Retroactive Detection of Malware With Applications to Mobile Platforms

Markus Jakobsson Karl-Anders Johansson
FatSkunk Inc
Mountain View, CA 94041

Abstract. We introduce a practical software-based attestation approach. Our new method enables detection of *any* active malware (e.g., malware that executes or is activated by interrupts) – even if the infection occurred *before* our security measure was loaded. It 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. We do not rely on heuristics when arguing our scheme's security. Accordingly, our approach represents a step towards greater rigor and security assurance.

1 Introduction

Traditional Anti-Virus (AV) software offers no guarantees of *retroactive* security – detection of past infections – which is worrisome given increased rates of malware evolution. The traditional model also relies on *constant* screening. This is a liability for mobile devices, which have limited power resources. Mobile devices are expected to become seriously targeted by malware [8] as they become the most common computing platform [6].

Software-based attestation (e.g., [3, 5, 7, 9, 10, 12–15]) promises to address this problem by allowing retroactive detection of *any* unwanted process – or rather, any process that remains active after the operating system has attempted to deactivate all processes but the attestation process. We introduce an efficient software-based attestation method, suitable for handsets, and prove it secure based on very simple physical assumptions. Our approach is based on the new notion of device *memory-printing* – a form of fingerprinting of device contents, whose security assurances are derived from the time it inherently takes to access memory of various types. (Due to space limitations, we refer to our extended version [9] for the security analysis.)

Using the techniques described herein, we enable the detection of *any* active malware. Once this has been done, a variety of methods can be used to detect *inactive* malware. Detection of inactive malware (which is not the focus of this paper) is easier than detection of active malware since the detection process cannot be interfered with. Many approaches are possible. For example, one can synchronize the device with a central repository, which can use whitelisting and blacklisting approaches to screen memory images. (To make sure that malware cannot hide by modifying the logs used for synchronization, one can explicitly clear any space claimed to be empty. If we know that there is no active malware, we also know that this deletion will occur.)

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 memory, we describe an algorithm that transforms the memory contents to a state that is then reported to a verifier. This computation is carefully timed by this verifier, who knows how long it should maximally take, since it knows the hardware configuration of the verified device.

In order to remain on the device, a malware agent either has to be active in RAM or reside in an inactive program. Doing the former, we show, introduces significant delays to generate the output expected from our algorithm (thereby resulting in detection); doing the latter is detected when secondary storage is screened. Thus, we *detect* infection, but do *not* block it from occurring in the first place.

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 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, and hold even if as little as only one byte is corrupted. Note that attackers can easily make computation take *longer* (which corresponds to even more obvious detection), but not make the computation to appear to take *less* time than it really did. This is because the client device does not report how long the computation takes, but the timing is done by the verifier. The attacker's only hope therefore becomes for the time increase caused by its presence to be dwarfed by the latency variance of process. We describe [9] how one can use SIM cards to almost entirely remove this variance.

The estimated time for the software attestation is on the order of a few minutes for the parametrization considered in the paper – a device with 128 MB RAM and one core – e.g., an Android G1. This makes it meaningful to run the detection a few times a day, when the device is not in use. Our technique is relatively stable with respect to developments in mobile computing, as it is essentially only the size and speed of the RAM that matters, and most devices use standard components. We have implemented a proof-of-concept of a close variant of our technique on a 256 MB RAM Android Beagleboard.

2 Related Work

Seshadri et al. [13] 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. Gardner et al. [4] found that their approach takes more than 30 minutes on a desktop (and longer on slower computers). In contrast, our solutions takes on the order of a minute on a handset.

Seshadri et al. [14] 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. [1] (although some of these findings are contested [11].)

Scandariato et al. [12] describe a general technique based on frequent updates of application *tags*, whose validity can be externally audited. This approach is heuristic, and hinges on the cost of reverse-engineering obfuscated code. Other obfuscation-based approaches were proposed by Hohl [7] and Shaneck et al. [15]. Garay and Huelsbergen [3] proposed an approach based on receiving executables from the verifier.

Memory-printing has structural similarities to memory-bound functions [2], which is a class of functions whose execution speed depends more on the bus speed than on the processor speed. While the two concepts have similarities, such as a built-in awareness of relative access speeds, they may still be more different than they are similar. The execution time of a memory-printing function, for example, depends critically on the amount of RAM available, and whether secondary storage is used, whereas these issues do not apply to memory-bound functions.

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[1]. We also assume that the verifying party can send data securely to the audited device, e.g., in a way that cannot be eavesdropped[2].

[1]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.

[2]This can be achieved using encryption/authentication using a

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 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:

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 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. **Transact & Restore:** Perform security-sensitive action (such as establish SSL connection, scan flash for inactive malware, etc); then restore RAM state by loading the contents that were swapped out during *setup I.*

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.

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.

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 describe a method to avoid issues with latency variance in section 6.2.)

If the verification (of both results and timings) succeeds, then the verifier knows that there is no active malware on the client. 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.

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 transaction performed after the scan (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:

1. **Outsource storage.** The data that was intended to be stored in the space where the malware agent resides is sent to secondary storage. When it is needed, it is transferred back[3].
2. **Compute missing data.** When the space where the malware agent resides is read, the malware agent computes what those contents *should* have been.
3. **Outsource computation.** The malware agent forwards session-specific data to an external computer, who computes the responses to be sent to the external verifier, and sends these to the malware agent.
4. **Replace detection code.** The malware agent infects the detection code, potentially stopping it from being executed.

