DIMACS Technical Report 2010-03
February 2010

Assured Detection of Malware With Applications to Mobile Platforms

by

Markus Jakobsson[1] and Karl-Anders Johansson
FatSkunk Inc
590 Mariposa Ave
Mountain View, CA 94041

[1]Permanent Member

DIMACS is a collaborative project of Rutgers University, Princeton University, AT&T Labs–Research, Bell Labs, NEC Laboratories America and Telcordia Technologies, as well as affiliate members Avaya Labs, HP Labs, IBM Research, Microsoft Research, Stevens Institute of Technology, Georgia Institute of Technology, Rensselaer Polytechnic Institute and The Cancer Institute of New Jersey. DIMACS was founded as an NSF Science and Technology Center.

ABSTRACT

We introduce the first software-based attestation approach with provable security properties, and argue for its importance as a component in a new Anti-Virus paradigm. Our new method is practical and efficient. It enables detection of *any* malware (that does not commit suicide to remain undetected) – even if the infection occurred *before* our security measure was loaded. Our new approach works independently of computing platform, and is eminently suited to address the threat of mobile malware, for which the current Anti-Virus paradigm is poorly suited.

Our approach is based on *memory-printing* of client devices. Memory-printing is a novel and light-weight cryptographic construction whose core property is that it takes *notably* longer to compute a function if given less RAM than for which it was configured. This makes it impossible for a malware agent to remain active (e.g., in RAM) without being detected, when the function is configured to use all space that *should* be free after all active applications are swapped out. Our approach is based on inherent timing differences for random access of RAM, flash, and other storage; and the time to communicate with external devices.

Keywords: anti-virus, audit, detection, handset, infection, lightweight, malware, memory-printing, mobile, post-mortem, retroactive, software-based attestation, timing

1 Introduction

Current Anti-Virus (AV) software works in a similar way to how TSA personnel screens air travelers as they enter the airport – based on their identification documents, belongings and behavior. This is a labor-intensive approach that needs constant updates of blacklists, and in which a single security breach can result in a undetectable (and therefore irrevocable) damage. After a successful infection, the malware agent can hide from the AV software and suppress all future alerts – like a rootkit [29] does. Thus, current AV software offers no guarantees of *retroactive* security. In the world of malware, this is worrisome, given the near-infinite number of ways in which malware can be obfuscated to avoid initial detection [28]. As it comes to *mobile* malware, an added problem is that it is too costly – in terms of system resources – to deploy traditional anti-virus software. Keeping with our analogy, this is similar to how it would be too expensive to deploy TSA personnel for each taxi, bus, and other form of public transportation.

This paper introduces a new Anti-Virus paradigm for which we can offer guarantees of security that are backed by proofs. We are assured detection of malware – including yet-unknown strains – even if the detection algorithm is loaded *after* the infection occurs[1]. To achieve this, we depart from the traditional approach of screening *events*, and instead audit the memory state of the device, comparing this centrally to the expected device state and to whitelists of good programs. Any discrepancy is evidence of corruption. An analogy of our approach is that of an air marshal that at given time intervals demands of all travelers to take a step out on the wing, reporting to an authority on the ground who was there, providing them evidence that the plane is empty – and first then, allows everybody to come in again. Our solution works even if the air marshal was delivered to the plane after takeoff.

To avoid forged "all clear" reports produced by malware agents wishing to avoid detection, we introduce the notion of device memory-printing – a form of fingerprinting of device contents, whose security guarantees are derived from the time it inherently takes to access memory of various types. It is different from previous timing-based attestation methods (e.g., [23, 24]), and does not suffer their limitations and vulnerabilities [4].

Our contribution. In contrast to the traditional software-centric approach to malware defense, we leverage knowledge of the hardware specifics to achieve our security guarantees. Supported by knowledge of bounds of how long it takes to read from and write to RAM, flash memory and other storage resources, we describe an algorithm that transforms the memory contents to a state that is then reported to a central authority. This process is carefully timed by this authority. In order to remain on the device, a malware agent either has to be active in RAM or modify legitimate programs in RAM, flash or other storage. Doing the former, we show, introduces significant delays to generate the output expected from our algorithm, and doing the latter causes immediate detection when the memory contents are inspected.

To avoid detection, a malware agent would have to quickly evaluate a given function of the entire memory contents of the corrupted device. To hide active code – which *must* take

[1]This paper deals with *detection* only, and not *removal*, which is an orthogonal issue.

up space in RAM – the malware agent either has to compute the expected contents of the cells it occupies, store these values elsewhere, or have the function results – or parts thereof – computed by an external device. Our solution is designed in a way that guarantees that any of these cheating strategies must take notably longer to carry out than the "legitimate" evaluation of the function. These guarantees are based on known physical limitations of the hardware used.

Our solution is based on a non-homomorphic function with a memory access strategy that severely – and inherently – increases the delays associated with accessing flash memory. It also uses periodic re-keying to avoid outsourcing of the entire task to a fast, external processor.

Our solution relies on the central authority being able to identify communication as coming from the intended client device, and to communicate securely with it. This can be achieved using a SIM card whose contents cannot be cloned (by a malware agent); or by a trusted path, such as what can be afforded by a physical communication link.

The estimated time for the software attestation is on the order of a minute. The camera-ready version will report on the results from an ongoing Android implementation of the proposed method, and will provide exact information on the time it takes perform an audit.

Outline. We begin by reviewing the related work (section 2). We then describe our assumptions and outline our approach (section 3), and detail the possible adversarial behavior (section 4). This is followed by an overview of relevant hardware characteristics (section 5). In section 6, we provide a detailed description of our solution. We argue why our solution is secure (section 7). We conclude and overview ongoing and future work in section 8. In the Appendix, we provide proof details and a detailed description of hardware parameters of importance.

2 Related Work

It is believed that less than 50% of networked computers have up-to-date AV protection [20], and that 12-25% are infected by malware [1, 7]. Malware is often used to circumvent other security measures (see, e.g., [9]). To many consumers, security against malware is complex and frustrating, and the concerns make many hesitant to engage in mobile commerce activities [10]. Yet, without good technical solutions, users do not manage their security well [16]. The threat is expected to be aggravated within 2-3 years as smartphones become the dominant computing platform [11, 26]. It is commonly believed that malware authors will target smartphones in earnest when this happens [12, 19, 18].

