Mobile Malware: Why the Traditional AV Paradigm is Doomed, and How to Use Physics to Detect Undesirable Routines

Markus Jakobsson Guy Stewart
FatSkunk Inc

Abstract

The traditional Anti-Virus paradigm focuses on signature-based and behavioral detection. These require substantial processing, which hurts the limited power resources of handsets. Also, carriers are reluctant and slow to deliver Firmware Over The Air (FOTA) patches, due to the rigorous testing they need to subject updates to, and the costs of over-the-air updates. A move to cloud-based screening fails to recognize that not all threats will be propagated over the backbone, may obfuscate themselves in transit; or fight back (as rootkits do) to evade reporting or use of techniques such as the "Google kill switch". Hardware vendors are evaluating security solutions with hardware support, such as TrustZone, but while this reduces the vulnerabilities, it still exposes an attack surface.

We describe a new approach that detects the presence of any undesirable routines – including corruptions of the code used to perform this detection. The security assertions we make are not based on heuristics, but rather, rely on the physical characteristics of the target device. We detail the approach, which does not rely on hardware modifications, and explain how to analyze its security.

Keywords: Cryptography, Hardware

1 Introduction

Current Anti-Virus (AV) software works in a similar way to how TSA personnel screen 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 for which a single security breach can result in a complete loss of control and irrevocable damage. While this approach may be sufficient to secure air travelers, it is becoming increasingly unsuitable to protect against malware. One reason for this is the constant evolution of malware, resulting in the declining benefit of signatures and behavioral models. A second reason is the spiraling cost of deploying these defenses; a third is the cost of patching; and yet another one is the burden of constant screening on devices with limited power resources.

In this paper, we describe an approach that detects any active malware, including zero-day attacks, simply based on the fact that all active routines take up space in the cache – whether in the form of code or an entry in the interrupt table. The approach does not rely on patching, and benefits from moving the attack surface off of the target device and onto a web server responsible for performing security assessments of the target device. This transfer of control improves security, since attentive organizations are better at managing their security than individuals are. It also aligns liability with control, as it allows relying parties to determine the security posture of consumer (and employee) devices, instead of having to blindly trust that the device owner makes prudent security decisions.

The basic underlying principle behind the approach we describe is to run a micro-kernel with full OS privileges, where the micro-kernel freezes all active routines and empties the cache, disables interrupts, and performs a computation that is configured to require all of the cache (except the space known to be used by the microkernel) in order to run quickly. The result of the computation is a checksum. If malware resists being disabled, thereby remaining at least in part in the cache in spite of the intentions to evacuate it to RAM or secondary storage, then the available space will be less than what was expected, and either the checksum result will be incorrect or the computation of it will take much longer than for a clean device. The algorithm used in the computation is designed in a manner that makes the time-space tradeoff extreme – thereby making any reduction of the available space cause a dramatic increase of the execution time. This delay is clearly distinguishable from deterministic but uncontrollable "hardware noise", such as the state of the hardware bus controller at the start of the scan, timer aliasing, and aliasing caused by different clock domains. Based on case analysis of the various ways that malware can stay in the cache, one can prove the minimum increase of time as a function of physical device properties – such as the number of processors; the processor speed(s); the bus speed; etc. The time of execution is measured at a location that is not exposed to malware – a good practical location for handsets, for example, is in the modem processor, which has a substantially smaller attack surface than the operating system does.

The checksum value and the time it took to compute this are then verified by an external entity in charge of performing the security assessment of the device. We call this the external verifier. Apart from verifying the results, the external verifier also generates the random challenge used as an input to the algorithm computing the checksum. The external verifier executes in a secure (trusted) location. Deterministic network latency improves the efficiency of the attestation process, so locating the external verifier close to the target is desirable. The external verifier can utilize a proxy executing on-board the device, provided that the proxy is running in a secure module on the device (such as the modem processor, as mentioned above, or a SIM card). The proxy provides device identity, cryptographic services, and secure high-resolution timing. Throughout this paper we refer to the remote off-board external verifier and its local on-board proxy as simply "the external verifier."

