Practical and Secure Software-Based Attestation

Markus Jakobsson Karl-Anders Johansson
FatSkunk
Mountain View, CA 94041
{markus,karl-anders}@fatskunk.com

***Abstract*—Software-based attestation can be used for *guaranteed* detection of any active malware on a device. This promises a significant advance in the battle against malware, including *mobile* malware. However, most software-based attestation methods are either heuristic or unsuitable for mobile computing – and often both. One recent software-based attestation method uses so-called memory-printing to produce a software-based attestation technique with provable properties. We describe a novel memory-printing algorithm that improves on that work by being more than an order of magnitude faster, while avoiding commonly used and questionable security assumptions. This results in a truly practical and arguable secure solution – taking less than 3 seconds on a 600 MHz processor with 256 MB RAM. Our work finds applications to malware defense and trusted computing in general, and mobile malware defense in particular.**

***Keywords*-detection, handsets, malware, software**

I. Introduction

For as long as computers have been able to communicate with each other, security researchers have been concerned with how to develop protocols and algorithms to avoid abuse of rights and resources. All of these efforts rely on one critical assumption: *That the protected systems do what they are supposed to do.* Malware unhinges this assumption.

An infected machine can be thought of as an unsafe execution environment. Kennell and Jamieson [12] proposed to observe side-effects of a computation in order to determine whether it was performed in the correct execution environment. While their approach later was found not to be secure [25], it inspired the work on software-based attestation by Seshadri et al. [22]. Software-based attestation aims at proving to a verifier that a given system is in a secure state, or more generally, that a given *process* is executing in a secure state. This can be achieved by being able to *guarantee* that no process – apart from the attestation process – is active (or able to get activated by interrupts.) As soon as this has been established, control can be ceded to a process that scans memory for unwanted applications, or more generally, that performs any desired computation. This allows detection of malware infection: If we know that no malicious process is active, then we also know that our scan results cannot have been corrupted.

Given the importance of detecting malware, and of being able to assert its absence, the last few years have seen substantial effort to design and analyze software-based attestation methods – see, e.g., [1], [2], [5], [6], [7], [8], [10], [11], [13], [19], [20], [21], [22], [23], [24], [26], [29]. Much of this work was done in the context of sensors and embedded devices, and much of it was heuristic.

Jakobsson and Johansson [10] recently proposed a provably secure software-based attestation solution, suitable for complex environments, such as smartphones. These are vulnerable platforms [18]. At the heart of the Jakobsson-Johansson approach lies the notion of *memory-printing*, i.e., a computation that can only be performed without notable delays if it is given access to all free RAM. Here, *free RAM* is defined as the portion of RAM which is *supposed* to be free after all applications – including the regular kernel – have been swapped out, leaving only a very small monolith kernel in RAM. This monolith kernel performs the memory-printing, which – if it computes the correct result within the expected time – proves that there *cannot* be any other active process.

While Jakobsson and Johansson's proposed implementation was *reasonably* fast for typical smart phones – estimated to take less than a minute to run – this might still place practical constraints on its use. Their implementation is less practical for typical netbooks and PCs, due to the greater amount of RAM these devices have. Their algorithm would have taken on the order of ten minutes to execute on such devices. We propose an improved memory-printing algorithm that brings down the execution time of the memory-printing to a few seconds for typical smart phones, and about twenty seconds for netbooks, laptops and similar devices.

We achieve the improved efficiency simply by avoiding accesses to secondary storage and minimizing the total number of operations performed. More particularly, we replace memory access tables with simple arithmetic operations, allowing us to avoid costly reads from secondary storage. Moreover, we replace a commonly executed sequence of four memory reads and writes with one single read operation, using registers alone to maintain state.

We argue that the resulting algorithm is secure. The security of our proposed solution does not rely on the *code optimality* assumption used in Seshadri et al. [22], [21] and Jakobsson-Johansson [10]. We simply base our security proof on the fact that random access of RAM inherently is faster than of secondary storage (e.g., flash.)

II. Related Work

Defense of mobile phones against malware is complicated by the fact that these devices suffer severe power restrictions, which places limitations on what type of detection can be performed. It is not evident that the traditional anti-virus paradigm – with its use of signature detection and behavioral detection – is well suited for mobile platforms.

As malware evolution increases in speed, it is becoming increasingly difficult to detect problems in a timely manner, independently of power limitations. Symantec's Quorum product addresses this problem by assigning reputations to applications, and *poor* reputations to both unknown applications and known malicious applications. Another way to deal with a spiraling pace of malware evolution is to implement some form of truthful reporting on devices, which offers the opportunity of retroactive detection. Truthful reporting can either be of *events* [11] or of *state*, where the latter approach is commonly referred to as *software-based attestation*.