Windows machines currently receive around a hundred updates a day, to account for close to forty thousand new and unique malware instances a day. The problem is escalating [15] as organized crime is moving in [5, 17, 25]. Since resource constrained devices will be strained to just receive updates at a pace matching the appearance of new malware varieties – and drained by performing the associated computation – it is clear that today's AV paradigm will not address the problem. Progress in battery developments does not promise to solve

this problem. Consequently, there is an increased need for lightweight AV protection, e.g., using centralized detection [14, 13, 21].

Whereas centralized threat detection can easily be built for some services, such as email and SMS, it is harder where connections may be encrypted or where the threats use Bluetooth to propagate [3]. One approach is to log all events on the client, and audit the logs remotely. Whereas a straightforward approach would be vulnerable to logs being tampered with by the malware agent, it is possible to use cryptographic audit techniques to address this problem [14]. The question remains of what types of events to log, though, and how to detect malware whose attack vectors makes it difficult to log in an efficient manner. Attacks relying on buffer overflow, for example, pose the thorny problem that whereas it is not realistic to log *all* activity, it may be difficult to know what activity can be ignored. An alternative approach, as we propose herein, is to audit *state* instead of *events*.

Gratzer and Naccache [8] proposed to write a pseudo-random string to a smart card, and then read it back and verify it in order to obtain an assurance that the device is not pre-loaded with undesirable code. Their security guarantees hinge on the fact that pseudo-random strings cannot be compressed – at least not a whole lot. While we also write pseudo-random strings to memory, our security guarantees do not rely on the difficulty to compress the pseudo-random strings, but on the time it takes to *compute* or *access* them. While similar at a high level, the solutions differ not only in terms of the underlying security guarantees, but also in terms of the achieved functionality. For example, Gratzer and Naccache's solution does not address situations in which client devices are able to communicate with their surroundings while being audited, but we do. Finally, and due to the fact that they communicate the entire pseudo-random string to the client device, and not a seed as we do, their solution is only practical for client devices with very limited amounts of memory. Otherwise, the bandwidth limitations associated with typical devices make the verification process too slow to be practical.

Seshadri et al. [23] propose a timing-based approach to heuristically assess the integrity of legacy computers. They compute a keyed checksum of the code that computes the checksum. (In contrast, we compute a checksum on all *but* our checksum code.) Their checksum is not only a function of the checksum code, but also of the program counter, the data pointer, and of CPU condition codes. A verifier determines whether the code was tampered with based on the result of the checksum computation and the time it took to perform it. After the checksum code has been established not to have been tampered with, control is handed over to a function that scans the entire device. In contrast to our solution, their solution does not protect against fast, external attackers that help an infected device perform the verification. Also, given their heuristic approach, they are not able to provide any security proof but instead argue how a collection of tried attacks were successfully detected.

Seshadri et al. [24] also propose a timing-based approach to scan embedded devices. Their solution does not address devices that can communicate with their surroundings (other than with the verifying device), and is therefore not suitable to address malware on typical mobile devices, such as smartphones. Some vulnerabilities of their solution were recently pointed out by Castelluccia et al. [4]; these are based on the fact that the audit code could be written

in a more compact manner.

Dwork, Goldberg and Naor [6] introduced the notion of memory-bound functions, i.e., functions whose time to compute depend more on the bus (and memory) speed of a device than on its processor speed. This was used to create so-called medium hard functions, with applications to spam prevention and, in general, access control of valuable resources. We introduce a new type of memory-bound function, which we refer to as a *memory-printing* function. It has the property that the speed to evaluate it depends strongly on the *type*, *quantity* and *location* (i.e., internal or external) of memory used. This allows us – using a timing-based approach – to ascertain that only fast internal memory is accessed when the function value is computed, and that no flash or other memory is used. This helps us create assurances of what processes are active at the time of the memory audit.

3 Overview

The security of our solution rests on two important assumptions:

Assumption 1: Secure device communication. We assume that the verifying party has some way of ascertaining that the device to be audited is in fact the device it interacts with[2]. We also assume that the verifying party can send data securely to the audited device, e.g., in a way that cannot be eavesdropped.

This can be achieved using encryption / authentication using a *device-specific key*, embedded in a SIM card. This key would be used to decrypt incoming messages and authenticate outgoing traffic, but cannot be read by malware.

It can also be achieved using a *physical assurance* that the correct device is being audited – e.g., requiring wired connection of the client device to the verifying agent or a trusted proxy thereof.

Assumption 2: Code optimality. We assume that the memory-printing algorithm is written in a near-optimal manner in terms of its footprint and execution speed, and that any modifications of the algorithm would make it notably slower to execute. For general software, this is not a meaningful assumption to make; however, given the simplicity of our memory-printing algorithm, it is quite realistic.

Definition: *Free* RAM. Our malware detection algorithm is implemented as a kernel/algorithm monolith that is stored in the instruction cache (where it fits in its entirety). It has an associated working space that is located in the data cache (and registers.) All other RAM[3] space is referred to as *free* RAM – whether it actually is free or not.

What is done: The malware detection algorithm involves the following steps on the client machine:

[2]Note that we make no assumptions regarding whether an infected client device outsources part of the task assigned to it by the verifying party; this will be clarified onwards.

[3]The cache is typically implemented using RAM, and is considered as being part of it herein.

1. **Setup I:** Swap out the contents of *free* RAM to flash, and perform a setup for the memory-printing (detailed in section 6.)

2. **Setup II:** Receive a cryptographic seed from the verifying party, and overwrite *free* RAM with the output of a pseudo-random function using this seed.

3. **Memory-printing I:** Receive a cryptographic seed from the verifying party, and use this to key a non-homomorphic function whose output is written to all *free* RAM. We detail this step in section 6. This process is timed by the verifier, both in its entirety and for shorter intervals. The verification is based on partial results that are transmitted to the verifier, where they are checked.

4. **Memory-printing II:** Receive a cryptographic key from the verifying party, and compute a keyed cryptographic hash [2, 22] of the entire memory contents, reporting the resulting value to the verifying party. This process is also timed by the verifier. The verifier compares the received value to a locally computed keyed hash of the expected contents.