For maximum security, the type of scans we describe are performed both in a scheduled manner (such as at 2am every night) and after they are triggered by events (such as accesses to internal or external resources, or after an application calls an API to initiate the scan). The scheduled scans are intended to limit the spread of malware using devices whose usage does not necessarily trigger scans by themselves, whereas the triggered scans are used to guard access to sensitive resources. Example internal resources include a credential storage used by a password manager, a key ring, and a decryption key for a file whose access is controlled by digital rights management software. Example external resources include a user account managed by a financial institution and a corporate network. In general, after a successful scan completes, a security sensitive task is performed. Depending on the context and the calling application, this may involve loading, verifying and activating a small set of routines; performing a trustworthy device backup to a server; and scanning secondary storage for undesirable software and data. Two examples of undesirable software are routines that, while not active malware, are either known to be harmful (such as a typical trojan), or not allowed under some terms of service.

We note that our solution does not *prevent* infection, but only detects it. Therefore, it is not intended to replace traditional AV technology, but rather, to complement it. One obvious use of traditional AV software is to limit infections in the first place; another is to search the secondary storage for passive malware after it has been determined that there is no active malware. On the other hand, since traditional AV technology only prevents a portion of attacks, our technology is needed to detect when it fails.

Our technology is also not intended to replace hardware solutions such as TrustZone. TrustZone can be thought of as a very powerful sandboxing technique, but is known to have a considerable attack surface. (Indeed, one of the companies working with us has requested that our technology be used to verify the security posture of TrustZone.)

2 The Problems with Conventional Methods

To understand why conventional methods are increasingly less helpful to defend against the threat of malware, it is important to understand how they work, what infrastructure they require, and how malware authors counter them. This section aims at providing a brief review of these issues.

Deployment. Conventional anti-virus methods are based on heuristics and blacklisting – whether of code or behaviors of code. Signature-based detection screens for code patterns associated with known offending code, and behavioral detection looks at unusual access patterns known to be used by undesirable routines. By their very nature, these defense methods require constant updating, starting with the gathering of intelligence and generation of descriptors, followed by testing that these descriptors do not generate false positives, and the eventual deployment of the descriptors. Based on the amount of testing that has to be done, the cycle – from initial identification of a threat to deployment of patches – takes anywhere from on the order of a day (for consumer-deployments to PCs) to something on the order of a month (the latter typically for corporate platforms, where the testing is much more intense), or more. The worst case is for Firmware Over The Air (FOTA) patches to mobile devices: In addition to being slow, they are also very costly. In contrast, the methods we describe are not based on heuristics or matching to known unwanted code, which avoids the need for patching in order to detect active threats. (Threats that are not active, such as typical trojans, are treated separately, after screening for active threats.)

Traditionally, the use of patches involves the communication of these to client devices, where these perform near-constant scanning of processes. To an increasing extent, this screening is instead performed using cloud services supported by thin clients on the user end, which makes patch deployment a bit easier. Independently of the deployment approach, however, malware authors use the detection systems as oracles to determine when new versions of their code are sufficiently different from older version. When the malware that is being tested is detected, the malware author simply creates a new version (an effort that can be automated) and tests again. This turns traditional AV systems into tools to evade detection. The ease with which this can be done is part of the reason for the ever increasing rate of malware evolution. The method we describe does not offer malware authors the opportunity to turn the defense system into a method to customize their threat, nor is it vulnerable to zero-day attacks.

On mobile platforms, the burden of deployment of patches is greater than for regular computer platforms. There are several reasons for this. One is that the testing of patches commonly is much more involved; another is the added network cost of patching over the air; and yet another is the fact that downloading large updates is burdensome to mobile devices with their limited bandwidth and battery resources. As mentioned, the approach we advocate is not based on patching – instead, it is pre-loaded at device deployment time, and requires no maintenance after that.

Scanning. Conventional anti-virus methods are based on near-constant screening of processes in order to keep malware off of protected devices. The ongoing processing translates to a burden on computational and power resources. In contrast, the methods we describe perform on-demand detection of the security posture of a device, which reduces the impact of device resources. Moreover, they allow an alignment of control and liability, by making it possible for a server associated with a relying party to assert that a given device is free from malware before service is provided to the device.