Seshadri et al. have made important contributions to software-based attestation on embedded systems [19], [20], [22], [23] and on legacy systems [21], [26]. Both of their principal approaches are based on timing a particular computation in order to assess whether it was carried out as intended. Their work on embedded systems only works for devices that can communicate *only* with the verifier – which cannot be guaranteed for handsets with Bluetooth or Wifi capabilities. The work on legacy systems depends critically on a code optimality assumption. Due to the complex code, this optimality is hard to assure – as highlighted by Castelluccia et al.[1], although later criticized by Perrig and van Doorn [14]. It is also a rather computationally demanding approach, taking on the order of half an hour to run on a powerful PC [6] – and longer on less powerful devices.

Gratzer-Naccache [7] and Choi et al. [2] independently of each other described how to detect infection on embedded systems by overwriting the entire space with a non-compressible sequence, and reading it back again. This is not a suitable approach for handsets.

Scandariato et al. [16] 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 [9] and Shaneck et al. [24]. Garay and Huelsbergen [5] proposed an approach based on receiving executables from the verifier.

Jakobsson and Johansson [10] introduced the notion of memory-printing, and used this to produce a software-based attestation solution suitable for handsets. In contrast to much of the prior work, it is not a heuristic approach, but more importantly, it applies to handsets, and not only to embedded systems. We improve on the work of Jakobsson-Johansson by easing the assumptions and by dramatically improving on the efficiency of their solution. The latter allows us to practically extend the same type of protection to typical netbooks, for which their solution was too costly due to their greater amounts of memory these devices have. The improvements associated with our solution are due to techniques we introduce to avoid generation of pseudo-random material during the "testing" period, and to much simplified memory access patterns.

Memory-printing has structural similarities to memory-bound functions [3], 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.

Looking beyond the family of software-based attestation solutions, one may also rely on trusted hardware, as was done in the (now inactive) NGSCB/Palladium effort [28] and the TCG system [27]. TCG faces potential problems relating to credential revocation, and has been criticized for threatening software that has not been signed by approved vendors with lock-out. A hybrid approach combining hardware and software-based approaches was proposed by Schellekens et al. [17].

III. Architecture and Adversary

Phases: Our solution involves two phases. In a first phase, a software-based attestation program is executed in order to establish whether there is any *active* malware. In the second phase, secondary storage can be screened (using whitelisting or blacklisting approaches, say); securely synchronized with a central repository; or some other sensitive function can be carried out. We know that the *second* phase is executed in a *secure state* if the *first* phase produces an output that indicates the absence of active malware. This paper addresses the first phase.

Architecture: In the following, we assume an architecture consisting of a *client*; an *external verifier*; and a *proxy*.

The client (which is the device to be audited) is represented by a monolith kernel that will swap out all other processes when it is activated, after which it will perform the computation that will provide evidence of corruption-freeness. (If the monolith kernel is corrupted by malware – which we have to assume is possible – then the above may not happen. However, our security guarantees still hold in that case.)

The proxy is used to reduce latency variance from the audited device, and may be a SIM card; a tethered cell phone; a cell phone tower, etc. At any rate, the SIM card may

be used to determine the identity[1] of the associated device. (As described below, we assume a software adversary, and so, do not have to be concerned with tampering of the SIM card. We assume that the adversary cannot convince the user to move the SIM card to another and more powerful device.)

The external verifier performs some initial computation, and communicates (using a secure channel) part of the information to the proxy, potentially using the (untrusted) client as an intermediary. The proxy interacts with the monolith kernel of the client, timing the computation performed by the client; these timing measures are reported back to the external verifier – again, using a secure channel, and potentially using the client as an intermediary.

We consider devices with secondary storage (that the user may be able to expand at will), and with wireless features –e.g., WiFi, Bluetooth, 3G/4G.