5. **Policy:** Execute the verification policy. This could involve reporting[4] the memory contents of the client device to the verifying party, or the execution of any task that should preferably be run in a safe environment. This may be done using whitelisting or blacklisting approaches, and may involve the use of heuristics to scan the swap space and secondary storage in general. *The implementation of the policy is beyond the scope of this paper, as we only deal with how to assert the absence of active malware herein.*

6. **Restore:** Restore RAM state by loading the contents that were swapped out during *setup I.*

In addition, the client machine will report state information from its computation in steps (3) and (4) at time intervals set by the verifying machine, and obtain updates of the seed resp. key used to compute these functions. These updates will be generated by the verifying party, and communicated to the client device on an as-needed basis.

The verifying machine will verify that the correct results – both final function value and partial results – are reported, and that these values are reported within acceptable time bounds.

Why it is done: Step 1 enables the restoration of context after the verification has completed. Step 2 simplifies the synchronization of state between the client and verifier at the same time as it provides random content later to be accessed and modified.

In step 3, *free* RAM is filled with a pseudo-random string that depends on keys obtained from the verifier at regular intervals. This function takes notably longer to execute if it is modified to use flash instead of RAM.

[4]This does not have to be done by communicating the entire memory contents to the verifying party; instead, only descriptions of changes since the last audit have to be communicated.

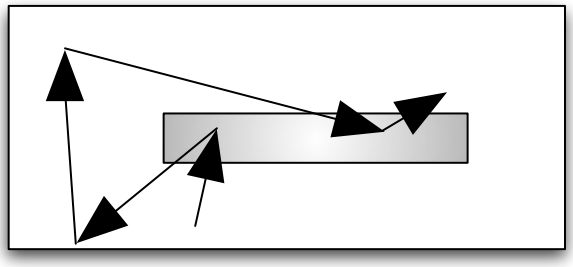

Figure 1: The figure illustrates the principles of memory-printing. All fast memory is accessed in a pseudo-random order, with a sequence of reads and writes to the accessed positions. The gray memory portion is used by malware, which will stop itself from being overwritten in order to survive. Later on, a keyed hash of the entire fast memory is computed. For this result to be correct, the "intended" contents of the grey portion have to be stored in secondary memory or recomputed on the fly, either of which causes a notable delay. That alerts the verifying server of the infection. The way memory is accessed intentionally causes dramatic slowdowns if flash is used instead of RAM. Additional techniques are used to cause dramatic slowdowns if external, wireless resources are used to store or compute needed values.

In step 4, the verifying party is given assurance that steps 2 and 3 were performed correctly, based on a function of the string computed in step 3, and the time it takes to compute this function. (We note that the timings will typically be performed over a lossy network with variable latency; this will be taken into consideration when the security determination is made.)

If the verification (of both results and timings) succeeds, then the verifier knows that there is no active malware on the client. Therefore, the result of the verification policy in step 5 is known to be valid, as it cannot have been tampered with. In step 6, the state from step 1 is restored.

The periodic timing checks bring assurance that the computation is performed fast enough. In particular, it guarantees that the pseudo-random string is truly stored in RAM (as opposed to the slower flash), and that the reported results are not computed by an external fast computer.

The use of frequent re-keying incurs round-trip communication delays for any externally supported communication. Namely, to make outsourcing possible, the malware agent would have to forward the seed / key updates to the external device, which would introduce a measurable delay. The exact delay depends on the latency of the network, but we will pessimistically assume that the delays are as short as they typically get on the given type of network.

An implementation note. To minimize the footprint of our malware detection algorithm, we can let the code consist of two components; one *loader* and one variable *algorithm segment*.

The task of the loader is to load algorithm segments from non-RAM storage, and hand over control to the loaded algorithm segment. After an algorithm segment has completed, it hands back the control to the loader. We will see the importance of this later.

We note that the techniques described above do *not* guarantee that the correct monolith kernel is run. Malware may, for example, suppress the entire execution of the audit code. However, the associated silence will be indicative of infection.

4 Adversarial Strategies

The malware agent needs to do one out of two things to remain resident on an infected machine. It either (a) has to remain active in RAM or swap space, or (b) modify legitimate programs, data or configurations of the client device to allow the malware agent to gain control after the audit process has completed.

To remain undetected in RAM, the malware agent needs to cause the verifier to accept the memory-printing computation, which requires that the correct responses are produced within the correct time bounds. Alternatively, to modify contents of secondary storage without being detected, the malware agent could corrupt the reporting of state (step 5 of the solution, as described in section 3). This requires being active in RAM at the time of step 5, whether as a unique process or as part of a corrupted version of the detection monolith kernel.

Therefore, *both* of the adversarial approaches above – (a) and (b) – require the malware agent to remain active in RAM and produce the right results within the right time bounds. The principal approaches a malware agent can take to achieve this are as follows:

Strategy 1: Outsource storage. The malware agent can rely on non-RAM storage or external storage to store (at least) the portion of the pseudo-random string generated in step 3 that was intended to be stored where the agent resides. The computation of the partial results used to time the execution would then be modified to use the outsourced storage instead of the space where the malware agent resides. In particular, a malware agent could use flash instead of RAM when computing parts of the memory-printing function.

Note that the malware agent does *not* need to maintain the same memory access structure as what is intended in the memory-printing function: What normally would have mapped to one and the same page or block could be stored in different pages or blocks, if this decreases the expected turn-around time. It also does not have to write back data to the cell where it came from, but can temporarily write it elsewhere, only to combine data later on. This may be beneficial for the adversary to attempt in order to avoid the delays associated with rewriting an entire flash block (see Appendix A.)

Strategy 2: Compute missing data. Instead of outsourcing storage of portions of the pseudo-random string, the malware agent can store a modified representation of this string (e.g., compressed or missing portions of the string). It can then attempt to reconstitute relevant portions of the string as they are needed during the computation of the temporary

values (step 3, detailed onwards) and the keyed hash (step 4). Since the malware agent has the seed from which the pseudo-random string is generated, it can use this – or later states – to regenerate required portions of data.

Apart from the computational delay this causes, it is also severely complicated by the fact that memory writes implicitly destroys old state.

Strategy 3: Outsource computation. A malware agent can forward relevant data to an external device, which we will assume has infinite[5] computational power and unlimited storage. The external device will receive data from the client device and compute the values that need to be reported to the verification authority, feeding these values to the malware agent on the infected client device.

We will pessimistically assume that the communication channels available to the client device will be used in an *optimal* manner to communicate data between the client device and the external colluding device.

