Secure Remote Attestation

Markus Jakobsson
Agari Inc.[*]

Abstract

More than ten years ago, a devastating data substitution attack was shown to successfully compromise all previously proposed remote attestation techniques. In fact, the authors went further than simply attacking previously proposed methods: they called into question whether it is *theoretically possible* for remote attestation methods to exist in face of their attack. Subsequently, it has been shown that it *is* possible, by relying on self-modifying code.

We show that it is possible to create remote attestation that is secure against all data substitution attacks, *without* relying on self-modifying code. Our proposed method relies on a construction of the checksum process that forces frequent L2 cache overflows if any data substitution attack takes place.

1 Introduction

Early malware is probably best described as *mostly harmless*. It typically made no effort to hide, and rarely did any damage to the systems it corrupted. For example, the *brain virus*, commonly credited for being the first malware instance, is representative: it simply printed a message containing the names and contact information of its creators. For years, malware development was in the hands of nerdy college students, seeking nothing but maybe a bit of fame among their peers.

The rise of the Internet, though, and its daily use for financial and political purposes, has provided ample motivation for malware authors to instead use their coding expertise to steal funds and data, whether for personal enrichment or political motives. This opportunity has led to the development of a plethora of innovative malware monetization strategies, whose functionality range from stealing or encrypting data to initiating financial transactions. As countermeasures to malware have been developed and deployed, malware and malware distribution techniques have evolved to avoid detection. Examples of such developments include obfuscation tools, such as crypters; and techniques

[*]Copyright 2018 Markus Jakobsson, all rights reserved. Patent Pending.

to hide and be persistent, such as those used by rootkits. Similarly, a thriving market has developed around zero-day attacks, fueled by the opportunities afforded by these and enabled by the large and complex body of software in use.

Whereas there is an array of heuristic methods developed to detect rootkits and—to some extent—zero-day attacks, traditional Anti-Virus methods do not offer any *assurance* of detecting these attacks. As a case in point, the custom malware used to attack Bangladesh Bank [15] was only discovered after its effects made its existence evident, as was the malware used to attack the Ukrainian power grid [2]. Similarly, Stuxnet was only discovered after it "broke loose" from its intended targets and started to spread in an unintended manner [8].

To enable detection of *unknown* malware—including any type of memory-persistent malware—the concept of *remote attestation* [5, 6, 7, 9, 10, 11, 12, 13, 14, 16, 19] was proposed. Remote attestation relies on a piece of code computing a checksum on itself in a way that, if modified, will either generate the wrong result or cause a measurable increase of the execution time. The checksum is reported to an external verifier that checks both its correctness and that it was computed "fast enough"—if both of these conditions are satisfied, then the checksum process is determined not to have been corrupted. This is used to bootstrap the computation on the audited system. The beauty of the concept underlying remote attestation is that it shifts from a paradigm based on blacklisting to one of whitelisting, thereby sidestepping the problem of unknown malware instances.

However, ten years ago, the very notion of remote attestation was shaken to its core by a cunning data substitution attack [18, 17] that defeated all then-existing remote attestation techniques developed for computer architectures with separate data and instruction caches[1]. The general principle of a data substitution attack is to make a *malicious* checksumming process compute the checksum of the legitimate checksum process, as opposed to a legitimate process computing a checksum on itself. Here, the two processes could be identical but for the action taken after the checksum is computed and approved: while the legitimate process would perform a desirable action, the malicious process would load and execute an unwanted routine.

While a general version of the data substitution attack can be detected by using the program counter and data pointer as inputs to the checksum computation [12], this countermeasure will not block the data substitution attack described in Wurster's Masters thesis [18] and an associated conference paper [17]. In these publications, multiple ways were shown using which an adversary can perform a data substitution attack by stealthily causing the L1 data cache and L1 instruction cache—when reading the *same* virtual memory location—to load *different* contents. One of the attack variants (see Figure 1) is based on changing the access control privileges for the checksum process from *code* to *data*, thereby causing an exception to be triggered when the processor attempts to load the process into the L1 instruction cache. An exception handler instru-

[1]Most remote attestation techniques fall in this class; one exception is remote attestation for simple smart cards [4].

mented by the adversary modifies the address to be accessed when triggered, thereby forcing the malicious process to be loaded. The exception is only triggered by read requests for the instruction cache, and no redirection takes place for read requests for the data cache.