Practical Adversarial Strategies: The adversary is a malware agent, and is assumed not to have physical access to the device to be verified. To be successful, the adversary must be able to perform the expected memory-printing computation within the expected amount of time, without ceding control to the legitimate memory-printing software. To achieve this goal, the adversary has the following general options, and combinations of these:

- **Guessing attack:** When unable to submit correct responses in time, the adversary outputs arbitrary responses of the appropriate length and distribution, with the hope that these – by serendipity – will be correct.
- **Secondary-storage attack:** Using secondary storage, the adversary maintains an image of the RAM cells whose contents are altered as a result of the adversary's presence, and accesses this image when these cells are referenced.
- **Computing attack:** As needed, the adversary *computes* the contents of the cells whose contents are altered as a result of the adversary's presence.
- **External attack:** The adversary collaborates with an external computer that computes the responses it needs to feed to the proxy. (We may assume that this external computer is infinitely fast; the connection to it, however, suffers regular network latencies.)
- **Switching attack:** The adversary modifies the monolith kernel to cause the wrong set of commands to be performed after the memory-printing has successfully completed. (However, by making modifications to the RAM contents, he *also* has to succeed with one of the previously mentioned attacks – guessing, secondary-storage, computing, or external.)
- **Hibernating attack:** The adversary modifies inactive legitimate programs in secondary storage, such that these – once activated – will revive the malware agent from hibernation. (We will not consider this attack herein, since it does not involve an active malware agent, and therefore, will be detected during the scan performed after the memory-printing has completed.)

[1]SIM cards have cryptographic keys that cannot be read from their host devices. These keys are used to authenticate transcripts to be sent to the external verifier.

Definition: Robustness: The monolith kernel corresponds to a program $F_{\mathcal{E}}$, parametrized for a known execution environment $\mathcal{E}$. The execution environment corresponds to the hardware configuration of the device. Executing $F_{\mathcal{E}}$ on input x produces a sequence of outputs $F_{\mathcal{E}i}(F_{\mathcal{E}}, x)$; each within a time $t_i(F_{\mathcal{E}}, x)$ from the start of the execution; and produces an ending state $s(F_{\mathcal{E}}, x)$. (Note that this is not the final *reported* state, but the contents of RAM and registers at the end of the execution of the monolith kernel.) Here, $x \in \mathcal{X}$, where $\mathcal{X}$ is the set of all legitimate inputs. Note that $F_{\mathcal{E}}$ is self-referencing, which is why its output is a function of both the input and of $F_{\mathcal{E}}$ itself.

The adversary provides a program F' to be run in $\mathcal{E}$. The adversary is successful if and only if (a) F' produces the *right* sequence of outputs; (b) where each output is produced within the *right* amount of time; and (c) it ends up in a *different* state than $F_{\mathcal{E}}(x)$ would have.

More formally, the we say that the adversary is δ-*successful* if and only if

$$\begin{cases} F'_i(F', x) = F_{\mathcal{E}i}(F_{\mathcal{E}}, x) \\ t_i(F', x) < t_i(F_{\mathcal{E}}, x) + \delta \\ s(F', x) \neq s(F_{\mathcal{E}}, x) \end{cases}$$

for all $x \in \mathcal{X}$, but for a negligible fraction, and for all $1 \leq i \leq n$, where n is the number of checkpoints. If there does not exist any adversary that is δ-successful with respect to $F_{\mathcal{E}}$, then we say that $F_{\mathcal{E}}$ is δ-*robust*.

IV. Solution – Big Picture

Terminology: We define *free RAM* as the portion of RAM that is supposed to be free after all applications and the standard kernel have been swapped out. The width of the bus is a *word*. We describe the size of memory in terms of the number of words, too; thus, we say that a 512 byte memory page has size 128 words on a standard handset where a word is 32 bits. Furthermore, we define a *chunk* as the length of the cache line. For example, if the cache line corresponds to 32 words, each of which is 32 bits, then a chunk is 128 bytes.

Goals: We wish to establish that there is only *one* active process on the client machine, namely an unmodified version of the audit process. We reach this goal this by performing *memory-printing*. This is a process that is configured to require all of *free* RAM to complete its computation in the expected amount of time. It has the property that any reduction of space leads to measurable slowdowns of the computation. Moreover, it is configured to access memory in a manner that results in a notable slowdown if the process is modified to sometimes access secondary storage instead

of RAM. *Therefore, if a malware agent is active – i.e., remains in RAM – then the process* will *be slowed down.* By measuring the time it takes to compute the correct result, it is determined whether the client machine is infected or not.

Process Overview: The malware detection algorithm involves the following steps on the client machine:

1) **Preparation:** Swap out the contents of *free* RAM to secondary storage[2].
2) **Setup:** Receive a value $seed$ from the external verifier, and overwrite *free* RAM with the output of a pseudo-random string generated from $seed$. (One particular way of doing this will be detailed in the next section.) Receive a value $step$ and a value key from the external verifier.
3) **Memory-printing:** Read RAM in a manner determined by $step$, and accumulate the results in $state$, the computation being keyed using key. This phase will be detailed in the next section. If the external verifier approves the memory-printing, the audited device is considered being in a safe state.
4) **Safe-State Execution:** The device executes the function that must only run in a safe state[3]. Depending on the context, the function may perform a scan of the contents of secondary storage; key establishment with a third party; secure synchronization of contents; etc.
5) **Restore:** Restore RAM state by loading the contents that were swapped out during the *preparation* step.