Strategy 4: Modify detection code. A malware agent can attempt to replace the monolith kernel code of the detection algorithm with modified code. This malicious code may be designed to suppress reports of compromised memory contents, or contain a hook for malware code to be loaded after the audit completes. The malware agent may attempt to incorporate these changes in the legitimate monolith kernel code without taking up more space by swapping out or compressing portions of this code, loading or unpacking it again as it is needed.

5 Hardware Characteristics

In this section, we will review the distinguishing characteristics that describe the different memory and network types of relevance; this is done in the context of the algorithm described in the next section. We provide a more detailed exposé in Appendix A.

Memory access. We use the term *chunk* to refer to the minimum amount of data that can be sent on the memory bus. For the Android G1 phone and many other computing devices, a chunk is 32 bits. We may sometimes refer to the chunk as a *32-bit chunk* for clarity. We are concerned with the time it takes to read and then write such a 32-bit chunk to various types of memory. Here, the locations for the read/writes are selected in a manner that intentionally does not allow an amortization of costs over consecutive operations.

On an Android G1, we have estimated the following access times (please refer to Appendix A for details and assumptions): It takes 5ns to read or write a 32-bit chunk it the data is in RAM cache, and 20ns to read or write if in regular non-cached RAM. Reading from on-board NAND flash using non-standard methods could *theoretically* be performed in approximately 1 μs (50x RAM time) and a write an be performed in approximately 2μs (100x RAM time).

[5]It is evident that these assumptions are not realistic; more reasonable assumptions, however, will only weaken the adversary's chances of avoiding detection. We make these assumptions to simplify the analysis of the attack and its likely chances of success.

If a block needs to be erased prior to writing the chunk, an additional 2ms penalty is added, totally dominating the write time. Faster external SD cards (30MB/s read/write) could - again, theoretically - allow for a chunk to be read/written in 133ns (6-7x RAM time) while maintaining the 2ms penalty for block erase.

Thus, *when accessed in the manner we do*, we see that access to RAM is dramatically faster than any of the alternatives available to an adversary. For more details, we refer to Appendix A.

Radio usage. The one-way latency time to communicate a basically empty UDP packet (header only) over Bluetooth is 15ms; over a local WiFi network 7ms; using 3G (WCDMA/HSDPA) 80ms. Note that out of the typical 80ms latency for 3G, the Time Transmit Interval (TTI) is about 2-10ms. This can be thought of as the latency seen between the handset and the cell tower. 4G/LTE is estimated to have total latency of only 5ms.

Early estimates indicate that we can send a short UDP packet every 5-10ms over 3G. The camera-ready version will include a more detailed analysis of network throughput.

The algorithm in our proposal does not use WiFi or Bluetooth, and therefore swaps out the platform code to support these features. However, we must consider the possibility that a malware agent blocks this in order to be able to take advantage of fast, local wireless communication.

The shortest possible roundtrip time for external wireless communication – given optimal conditions – is currently 14ms for a small packet using WiFi on a local network.

6 Our Solution

In the following, we will describe how memory-printing works. We will first focus on the description of the memory-filling technique. We will then describe how the periodic timing is performed.

6.1 Filling Fast Memory

We will now describe a memory-printing function that satisfies the requirements needed to detect the various forms of adversarial abuse. It will be used to fill *free* RAM. (It can also be used to fill other types of fast memory, should these come to compare with RAM in terms of access times in the future.)

Setup and Memory-printing. In order to fill *free* RAM with a pseudo-random string, there are two main steps:

1. First, a setup function is run. (For practical reasons, this is done as part of step 1, as described in section 3.) This determines the random order of memory accesses to be made by the memory-printing function, using a seed obtained from the verifier to generate pseudo-random values. The table is stored in flash, and the program space used by the setup function is cleared after the setup completes.

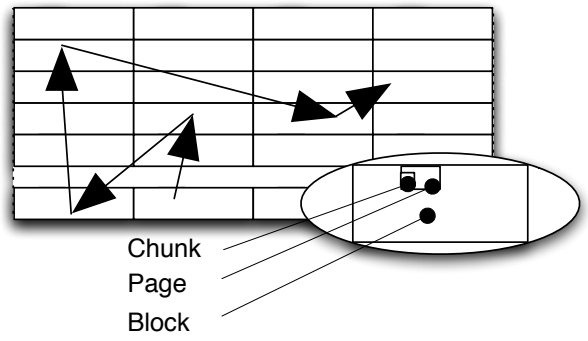

Figure 2: The figure illustrates how memory-printing is performed. A pseudo-random sequence is XORed in to *free* RAM in a pseudo-random order; later, a keyed hash of the entire contents of RAM is computed. Even though RAM does not use blocks and pages, we can divide it into "virtual" blocks and pages, corresponding to those of flash. Note that we do not access consecutive chunks in a page or block – this makes the access slow in flash, but still fast in RAM.

2. Second, the memory-printing function is used to fill all *free* RAM. Its execution is timed, both from beginning to end and in shorter intervals.

These functions are detailed below. Machine code for the Android for the two last functions is provided in Appendix B; the instruction count for these functions is of important to assess the runtime of the memory-printing. (Real timings will also be provided in the camera-ready version.)

Parameters. Let *number_chunks* be the number of chunks in RAM, which is 2^{25} (= 128 MB / 32 bits) for the G1 phone. We assume that the micro-code and its working space are located in the part of RAM with highest-numbered addresses[6]. We let *chunks_taken* be the number of chunks they use. Moreover, *free_chunks* is the difference between *number_chunks* and *chunks_taken*, i.e., the number of chunks that *free* RAM consists of. Finally, *chunks_per_block* is the number of chunks contained in a flash block, equaling 32768 (=128kB/ 32 bits) for the G1 phone.

Setup function

```
% Generates the permuted order of RAM accesses
% Outputs a vector named position
 where ← 0
 for j ← 0 to free_chunks -1
    % not occupied, so no need to jump to other chunk:
    jump[k] ← 0
 for j ← 0 to free_chunks -1
```

[6]This is a simplification to simplify the description; in reality, the working space would be in the data cache, and the code in the instruction cache.

