider (*e.g.,* the "kill pill" message on Blackberry phones [39]). If hardware support is not available, the provider could also simply deny service to the device, which would (hopefully) inform the user that something is wrong.

Compromised devices can be restored to a pristine state using the data held at the replica. Data-loss can be kept at a minimum, as an exact copy of all user data exists in the cloud. Furthermore, using multiple intrusion detection techniques we can accurately detect the moment of the attack, to restore the really last clean state of the system. Unfortunately, recovery over the network cannot be guaranteed, so we adopt an approach similar to current systems such as the iPhone, where the device needs to be plugged-in a PC to be recovered.

2.6 Handling Data Generated On The Device

While we can proxy the data that is already available 'in the network', we cannot do so 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 4 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, of-

ten 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 yet applied.

Pictures and videos taken using the device may incur significant overhead in transmission. *PA* caters more to security sensitive environments like corporations and government institutions, where such data are encountered less frequently. Nevertheless, 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 also proxy the uploaded data and make them available to the replica.

3. IMPLEMENTATION

In this section, we discuss a prototype implementation of *PA* for Google's Android system. While it is possible to implement the *tracer* and *replayer* 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 to serialise accesses to shared objects (discussed in 2.1). However, it is hard to maintain such an implementation, as it requires frequent updates to keep it operational with new kernels, and it requires that a new boot image is installed on the device every time the *tracer* is updated. This motivated us to implement *PA*'s prototype in user space.

Our implementation is transparent to applications and the OS, and only requires process tracing functionality, comparable to the one offered by Linux's *ptrace*, which enables us to attach to arbitrary processes, and intercept system calls and signals. Similar interfaces are also support by BSD- and Windows-style OSs used on other devices, such as the iPhone OS and Windows Mobile.

3.1 Recording And Replaying

In this section, we explain the novel aspects of implementing execution recording and replaying on Android.

3.1.1 Starting The Tracer And Everything Else

In UNIX tradition, Android uses the *init* process to start all other processes, including the supporting framework and user applications. 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 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 calls *exec* to start the appropriate binary with the corresponding parameters.

3.1.2 Scheduling And Shared Memory

In Section 2.1, we briefly mentioned that we serialise accesses to shared objects using a modified task scheduler that operates in a deterministic way. Unfortunately, we can only do so with coarse granularity, as we operate entirely in user space. Our scheduling algorithm is quite simple and far from optimal, but sufficient for our purpose, as it is reproducible. Furthermore, it does not require us to log any additional information in the execution trace. It operates by ensuring 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, Android does not use spinlocks as they are wasteful in terms of CPU cycles. Instead, locking is performed using mutexes, which results in 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 traditional CREW 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 a modified CREW protocol that will track 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 user space. Fortunately, we have had no need for this in our implementation.

3.1.3 Ioctls

I/O control, mostly performed using the *ioctl* system call, is part of the interface between user and kernel space. Programs typically use ioctls to allow userland code to communicate with device drivers. Each 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. Unfortunately, due to backward compatibility issues and programmer errors actual ioctl numbers do not always follow this convention. Furthermore, Android performs most of its IPC through the kernel using the binder framework [33]. Many of the binder operations actually result in one or more ioctls on the *"/dev/binder"* device. Thus, it is important that we can access the Android kernel source code to check the semantics of the various ioctls being used. Fortunately, smartphones make use of fewer ioctls than PCs, but the procedure is still a tedious one. Our prototype handles about two hundred ioctl commands.

3.2 Execution Trace Compression

One of our primary goals is to minimise transmission costs, which requires minimising the size of the execution trace. Here we discuss the rules we applied to reduce the size of the trace:

- *Record only system calls that introduce nondeterminism.* Phone and replica execute the same instruction stream, so there is no need to record system calls that have identical effects in both (*e.g.,* creating a socket, opening a file, reading from local storage, *etc.*). We also do not record the results of systems call used for IPC between processes, as the mirror processes on the replica will generate the same data.
- *Use a network proxy so that inbound data are not logged in the trace.* The data received by the phone over the network are not directly seen by the replica (*e.g.,* a received email). We introduce a transparent proxy that logs all Internet traffic towards the phone, and makes it available to the security server upon request. This way the phone does not need to waste precious energy to log and transmit them to the replica.
- *Compress data using three algorithms.* First, we encode time related data returned by calls such as *gettimeofday* and *clock_gettime* using delta encoding, replacing the actual time data in the trace with the differences of consecutive values. Second, we employ Huffman encoding to represent frequent values in the trace. For instance, we use a bit to represent a system call that returned zero, another one to indicate that a log entry has been produced by the same thread as the previous entry, *etc.* Finally, we employ general purpose compression using the DEFLATE algorithm (also used by the *gzip* utility).