Another common traditional approach is to monitor or filter network communications on the backend, whether to block or detect corruptions. An example of the former is screening of email attachments by ISPs and carriers; an example of the latter is detection of requests indicative of the requesting device being infected – such as requests to known command and control centers. Whereas both of these techniques are great complements to other security measures, they share many of the drawbacks of signature-based and behavioral techniques (such as being based on knowing the exact nature of the problem); and also, do not address attacks that do not use the regular backbone for infection and configuration.

In the context of conventional anti-virus methods, offending or unwanted code is typically limited to malware. Traditional methods are often unable to detect other potentially unwanted code, such as code associated with jailbreaking, or more generally, code that breaks terms of service. However, the method that is the focus of this paper detects any active code – independently of how it was installed, and independently of whether it affects the client-side detection code. In particular, we assume our client side code to be potentially corrupted. In the context of the traditional detection methods, this is counter-intuitive to most, but possible to achieve by a complete shift of the underlying detection paradigm.

3 Technical Approach

Our approach is a form of software based attestation [2, 3, 4, 5, 6, 7, 8, 10, 11, 12], meaning that it uses software to determine whether any other software is running on a device. The last few years have seen an abundance of proposals of how to perform software based attestation; these proposals differ in terms of the underlying technical approach, the protected platform and the execution time[1]. There are two principal technical approaches – code that only does self-introspection (e.g., [5, 7]) and code that also writes to larger portions of memory (e.g., [1, 3].) Our approach is of the latter type, as we believe that this dramatically simplifies the design, which in turn makes it easier to prove the security properties of the algorithm. In terms of platforms differences, there are designs for smart cards [1], embedded systems [5], cars [9] legacy systems [7] and mobile devices [3]. These differ quite dramatically from each other in terms of technical principles and assumptions on the systems. Most of the methods can only be used for the platform they were designed, as they rely on particular platform features. Our method is applicable to any computational device with a unified cache, although our primary focus is attestation of mobile devices. The execution time for our approach is below 1.5 milliseconds for a typical smartphone sold in 2013.

The security of our solution rests on several important assumptions:

Assumption 1: Secure device communication. We assume that the external verifier (or its proxy) has some way of ascertaining that the device to be

[1]It is more meaningful to speak of execution time than computational overhead, since software based attestation techniques are only run before a security-sensitive boundary is entered, as opposed to all the time.

audited is in fact the device it interacts with. We also assume that the external verifier can send data securely to the audited device, e.g., in a way that cannot be eavesdropped or tampered. Both of these aspects correspond to devices being associated with unique keys that cannot be extracted by application processes[2].

Mobile devices have such keys (used for billing purposes), accessible by their modem processors; as do Intel processors, as part of their Identity Protection Technology.

Assumption 2: Code optimality. We assume that the checksumming algorithm used in the scan is written in a near-optimal manner in terms of its footprint and execution speed, and that any modications of the algorithm would make it notably slower to execute.

For general software, code optimality is not a meaningful assumption to make; however, given the simplicity of our scan algorithm, it is quite realistic. In particular, the inner loop of our checksumming algorithm is only eight instructions long – such as short routine can be manually verified to be optimal.

Assumption 3: Deterministic hardware. We assume that the hardware will present largely the same performance characteristics each time the attestation is executed.

We note that the attestation process can tolerate a high noise threshold, such as background DMA processes utilizing a shared SoC bus, but the quantity of the noise must be possible to bound a priori.

Given these assumptions, we are now ready to describe the scan algorithm we use, starting with a definition:

Definition: Free cache. 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 cache space is referred to as *free cache* – whether it *actually* is free or not.

Since all legitimate routines will agree to be terminated, it is only active malware that will reside in free cache. At a high level, our routine simply determines whether all of free cache is indeed free. This is done by running an algorithm that requires all of the free cache space in order to execute quickly – and to measure the time this algorithm takes to run using a timer that cannot be corrupted by malware.

