Proceedings of the 12th USENIX Security Symposium

Washington, D.C., USA
August 4–8, 2003

Establishing the Genuinity of Remote Computer Systems

Rick Kennell & Leah H. Jamieson
Purdue University School of Electrical and Computer Engineering
{kennell,lhj}@purdue.edu

Abstract

A fundamental problem in distributed computing environments involves determining whether a remote computer system can be trusted to autonomously access secure resources via a network. In this paper, we describe a means by which a remote computer system can be challenged to demonstrate that it is *genuine* and trustworthy. Upon passing a test, it can be granted access to distributed resources and can serve as a general-purpose host for distributed computation so long as it remains in contact with some certifying authority. The test we describe is applicable to consumer-grade computer systems with a conventional network interface and requires no additional hardware. The results of the test can be conveyed over an unsecured network; no trusted human intermediary is needed to relay the results. We examine potential attacks and weaknesses of the system and show how they can be avoided. Finally, we describe an implementation of a genuinity test for a representative set of computer systems.

1 Introduction

For most types of valuable real-world objects, there are generally accepted methods of assessing their genuinity in a non-destructive fashion. Archimedes determined that King Heiron's crown was not made of pure gold by noticing that the mass and volume of the crown did not match the known density of gold. We can discern real diamonds from imitations by examining their electrical characteristics and their indices of refraction. We can determine that money is not counterfeit by carefully studying its watermarks and other identifying features.

Unfortunately, we have few such measures for computer systems. When answering the question of whether a computer system is real, we can only verify that it looks like a computer and acts like a computer. Unfortunately the dynamic nature of a programmable computer means that it may not always behave the same in the future. Furthermore, when that computer system is moved from our immediate presence, we have few guarantees that it has not been physically modified or reprogrammed.

We introduce the need for a remote system genuinity test with a motivating example: Suppose Alice is the conscientious administrator of a network of computer systems that rely on a central NFS [42] server. Only trusted systems are allowed to act as NFS clients. Bob and Mallory use the systems in the network for large distributed applications that manipulate data on the file server. In addition to accomplishing normal work, Mallory would like to either steal or corrupt Bob's data by subverting an NFS client. Mallory has a deadline to perform an especially large computation and has made an arrangement with a distant colleague to borrow several hundred new computers to join the computer network temporarily in order to assist with the work. This necessarily means that they must also be able to access his data on the NFS server.

How can Alice determine which systems should be given access to the NFS server? She could travel to the location of each of the new systems in order to configure and physically secure them, but this might take more time than she is able to commit. Allowing Mallory to specify the systems eligible to be NFS clients would give him an opportunity to deliberately misconfigure a system in order to allow him to access Bob's data. Knowing the identity of the remote systems gives Alice no assurance of how the systems will behave.

What Alice really wants is an automated way of determining that a remote system has been properly configured before it is granted access to the NFS server. For the specific case of NFS, "properly configured" means that only Alice has administrative control over the client. In a broader sense, it means that the system's hardware and software will act in a deterministic fashion that corresponds with Alice's expectations. To measure this determinism, it is necessary for Alice to be able to decide whether the system's hardware and software are *genuine*. For instance, it is easy to see that the correct system software running in the context of a simulator can not always be trusted because it cannot be known whether the simulator will always act the same as real hardware. Moreover, anyone who controls the simulator will be able to spy on or manipulate the data resident in the simulated environment. The same lack of trustworthiness is more obviously apparent for compromised software running on real hardware.

We demonstrate a method by which we can simultaneously measure the genuinity of a system's hardware and software, remotely, without the need for a trusted human intermediary. This method can be used to decide on a level of trust that can be imparted to the system in question. Our mention of NFS (without special encryption or credential support) in the previous example is deliberate. In Section 5 we describe our implementation of a genuinity test that allows us to trust a remote system for use as an NFS client. Although attacks against our method are possible, their risk may be acceptable in some environments.

This paper is organized as follows. In Section 2 we examine the basic methods for determining the genuinity of software, hardware and combinations thereof. In Section 3 we show how the results of such a test can be delivered through an insecure network to an observer. Section 4 describes several forms of attack against the system as well as our methods for reducing the risk of those attacks. Section 5 shows various results of an implementation of our system. We conclude our paper with an overview of related work in Section 6 and a summary of our results and future plans in Section 7.

2 Tests of Genuinity

In order for some secure authority to be able to trust that a remote computer system will always act in a deterministic manner, it is necessary to verify two things:

- The computer must be a real computer—not a simulator or emulator. If the computer was not genuine, as in the case of a simulator, the environment would then be subject to potential manipulation by the simulator's controller, whereas a genuine system would be governed only by the definition of the machine's instructions.

- The computer must be running exactly the software that the secure authority expects. If the software environment is known completely then the behavior of the system can be known as well, but without perfect knowledge of the running software, the action of the system is potentially unpredictable.

We first examine the challenge of building a test for the correct software while assuming that the hardware can be trusted. We then integrate that test with a microprocessor genuinity test. This combined test constitutes an architecturally-sensitive execution fingerprint.