% select a new random position but avoid same block:
where ← (where + random[chunks_per_block, free_chunks - chunks_per_block]) mod free_chunks
% also avoid already taken positions:
where ← where + jump[where] mod free_chunks
% and store the result:
position[j] ← where
% this chunk is now taken, and should be avoided:
jump[where] ← jump[where+1 mod free_chunks] +1

Memory-printing function
% Fills RAM with a random string
state ← 0
for j ← 0 to free_chunks -1
% modify_memory(x,y) XORs the value y
% into the x*th chunk position of RAM.*
modify_memory(position[j],next_chunk)

***next_chunk* function**
% Returns next chunk
state ← contents of the RAM chunk with location (state + seed) mod number_chunks.

About the computation of new chunks. The memory-printing function calls the function *next_chunk*. This function returns a pseudo-random chunk of bits that will be XORed in to the selected memory position. The function *next_chunk* computes a new state and an output from an old state, using a keyed non-homomorphic function. As a concrete example, and as described above, we can let the output be the contents of the RAM memory cell at a position determined by the previous output and the seed. We use a modulo *number_chunks* in the computation of state in the *next_chunk* routine. Using the modulo *free_chunks* instead would guarantee that only pseudo-random content, and no code values, were used to offset memory values in *free* RAM. However, doing so would double the time to execute an iteration of the loop, as it is a more complicated modulo to compute. Occasionally using program code and data as offsets is estimated not to introduce any vulnerability – especially since the adversary cannot anticipate when this will be done.

Note here that a given state or output *cannot* be computed from an initial state using random access. Instead, it requires iterated application of the function.

Code optimality. Given that unused code is erased from RAM, the application footprint at the time of the memory-printing is very small. It consists of (a) the code for the memory-printing function and next_chunk functions (see Appendix B); (b) one page of position values; (c) a routine that communicates with the verifier; and (d) a short loader routine to be run after each phase of the the memory-printing has completed. This minimality is what makes Assumption 2 realistic.

Bad for flash. We note that the above memory access structure causes hits to different pages for each access. This will dramatically increase the cost of a flash-bound computation in comparison to the RAM-bound alternative available to the honest execution of the algorithm.

Execution time. Based on the time to execute the inner loop (see Appendix B), the time to perform memory-printing for a typical smartphone (such as the G1) will be approximately one minute. While this is significantly more demanding that traditional AV approaches, it would only be run occasionally (such as when the device is at rest, potentially as it is being charged.) This would limit the degree to which the detection would affect the user experience. However, and as noted briefly in section 8, ongoing work offers hope that faster memory-printing algorithms are possible to devise.

6.2 Performing Timing

The verifying party times the execution of steps 3-4 of section 3. This is done to identify attempts to outsource storage; compute missing data; and outsource computation.

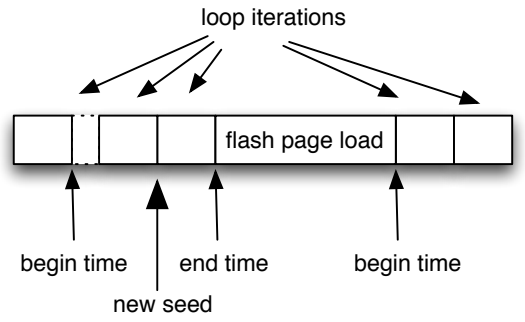

Figure 3: For each round of the loop of the memory-printing, a small number of RAM accesses are made, but no flash accesses. At regular intervals, a new page of RAM positions to modify is read from flash, replacing the previous such page. These scheduled flash accesses do not cause timing delays, as they are known by the verifier, and the timing intervals can be set accordingly. However, "unwanted" flash accesses (i.e., those that are made only by the malware agent) will be detected, as they make the timely reporting impossible. New seeds are dispensed by the SIM card, who receives encrypted lists of seeds and discloses these one by one, after receiving the triggering state information from the handset.

The verifying party will obtain checkpoint state information from the client device at frequent intervals, whose starting and ending points are set by the verifying party. (There can be more than one seed update per time interval, if needed, and the disclosure time can be pseudo-random.) As shown in figure 3, this is done in a way that avoids having *intentional* flash accesses (to load new pages of position vector elements) be counted as delays.

The seed values and associated triggering values are generated by the external verifier, and sent to and decrypted by the SIM card. The checkpoint state information received from the handset is compared with the triggering values, and if two values match, the associated seed value is disclosed to the handset.

The computation can be timed by an external entity, such as the external verifier, or a proxy thereof – e.g., the base station that the handset interacts with. To lower the impact

of latency variance, the timing can be performed by the SIM card. This can be achieved by maintaining a counter on the SIM card, increasing it by one in a loop while waiting for the next value (so-called C-APDU) from the handset, and recording the value of the counter for each such C-APDU[7]. At the end of the computation, the entire vector of checkpoint values and associated counter values would be authenticated and sent to the external verifier.

7 Security Analysis

We will now assess the security of our proposed approach by reviewing each of the adversarial strategies.

Defending against adversarial strategy 1 – outsource storage. Recall that on the G1, each cache access takes 5ns, while each other RAM access takes 20ns. For each iteration of the loop in the memory-printing function, three memory accesses are made from RAM. One of these (loading the position value) is to cache, whereas the other two most likely are not to cache, but to general RAM. Therefore, these memory accesses take at most 45ns.

Based on the sample memory-printing machine code described in Appendix B, the execution of each iteration involves 32 cycles, and *excluding the memory accesses mentioned above*, is estimated to take 35ns. (Recall that the code resides in the instruction cache.)

This results in a total duration of 80ns per iteration of the loop for the *legitimate* execution. We do not count the occasional access to flash to load a new page of position values, since this is not inside the timing interval; see figure 3.

Now, let us consider the additional delay incurred by a malware agent that causes a flash access. Under optimal conditions, and using a modified flash reader (see section 5 and Appendix A), one *read* access may be possible to perform in 1μs for internal flash and 133ns for fast, external flash. For the internal *write*, the delay is 2μs, whereas the external write to fast flash remains 133ns.

The above assumes that the malware agent modifies how flash is read/written. This may not be feasible to do, and is unlikely to be practical. If the malware agents reads flash in a "normal" way, the delays are *considerably* longer.

The above delay is in *addition* to the time to execute the code, which we pessimistically assume will not take any longer than the legitimate code. Thus, the extra delay is approximately 166% of the time to execute an entire loop iteration in the worst-case situation (external, fast flash with modified reading/writing).