[2]We do not address hardware trojans or other attacks in which an attacker uses physical access to components to corrupt the hardware of a device.

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

1. **Setup:** Flush out the contents of free cache to RAM, disable interrupts, pause all processing for the duration of the attestation process, and perform a setup for the checksumming.

2. **Cache fill:** Receive a cryptographic seed from the external verifier, and overwrite free cache with the output of a pseudo-random function using this seed. (The external verifier may reside on the network and use a local proxy in the baseband station, performing timing.)

3. **Checksumming:** Receive a cryptographic key from the external verifier, compute a keyed nonlinear checksum of the entire memory contents, and report the resulting value to the external verifier.

4. **Verification:** The external verier compares the received value to a checksum it computes locally on the expected contents of the device. It also compares the time the device computation took to an expected value. If either comparison fails, then this is indicative of the device being infected; otherwise, it is free from active malware.

5. **Transact:** Perform a security-sensitive action (such as establish SSL connection, scan RAM and flash for inactive malware, etc). The external verifier controls whether the action is allowed or not, e.g., by sending/withholding decryption keys used to access sensitive data on the device.

Why it is done: Step 1 attempts to establish an isolated execution environment. If successful, this step ensures no other software will execute, no cache evictions will take place and the attestation scan will complete in optimal time. The subsequent attestation process verifies the integrity of this isolated execution environment in order to establish a Secure eXecution Environment (SXE).

In step 2, free cache is filled with a pseudorandom string that depends on keys obtained from the external verifier in step 1.

In step 3, the cryptographic checksum is computed from the contents of the cache – the pseudo-random string and the correct (good) scanning code. Attempts by malware to use contents that are not in the cache (in order to hide itself in the cache) will result in dramatically increased computation time. (This is described in greater detail in the next section.)

In step 4, the external verifier is given assurance that steps 1 through 3 were performed correctly, based on the checksum value and the time it took to compute it. One can utilize various techniques to avoid issues with latency variance.

In step 5, after the verifier asserts the integrity of the isolated execution environment, it releases keys or tickets used to access the security sensitive resource. A loader loads a payload and verifies that its digest is correct, based on a digest that is part of the micro-kernel (and which is therefore already verified for validity, using the checksum.) The payload consists of a whitelisted application (whitelet) with optional data. Sensitive data contained in the payload is protected, and access to the data requires one or more security keys, provided by the external verifier (but only after a successful scan.) These keys are deleted after they are used. The whitelet executes entirely in the context of the SXE with assurance of confidentiality and integrity.

The attestation process can be run during boot time to create a static root of trust (an OS secure boot), or at any time during normal system operation to create a dynamic (temporary) root of trust. A dynamic root of trust allows a whitelet to execute in a secure environment even when the host operating system is infected.

4 Adversarial Strategies

The detection mechanism described above evicts all cache contents (except for what constitutes the detection mechanism itself); fills the free space with non-compressible data; and then computes a keyed checksum of the entire cache contents (including the detection code itself.) If the scan indicates the absence of active malware, then a loader will be run, which loads and verifies a collection of processes to be executed, allowing only those matching the stored message digest to start. These processes will be granted access (by the external verifier) to the resources they need to run. Encrypted internal resources are unlocked by communicating the appropriate key; external resources are unlocked either using keys or traditional tickets.

Passive malware (i.e., malware that does not interfere with the scanners control of hardware resources) will not be able to interfere with the processes to be run after the detection mechanism is run, since passive will be evicted by the detection mechanism and the detection mechanism turns off interrupts. Moreover, the contents of the interrupt table will not be restored until after the all security sensitive tasks have completed, and so, passive malware will be disabled for the entire duration of the security sensitive task. (If this task involves scanning RAM and secondary storage for undesirable code and data, or synchronize the contents of the device with a cloud server that performs such as scan, then the undesirable processes can be killed and removed while being evicted.)

Active malware, on the other hand, will fight back. It may interfere with the task of the detection mechanism. We make the very pessimistic assumption that the active malware may have corrupted the detection code, which is assumed to be known in full detail by the malware author. Based on this assumption, we will describe an array of adversarial approaches, and briefly argue why each one will fail.