3.3 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.

3.3.1 Virus Scanner

One of the security measures we apply at the server is antivirus scanning. For this purpose, we modified the popular open source anti-virus ClamAV to run 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 by Oberheide *et al.* [30].

3.3.2 Dynamic Taint Analysis

PA can go a lot further than just running multiple antivirus software in the cloud. We modified the Android emulator to apply dynamic taint analysis (DTA) on the replica [10, 27]. DTA is a powerful, but expensive method to detect intrusions. The technique flags all data that arrive from a suspect source (like the network) as tainted. Tainted data are tracked throughout the execution of the system, so that all data the depend on tainted data are also flagged as tainted. For instance, when tainted values are used as source operands in ALU operations or copied, the destination is also tainted. When an attacker exploits a vulnerability (*e.g.,* a buffer overflow, a format string attack, a dangling pointer, *etc.*) to inject and execute arbitrary code, or simply arbitrarily redirect the execution flow of the vulnerable program (*e.g.,* using return-to-libc, or return oriented shellcode), DTA identifies the illegal use of tainted data and raises an alert. For instance, an alert is raised when tainted data are executed, or used as an operand of a *CALL* instruction.

DTA works against a host of exploits, including zero-day attacks, and incurs practically no false positives. Unfortunately, the overhead imposed is very high, making it an impractical solution to deploy on production systems (both PCs and phones). By applying DTA on a smartphone's replica, we manage to hide its overhead from the end user, and concurrently exploit the more powerful hardware in the cloud to accelerate it.

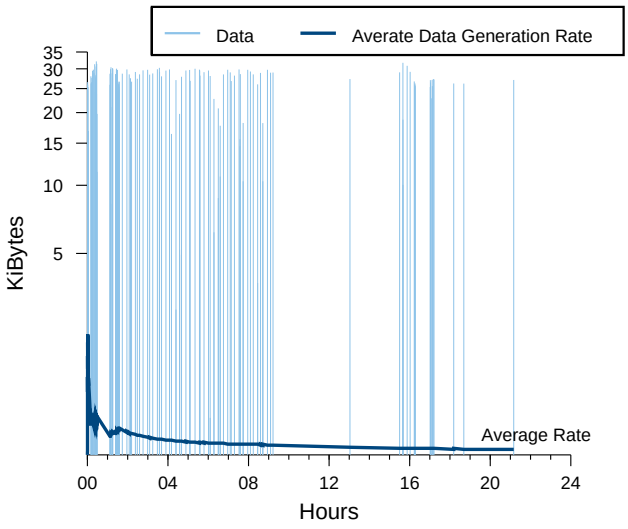

Figure 2: Data generated by *PA* on a user operated HTC G1 for a day.

4. EVALUATION

We evaluate 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 run the *tracer* on the HTC G1 developer phone, while the *replayer* is hosted on the modified QEMU [1] emulator that comes with the official SDK. We do not perform a security evaluation of our taint analysis implemention on QEMU, as it has been sufficiently demonstrated by [35].

4.1 Data Volume

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

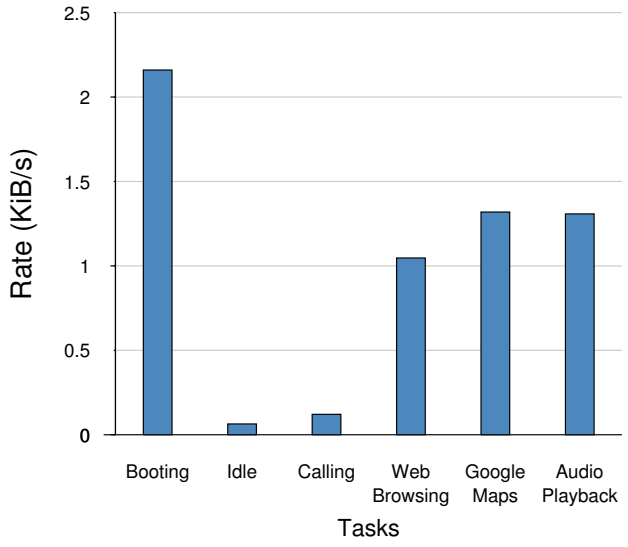

Figure 3: Average data generation rate, when performing various tasks.