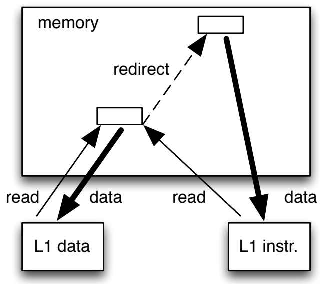

Figure 1: *The Figure illustrates a data substitution attack in which two read requests to the same address result in different content being loaded. The request from the data cache is served without a redirection. However, since the access control privileges for the contents have been set to "data", the read request from the instruction cache triggers an exception and a subsequent redirection.*

Later, it was shown by Giffin et al. [3] that the attack can be overcome using self-modifying code. However, self-modifying code presents other difficulties, among other reasons due to its use in malicious software, and so, a remote attestation method that does not rely on this approach would be beneficial. We introduce a new design approach to counter *all* data substitution attacks, and without relying on self-modifying code. Its security is based on how hierarchical memory accesses work, and its computational efficiency is comparable to that of the fastest types of remote attestation previously proposed. Our approach uses no new hardware, and is not specific to any particular instruction set or device type. We focus on architectures using *inclusive caching*, i.e., where the contents of the L1 cache are duplicated in the L2 cache.

2 New Design Principles

Wurster [18, 17] describes several versions of their redirection attack, for a collection of common processor architectures. The result of all of the versions is that read requests made for the L1 instruction cache and the L1 data cache—associated with the *same* virtual memory address—resolve to *different* physical addresses. This is sometimes achieved by redirecting reads for the instruction cache, sometimes by redirecting reads for the data cache. Independently of what adversarial approach is taken, it results in a data substitution attack that cannot be detected simply by monitoring the program counter and data pointer.

Understanding the problem. We assert that one can detect *any* data substitution attack by forcing any memory redirects to cause "additional" cache misses. This is not a property previous remote attestation proposals have. For example, neither the Pioneer proposal [12] nor Qualcomm's FatSkunk approach [6, 7] avoid attacks based on memory redirects. Looking at why these approaches fail is instructive. Both involve a checksum process that fits in the L1 cache, and so, the critical memory redirect can take place before the remote attestation has even started. If the adversary causes the corrupt version of the routine to be loaded into the L1 instruction cache using a memory redirect, then any potential delay incurred from the redirection will very obviously not affect the execution time of the routine. While it is not sufficient to make the checksum process too large to fit in the L1 cache for us to counter the attack, we do not believe any process that *does* fit can be secure against redirection-based data substitution attacks.

Forcing cache misses. The example above shows that in order to detect the delays associated with malicious memory redirection, the memory redirects need to take place *after* the checksumming process has started. Practically speaking, this corresponds to a checksum process that is substantially larger than the L1 cache, thereby causing frequent flushing of the L1 cache. Moreover, it requires that the adversary is unable to redirect reads from one L2 location to another, as the delay associated with this would be limited to the diminutive delay[2] associated with the operation of the exception handler. For maximum detectability, we therefore need to ascertain that the use of a memory redirection attack results in an L2 cache miss—and that the "legitimate" execution of the checksumming process does not.

Figure 2 shows an attempt to a data substitution attack based on redirection, where the checksum process takes up the entire L2 cache and where the redirected read results in an L2 cache miss. One must remember that an adversary can also force redirects to be made from one L2 location to another, *even if* the checksum process takes up the entire L2 space. However, as shown in Figure 3, if the process is the size of the L2 cache, this necessitates the displacement of a portion of the checksum process from the L2 cache to slower memory. When the displaced portion is requested, *that* results in a delay.

On one typical PowerPC CPU, an L1 cache miss results in a 40 cycle penalty, whereas an L2 cache miss is associated with a penalty of at least 500 cycles; therefore, if one were to force approximately 4000 L2 cache misses on average for a process running on a 3.4GHz processor, a 0.5ms delay would be incurred. The typical network latency variation for a LAN is about half of that, making remote attestation even over a LAN entirely practical, and remote attestation carried out across a connection with lower latency variance a breeze.