2.1 Software Genuinity

In order to introduce the approach to verifying the genuinity of a system's software, we consider a naïve example of some computational *Entity* running a program that must check that its own instructions have not been tampered with. (We assume throughout this paper that the program to be verified is an OS kernel.) The intent might be to demonstrate to an *Authority* elsewhere on a network that the program is correct. If we can make a closed-world assumption that the software to be tested exists entirely in a known section of memory, we can accomplish this by including a subroutine in the program that would compute a checksum of that memory space. The subroutine can send the outcome to the Authority so that it can be compared with a known-good result. Any attempted forgeries are easily detected by using a cryptographically-secure hash to compute the checksum so that the likelihood of changing the instructions to produce a correct checksum is extremely small.

The possibility of a replay attack is reduced if the Authority is allowed to challenge the Entity by specifying one or more subregions of the program's instruction address space. Such a method has been used for several collaborative network applications to verify the integrity of the client software. For instance, the AOL Instant Messenger service can discriminate against rogue clients by requesting a cryptographic hash of a section of the client program's text segment [3, 38].

It is a more difficult problem to guarantee that the code that was executed to compute the checksum was really the subroutine in the program. For instance, consider the possibility that the Entity contained the expected program in its memory but executed a rogue program from some other location. Detecting an attack of this nature is equivalent to determining whether a piece of code is presently executing at a specific location. A test of this nature is an example of *Software Tamperproofing* [12, 21]. Normally, this technique is useful for situations where, for example, one wishes to ensure that a piece of software cannot be illicitly duplicated in another application. Here, we want to use it to ensure that the code we are interested in has not been modified to run in any place other than the one in which it was intended [13]. There are several architecturally-sensitive ways of constructing such tests using modern microprocessors, and we will address this thoroughly in Section 2.4.

A basic limitation of our closed-world assumption is that we cannot incorporate dynamically-allocated data or any other non-static information into the checksum in a meaningful fashion. However, if we can determine that the instructions of the software are correct (and that these instructions are being executed) we may be able to trust the software to further interrogate and verify the integrity of its data. We will address this in further detail in the context of our implementation in Section 5.1.

2.2 A Microprocessor Genuinity Test

In order to determine the genuinity of a system's hardware, we first make a limiting assumption that the primary item of importance is the microprocessor specifically rather than trying to look at the entire system. In order to determine a microprocessor's genuinity, one must exercise a representative subset of its functions to check that it fully conforms to all of its specifications. In conducting such a test, it is very easy to discriminate between microprocessors that have different instruction set architectures, but it is considerably harder to tell the difference between two implementations of the same architecture. Nevertheless, we maintain that there are always differences between implementations that can be observed by software at some level even if the instruction set is the same. For instance, in determining the difference between two microprocessor implementations that differ only by their cache geometry, an instruction sequence with a particular memory reference pattern would take longer to execute on one of the processors. If the microprocessors had performance counters capable of measuring the number of cache misses or if they had direct means of evaluating the cache contents, a test could be constructed to discriminate between the two implementations without resorting to timing analysis.

Determining the difference between a real microprocessor and a simulator is an apparent contradiction to Turing's thesis [45] that any computer is theoretically capable of any mechanical calculation—including simulation of another computer. However, even modern high-performance simulators are typically one or more orders of magnitude slower than real computers [32]. If the Authority establishes a time limit for the Entity's response, it will be able to ascertain whether the Entity was being run in the context of a simulator. Furthermore, by constructing the checksum using operations that are most difficult to simulate, the disparity in execution time between simulator and real computer is maximized. The goal then is to establish a repertoire of tests that execute much more slowly on a simulator than on the real computer.

In general, any test to be used for the purpose of determining genuinity will exercise a function of the CPU that has the following ideal characteristics:

- The function will occur automatically as a side-effect of instruction execution. This ensures that a simulator will be forced to do multiple jobs at once—execute instructions as well as accurately model their lower-level architectural impact.
- The function must be deterministic and predictable. The Authority must still be able to either obtain the result of a genuinity test from a trusted reference platform or (slowly) simulate and predict the result on its own.
- The effects of the function can be easily measured. This allows the genuinity test to run as efficiently as possible on the real CPU.
- The function will have a good deal of implicit parallelism. This minimizes the chance that a simulator will be able to mimic it quickly.

Consider the previous example of a CPU that includes hardware performance counters that can measure the occurrence of cache misses. For such a CPU, modifications to the cache's state will occur as a side-effect of all load and store operations. In the case of a miss, there is some implicitly parallel activity that occurs to determine the cache line that should be replaced, and this parallelism is largely hidden from the instruction set architecture. If we understand the cache geometry and replacement policy we can accurately predict, for a particular access pattern, which cache lines will be replaced and when, but doing so generally requires an effort proportional to the *associativity* of the cache in question. At the conclusion of a correctly-executed test that exercises this pattern, the hardware counter should contain the number of misses we predicted to occur, and the final state of the cache should also be as we predicted.