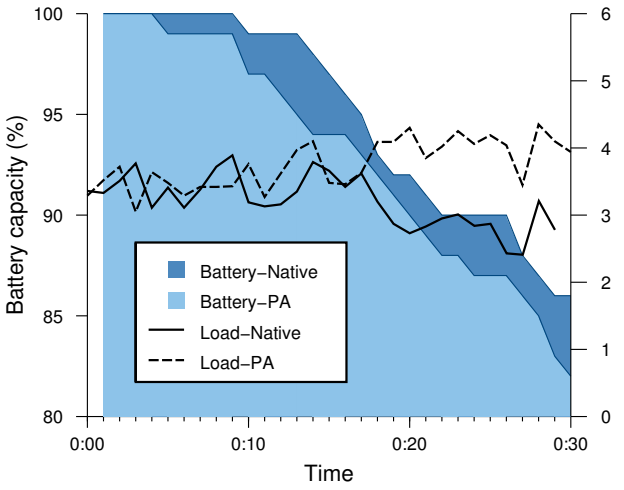

Figure 4: Battery level and CPU load average while browsing on the HTC G1 developer phone. We draw two independent experiments, where we browse URLs from [7] natively and under *PA*.

available download bandwidth.

Our traces collected from actual users using their phones show, not surprisingly perhaps, that mobile devices are mostly idle, or used for voice calls. A plot of the amount of data generated by *PA* over time is shown in Figure 2. Meanwhile, Figure 3 shows that the data generated when the device is idle or the user is making a call is negligible, with an average of 64B/s and 121B/s respectively. Even when performing more intensive tasks, such as browsing or listening to music, the *tracer* generates less than 2KiB/s. For instance, 5 hours[2] of audio playback would generate about 22.5MB of trace data. Transmitting such an amount of data solely over 3G may burden users with excessive costs, specially when operators cap their bandwidth and charge extra for data transfers over the cap, but it can be easily stored locally on the smartphone (devices already offer relatively large amounts of storage; *e.g.,* the iPhone 4 offers 32GB of storage) until a WiFi connection is available. Taking into account that many users spend most of their time at home or at the office, it is very likely that WiFi connectivity will be frequently available to synchronize with the server.

4.2 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 transmission of the execution trace to the server. To quantify these costs, we monitored the device's CPU load average, and battery level, while randomly browsing URLs from [7]. We performed this task natively as well as under *PA*, and show the results in Figure 4.

Figure 4 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 justify for such a divergence. The figure also shows how battery capacity drops in time while browsing. As expected power is consumed faster when using *PA*. When idle or in light use, the additional battery consumption is minimal, but heavyweight tasks, such as browsing may consume up to 30% more energy.

However, most of this overhead seems to be due to the additional CPU cycles consumed by the user space *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 level. We investigate the cause behind this increase in CPU load and battery consumption, and discuss our findings in 4.4.

4.3 Server Scalability

Chun *et al.* [6] has shown that simply moving computation from a smartphone to faster hardware such as a PC, even when running on an emulator, can increase performance up to 11.8 times. While we cannot replay execution faster than it is recorded, the significant difference in processing power between smartphones and PCs enables us to host multiple replicas on each security server.

We corroborate our assumption by measuring the number of phone replicas that can be run concurrently on various hardware configurations. Each replica was run on the Android Qemu-based emulator, executing the same task as the original. It is also in-sync with the replayed device, *i.e.,* the replica has to wait for trace data from the device. While running the replicas, we did not introduce any detection mechanism or instrumentation, which represents an optimal scenario for this experiment. The results are shown in Figure 5, where we draw the number of replicas that can be run under these conditions using different hardware configurations. Particularly, we used a dual-core (x2 2.26GHz P8400) HP HDX18 notebook with 4GB of RAM, a four-core (x4 2.40GHz Q6600) desktop PC with 8GB of RAM, and a high-memory extra-large instance on Amazon's Elastic

[2]Typical battery life when browsing or reproducing audio can range from 3 to 8 hours depending on the device.