[2]From a practical perspective, very short delays may be hard to detect due to the variance in communication times between the audited system and the external verifier.

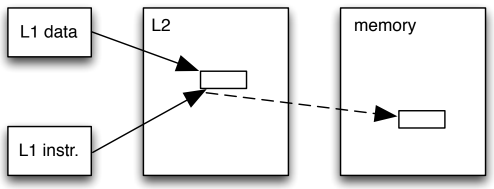

Figure 2: *The Figure shows an example data substitution attack mounted on a process residing in the L2 cache, where some read operations get redirected from the L2 cache to slower memory. The resulting delay is detected.*

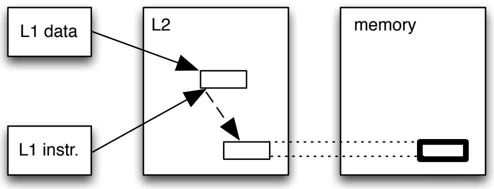

Figure 3: *The Figure shows an example data substitution attack mounted on a process residing in the L2 cache, where some read operations get redirected from one L2 location to another. However, the legitimate checksum process requires the entire space of the L2 cache, so the malicious segment must displace some other segment of the checksum process. When displaced data (moved to the location of the bold rectangle) is requested, an L2 cache miss occurs, and the resulting delay is detected.*

Amplifying the adversary's penalty. To amplify the delay associated with redirection attacks, it is beneficial to maximize the number of expensive cache misses suffered by a malicious process. This can be achieved in various ways:

- **Reduce locality of reference.** It is beneficial for the checksumming process not to exhibit strong locality of reference—whether for instruction or data reads. The result of low locality of reference is an increased number of L2 cache misses incurred by a malicious process that uses redirection. As a special case of this goal, it is desirable that the L1 instruction cache is automatically flushed for each iteration of the checksum process.

  It is crucial that this absence of locality of reference is an *inherent* property of the process, as opposed simply to how it is implemented, or an adversary may be able to create a malicious process with better runtime behavior than the legitimate process, e.g., by changing the memory layout of the malicious process. Such a process would potentially go undetected.

- **Understand cache prediction.** Cache predictors affect the execution time of processes—whether malicious or not. Whereas it may at first seem that successful cache prediction is always a liability in the context of remote attestation, the truth is more complex. Successful *L2* cache prediction is undesirable as it makes detection harder by speeding up the execution of a malicious process. Successful prediction of L1 misses, on the other hand, is not harmful—and is in fact slightly beneficial as it reduces the computational time for the legitimate checksum process.

A good way to balance these conflicting design criteria is to compute the location of the next access, then perform a computation taking the same number of steps as an L1 cache miss, followed by the access itself. An L2 cache miss would cause a notable delay.

3 A Detailed Construction

We are now ready to start describing the details of the solution we propose, following the requirements outlined in section 2. This description will be broken down in terms of *computational structure*, *memory layout*, and how to *configure our proposed remote attestation process.*

3.1 Computational structure

The checksum process can be broken down into *segments of code* used to read input values, compute a checksum, forcing cache misses and amplifying the penalties associated with cache misses.

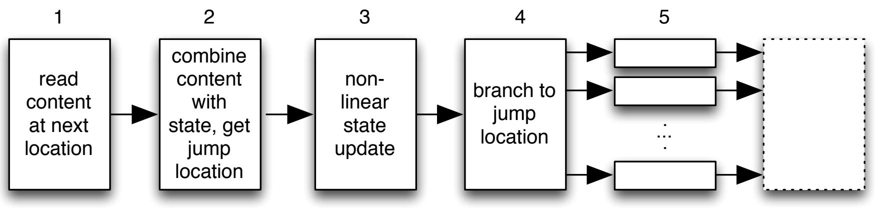
1
2
3
4
5
read content at next location
combine content with state, get jump location
non-linear state update
branch to jump location

Figure 4: *The Figure shows the principal execution flow associated with the checksumming process.*

Referring to the steps shown in Figure 4, the checksum process can be described as follows:

1. The contents of a selected L2 cache location are read to a register, causing the corresponding cache line to be stored in the L1 cache.