The process will be timed from the beginning of step 2 until the end of step 3; in addition, partial computation segments of step 3 may be timed individually.

V. Solution – Details

Setup of free RAM: During the setup phase, free RAM is overwritten with a pseudo-random string generated from the value $seed$, obtained from the external verifier via the proxy. The computation of the pseudo-random string is is done in a way that *guarantees* that recomputation of the value of *any* cell will take a minimum effort – whose time exceeds the threshold δ associated with detection of cheating. It can be done as follows:

1) **Generating:** Using a hash function h (such as MD6 [15]) with output size n bits, generate value $x_i = h(seed, i, aux)$ for $0 \leq i \leq n-1$, and some value aux. This generates n^2 pseudo-random bits.
2) **Shuffling:** Compute $y_j = \prod_{i=0} 2^i \, BIT_j(x_i)$, $0 \leq j \leq n-1$, where BIT_j is a function that returns the jth most significant bit of the input. This shuffles the bits in a manner that requires computation of all n hash function applications to reconstitute any one of the values y_j.
3) **Blending:** Compute $z_j = h(y_j)$, for $0 \leq j \leq n-1$. This assures that each bit of the output is a function of all the n input bits, each one of which required one hash function evaluation to compute (as performed in step 1 above).

The above process generates a n^2-bit pseudo-random string with the particular property that the computation of any *one* of the output bits will take at least half as long as the computation of the *entire* block of bits – or at least 512 applications of MD6 in 512 bit mode.

The above will be repeated (with different values for aux) until the resulting strings fill the entire free RAM[4].

Memory Access Scheduler: The memory-printing will cover the entire RAM. We let s_{RAM} be the size of RAM, measured in *chunks*. (Recall that we have defined a chunk as the width of the cache line.) The external verifier will select a random value $step$ in the range $page < step < s_{RAM} - page$, such that $step$ and s_{RAM}/α are relatively prime. (If s_{RAM}/α is a power of two it is sufficient that $step$ is an odd value.) Here, $page$ denotes the size[5] of one memory page in secondary storage, also measured in chunks.

The memory-printing process will involve a loop in which memory is accessed and the results combined to form a keyed memory checksum. For each iteration of the loop, the access location is increased by the value $step * \alpha$, modulo s_{RAM}, where α is the cache line size. This means that all RAM memory locations will be accessed exactly once. (This holds since $step$ and s_{RAM}/α are relatively prime.) Moreover, we know that the access order will be unknown to the adversary until the value $step$ is disclosed by the external verifier. An infected device would not know the value ahead of time.

We note that on a multi-core processor, or on a machine with many processors, the memory-printing can either be constructed in a way that is inherently serial (and which will therefore obstruct the use of multiple processors) or adopted for multiple processors. One way of doing the latter is to start off several computations with offsets such that each thread corresponds to a different portion of memory, and where there are no collisions. This is straightforward using a step size that is a k-tuple of a random odd number, given k cores. However, in this paper, we assume the use of single-

[2]This may involve swapping out the kernel, or parts thereof. For simplicity, we assume that the kernel will be swapped out in its entirety, and crucial features – such as the device drivers for serial communication – are re-implemented in the monolith kernel.

[3]If the code of the safe-state function was kept in secondary storage during the memory-printing (i.e., it is not part of the monolith kernel) then its digest has to be compared to a value stored in the monolith kernel or on the proxy. The function is activated only if these values match. Note that the monolith kernel does not need to contain code to compute the message digest if this computation can be performed by the proxy.

[4]It is possible to shuffle and blend each such block of strings to still further increase the cost of computing any part of the final string; however, this is not necessary for our use of the pseudo-random string.

[5]In the case where there are several pages sizes, e.g., if there are several components that constitute secondary storage, then we consider the largest of the page sizes.

core processors for the sake of denotational simplicity.

Accumulator: Memory contents will be accumulated in a register using a simple non-linear function that combines the previous register contents (which we refer to as $state$) with the newly read memory contents ($data$), one by one. One may use a hash function (e.g., MD6 [15] or Skein [4]); a non-linear shift-back register; or even a very simple function such as $state \leftarrow \text{ROR}(state + data)$ to achieve this. The latter function corresponds to a function $\text{ROR}(\ldots(\text{ROR}(state_0 + data_1) + data_2) \ldots + data_n)$, where + refers to regular addition, and ROR rotates the contents of the register one bit to the right. (We note that this function in itself may not have sufficient non-linearity, but will have this in combination with the a priori unknown step size.)