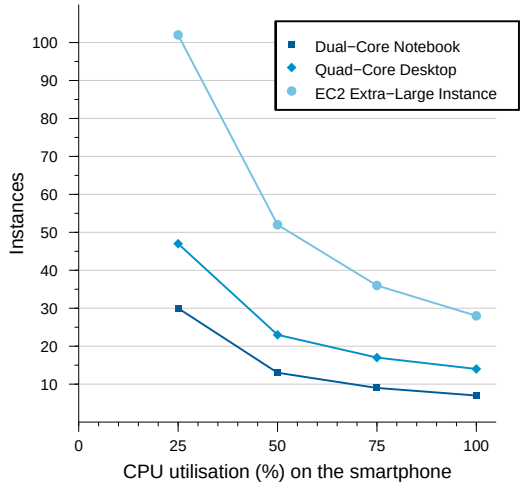

Figure 5: Number of replica instances that can be run on a server without delay. As CPU utilisation increases on the phone, fewer replicas can be executing in sync with the phone.

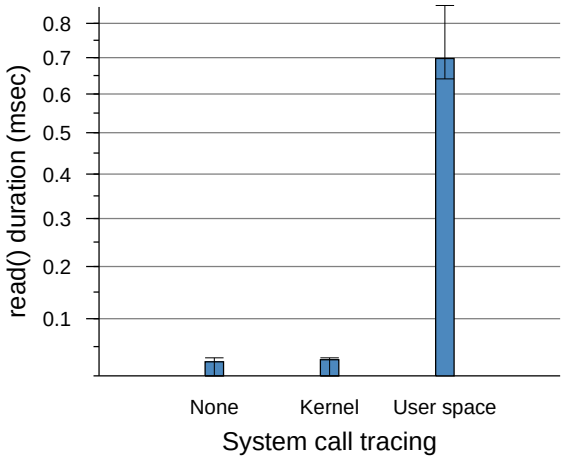

Figure 6: The time it takes to read 4Kbytes of data from */dev/urandom* natively, and when tracing.

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: Top executing functions in the *tracer*.

Cloud (EC2) service with 68.4GB or RAM. *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, since it determines the number of replicas that can be run without delays, as computation is relatively expensive when running under the emulator.

Determining the average CPU utilisation of smartphones in a realistic scenario is not a trivial task, and we are not aware of any preexisting studies on the subject. Nevertheless, we can look at the intensity of different tasks commonly performed on these devices. For instance, on the HTC G1 developer phone we measured 90%-100% CPU utilisation when running a game, 20%-25% when reproducing audio, 30%-100% when browsing, and finally 0%-5% when the phone is idle. 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.*).

We already mentioned that the results presented in Figure 5 present an optimal scenario, as no security mechanism is applied. *PA*'s scalability actually depends on the type and number of mechanisms introduced at the server. For instance, previous work that implemented DTA for the x86 architecture using the Qemu emulator reported a x2-x2.5 slowdown compared with execution under Qemu alone. We obtained similar results implementing DTA for the ARM architecture using Android's Qemu-based emulator. As such, we estimate that if DTA is applied on every replica, we would be able to run roughly half of the instances reported in Figure 5 without any delay. Finally, we have tested our scheme on Amazon's EC2 cloud service to demonstrate the scalability of our approach. In practice, organisations that are willing to invest in smartphone security, will most probably host their own security servers as well as proxies to ensure that privacy sensitive data remain within the organisation, and to reduce costs[3].

4.4 Overhead Imposed By Ptrace

We mentioned earlier that we observed an increase in CPU load and consequently battery consumption under *PA* that was unexpected. We investigated further by profiling the *tracer* to identify its "expensive" functions, and list the top five functions in Table 1. We see that compression (performed by *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, we use ptrace at least once for every event to retrieve additional information (*e.g.,* the system call number). These two calls cause a large number of context switches between the *tracer*, traced processes, and the kernel, and incur the larger 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).

We are confident that moving event reception and the initial part of event handling of *PA* in the kernel, would greatly improve performance. This is supported by what we see in Figure 6. Even when tracing a single system call like *read*, using *ptrace* incurs a huge overhead when compared with

[3]Products that offer data proxying are already available for devices like BlackBerry smartphones [3].

native execution. On the contrary, tracing the same system call, including copying the returned data, within the kernel imposes no observable overhead. Future work on *PA* will focus on moving part of the implementation in the kernel.