2. The contents that were read are combined with a checksum state s_1.

3. The checksum state s_1 is combined with the contents of selected registers, and stored in a secondary state s_2.

This step preferably takes the same amount of time as an L1 cache miss incurs. Given the use of cache predictors for the instruction pipeline, that leads to an efficient execution with no delay after the execution of the multi-branch instruction in step 4—unless an attacker is performing a redirect-based attack. However, if a redirection attack is taking place there will be a delay, as loading a cache line to the L2 cache takes much longer than to the L1 cache.

4. The checksum state s_1 is used to algebraically select a branching location for a multi-branch with a large number of different procedure locations. These procedures correspond to the rectangles shown in step 5.

   A register j is used to store the location to jump to at the end of the procedure. The location j is independent of the selected procedure, and corresponds to the dotted-line component shown at the very right in Figure 4.

5. In each procedure instance shown in Step 5 of Figure 4, the state s_2 is modified in a different way and the result of the operation is stored in the register for state s_1. The procedure ends by jumping to the location j, set in step 4. This corresponds to starting the execution of the next code segment.

We refer to the code of steps 1-4 above as the *control portion* (as this part controls the execution flow), and the code of step 5 as the *procedure portion* (as it consists of a large array of procedures, each causing different state modifications.)

A sequence of code segments, as described above, is daisy chained to form a circular structure, a portion of which is shown in Figure 5. Here, different segments of the chain use distinct control elements, while the elements corresponding to the procedure portion are reused for each segment in the daisy chain. At the "end" of the daisy chain, it is determined whether to start a new iteration based on whether the entire input string has been checksummed.

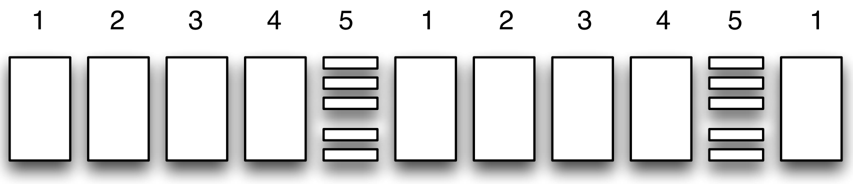

Figure 5: *The Figure shows daisy chaining of code segments corresponding to the outline shown in Figure 4.*

By design, the amount of memory required to store the instructions executed for one full rotation of the daisy chain exceeds the size of the L1 cache, thereby automatically forcing the L1 instruction cache to be continually flushed. In particular, the amount of space needed for one full cycle of the daisy chain corresponds to the amount needed to store the instructions of steps 1-4 and the instructions for the selected procedure of step 5, times the number of instances.

3.2 Memory layout

We partition the L2 memory space into *sections* of contiguous bits. Each memory section contains code corresponding to *setup*, a *control portion*, a *procedure portion*, *iteration control* and *concluding code*, or a combination of these. The setup code turns off interrupts, receives initialization values and applies these. The control portion updates the checksum state—where each control portion instance does this in a slightly different way[3]. It then uses the state to select a procedure to which it branches. Each procedure, of which there is a very large number, modifies the checksum state in a distinct manner. The iteration control is used to determine when the checksumming has completed, and the concluding code transmits the checksum result to an external verifier, then loads the routine to be run after the checksum has been approved by the external verifier. The validity of the loaded code is checked using a hash function, where the result is compared to the expected result, which is stored as part of the checksum process. For simplicity, we assume hardware support for a hash function, but note that this code would otherwise be part of the concluding code.