The memory hierarchy makes an excellent device to satisfy the characteristics of a good test since it has traditionally been found difficult to simulate precisely [7, 33], and, when memory is modeled at its fullest detail, the result is that the simulator runs at least an order of magnitude slower [32, 49] than if it had a memory model adequate only for simulation of the instruction set architecture. We expect this disparity in speed to only increase in the future as CPU implementations continue to use memory hierarchies with higher levels of associativity and more complicated replacement policies.

2.3 A Combined Software/Hardware Genuinity Test

By incorporating instructions that characterize the system's execution into the checksum procedure, we can build a mechanism that can check the integrity of its own instructions while simultaneously ensuring that the instructions are running on a real computer. Each load that the checksum procedure performs has several side effects: cache lines are evicted and refilled, translation lookaside buffer (TLB) entries are evicted and refilled, and bus transactions are performed. For each group of instructions fetched, another TLB is consulted for virtual memory mapping of the instruction space. Each branch in the checksum procedure is predicted by the CPU to be either taken or not taken. Many sources of meta-information about the CPU's execution exist; however, our selection criteria indicate that the best candidate function is the memory hierarchy. In particular, the TLB is a good choice since, for most architectures, it has a higher associativity than the caches. It also has a deterministic replacement policy whose effects can be easily measured on most platforms.

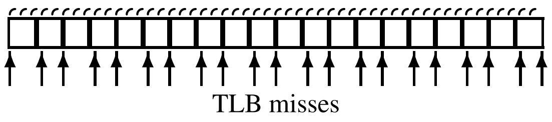

Figure 1: Linear memory traversal

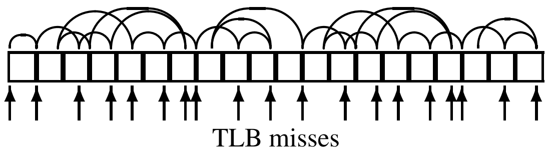

Figure 2: Pseudorandom memory traversal

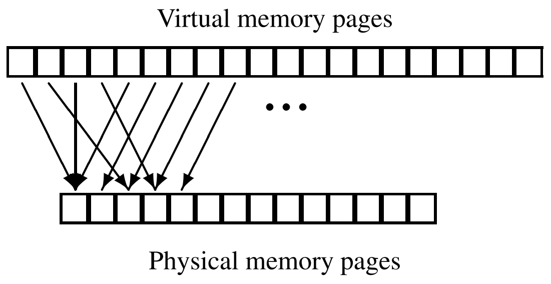

Figure 3: Pseudorandom memory mapping

In addition to making the genuinity test difficult to simulate quickly, we also need to make sure that the architectural sensitivity is inseparable from the memory checksum and that the results from one genuinity test give no clues to an observer as to how a future test will be computed. A checksum procedure that traverses memory in a linear fashion would be unacceptable since it would cause a data TLB (DTLB) replacement pattern that could be predicted without simulating the DTLB. Figure 1 graphically depicts a linear access pattern across pages of memory. Pages of memory are shown as blocks and the access pattern is shown by the small arcs above. The TLB misses predictably occur on the first access to each page. We need to introduce as much run time complexity as possible into the test; therefore, we construct the checksum so that it traverses memory in a pseudorandom fashion as in Figure 2. Doing so creates a greater run time uncertainty about whether the page of the word that is loaded is presently mapped by the DTLB. We further complicate the test by aliasing the pages of the physical memory region we intend to check multiple times through a much larger virtual region (Figure 3) rather than using a one-to-one mapping from virtual space to physical space as is customary. Doing so increases the duration of the checksum and constitutes a better discriminant against the possibility of a simulation attack. By varying the pseudorandom walk and the mapping from virtual to physical space for each new invocation of the genuinity test, the checksum result will be different each time and unrelated to other invocations.

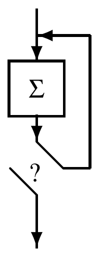

Figure 4: Simple checksum representation

As the checksum procedure makes the pseudorandom walk through the virtual region, it consults a hardware performance counter to determine if the previous load caused a hit or miss in the DTLB. This information is incorporated into the checksum result, and the procedure continues in this manner until the pseudorandom walk is complete.

It is important that one or more sources of meta-information about the procedure's execution be incorporated continuously into the checksum result in order to ensure that the checksum of the memory contents and the test for genuinity of the execution environment cannot be separated. This combination has a benefit of allowing us to relax the requirement of a cryptographically secure hash for computation of the memory checksum. Each word read from memory by the checksum procedure can be added to the running result as long as the DTLB hit/miss information is incorporated in a non-additive manner. For instance, after each checksum addition the DTLB miss count could be XOR-ed into the result. Since the memory checksum and the DTLB miss count are not linearly related, the integrity of the memory checksum is enforced. Graphically, we can depict this simple procedure as in Figure 4 where the checksum component might consist of

```
sum = sum + [ PseudoRandomLocation ]
sum = sum XOR TLB_MissCount
```

and the decision element simply determines whether the last value in the pseudorandom sequence has been read.