The accumulation process should be keyed. This can be achieved by offsetting the value $state$ with a new value key (obtained from the external verifier or the proxy) at regular intervals – for example, by adding the new value key to the current value $state$. An infected device would not know the updated key value until this is released – whether directly by the external verifier, or by a proxy, as described in section III.

Detailed Memory-Printing: For practical purposes, and to maximize the execution speed for the legitimate client, we make full use of the data cache. To do that, we process data in chunks the width of the cache line, whose size typically ranges from 8 to 512 bytes. In the following, the parameter α denotes the width of the cache line, in words.

The parameter $period$ corresponds to the length of the timing periods; this must be set to correspond to a time that is shorter than an optimistic round-trip latency from the client device to a potential external helper. The reported values, and the time at which they are reported, would be recorded by the proxy (see Appendix A for more details on the proxy.)

For ease of denotation, we refer to $state$ as a vector with α elements. The memory-printing (step 3 of the process overview in section IV) can be performed as follows:

$state[0 \ldots \alpha - 1] \leftarrow 0$
$location \leftarrow 0$
repeat $(s_{RAM}/period/\alpha)$ times:
    receive value key from the proxy
    $state[0 \ldots \alpha - 1] \leftarrow state[0 \ldots \alpha - 1]$ XOR key
    repeat $period$ times:
        $data \leftarrow$ RAM[$location$]
            % read cache line
        $state[0] \leftarrow \text{ROR}(state[0]$ XOR $data)$
            % accumulation
        $data \leftarrow$ RAM[$location + 1$]
            % read from cache
        $state[1] \leftarrow \text{ROR}(state[1]$ XOR $data)$
            % accumulation
        ...
        $data \leftarrow$ RAM[$location + \alpha - 1$]
            % read from cache
        $state[\alpha - 1] \leftarrow \text{ROR}(state[\alpha - 1]$ XOR $data)$
            % accumulation
        $location \leftarrow (location + step * \alpha)$ mod s_{RAM}
            % next location?
    report array $state$ to the proxy

Parameter Choices and Timing Results: We will consider a single-core device with 256 MB RAM; 32 bit words; a CPU frequency of 600 MHz; a memory frequency of 166 MHz; a CAS latency of 3 cycles; RAS-CAS latency of 3 cycles; Dcache line width of 128 bytes (i.e., $\alpha = 32$); Dcache size 256 KB; a databus lane width of 16 bits; and 3 instructions per Dword on average. This describes an Android Beagle board, a common developer platform that we used to implement our algorithm.

The above hardware configuration results in an estimated 8388608 cache misses for the execution of the unrolled version of the algorithm above (see Appendix A for code). The cache row read/write time is 14 cycles, giving a total cache latency (read + write) of 1.41 seconds. The execution time of the unrolled version of the code is 1.34 seconds, not counting the cache latency. This gives an approximate execution time of 2.76 seconds.

We will assume that round-trip latencies of approximately 2 ms are possible for a malware agent wishing to communicate with an external adversary, and that the external adversary is infinitely fast. We want to set the value of $period$ so that the time to execute one iteration of the outer loop is no more than the round-trip latency. This corresponds to setting $period = 1024$, which is the number of iterations of the inner loop. Given $\alpha = 32$, and $s_{RAM} = 2^{28}$, we get an outer loop that iterates 8192 rounds. One iteration of the outer loop takes approximately $0.33ms$.

VI. Security Analysis

To succeed, the malware agent needs to be active and remain undetected. It can either reside in space that should be empty after all legitimate processes have been paged out (cases 1-3 below) or in space that is intended for "whitelisted" code, such as our monolith kernel (case 4), or a combination of these.

We know that to succeed, an adversary has to provide the correct responses $F_{\mathcal{E}i}(x)$ within the expected time $t_i(F_{\mathcal{E}}, x) + \delta$. This can only be done with a better than negligible probability by (1) having an external party help generate the responses or by generating the responses locally (i.e., using only the audited device). The latter can only be done in a way that (2) relies on secondary storage, or (3) using RAM only. We will analyze these cases separately, with respect to the proposed solution:

Case 1: External support: The values key and $step$ are communicated to the proxy over a secure channel. The value of $state$ is reported for each iteration of the outer loop. The computation of this value depends on the value key. Therefore, it is not possible for an external device to help with the generation of the responses if the roundtrip latency (from sending out the value key to receiving the result of the computation) exceeds the time threshold $t_i(F_{\mathcal{E}}, x) + \delta$, where $t_i(F_{\mathcal{E}}, x)$ is the time for $F_{\mathcal{E}}$ to generate the response.