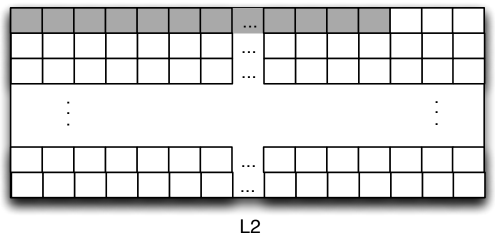

Figure 6: *The Figure shows the memory layout of the checksum process. The L2 memory is divided into N_2 code sections, a portion of which (marked in grey) contain code for setup, the control portions, iteration control, and the concluding code. The remaining code sections (in white) contain code for the procedure portion.*

The control portion, in turn, can be broken down into setup and concluding code on one hand, and code executed in the loop on the other hand. If the control portion does not fit in one memory page, the "loop code" should be distributed as evenly as possible between the pages used for control code so that the number of accesses per page performing the checksumming is approximately the same.

[3]The differences between the instances makes an adversary unable to represent the code using less space without inflicting a run-rime "unpacking delay".

Without lack of generality, we assume that the control portion requires c pages and that the loop code is distributed evenly between these.

3.3 Configuring the remote attestation process

Determining the length of the daisy chain. As shown in Figure 6, the L2 memory space is divided into N_2 code sections, each one of which has size S(section), where a section consists of an integer multiple of cache lines. Here, $N_2 = S(L2)/S(\mathit{section})$, where $S(L2)$ is the size of the L2 cache. Denoting the size of the L1 cache by $S(L1)$, we see that the number of code sections that fit in the L1 cache is $N_1 = S(L1)/S(\mathit{section})$. To guarantee that the L1 instruction cache is flushed for each iteration of the checksum process, each iteration should therefore execute at least N_1 sections. By setting the number of daisy chained code elements to N_1, this is guaranteed.

A concrete example. For clarity, let's consider one possible configuration, in which each section corresponds to 16 words, each word 32 bits long. Thus, $S(\mathit{section}) = 16$, where the size is measured in 32 bit words; this coincides with a typical cache line size for a mobile device.

A typical mobile device has a 1 MiB L2 cache (2^{18} words) with a 4KiB page size (1024 words), and one or more processors with a 4KiB L1 instruction cache and a 4KiB L1 data cache. Therefore, S(L2)=2^{18}, S(L1)=1024 and $S(\mathit{page}) = 1024$. Thus, we have $N_2 = S(L2)/S(\mathit{section}) = 16384$ and $N_1 = S(L1)/S(\mathit{section}) = 64$.

To continue with the example, we'll assume that the code (marked in grey in Figure 6) that does not correspond to procedures requires space corresponding to 105 sections—corresponding to two pages (i.e., $c = 2$). This leaves $16384 - 105 = 16279$ sections for the procedures. In each control section, state s_1—which is a 32 bit number—is converted to a branching location. One can compute a tentative branch location as (s_1 `AND FFFFFFF0`)—effectively generating one of the 16384 section addresses. It is then determined whether the result is less than $105 \times 16 = 1696$, which corresponds to a location outside the procedure space; if so, then one of the 28 most significant bits of the tentative branch location is set to 1, thereby remapping the location into the range of procedure addresses. Practically speaking, this can be approximated with a uniform distribution.

A look at the locality of reference. We note that the process we described exhibits a very limited locality of reference for execution, beyond the execution of instructions within a code section. Therefore, if the adversary has instrumented the system to trigger an exception when code corresponding to one or more pages is accessed, then *each* cache line read within such pages will trigger the exception.

To achieve a similar absence of locality of reference for the data accesses, we avoid linear reads of the L2 cache. This is achieved if the distance between two consecutive data reads is $d = S(\mathit{cache\,line}) + 1$ before the reduction modulo the size of the L2 cache. This way, no N_1 consecutive read accesses will be made to one and the same cache line. Since d is odd, all $S(L2)$ addresses will be read without getting stuck in a smaller cycle. Coming back to the example above, where we assume $S(cache\,line) = 16$, this would mean that $d = 17$.

Function properties. Step 3 of Figure 4 combines the content of a collection of registers with the state register s_1. These registers are modified in the procedures. For example, in one procedure instance, the existing contents of register r_5 may be XORed with the state s_2; in a second procedure instance, the contents of register r_6 are rotated two bits to the right; while in a third instance, the contents of r_5 and r_6 are added to each other and stored in register r_3; and in a fourth instance, the contents of the processor flags are XORed with the contents of the register keeping state s_1. Each procedure instance comprises a different set of such operations, modifying the contents of the registers. The operations are selected so that the effect of a large portion of pairs of procedures are non-commutative. This implies that the order in which the procedures are invoked matters to the final outcome. Consequently, the checksum function implements a non-linear accumulator.