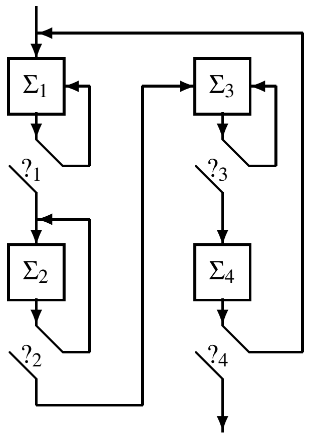

Figure 5: Complex checksum representation

Reducing the frequency of integration of the TLB miss count would not reduce the burden of an attacking simulator to continually track the TLB contents. The procedure could be easily modified to incorporate the TLB miss count into the checksum after every few iterations. This would allow the genuinity test to run more efficiently with only a minor risk of separability of architectural sensitivity from the checksum calculation, but it would still require almost the same computational overhead for an attacking simulator. Furthermore, if several unrelated sources of meta-information were available, each source could be incorporated into the checksum in a round-robin fashion at various points in the genuinity test. Adding more information sources would only require an attacking simulator to continuously simulate more functions that already occur automatically in the real CPU. For instance, if the procedure were partitioned into multiple code sections spread out over several pages of memory as in Figure 5, the instruction TLB (ITLB) would register a hit or miss each time a new code section was invoked.

Further refinements to the checksum procedure shown in Figure 5 can be made, such as making the decision points dependent on the checksum (rather than only the pseudorandom location pointer) or making each of the checksum and decision points contain unique functionality.

The end result is a memory checksum of the running application which has been perturbed by meta-information about the execution of the procedure. The perturbation is difficult to simulate in a timely manner due to the typically large associativity of the TLBs. If either the memory contents or the execution of the procedure is not correct, the incorrect result will be computed. If the procedure is evaluated by a simulator, it cannot be done quickly enough to meet the Authority's deadline. The result will differ for each combination of pseudorandom walk and virtual-to-physical memory mapping thereby allowing an Authority to challenge each new Entity with a unique test every time. Since the checksum procedure cannot discern at run time whether the result is correct, its result constitutes a *zero-knowledge proof* that the system is a real computer executing the expected software. The result can then be delivered to an Authority that can compare it to a precomputed, known-good result to determine the genuinity and, therefore, the trustworthiness of the system.

2.4 Additional Requirements of the Genuinity Test

There are several additional aspects of the genuinity test that are important for the veracity of the test. In particular, some steps must be taken to ensure that the execution environment is sufficiently isolated from outside influence. Since the mechanism presented relies on the fact that only the memory references pertaining to the checksum are to be used to predict the behavior of the DTLB, there can be no other memory references performed by the procedure. Therefore, all temporary data needed by the procedure must be held in CPU registers for the duration of the test. Furthermore, since any temporary diversion from the procedure might cause extraneous memory references, the checksum procedure must be run with all external interrupts disabled. After the genuinity test has been satisfied, interrupts are re-enabled, and the machine is allowed to function as a normal execution host.

If the secure Authority and the Entity are located far apart, there will be some (potentially unbounded) network transport delay to issue the challenge to the Entity and return the results to the Authority. In theory, an allowable delay should be no longer than the marginal time difference between the checksum procedure running on the target CPU compared to the same procedure running on the highest performing simulator. The longer the test that we construct runs, the better this margin becomes, so a long procedure is desirable. In practice, however, we can assume that the network transport delay is small and any pathological delays will simply be treated as failures. Hopefully, such delays will be only temporary, and a new genuinity test can be retried at a later time.

The problem of determining whether the program being verified is actually running is addressed by incorporating a suitable degree of *introspection* into the verification procedure. The simplest way of doing so would be to introduce a section of the verification procedure that would incorporate the present value of the program counter (PC) into the checksum value; however, the location of the checksum code is known in advance, and it does not require much program analysis to replace each PC incorporation by a constant-valued expression in a rogue program running elsewhere. Because software running in a location other than its intended location or running as a part of another application will produce a different ITLB miss pattern than the expected one, we can employ architecturally-sensitive techniques to identify such an attack. In particular, on architectures that support such operations, we can modify the genuinity test to incrementally incorporate the contents of the ITLB into the checksum. This not only ensures that the code was run in the expected location but also improves the certainty of the test by incorporating one more piece of architectural meta-information into the checksum.

We can also rely on more complicated architectural state in order to make such an attack possible only in the context of a full microprocessor simulator. For instance:

- Some microprocessors write information to the virtual memory page tables to indicate accessed or dirty pages. For instance, an x86 CPU marks a page table entry (PTE) as "accessed" when it is read or executed. This progression of PTE modification for a particular test is deterministic and predictable and we can incorporate it into the checksum calculation by including the page tables in the memory mapping. An attacker would certainly be forced to simulate the entire virtual memory subsystem in order to keep track of this information.
- Some microprocessors allow privileged applications to directly examine the TLB's Content-Addressable Memory cell (CAM) contents. By incorporating this information into the checksum value, the outcome will only be correct if the ITLB contains values corresponding to the correct execution location.
- Some microprocessors have hardware-based last-called stacks that can be interrogated to obtain the address of the last control-modifying instruction (such as a JUMP or CALL instruction). Since our checksum algorithm has pseudorandom control-flow that is based on the contents of memory, this information is not expressible as a constant. By incorporating this information into the checksum value, we force an attacker to either simulate this hardware stack or fully simulate the instruction stream.