The parameter choices of section V were made to make this type of attack impossible – the time between key updates is $0.33ms$ and the roundtrip latency several times greater. If roundtrip latencies were to fall, the parameters will be readjusted accordingly to maintain security against this attack. As long as a new key can be obtained from the proxy (i.e., SIM card) in less than this time, this will not cause any difficulties.

We note that the use of the external helper does not preclude the need to authenticate outgoing transcripts, which is assumed to be done by the SIM card. (This is so since the external helper does not have access to the cryptographic key of the SIM card.) For this reason, it does not matter whether the latency between the device and the external helper is less than between the device and the SIM card: If an external helper is involved, the associated latency is *in addition* to the latency associated with the SIM card.

Case 2: Using secondary storage: Assume that the adversary has modified some k words of data in RAM, and that an image of the "intended" contents of this area is kept in secondary storage. When $F_{\mathcal{E}}$ would have attempted to access one of the modified cells, F' instead accesses the image in secondary storage. This can be done in three ways: (a) The *entire* RAM can be imaged in secondary storage, and each access is made there; (b) Each memory access of $F_{\mathcal{E}}$ can be replaced with a comparison of the address to a range, with select access attempts only being rerouted to secondary storage; and (c) the Translation Lookaside Buffer (TLB) can be used to automatically reroute accesses with selected ranges.

In the first case, *each* memory access will suffer the delay associated with accessing secondary storage instead of RAM. In the second case, only rerouted accesses suffer this delay, but for *each* memory access, a comparison has to be made. In the third case, there is only a delay imposed for rerouted accesses. However, due to the somewhat coarse granularity of the TLB, there is a minimum number of cells for which accesses can be rerouted. (For the Beagle board, this is 4kB, or 1024 words.)

For the Beagle board that was used for our tests, the cost of reading one word of secondary storage is at least 144 ns (see [10]), whereas the cost of accessing RAM is at most 20 ns. The increased cost for each rerouted access is therefore at least 124 ns. The minimum delay associated with the first case is therefore at least $124ns \times s_{RAM} \approx 32ms$.

Consider now an adversary who adds a comparison for each new iteration of the inner loop, to determine whether to access RAM or secondary storage for the reads of that iteration. To minimize the number of changes to physical memory cells as this comparison is added, it is necessary for the adversary to add a jump to the end of the monolith kernel and perform the comparison there (along with any instruction that was replaced by the jump instruction). At the end, there must be a jump back. At the very least, two instruction memory areas will be affected by this change, and need "re-routing". This results in an increase of two jumps and at least one comparison – or at least four instruction cycles – for each iteration of the inner loop. Therefore, the cumulative delay associated with the second case is at least $4 * s_{RAM}/\alpha$ instruction cycles, or $4/32 * 5ns \times 2^{16} \approx 125\mu s$ – plus the delay associated with accessing secondary storage for each rerouted access.

In comparison, the minimum cost of the third case is the extra cost of rerouting at least 1024 memory accesses, or $124ns \times 1024 \approx 127\mu s$. This is the shortest delay. Therefore, as long as $\delta < 127\mu s$, the adversary will be detected. This will become our *theoretical lower bound* for δ.

However, we note that the above are not tight bounds, as it assumes that SD flash can be read in single words, and without the need for error correction. Normal reads to SD flash, in contrast, impose approximately $60\mu s$ of delay per read, causing a total delay of at least $60ms$ under reasonable circumstances for the third option (and correspondingly more for the other options). This affects the *realistic* value of δ, which is beyond the scope of this paper to compute.

Case 3: Using RAM: Due to the construction of the pseudo-random string that is used to fill free RAM, we can see that computing any one cell will require the computation of at least 512 hash function evaluations, since the value in each cell is a portion of the second-layer hash function output, whose input is a function of 512 first-level hash function evaluations.

From the MD6 report [15], it can be seen that there is a setup cost for each hash function computation, and that this is at least 1156 clock cycles. At the same time, the hash computation for MD6-512– once the setup is completed – takes 155 cycles per byte, or 9920 cycles for a 64 bit input. Both of these are for optimized code and for a typical 32 bit processor. The total cost per hash function evaluation in this setting is therefore 11076 cycles. It is not likely that this can be dramatically reduced, but just for the sake of an extreme argument, we assume that an adversary can do this in half of that time – in 5538 cycles.