If there were no variance in network latency, this would always be detected. However, since the 3G Time Transit Interval is 2-10ms (see section 5), it is clear that a 133ns delay is not possible to identify by an external verifier. (However, to reduce the variance due to TTI, the execution of the time-printing can be synchronized with the transfer of the transport block set. For simplicity, we do not do that herein.)

[7] This cannot be done using standard Java Cards as they only let SIM card applications remain active between a C-APDU and the resulting response, or R-APDU. However, modified Java Cards and proprietary operating system cards can perform this task.

Therefore, we identify the duration not only of *single* timing intervals, but of several consecutive intervals, too. The reason is that delays caused by adversarial flash reads will *not* vanish over several observations, but noise due to latency variance *will.* We note that the malware agent has to send a large number of accessed to flash – maybe on the order of 256 times, corresponding to a 1kB resident code size. Thus, in this example situation, the *accumulated* delay would be around 34ms for each report packet. This would be trivially detected.

The camera-ready version will include a statistical analysis of how many observations are necessary to detect an adversarially caused delay with certainty.

We observe that there are clear time-space tradeoffs for malware authors. Long malware may be able to make reprogrammed, fast flash accesses, but at the cost of larger code size. This translates into a greater number of flash hits, as a bigger program space has to be "defended" by the malware agent. In contrast, short malware is less likely to be able to make advanced flash reads, in which case the flash accesses will be longer. This is particularly so for any approach that is so simple that it does not avoid flash block writes. We refer to Appendix A for more details on timing estimates for these scenarios.

Defending against adversarial strategy 2 – compute missing data. Assume that the malicous code and its working space takes c chunks of *free* RAM. It may also use space assigned to our monolith kernel, and space in flash, but that is irrelevant here.

As soon as the setup routine has been run, the malware agent knows when each cell of *free* RAM will be modified. However, it will not know what values these cells will be modified with, since that depends on the seed values used. It will learn what memory cells values will be read from – and what these values are – very close to the time when the access to its space would be made.

Consider a moment when a given chunk that belongs to the malware agent's code and data space is about to be written. The malware agent has to stop this from happening, or it will be overwritten. However, it must also later be able to retrieve the value that would have been stored in the chunk to be written. It has two options: either (a) it stores information about the address of the chunk that contains the value to be XORed in, or (b) it stores that value. (In addition, it must store some state information from the pseudo-random generator used to initially overwrite *free* RAM, but we ignore that for now.)

Case 1: First, imagine a malware agent that always uses option (a), storing the address of the cell that contains the value. Recall that the malware agent cannot know, a priori, what these addresses are, and therefore, will not know whether their contents will be modified later on or not. If the malware agent stores the address to a chunk whose contents will be modified as a result of later operations, then it will retrieve the wrong value when it needs to generate the value of the chunk.

Since the malware agent cannot move itself in an opportune manner to avoid this – it does not know where to move to until it is too late – we see that this will result in an incorrect output for a given chunk with a 50% probability. Given that the malware agent takes up c chunks, the probability of success is only $1 - 2^{-c}$. For any c larger than, say, 40,

this is not a meaningful attack. At the same time $c = 40$ corresponds to a 160 byte malware space (code *and* data), which is shorter than what is reasonable to even protect itself from being overwritten – much less to carry any payload.

Case 2: Imagine then a malware agent that always uses option (b) and stores the value needed. However, since the value it needs to store takes 32 bits, the malware code would have to add one chunk to its data space for each chunk it wants to "protect". This is not possible since these new chunks of data space will also have to be protected.

Case 3: Finally, imagine a malware agent that uses option (a) when the cell will not later be overwritten, but option (b) otherwise. To be able to achieve this, the malware agent clearly has to know whether the address that the value should be read from *has been* re-written already or *will be* re-written. It needs to know this for a large fraction of addresses – close to 1. This cannot be stored in *free* RAM, as it would take up close to $\frac{1}{32}$ of *free* RAM. Therefore, it has to be stored in flash, and as a result, each time a chunk of the malware agent's space is about to be written, one flash access will be incurred. (Preloading is not meanigful due to the unpredictable memory access order.) Being required to access flash therefore incurs the same cost as strategy 1 – outsource storage – which we know will be detected.

Defending against adversarial strategy 3 – outsource computation. According to assumption 1, it is not possible for the adversary to cut the client device out of the loop. Therefore, without considering the variance of latency due to TTI, we see that the one-way latency incurred by the adversary would be on the order of 5-7ms (for 4G resp. WiFi), with round-trip latencies of twice this duration. Thus, the minimal round-trip duration would still be at least 10-14ms. A new seed value is obtained from the SIM card at a random time of the interval (scheduled by the verifier). If seeds are replaced once per interval, this gives the malware agent only a half interval (or 2.5ms) to transmit the new seed and get the response. The outsourcing of computation therefore fails.

Defending against adversarial strategy 4 – modify detection code. The security against adversarial strategy 4 follows directly from assumption 2 (code optimality), with the exception of a "kamikaze strategy" in which the adversary corrupts the execution of some of the steps (as described in section 3), and then willingly loads legitimate code and removes itself. Such an adversary could only corrupt step 1 of the process, as it will have to be overwritten during step 2 to avoid detection. Moreover, it needs to correctly perform the setup in step 1; this means that the only harm it can do is to cause an incorrect state to be swapped out in step 1. It can write anything it wants to to swap space. It can place a copy of itself in the swap space, or a copy of a legitimate but vulnerable application, with an input triggering an opportunity for malware to be loaded. However, the swap space will be scanned along with all other memory during step 5, and any known malicious configuration will be detected.

Combinations of the four adversarial strategies will fail, since each of them will be detected and combinations of them do not change the underlying device-specific limitations.

8 Conclusion and Future Work

We have presented the first software-based attestation approach that is suitable for use on handsets. It does not rely on heuristic assumptions, but its security properties can be proven based on measurable physical limitations of devices. The execution time of our proposed system is estimated to be on the order of a minute for typical smartphones. Ongoing work aimed at reducing the computational effort offers hope that significantly faster solutions are possible, which may eventually allow for a continuous user experience. Yet other ongoing work on server-side processing of usage patters of infected devices suggest that it will be possible to use our techniques to rapidly detect and classify epidemics, without access to malware code, but simply based on the network spread patterns.