If the checksum obtains the correct result, it will have demonstrated that its own exit path is trustworthy. This exit path will be responsible for invoking the code that delivers the result to the Authority via the network and for confirming that critical data values of the computer system are intact (e.g., the call stack and interrupt vectors) in order to prevent any code entry points that are not already known to the OS kernel. The kernel that invokes the checksum procedure is expected to remain in operation until the end of the session in which it participates with the Authority.

3 Demonstrating Genuinity Via an Insecure Network

In our work, we assume that the distance between the Entity and Authority does not permit the establishment of a secure communication link prior to the genuinity test. Furthermore, we intend our system to be used without the benefit of a trusted human who could serve as the courier of a shared secret. In order to prevent an interposition attack by a malicious router, we can leverage the genuinity of the Entity, as indicated by its valid checksum computation, to negotiate a public key exchange with the Authority. To do so, we first ensure that the public key of the Authority is embedded into the verified memory space of the Entity's genuinity test; a properly computed checksum value will be an indication that this is true. When the Entity sends the computed checksum to the Authority it can first encrypt the result, along with an identifier chosen at random by the Entity, using the public key of the Authority. Although it is critical that the Entity deliver this information to the Authority in a timely manner, a single public key encryption does not greatly delay the result. Thereafter, the Entity would have an identity known only to itself and the Authority. At some later point, the Entity will generate a new public key and send it to the Authority, along with the random identifier as an indication that the new key really belongs to the authenticated system and not an intermediary. The knowledge of genuinity of the remote system represents a convenient improvement to the problem of key exchange in the presence of an interlocutor [8, 15, 16, 39].

The primary difficulty in such an exchange is obtaining enough entropy to generate a random identifier that cannot be easily guessed by an adversary. Although the genuinity test must be deterministic, there are still events present in the Entity that are random. In particular, we have found that periodic sampling of the timestamp counter during the course of the test will produce sufficient entropy (even for repeated invocations of the same test code). The minor variations in memory and CPU pipeline timing force the CPU to sample the free-running timestamp counter at unpredictable intervals. Such a random number generator is similar in some respects to the TrueRand function in CryptoLib [29]. Experimental results for our random value generator are shown in Section 5.4.

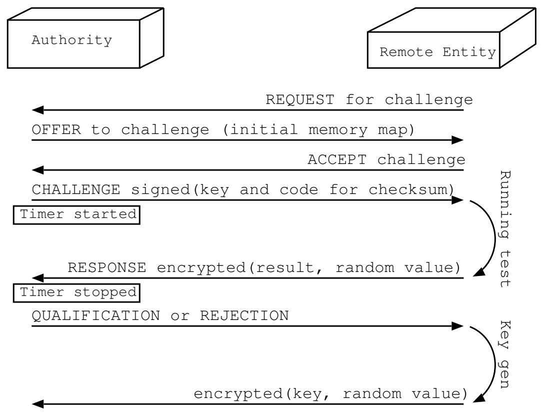
Authority
Remote Entity
REQUEST for challenge
OFFER to challenge (initial memory map)
ACCEPT challenge
CHALLENGE signed(key and code for checksum)
Timer started
Running test
RESPONSE encrypted(result, random value)
Timer stopped
QUALIFICATION or REJECTION
Key gen
encrypted(key, random value)

Figure 6: Network protocol

Figure 6 illustrates the following negotiation protocol initiated by the Remote Entity.

1. The kernel running on the Entity sends a network packet to the Authority containing (cleartext) information about its microprocessor and requesting a genuinity test.
2. The Authority generates a suitable test and sends a cleartext message offering to challenge the Entity with a genuinity test. The message contains the memory mapping to apply for the test.
3. The Entity sends a cleartext message to the Authority accepting the offer and initializes the virtual memory mapping.
4. The Authority sends a cleartext packet that contains the code to be run and a public key for the response. It also records the time at which it sends the packet.
5. The Entity receives the code and key, and loads both into memory. It then transfers control to the test code. When the checksum completes, it jumps directly to a (now verified) function in the Entity's kernel that encrypts the checksum result and the random quantity and sends them to the Authority. This step is carefully designed to minimize the time that the unencrypted values are resident in memory. Most likely the unencrypted values will never be visible on the external memory bus.
6. The Authority receives the reply, registers its time, and decrypts it. If the reply contains the correct checksum result and was received within an allowable time, it marks the Entity as a genuine host. It sends a reply packet indicating the Entity's qualification.
7. The Entity generates a new session key, concatenates it with the previous random value, encrypts them with the testcase public key and sends them to the Authority. Here the random value acts as an authenticator that the Authority can use to know that the session key it is receiving is from the same system that showed itself to be genuine. The session key can then be used to establish a secure virtual private network or some other kind of secure communication channel.