We see that computation of the contents of only *one* cell in RAM will take at least (512+1)*5538 cycles, due to minimal computational effort to compute one word. The time to do this is a bit more than 56 ms. This does not take into consideration the time to fetch instructions; cache misses; the number of cells one needs to compute; etc. We

do not need to do this since the time to perform this attack clearly will exceed the previously computed value for δ.

Case 4: Malware in whitelisted space: In the case where at least part of the malware is stored where whitelisted code should reside, this will cause the checksum to be incorrect unless the malware agent "corrects" the memory-printing computation by having it computed on the *expected* contents. This can only be achieved by modifying the memory printing code, whether by inserting a comparison, changing the loop variable to make the loop halt early, or unrolling the loop. Inserting one extra instruction to be executed each loop will increase the time of execution by $s_{RAM}/period$ cycles, which is on the order of 1ms. Changing the loop variable will affect the cells operated on – unless these are offset – since the loop variable is used to determine the location to be read. Offsetting the location will add at least one operation per iteration of the loop. Finally, the code can be unrolled into two or more segments. Each time it is unrolled corresponds to the execution of one special case. However, each time it is unrolled also increases the amount of modified code, and therefore increases the need to *hide* modified code. Data is processed in segments of α words. As long as the increase of modified code is at least α words, increasing unrolling will make the situation *worse* for the adversary. Looking at the pseudo-code (section V), we see that there is more than one instruction in the loop for each of the α elements of the cache line. Therefore, unrolling the code cannot help the adversary.

Taking a step back: From the above cases, we see that our solution, when executed on the platform described, is δ-robust for $\delta = 127\mu s$. This is a drastic lower bound, given practical considerations – such as the fact that typical malware agents will not contain lowel-level routines for how to access words from flash in an optimal manner, and the fact that flash is very error prone. In the above we took the pessimistic view that an adversary could read words without errors at an optimal speed.

While practical network latency variance may very well exceed this value of δ, there is no measurable latency variance to communicate with a SIM-card proxy. While the communication is serial, and therefore slow, the *variance* can be neglected. This makes it practical to perform timing using a SIM card (see Appendix A for a description of this process.)

VII. Conclusion and Future Directions

We have described a more efficient approach to perform memory-printing, with applications to trusted computing, and to detection of mobile malware in particular. We have proven our solution to be secure based on physical limitations on hardware, and analyzed the security in the context of one meaningful hardware configuration. By doing this, we have established that provably secure software-based attestation can be practical and efficient.

We do not believe that our solution is *anywhere* close to optimal. We believe that there is space for further significant improvements in terms of the running time of the algorithm. Moreover, by gaining a more realistic understanding of the reasonable limitations (as opposed to the theoretical limitations) of the adversary, one may be able to make better parameter choices and corresponding algorithm improvements. In that context, it is meaningful to develop statistic models for how to identify adversarially caused delays over a regular network connection (as opposed to using an on-board SIM proxy). This is of significance for us to be able to extend detection guarantees to devices that lack hardware SIM cards.

For practical purposes, the massive deployment of memory-printing techniques requires the selection of parameters for each platform, and the writing of monolith kernels adapted to these. We have implemented our solution on an Android Beagle board. In our experience, the greatest difficulty was swapping out the kernel and running applications before execution of our monolith kernel, and then swapping the kernel and applications back in afterwards.

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] Y.-G. Choi, J. Kang, and D. Nyang. Proactive code verification protocol in wireless sensor network. In *ICCSA (2)*, pages 1085–1096, 2007.

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

[4] N. Ferguson, S. Lucks, B. Schneier, D. Whiting, M. Bellare, T. Kohno, J. Callas, and J. Walker. The Skein Hash Function Family.

[5] 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.

[6] 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.

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

[8] L. Gu, X. Ding, R. H. Deng, B. Xie, and H. Mei. Remote attestation on program execution. In *STC '08: Proceedings of the 3rd ACM workshop on Scalable trusted computing*, pages 11–20, New York, NY, USA, 2008. ACM.

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

[10] M. Jakobsson and K.-A. Johansson. Retroactive detection of malware with applications to mobile platforms. In *ACM HotSec 10*, 2010.

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

[12] R. Kennell and L. H. Jamieson. Establishing the genuinity of remote computer systems. In *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium*, pages 21–21, Berkeley, CA, USA, 2003. USENIX Association.