[3]Most pessimistically, let's assume that the malware only changes *one* byte of the RAM contents. If the adversary uses the Translation Lookaside Buffer (TLB) to reroute accesses to that cell to flash, that will affect *all* words within the TLB block. Since typical devices use TLBs with 1024 word size, that increases the flash-bound delay for all of these words. If the adversary rewrites the memory-printing code to "surgically" reroute some accesses, that incurs (at least) the cost of one comparison per iteration of the loop. For more details, see [9].

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.

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 the following access times: 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). 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.

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. One can send a short UDP packet every 5-10ms over 3G. 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. More detailed hardware characteristics are available in [9].

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.

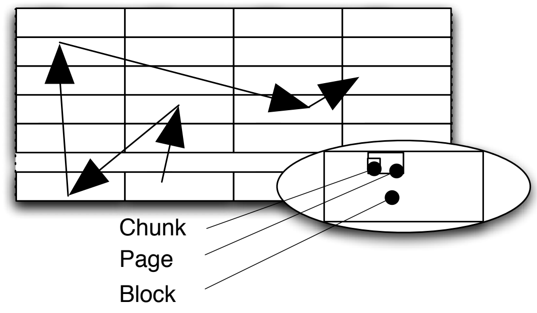

Figure 1: The figure illustrates how memory-printing is performed. A pseudo-random sequence is XORed in to *free* RAM in a pseudo-random order; later, a keyed hash of the entire contents of RAM is computed. Even though RAM does not use blocks and pages, we can divide it into "virtual" blocks and pages, 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.

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

1. First, a table of positions to access is created, using a pseudo-random selection of positions that have not yet been visited (see [9] for more details.) The table is stored in flash.

2. Second, the memory-printing function is used to fill all *free* RAM, in the order indicated by the table of positions. The value stored in the indicated position is the XOR of the old value and the value state, where state is set to the contents of the RAM chunk with location (state + seed) mod number_chunks, for the previous value of state. The execution of this step execution is timed, both from beginning to end and in shorter intervals.

Machine code for the above two functions is available in an extended version of this paper [9].

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[4]. 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.

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.

Slow 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. The inner loop of the memory-printing constitutes approximately 32 CPU cycles, therefore running one iteration in about 80ns out of which 45ns are pure memory access. Based on this, 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.

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.

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. As shown in figure 2, this is done in a way that avoids having *intentional* flash accesses

[4]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.

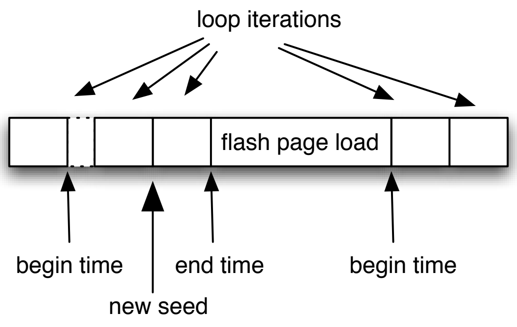

Figure 2: For each round, a small number of RAM accesses are made, but no flash accesses. At regular intervals, a new page of RAM positions to modify is read from flash, replacing the previous such page. These scheduled flash accesses do not cause timing delays, as they are known by the verifier, and 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. See [9] for more details.

(to load new pages of position vector elements) be counted as delays.

The seed values are generated by the external verifier, and sent to and decrypted by the SIM card. The SIM card reports checkpoint state information and associated time stamps back to the verifier.

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[5]. 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.

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.

Acknowledgments. We wish to thank Dan Boneh, Herbert Bos, Ron Rivest and the HotSec program committee for their feedback and suggestions.

[5]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.

References

[1] 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.

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

[3] J. A. Garay and L. Huelsbergen. Software integrity protection using timed executable agents. In *ASIACCS '06: Proceedings of the 2006 ACM Symposium on Information, computer and communications security*, pages 189–200, New York, NY, USA, 2006. ACM.

[4] R. Gardner, S. Garera, and A. D. Rubin. On the difficulty of validating voting machine software with software. In *EVT'07: Proceedings of the USENIX Workshop on Accurate Electronic Voting Technology*, pages 11–11, Berkeley, CA, USA, 2007. USENIX Association.

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

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

[7] F. Hohl. Time limited blackbox security: Protecting mobile agents from malicious hosts, 1998.

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

[9] M. Jakobsson and K.-A. Johansson. Assured detection of malware with applications to mobile platforms, 2010.

[10] L. Martignoni, R. Paleari, and D. Bruschi. Conqueror: tamper-proof code execution on legacy systems. In *Proceedings of the 7^{th} Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, Lecture Notes in Computer Science. Springer, July 2010. Bonn, Germany. To appear.

[11] A. Perrig. Refutation of "on the difficulty of software-based attestation of embedded devices".

[12] R. Scandariato, Y. Ofek, P. Falcarin, and M. Baldi. Application-oriented trust in distributed computing. In *Third International Conference on Availability, Reliability and Security, 2008 (ARES 08)*, pages 434–439, 2008.

[13] 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.

[14] A. Seshadri, A. Perrig, L. V. Doorn, and P. Khosla. SWATT: SoftWare-based ATTestation for Embedded Devices. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2004.

[15] M. Shaneck, K. Mahadevan, V. Kher, and Y. Kim. Remote software-based attestation for wireless sensors. In *ESAS*, pages 27–41, 2005.