Note that while the kernel on the Entity is waiting for a challenge message it has no means of reliably determining the sender of a packet. Since the challenge consists of runnable code that is to be inserted into the Entity's kernel memory, it is important to first check the message's validity in order to avoid running code sent by an attacker. The risk of such an attack can be reduced by generating a second key pair and embedding the public key into the kernel that is loaded onto the Entity. The Authority can sign the messages that it sends, and the Entity can then discriminate against bogus messages. Since the genuinity test negotiation protocol is already sensitive to the exact type of kernel running, this protocol addition poses no further complications to the test other than the time needed to validate the challenge signature.

4 Potential Attacks

It is necessary to elaborate on some potential attacks against our genuinity test and ways of guarding against them in our implementation. In particular, there are several forms of simulation that could be attempted as well as some methods of ensuring that they cannot be effective. Since a primary discriminant in our method is time of execution, we are faced with the fact that there is a definite lower bound on the type of target CPU that can be verified. If a system that lies below that bound requests a genuinity test, it must be rejected. Furthermore, because the performance of new systems (on which a simulator can be run) is constantly improving, the lower bound on the target CPU is also increasing. Finally, as the state of the art in simulator construction improves, target CPUs that were once a safe bet for a genuinity test are placed into question. In order to assess an appropriate lower bound, we first need to examine the kinds of simulators that are possible.

4.1 Algorithmic Simulators

A full simulation of the instruction execution and any architectural side-effects would be too slow to pass the test that we have created. There remains the possibility that the simulator does not pretend to be a microprocessor but is custom-tailored to simulate only the checksum procedure's *effect* on the microprocessor. For instance, if it is known that a remote system with particular TLB characteristics is to be shown genuine, the effect of the test on the target CPU can be efficiently simulated by a specialized application running on a very fast machine. We refer to this as an *ideal simulator*.

Here, again, it is necessary to assert that the checksum code must be executed in order to obtain the correct result. To do so, it is necessary for the Authority to deliver the challenge to the Entity in the form of unique, executable instructions in order to prevent the simple interpretation by a dedicated algorithmic simulator. For instance, instead of delivering values indicating the configuration of the pseudorandom sequences in the challenge, the Authority must send the executable code to perform the checksum. If the checksum procedure was sufficiently intertwined with its containing application, a malicious Entity would need to either do a great deal of program analysis to quickly *construct* the simulation algorithm or use full instruction simulator to initiate the test. Such safeguards are another example in the study of software tamperproofing [12, 21].

4.2 Virtualizing Simulators

Some modern system simulators [25, 30] are capable of efficiently running an entire operating system within an application by means of a technique called virtualization. This involves running the simulated OS kernel at an artificially low privilege level in such a way that execution is trapped and simulated only when a privileged operation is attempted. This allows the majority of instructions to execute natively on the CPU. However, the checksum procedure running on a virtualizing simulator would fail after the first privilege trap since it would be necessary to save some of the CPU state and doing so would involve referencing memory. Referencing any memory would alter the TLB miss patterns and invalidate the checksum.

For many microprocessor architectures, it is also difficult to build a virtual machine monitor that does not leak information about its supervisor [40]. Assuming that such information can be obtained in an efficient manner, it too can be incorporated into the checksum procedure in order to discriminate against such a simulator.

An improvement to the idea of virtualization involves dynamic translation of the privileged instructions into subroutine calls that mimic their operation in the context of the simulator. For purposes of an attack on a checksum procedure, this reduces the complexity of the problem to that of an ideal simulator. However, our results in Section 5 indicate that even if the virtual simulator could quickly converge on dynamically-generated native code that efficiently simulated the multiple implicitly-parallel architectural features, it could not execute it quickly enough to succeed unless there was a very wide disparity between the speeds of the simulator's host CPU and target CPU.

4.3 Hardware Attacks

The ultimate attack on our genuinity test involves modifying the remote system in such a manner that it can successfully compute the correct result for the checksum procedure but still allow a third party to physically inspect and alter the system after the test has completed. This can be accomplished by attaching hardware to the coherent memory bus of a system or attaching an analyzer to the microprocessor's probe ports. For many operating environments, the risk of such an attack is acceptable since undertaking it would require significant skill and equipment.

Alternatively, one can imagine the same type of attack occurring in a symmetric multiprocessor system where the primary CPU is unable to detect the presence of a malicious secondary CPU. Since conventional CPUs can do nothing to ensure or evaluate the security of their own external interfaces, there can be no guarantee of security in the presence of such an attack. This is an open problem in our work, and we continue to examine such a possibility.