Setup and Concluding code. The setup code turns off interrupts; receives initialization values for the states s_1 and s_2. Since these two states correspond to multiple registers, and the state impacts both the computation and the selection of procedures, the entropy makes it impossible for an adversary to exhaustively pre-compute any non-negligible portion of results or anticipate the program flow. Moreover, to increase the entropy of the system, the first read operation would be performed from a location *start*, selected by the external verifier as an integer in the the range $0 \leq start < S(L2)$. This value is also received as part of the setup. In addition, the setup code initializes the counters used for the iteration control.

The concluding code transmits the final checksum value to the external verifier; awaits a response; then loads and verifies the correctness of the process to be run in a safe state. In some instances, the response contains a key (to be used to decrypt a resource); in other cases, it contains the result of an operation that the audited device requested.

Some remarks.

- The process we have described works both for single-core and multi-core processors; however, it is worth noting that multi-core implementations cause bus contention every time two or more cores make simultaneous accesses to data in the L3 cache or DRAM. This amplifies the delay associated with attempting a data substitution attack. It should also be noted that this type of contention takes place —even for the legitimate checksum process—on systems with shared L2 caches. Legitimate L2 contention can simply be taken into consideration when determining what the expected computational time is.

- We have not discussed how to warm the L2 cache prior to the execution. One straightforward way to do this is to run the checksum process twice in a row—receiving different initialization values for each time—and ignoring the checksum value from the first computation.

- To increase the entropy for the data accesses, one can instead of using a fix value for d use a random odd number set by the external verifier. The number would be chosen to avoid multiple reads during a sequence of N_1 consecutive reads, where any two of these access the same cache line. To keep the security argument simple, however, we do not use this approach.

4 Security Analysis

We provide a security argument that is not specific to a particular instruction set, nor to the performance of the hardware or the size of the caches. Thus, our argument is, with necessity, based on the *structure* of the building blocks and their effects on the memory management, as opposed to the specific sequence of instructions in an implementation. We will break down our security argument into a collection of separate arguments, covering the principles that, when combined, results in the security of the proposed approach.

The computation cannot start early. The computation of the checksum function depends on the initialization values received during the setup phase. These have a sufficient entropy that it is not feasible to precompute a non-negligible portion of the potential $(init, checksum)$ pairs a priori, where $init = (s_1, s_2, start)$.

Moreover, the operations of the procedures of the checksum function are non-commutative[4], by virtue of of consisting of computational elements that by themselves are non-commutative. For example, using a combination of XOR and bitwise shifting in the function makes it non-commutative. Therefore, it is not possible to pre-compute any portion of the checksum function ahead of receiving the initialization values.

The absence of efficient compression. The checksum process consists of code sections with substantial individual variations. This protects against the creation of a more compact process. In contrast, assume for a moment that the code for a large number of different control segments were identical to each other, they could be *"rolled"*—by which we mean the opposite of *unrolled*—with a rather small computational penalty. Similarly, if the individual procedures all belonged to the same family of functions, that could lead to more compact representations of the code. For example, if the ith procedure were to perform the

[4]Note that the checksum function does not implement a hash function, and does not satisfy collision-freeness, since this is not needed to guarantee security.

simple function of adding the number i to the input, then this would enable a notable code size reduction without an associated ballooning of the computational costs.

While computationally expensive compression methods could potentially be used to reduce the effective size of the code to some extent, there is a substantial run-time cost associated with such methods. The individual variation of the code elements means that the checksum code cannot be modified in a manner that both reduces its size and maintains its approximate execution time. In particular, it is implausible that there exists a more compact representation of the checksum process that is at least one memory page smaller than the original version, and where the modified checksum process has a comparable execution time to the original.

The guarantee of delays. The checksum process is the same size as the L2 cache. Since redirection attacks operate on the granularity of pages, an attacker would have to cause an L2 cache miss for at least one page (as shown in Figures 1 – 3) as a result of performing a redirection-based data substitution attack.