- **Silent attack.** The adversary causes the corrupted device to not communicate with the external verifier. This is not a helpful strategy, though, since the external verifier will not grant access to the sensitive resources without first running the detection algorithm. If a scheduled scan is missed (which could happen for legitimate reasons), the external verifier will flag the device, demanding a scan to be performed when the device shows up on the network next.

- **Naive corrupted code attack.** The adversary modifies or augments the detection code (or its associated data) to perform a different task than what is desirable – for example, to permit a corrupted process (whose digest will not be correct) to be loaded and run. In the naive corrupted code attack, the adversary does not attempt to hide the modifications. Therefore, as the external verifier compares the checksum of the cache contents to the checksum it computes, the modification will be apparent, except with a negligible probability.

- **Advanced corrupted code attack.** The adversary modifies or augments the detection code (or its associated data) to perform a different task than what is desirable, and to hide this fact to the checksumming function. For example, the code modification can displace (e.g., to RAM) some of the non-compressible data that is generated by the detection mechanism. When the checksum operation is performed, cache misses will bring in these values to the cache. However, based on the order in which the checksum operation accesses the cache, each such access will result in a cache miss, the aggregation of which will correspond to a measurable delay. (A version of this attack generates the non-compressible data on demand, when needed by the checksumming code. Due to the manner in which the non-compressible data is generated and stored, though, this will result in an even more substantial delay.) The delay, no matter what caused it, is evidence of an active attack.

- **Evil twin attack.** In the evil twin attack, a virgin version of the detection code is kept in the cache, but is not executed. Instead, a corrupt routine is run. The corrupt routine is potentially limiting its footprint by making calls in to the virgin code using Return Oriented Programming (ROP) techniques. The corrupt code must take up some space, and must displace either code or the non-compressible data that is needed for the computation of the checksum. According to the same argument as above, this will be detected.

- **Evil interrupt attack.** The evil interrupt attack works like this: The adversary replaces the virgin detection code with a version that (1) leaves the interrupts enabled (whereas the virgin code *dis*ables them), and (2) rewrites itself to the virgin code after the execution has passed the location where the interrupts normally would have been turned off. This results the correct checksum being computed, but without the interrupts being turned off. An interrupt is then triggered after the (correct) checksum has been computed. The associated routine performs a malicious task, such as rewriting the loader. This attack fails since the code loaded by the interrupt will be stored in cache and will therefore displace other data; which will be detected as above. There are many alternative ways to address this problem, although these are beyond the scope of this paper.

- **Evil processor attack.** A processor that is not used by the detection mechanism is controlled by the adversary, and is used to perform a malicious task after the correct checksum has been computed. To avoid causing a displacement of data from the cache, the code running on this processor may be flagged as non-cacheable, and therefore, run directly from RAM; while this will cause the execution to be slow, that will only affect the execution speed of the corrupted processor. The simplest way to avoid this attack is to let the detection mechanism make use of all processors that are potentially reachable by malware.

The list of potential attacks provided above is not exhaustive; instead, it is illustrative of our approach. While the countermeasures we describe are sufficient to address the above attacks, the system we describe incorporates additional countermeasures. These are beyond the scope of this paper.

About the authors

Markus Jakobsson is a recognized fraud researcher who has contributed to research on phishing, authentication, mobile security and cryptographic protocols. He is the CTP and founder of FatSkunk. Guy Stewart is a seasoned software professional with field proven experience in distributed systems and strong security solutions. Guy is the VP of Engineering of FatSkunk.

FatSkunk is a software company producing malware detection technology suited for devices spanning mobile platforms, the Internet of things and traditional PCs. What makes FatSkunk's approach markedly different from the traditional anti-malware techniques is that it is not heuristic, and comes with an assurance based on physics, and more particularly, based on knowledge of the performance of hardware components, such as the processors, the bus and various components of the memory structure. It is not limited to detection of malware, but can be used to detect any unwanted active process, such as code associated with jailbreaks.

References

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

[2] 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.

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

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

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.