In order for any direct attack against the microprocessor to be successful, it must either be done transparently while the machine remains in operation or done very quickly while the machine is temporarily off-line. An attacker might pause the CPU (for example: a notebook computer's suspend-to-disk operation) long enough to interrogate the memory and determine the shared secrets for the secure network connections and then substitute them into another machine that he or she controlled. To guard against this possibility, we require that the remote machine remain in active contact with the Authority with a period no longer than the time necessary to pause the machine to obtain information. If the contact time exceeds an appropriate timeout, the remote machine is no longer considered trustworthy and must go through another genuinity test before trust is restored. Similarly, the trusted software running on the remote machine should destroy its secrets if it falls out of contact with the Authority.

There is also the possibility that the user has preconfigured the Entity to allow a *hotkey* sequence to cause a subversion of the system software. Ideally, there would be no systems for which we could not disable a suspension request or some other type of subversion mechanism. Because most systems implement user input using interrupt-driven methods, we believe we can avoid most problems by carefully checking the interrupt vectors and the relevant I/O configuration for device drivers to make sure that they are valid and safe. Section 5.1 describes this process in greater detail.

5 Implementation

We created a trial implementation of our hardware/software genuinity test using the Linux 2.5 kernel as a host application. The kernel provided us with a convenient and portable environment for manipulating virtual memory and network interfaces. We initially made only minor modifications to the kernel in order to incorporate our test mechanism. In particular, we inserted a few pages of empty space near the beginning of the text segment to hold alternate page tables and added a set of functions to perform the network test negotiation. The test negotiation was set up to be invoked automatically during kernel initialization—before a filesystem existed or 'init' was started. We implemented an in-kernel 128-bit RSA algorithm to perform the public-key encryption of the return result.

Our Authority consisted of the following components:

- A generator to build RSA-128 key-pairs. Each new test used a unique public key.
- A testcase generator that randomly combined predefined code chunks into a unified image. A cache of tests were precomputed for different models of x86 microprocessors.
- A network server that handled the negotiation of genuinity challenges and selection of tests based on microprocessor type. This server also maintained an indication of the status of a remote host (e.g. whether it had been recognized as genuine yet or had been rejected).

Most specifications for our system were flexible and chosen to best illustrate the validity of our approach. For instance, a 128-bit RSA key would be too small in practice. We chose this size in order to minimize the time needed by the Entity to perform the encryption. Our RSA implementation was not optimized for speed and a larger key size would be easily substituted assuming some modest improvements to the code. We also made the assumption that the Entity to be tested would operate without non-volatile storage. The kernel was initially loaded via the network using GRUB [19] or Etherboot [36]. The key exchange at the conclusion of the genuinity test was used to negotiate keys for an IPsec [28] session. The Authority then allowed NFS exports to the Entity. It may be desirable to allow the use of a non-volatile storage device for swap space or local executables. Various methods exist for ensuring that secrets swapped to disk are not compromised by an attacker [37] and that executables on a non-volatile file system can be trusted [46]. By combining these elements, we can create a trusted system that uses data served securely by the Authority.

5.1 Securing the Non-Static Information

One disadvantage of using a Unix-like kernel as the host application for a genuinity test is that it has a great deal of non-static and allocated data that could be useful to an attacker as a means of subverting the kernel after a successful completion of the genuinity test. (Recall that only the static or predictable contents of memory are incorporated into the genuinity test.) For instance, consider the possibility that an attacker loaded the kernel, modified a pointer in an unchecked data region and allowed the kernel to initiate the genuinity test. Later, the attacker could invoke some behavior to exploit that pointer.

The areas of the kernel in question are the non-constant data segment, the initially-zero segment (or 'BSS' in the Unix vernacular), and the driver initialization text. In addition, the kernel dynamically allocates virtual memory for its data structures. We use a layered approach to ensure the integrity of this data.

First, much of the non-constant data segment is designated non-constant only as a convenience. For instance, the syscall table is specified in an assembly language .data section even though it is (hopefully) not modified during regular kernel operation. By adding a section identifier, it was moved to the read-only data section and was subjected to the genuinity test.

When the genuinity test completes, it jumps back into a verified section of code in the kernel that will restore interrupt handling, encrypt the results and send them to the Authority via the network interface. Before interrupts are reenabled, this code is guaranteed to function deterministically and can be entrusted to perform checks on the data that could not be subjected to the genuinity test. In order to maximize the time available to perform these checks, the code encrypts the results and sends them over the network and verifies data *before* re-enabling interrupts. Items verified by this code include the interrupt vectors and root virtual memory page table. In doing so, we can verify that no malicious agents are present in memory that were placed there before the invocation of the genuinity test.

The Linux kernel moves much of the device driver initialization code into a segment of virtual memory that is freed after use. In order to verify the integrity of this code, we disabled the function that freed the space and modified the linker script to move the text of the initialization code into the kernel text section. Doing this also allowed us to later use a known-genuine system to find the result of a checksum computation for another system.

We made an attempt to invoke the genuinity test as early as possible in the kernel's initialization stage to avoid the proliferation of allocated data. Nevertheless, because relaying of the test results requires a working Ethernet adaptor as well as system bus and chipset functionality, some data structures must be allocated to support access to them. One way of eliminating this data would be to restart the kernel initialization process at a point before any memory was made allocatable.