Similarly, performing a data substitution attack that does not rely on redirections necessitates L2 cache misses, since—simply speaking—to use one process to checksum another process, both of these processes need to be stored and accessed.

The amplification of delays. The goal of the proposed construction is not only to make delays an inevitable result of any data substitution attack, but also to maximize these delays. Recall that the delay is incurred when information residing neither in the L1 nor L2 cache is requested. Thus, delays are maximally amplified if two reads to the same location *always* result in two L1 cache misses. In other words, it is important that, during the execution of the checksumming process, both the L1 data cache and the L1 instruction cache are *"constantly flushed"*.

Establishing that the L1 data cache is constantly flushed is relatively straightforward. This is because the distance d between two read locations is $d = S(cache\,line) + 1$. Thus, if the a first read operation is performed at a location x, then the next i reads (for $i \leq N_1$) will be made from locations $[x + i \times (S(cache\,line) + 1)]_{S(L2)}$, where $[\,]_{S(L2)}$ denotes reduction modulo $S(L2)$. As long as $S(L2) > (S(cache\,line) + 1) \times N_1$, any N_1 consecutive read requests will be from different cache lines. This is always satisfied by existing cache hierarchies.

Since d is odd and $S(L2)$ is a power of two, it will take $S(L2)$ additions of d modulo $S(L2)$ before the starting point is arrived at. In other words, the checksumming process reads the entire L2 cache, never reading two words from the same page before first having flushed the L1 data cache. Therefore, each data read request will result in an L1 cache miss. Since all words of the L2 cache are read, each page will be requested the same number of times.

Consider now the L1 instruction cache. Recall that there are N_1 code elements in the daisy chain, each one of which comprises a control portion and a procedure portion. Here, $N_1 = S(L1)/S(section)$, where $S(L1)$ is the size of the L1 cache. For each round of the daisy chain, we know that we will execute N_1 *control* segments and N_1 *procedure* segments. The former are guaranteed to be distinct, and the latter are *likely* to be distinct—but not guaranteed to be so. However, it is plain that the N_1 control segments will flush the L1 cache, since each one is of size $S(section)$ and $S(L1) = N_1 \times S(section)$. Thus, the instruction read accesses the execution of the checksumming process never reads two words from the same page before first having flushed the L1 instruction cache. In other words, this means that the L1 caches will be automatically flushed between two accesses to the same cache line, whether we are considering the instruction or data cache, which, in turn, implies the maximum L2 cache miss penalty, should any data substitution attack take place.

Turning now to the number of accesses per page, we note first that the control portion is expected to fit in a small number of pages, with the procedure code taking up the remaining pages in the L2 cache. In our example in section 3.3, the control portion fits in two pages. More generally, assume the control code requires c pages, where the loop component is evenly divided between these. This means that each "control page" will be accessed $S(L2)/c$ number of times since the loop will be executed exactly once for each of the $S(L2)$ words in the cache, as these are read.

We can prove that the "procedure pages" will be accessed *at least* $S(L2)/c$ number of times (for $c > 1$) with all but a negligible probability. Recall that that the selection of the procedure is pseudo-random with a close-to-uniform distribution[5]. The distribution of accesses among the pages containing procedure code is governed by a binomial distribution, where each page corresponds to a different-colored ball and where $S(L2)$ balls are drawn with replacement. If we wish to compute the probability of less than $S(page)/c$ accesses of one control page, we can use Chernoff's inequality for estimating the cumulative distribution function. This states that $F(k; n, p) \leq exp(-\frac{1}{2p}\frac{(np-k)^2}{n})$ where $k = S(page)/c$ is the minimum number of page requests, $n = S(L2)$ the total number of read accesses, and $p = S(page)/S(L2)$ is the probability that a particular read access will be made to the page of interest[6]. Simplifying the above expression gives us $F(k; n, p) \leq exp(-(\frac{c-1}{c})^2 \times S(page)/2)$. Using the example in section 3.3, where $c = 2$ and $S(page) = 1024$, we get an upper bound on the probability of $exp(-128)$, which is less than 2^{-184}. This is the probability that there will be

[5]We make the simplifying assumption that the distribution is uniform here in spite of the slight bias introduced by the procedure selection method described in section 3.3. This uniform distribution does not apply to potential pages that contain a combination of control content and procedure content. However, these have already been considered in the "control" case, and can be ignored here.