5. RELATED WORK

The idea of decoupling security from execution has been explored previously in a different context. Malkhi *et al.* [22] 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 additional engineering.

Ripley [40] is another system that proposes the replication of an application in a server to automatically preserve its integrity. Unlike *PA*, it focuses on distributed web 2.0 applications, and particularly AJAX based applications. Attacks are detected by comparing the results of the replica with the client's. A discrepancy indicates an attack, so Ripley is in fact investing on the two executions deviating. Furthermore, it does not apply to a broad range attacks like *PA*, and it is not transparent to the application.

The idea of off-loading execution from smartphones to the cloud was first proposed in CloneCloud [6]. The main focus of this work is the acceleration of CPU intensive and low interaction applications. While the authors recognize its potential use for decoupling security from phones, they do not investigate the effects of disconnected operation on security (*i.e.,* the need for secure storage), nor do they investigate the cost of replication for the phone and the server. Finally, CloneCloud is not always transparent to applications.

Decoupling security from smartphones was first explored in SmartSiren [4], 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.* [30] 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 *PA*, and we show that such a design is feasible and useful. However, both these approaches can only protect against a limited set of attack vectors.

Other work on smartphone security includes VirusMeter by Liu *et al.* [21]. This work also identifies that traditional defences do not perform as well on smartphones due their limited resources. They propose using power consumption levels to identify potentially malicious software operating on a smartphone. Their solution uses very little resources, but it may incur false positives. Enck *et al.* address the issue of malicious applications downloaded on smartphones with Kirin [13]. They propose a system that can automatically analyse applications submitted to application stores (*e.g.,* Google's *Marketplace* and Apple's *Apple Store*) for potentially malicious behaviour. Kirin is orthogonal to our system, and could in fact be used in combination.

Our architecture also bears some similarities to BugNet [26] 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.

6. 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 apparent, preferably using multiple and diverse attack detection measures. Our security model 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 locally, the phone records a minimal execution trace, and transmits it to the security server, which faithfully replays the original execution.

The evaluation of a user space implementation of our architecture *Paranoid Android*, shows that transmission overhead can be kept well below 2.5KiBps 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 our architecture is suitable for protection of mobile phones. Moreover, it offers more comprehensive security than possible with alternative models.

Acknowledgments

This work has been supported by the European Commission through projects FP7-ICT-216026-WOMBAT and FP7-ICT-257007 SYSSEC. Also, with the support of the Prevention, Preparedness and Consequence Management of Terrorism and other Security-related Risks Programme European Commission - Directorate-General Home Affairs. This publication reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

7. REFERENCES

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

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

[3] BlackBerry, Inc. BlackBerry Enterprise Server. http://na.blackberry.com/eng/services/business/server/full/.

[4] 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.

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

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

[7] A. T. W. I. company. Top 500 global sites. http://www.alexa.com/topsites.

[8] 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.

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

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

[11] 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.

[12] 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.

[13] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *Proc. of CCS*, pages 235–245, 2009.

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

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

[16] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek, and Z. Zhang. R2: An application-level kernel for record and replay. In *Proc. of OSDI*, 2008.

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

[18] 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.

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

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

[21] L. Liu, G. Yan, X. Zhang, and S. Chen. VirusMeter: Preventing your cellphone from spies. In *Proc. of RAID*, pages 244–264, 2009.

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

[23] 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.

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

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

[26] 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.

[27] 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.

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

[29] 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.

[30] 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.

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

[32] 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.

[33] I. PalmSource. OpenBinder. http://www.angryredplanet.com/~hackbod/openbinder/docs/html/index.html, 2005.

[34] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos. Paranoid Android: Zero-day protection for smartphones using the cloud. Technical report, Vrije Universiteit Amsterdam, 2010.

[35] G. Portokalidis, A. Slowinska, and H. Bos. Argos: an emulator for fingerprinting zero-day attacks. In *Proc. of ACM EuroSys*, April 2006.

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

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

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

[39] V3.co.uk. BlackBerry 'kill pill' vital for IT security. http://www.v3.co.uk/vnunet/news/2159105/blackberry-kill-pill-vital.

[40] K. Vikram, A. Prateek, and B. Livshits. Ripley: automatically securing web 2.0 applications through replicated execution. In *Proc. of CCS*, pages 173–186, 2009.

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

[42] 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.