References

[1] M. Barrett. Cybercrime – and what we will have to do if we want to get it under control, July, 2008.

[2] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. In *CRYPTO '96: Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, pages 1–15, London, UK, 1996. Springer-Verlag.

[3] L. Carettoni, C. Merloni, and S. Zanero. Studying Bluetooth malware propagation: The BlueBag project. *IEEE Security and Privacy*, 5(2):17–25, 2007.

[4] C. Castelluccia, A. Francillon, D. Perito, and C. Soriente. On the difficulty of software-based attestation of embedded devices. Proceedings of the 16th ACM conference on Computer and Communications Security (CCS), 2009.

[5] K.-K. Choo. Organised crime groups in cyberspace: a typology. *Trends in Organized Crime*, 11(3):270–295, 2008.

[6] C. Dwork, A. Goldberg, and M. Naor. On memory-bound functions for fighting spam. In *In Crypto*, pages 426–444. Springer-Verlag, 2002.

[7] Georgia Tech Information Security Center. Emerging cyber threats report for 2009, October, 2008.

[8] V. Gratzer and D. Naccache. Alien vs. quine. *IEEE Security and Privacy*, 5(2):26–31, 2007.

[9] S. Hansell. How hackers snatch real-time security ID numbers, August 20, 2009.

[10] Harris Interactive Public Relations Research. A study about mobile device users, June 2009.

[11] S. Havlin. Phone infections. *Science*, 324(5930):1023–1024, 2009.

[12] M. Hypponen. Malware goes mobile. *Scientific American Magazine*, pages 70–77, 2006.

[13] M. Jakobsson. A central nervous system for automatically detecting malware, September, 2009.

[14] M. Jakobsson and A. Juels. Server-side detection of malware infection. In *New Security Paradigms Workshop (NSPW)*, 2009.

[15] Kaspersky Labs. Kaspersky labs forecasts ten-fold increase in new malware for 2008.

[16] T. Kee. Study: Smartphone users get an 'F' when it comes to security, August 17, 2009.

[17] B. Krebs. European cyber-gangs target small U.S. firms, group says, August 25, 2009. Washington Post.

[18] J. Leopando. Signed malware coming to a phone near you?, July 15, 2009.

[19] R. McMillan. Android security chief: Mobile-phone attacks coming, August 12, 2009.

[20] E. Mills. Microsoft to offer free consumer security suite, November, 2008.

[21] J. Oberheide, E. Cooke, and F. Jahanian. CloudAV: N-Version Antivirus in the Network Cloud. In *Proceedings of the 17th USENIX Security Symposium*, San Jose, CA, July 2008.

[22] R. L. Rivest. The MD6 hash function – a proposal to NIST for SHA-3. Submission to NIST, 2008.

[23] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 1–16, New York, NY, USA, 2005. ACM Press.

[24] A. Seshadri, A. Perrig, L. V. Doorn, and P. Khosla. Swatt: Software-based attestation for embedded devices. In *In Proceedings of the IEEE Symposium on Security and Privacy*, 2004.

[25] Symantec Report on the Underground Economy, November 2008.

[26] P. Wang, M. C. Gonzalez, C. A. Hidalgo, and A.-L. Barabasi. Understanding the spreading patterns of mobile phone viruses. *Science*, 324(5930):1071–1076, May 2009.

[27] Wikipedia entry on NAND flash, Accessed Aug 22, 2009.

[28] Wikipedia entry on polymorphic code, Accessed Aug 22, 2009.

[29] Wikipedia entry on Rootkit, Accessed Aug 22, 2009.

A Detailed Hardware Specifics

About the bus. Typical mobile devices have 32-bit buses. That means that the smallest portion of memory that can be accessed is 32 bits long and that larger segments will be accessed in portions of 32 bits. The speed of the bus affects the access time for all memory, except registers.

About registers. Registers are associated with the CPU, are fast to access. They are used by active algorithms, and offer very limited amounts of storage. A typical smartphone processor, such as the Android G1 uses a ARM1136EJ-S processor, which has sixteen 32-bit registers. Very small malware – whose purpose simply is to load a larger payload portion – requires tens of bytes of storage. Therefore, the registers cannot be used to house a malware agent[8].

About RAM. There are many types of RAM, all allowing random access for both reads and writes. Typical access times for RAM is 5-70ns, where faster types of RAM are used as memory cache. A typical smartphone has 32kB of instruction cache, 32kB of data cache, and 128 MB of main system RAM. The Android G1 has 128 MB of system RAM and is believed to be configured at 32kB / 32kB cache size) The G1's CPU cache takes around 5ns to read or write per 32-bit word (two CPU cycles), while its other RAM takes 20ns to read and write, assuming reasonably sequential access.

About Flash memory. There are two types of flash memory – NAND [27] and NOR flash. Cell phones use NAND flash. It is only possible to erase flash in entire *blocks*, where erasure corresponds to setting all bits to 1. A typical block size is 16, 128, 256 or 512 kB. However, flash can be read – and written, as long as it only changes 1's to 0's – in smaller portions, referred to as *pages*. Typical page sizes are 512 or 2,048 or 4,096 bytes. Flash memory used in mobile phones is typically 16 bits wide and the flash memory bus is typically clocked at 64MHz. A typical smart phone uses so-called small page flash memory, i.e. 512 bytes/page (excluding built-in error correction etc) and 32 pages/block (and therefore 16kB / block). The NAND controller in the MSM7k base band chip used in the G1 is initially designed for small page flash and has a 512 byte built in buffer. The flash memory used in the G1 is, however, a "large page" flash with a native 2kB page size and a 128kB block size. Each 2kB page consists of four 512 byte sectors. The G1 has 256MB of NAND flash, i.e 2,048 blocks, each containing 128kB. The G1 suffers from a memory architecture where the NAND flash is attached to the signalling processor rather than the application processor, so all communications go through RPC between the processors. This reduces effective read access speed to around 4MB/s. Write is estimated at 2MB/s.