[6]Note that the probability is a *lower bound* of Chernoff's lower bound since we are making the simplifying assumption that the entire L2 cache is occupied by procedure code, whereas in reality, only a portion $(N_2 - c)/N_2$ of it is. The smaller the procedure portion, the greater the probability of any particular portion of it to be accessed for each round.

$S(page)/c$ or fewer accesses to a page that is corrupted by an adversary, each one of which will result in a delay.

When $c = 1$, we can prove that the pages will be accessed at least $3/4 \times S(L2)$ times, but for a negligible probability. The argument is the same as above, except that we use $k = S(page) \times \frac{3}{4}$. This results in a probability upper bounded by $F(k; n, p) \leq exp(-(S(page)/32))$, which is even smaller than the probability for $c > 1$. Thus, we can conclude that each page will be accessed at least $S(page)/c$ times when $c > 1$, or at least $S(page) \times \frac{3}{4}$ times when $c = 1$. The delay associated with accessing a page not in the L2 cache is guaranteed to equal the product of this count and the delay per access to the corrupted page.

The memory layout cannot be optimized. An attacker wishing to speed up the computation by remapping memory before the start of the checksum process will not be successful. Considering instruction accesses first, it is clear that the access pattern for the procedure calls is not a priori predictable to an adversary, since it depends directly on the initialization values received from the external verifier. Moreover, the adversary cannot change the L2 memory mapping in the context of data access without also changing it for instruction access, and vice versa. Therefore, at most, the adversary can change the ordering of the sections (which are units of S(section) words); doing so, however, will incur the tremendous computational cost of performing a reverse mapping for each multi-branch instruction.

The guaranteed detection of attacks. Based on the above analysis, we can conclude that a data substitution attack will be be detected as long as the guaranteed delay exceeds the latency variance between the external verifier and the audited device. For situations where this does not hold, however, the standard approach of iterating the checksum process multiple consecutive rounds brings up the necessary delay above the detection threshold. Security, in other words, has been reduced to a matter of proper configuration in terms of the number of iterations, and depends directly on the measured network latency variance.

References

[1] Cyware. UK: Multiple Hospitals Cancel Hundreds of Operations Due to a Computer Virus, `cyware.com/news/uk-multiple-hospitals-cancel-hundreds-of-operations-due-to-a-computer-virus-fbecf7d7` Nov 1, 2016.

[2] E-ISAC and SANS. Analysis of the Cyber Attack on the Ukrainian Power Grid Defense, `www.nerc.com/pa/ci/esisac/documents/e-isac_sans_ukraine_duc_18mar2016.pdf`.

[3] J. T. Giffin, M. Christodorescu, and L. Kruger. Strengthening software self-checksumming via self-modifying code. In *ACSAC*, pages 23–32. IEEE Computer Society, 2005.

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

[5] 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.

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

[7] M. Jakobsson and G. Stewart. Mobile Malware: Why the Traditional AV Paradigm is Doomed, and How to Use Physics to Detect Undesirable Routines. In *BlackHat*, 2013.

[8] D. Kushner. The Real Story of Stuxnet—How Kaspersky Lab tracked down the malware that stymied Iran's nuclear-fuel enrichment program, IEEE Spectrum, `spectrum.ieee.org/telecom/security/the-real-story-of-stuxnet/`.

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

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

[15] S. Shevchenko. Two Bytes To $951M, `baesystemsai.blogspot.com/2016/04/two- bytes-to-951m.html`.

[16] 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.

[17] P. C. van Oorschot, A. Somayaji, and G. Wurster. Hardware-assisted circumvention of self-hashing software tamper resistance. *IEEE Trans. Dependable Sec. Comput.*, 2(2):82–92, 2005.

[18] G. Wurster. A Generic attack on hashing-based software tamper resistance. Master's Thesis, Carleton University, April 2005.

[19] 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.