[13] F. Monrose, P. Wycko, and A. D. Rubin. Distributed execution with remote audit. In *In Proceedings of the 1999 ISOC Network and Distributed System Security Symposium*, pages 103–113, 1999.

[14] A. Perrig and L. van Doorn. Refutation of "On the Difficulty of Software-Based Attestation of Embedded Devices".

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

[16] 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.

[17] D. Schellekens, B. Wyseur, and B. Preneel. Remoteattestation on legacy operating systems with trusted platform modules. In *In Science of Computer Programming*, pages 13–22, 2008.

[18] N. Seriot. iPhone Privacy. In *Black Hat*, 2010.

[19] A. Seshadri, M. Luk, and A. Perrig. SAKE: software attestation for key establishment in sensor networks. pages 372–385. 2008.

[20] A. Seshadri, M. Luk, A. Perrig, L. van Doorn, and P. Khosla. SCUBA: Secure Code Update By Attestation in sensor networks. In *WiSe '06: Proceedings of the 5th ACM workshop on Wireless security*, pages 85–94, New York, NY, USA, 2006. ACM.

[21] 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.

[22] 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.

[23] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla. Using SWATT for verifying embedded systems in cars. 2004 Embedded Security in Cars Workshop (Escar 2004), 2004.

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

[25] U. Shankar, M. Chew, and J. D. Tygar. Side effects are not sufficient to authenticate software. In *In Proceedings of the 13th USENIX Security Symposium*, pages 89–101, 2004.

[26] E. Shi, A. Perrig, and L. V. Doorn. Bind: A fine-grained attestation service for secure distributed systems. In *SP '05: Proceedings of the 2005 IEEE Symposium on Security and Privacy*, pages 154–168, Washington, DC, USA, 2005. IEEE Computer Society.

[27] Trusted Computing Group (TCG). http://www.trustedcomputinggroup.org/.

[28] Wikipedia entry on NGSCB. Next-generation secure computing base. Accessed Dec, 2009.

[29] Y. Yang, X. Wang, S. Zhu, and G. Cao. Distributed software-based attestation for node compromise detection in sensor networks. In *SRDS '07: Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems*, pages 219–230, Washington, DC, USA, 2007. IEEE Computer Society.

APPENDIX

The following code is the ARM assembly inner loops of the generic memory-printing code (for any α, and kept non-unrolled here out of space considerations). Note the dramatic difference in simplicity compared to the inner loop code of Jakobsson-Johansson [10].

```
/* Inner loops for non-unrolled version of
 * memory printing algorithm.
 *
 *   Input:
 *      r0 = period / alpha (number of loops)
 *      r1 = state vector address + alpha * 4
 *      r2 = location + alpha * 4
 *      r3 = step (in bytes)
 *      r4 = size of RAM in bytes, minus 1
 *      r5 = alpha
 *
 *   Temporary:
 *      r6, r7, r8
 */

lsl r6, r5, #2
rsb r5, r6, #0

outer_inner_loop:
mov r8, r5

inner_loop:
ldr r7, [r8, r1]
ldr r6, [r8, r2]
eor r7, r6, r7
ror r6, r7, #1
str r6, [r8, r1]
adds r8, r8, #4
bne inner_loop
```

```
subs r0, r0, #1
add r7, r2, r3
and r2, r7, r4
bne outer_inner_loop
```

In the description of the memory printing (section V) values are received from and reported to a proxy, such as a SIM card. A slightly modified Java Card can be used for this purpose, where the modification permits the process to remain active after responding to a request. The proxy will have received a vector of values $input$ and $output$ from the external verifier, and produces a vector $duration$ that is transmitted to the external verifier at the completion of the execution. (All the communication between the proxy and the external verifier is assumed to be authenticated and encrypted.) This computation, which constitutes the external timing of the memory-printing, can be done as follows:

$counter \leftarrow 0$
$time \leftarrow 0$
repeat $period$ times:
    $counter \leftarrow counter + 1$
    repeat until value $state$ is received:
        $time \leftarrow time + 1$
        if $time = maxtime$ then
            $duration(counter) \leftarrow \perp$
            $counter \leftarrow counter + 1$
    if $state = input(counter)$ then
        $duration(counter) \leftarrow time$
        respond with value $output(counter)$
    else
        $duration(counter) \leftarrow \perp$

In the above, the value $\perp$ corresponds to an error message indicative of an attempt of a malware agent on the client to cheat. Upon receipt, the external verifier will determine if the values in the vector $duration$ all fall within the tight bounds suggesting successful completion, and will conclude that the client is in a safe state only if that holds.