The smallest data transfer from a flash memory is typically a sector, i.e. 512 bytes as it is on this level the necessary error correcting codes (ECC) operate. On the G1 it takes no shorter than 125μs to read a sector (512 bytes, plus out-of-band data; 16 bytes, for ECC

[8]If future developments make register-housed malware a reality, then an approach analogous to the one used to distinguish between RAM and flash can be used to perform computation that flushes the registers by introducing timing-distinctions between registers and RAM.

among other things) from on-board NAND flash memory. Since we will typically not make consecutive memory accesses, the 512 byte sector read becomes the actual cost of one single 32-bit read in a naive malware implementation. We must however assume that an attacker could find a way to avoid accessing the entire sector, possibly by inventing their own ECC scheme. For simplicity we will assume that a 32-bit read could be performed in 4/512:th of the time of the sector read, i.e. approximately 1μs. Note that this is under "better than ideal conditions" as we basically assume zero setup time and no overhead or error correction. That is still 50 times slower than RAM.

Writing to flash memory is generally 4-6 times slower than reading, but on the G1 we estimate sector write time to about 250μs. This is not because writing to NAND flash is especially fast on the G1 but rather that the architecture of accessing NAND through another CPU cap performance and mainly lowering read speeds.

Note that write speed to a file system residing on NAND flash lowers dramatically as the file system gets fuller and when files are overwritten with new content. This is because NAND pages (or sectors) cannot be overwritten without first erasing the entire block, so instead the data is written to a new NAND block thus leaving "garbage" behind in the old block. Sooner or later this garbage must be collected, and this is done by compacting the still relevant pages into new blocks and erasing the blocks that now only have garbage in them. As the file system gets fuller and fuller this takes longer and longer time, thus further increasing the gap between RAM and Flash speeds. Every time a NAND block needs to be erased, there is a penalty of about 2ms.

About SD card memory. SD (Secure Digital) cards are popular removable data storage cards that are commonly used in mobile phones and other consumer electronic devices such as digital cameras, camcorders and handheld computers. They use NAND flash to store data in blocks just like the internal NAND flash of the typical smart phone. SD cards come with a *Speed Class rating* that indicate the *minimum* write speed in MB/s. The Speed Classes officially defined by the SD Association are 2, 4, and 6, corresponding to 2.0MB/s, 4.0MB/s and 6.0MB/s respectively, although classes 8 and 10 are now commonly referred to as well by manufacturers. Note that these are the *minimum* rated *write* speeds, and also note that read speeds typically are higher. As of this writing the fastest SD cards announced (SanDisks 32GB Extreme SDHC Card) claims speeds of approximately 30MB/s for both read and write under ideal conditions (unverified). The G1 can not make use of this bandwidth and tends to bottom out at around 10MB/s read speed. We do however anticipate that mobile phones soon will start to reach these speeds.

About other secondary memory. Flash is considered as secondary memory for cell phones. For traditional computing platforms, hard drives are the secondary memory of choice. Typical access times for hard drives – taking average rotational latency into consideration – are dramatically longer than RAM access times. If access locations are set pseudo-randomly then this increases the expected access time.

About external communication. Theoretical Wi-Fi rates go up to 22Mbps, with typical rates at 5Mbps. Similarly, theoretical Bluetooth 2.0 with EDR (Enhanced Data Rate)

exhibits transfer rates of up to 3Mbps, with typical rates at 500kbps. Note that we are not concerned with the turn-around times, but rather the maximum practical throughputs, as we are making the pessimistic assumption that all communication is optimally scheduled by an external host. Thus, these typical access times are on the order of 45-333 slower (for Wi-Fi resp. Bluetooth) than RAM access, under conditions that are the most favorable to an adversary, and under the rather pessimistic RAM rates of 1000Mbps.

About changes. It is worth noting that all of the above access speeds are likely to change over time; however, dramatic changes are unlikely, and modest changes will only force us to make new parameter choices. Since parameter choices have to be made for each particular device type in the first place, that is not a concern.

B Memory-Printing Machine Code

In order to estimate the run time of the memory printing, it is meaningful to consider possible code. In a production version, this code would be further optimized, but the code below allows us to provide a reasonable estimate for the execution time per loop.

```
memory_printing:
    @ Number of rounds in arg0 (r0)
    @ Number of chunks in arg1 (r1)
    stmfd    sp!, {r4, r5, r6, r7, r8, sl, lr}
    subs    sl, r0, #0
    ldmlefd    sp!, {r4, r5, r6, r7, r8, sl, pc}
    cmp    r1, #0
    add    r3, r1, #127
    movge    r3, r1
    mov    r7, r3, asr #7
    mov    r8, #0
.LRounds:
    cmp    r7, #0
    ldrgt    r4, .LGlobals
    ldrgt    r5, .LGlobals+4
    movgt    r6, #0
    ble    .LNoPages
.LPages:
    mov    r0, r6
    bl    get_position_page_from_flash
    mov    ip, #0
    mov    lr, r0
.LInner:
    ldr    r3, [r4, #0]
    ldr    r2, [r5, #0]
```

```
    add    r3, r3, r2
    ldr    r0, [ip, lr]
    bic    r3, r3, #0xFE000000
    bic    r3, r3, #0x00000003
    ldr    r1, [r3, #0]
    ldr    r2, [r0, #0]
    add    ip, ip, #4
    eor    r2, r2, r1
    cmp    ip, #512
    str    r1, [r4, #0]
    str    r2, [r0, #0]
    bne    .LInner
    add    r6, r6, #1
    cmp    r7, r6
    bne    .LPages
.LNoPages:
    add    r8, r8, #1
    cmp    sl, r8
    bne    .LRounds
    ldmfd    sp!, {r4, r5, r6, r7, r8, sl, pc}
.LEnd:
    .align    2
.LGlobals:
    .word    state
    .word    seed
```

The largest portion of time is spent between the *.LInner:* label and the *bne .LInner* conditional branch, i.e., in the inner loop. The three first memory loads are the global variables *state* and *seed*; and one entry of the paged-in *position* vector, all residing in cached RAM. The parts that consume the most time in the inner loop are one load from uncached memory for updating the state variable in the *next_chunk* function; and one uncached memory read at the position pointed out by the entry in the position vector. The subsequent store operation is faster, because then the cache has been activated for that memory position. The modulo operator in the *next_chunk* function has been replaced by a hard coded bitwise AND (implemented using two BIC instructions) to force the memory address within the available 128MB RAM (the first BIC) and aligned to an even 32-bit boundary (the second one). We estimate that the inner loop constitutes approximately 32 CPU cycles, therefore running one iteration in about 80ns out of which 45ns are pure memory access.