5.2 Precomputation of Checksum Results

In order to be able to quickly certify a remote Entity, our Authority must precompute checksum results for each genuinity test. This can be done by using an off-line simulator that is able to do an adequate job of emulating the architectural features of the target microprocessor. This may take a significant amount of time, but can be amortized over the expected running time of the certified remote hosts. The Authority need only maintain enough results to handle an initial surge of requests for genuinity tests.

In some cases, genuinity tests may exploit instruction execution side-effects that are deterministic, but are unpredictable. For instance, if the microprocessor has an undocumented TLB replacement policy it may not be possible to construct a simulator to use for result computation. In such cases, the Authority can use an existing genuine system to compute the results for new tests. When doing so, it is important that the exit path of the checksum algorithm does not cause a re-initialization of a known-genuine execution host, but we also do not want to open the possibility for a non-genuine host to intercept its checksum results and use them for an interposition attack.

We solve this dilemma by adding a flag to the challenge delivered by the Authority that indicates whether the Entity should deliver the results over the network and reinitialize itself in preparation for becoming a known-genuine host. The exit path of the checksum algorithm encrypts the checksum result and random identifier and then checks this flag. If the flag is not set the system destroys the value of the checksum result and random identifier but preserves the encrypted version and then returns to its prior execution environment. The encrypted version of the information is only useful to the Authority, and the random identity serves as a *nonce* to prevent reuse of the values. The Authority can use a secure channel to both initiate a new test and procure the result. The only side effect on the remote host will involve a momentary pause of other activity while its interrupts are disabled.

By allowing the existing base of genuine hosts to compute the results of new tests, the Authority can generate tests for new systems on a demand basis. It is also much easier to use one CPU to compute results for another equivalent CPU than it is to construct a suitable simulator. In practice, we precomputed most of our checksum results in this manner.

5.3 Benchmarks

In order to establish a time limit for the response delay for a correct checksum result we used simulators to evaluate a particular genuinity test. It is relatively easy to tell the difference between a fast CPU and a simulator running on a similarly-fast host CPU. Therefore, we chose a particularly conservative example where we used a 133MHz Intel Pentium as a target CPU. We ran our simulators on a 2.4GHz Intel Pentium 4 system. In order to sustain an artificial performance advantage of the real CPU over the simulators, we incorporated as much architectural meta-information as possible. We used the Authority software to generate the following random genuinity test:

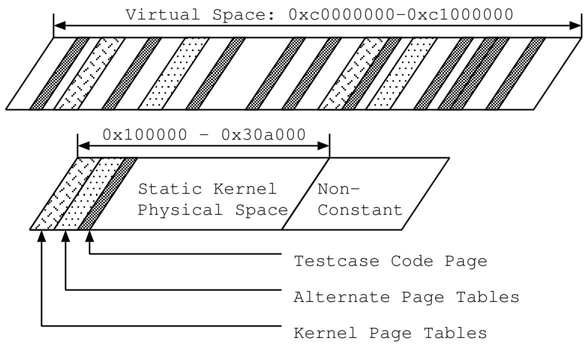

Figure 7: Memory Map for Benchmark Testcase

- The static region of the kernel's physical memory (roughly 0x00100000 – 0x0030a000) was mapped into the 16MB region from 0xc0000000 – 0xc1000000 using the alternate page tables in the beginning of the kernel text segment. This was a pseudorandom mapping that favored the specific page where the checksum code was loaded. On average, most pages of the kernel were aliased by three virtual pages. The code page was aliased by 2661 virtual pages. The kernel page tables, as well as the alternate page tables were mapped in the virtual memory space. Figure 7 depicts the mapping. Note that this 16MB region is the only memory accessible for the duration of the testcase. Instructions, page tables and data are all visible in this space. The self-contained nature of this region makes it very difficult to subvert the genuinity test by patching in malicious code since there is no accessible place that it can be hidden.
- The checksum code was constructed in nodes spread over 22 of the code page mappings in virtual memory. Each of the nodes branched to one of two other nodes on the 22 pages. The sections of code were offset so that they did not overlap in the single physical code page that they were all aliased to.
- The checksum code nodes were selected as follows:
  - One node accessed the ITLB CAM cells to determine whether an instruction fetch for a selected target would be a hit or miss. In the case of a hit, the physical page and replacement information were extracted from the CAM cell. This information was incorporated into the checksum value.
  - One node made a similar access to the DTLB CAM cells.
  - One node accessed the tag and replacement information for the data cache and instruction cache cells.
  - One node read a performance counter that counted branches and incorporated it into the checksum value.
  - One node read a performance counter that counted instructions executed and incorporated it into the checksum value.
  - One node sampled the timestamp counter probabilistically one out of every 4096 times it was invoked and incorporated it into the generated 32-bit random identifier.
  - The remaining sixteen nodes read a byte of memory, added it to the 32-bit checksum value and advanced the memory pointer. A compact linear feedback shift register algorithm was used to advance the pointer in a pseudorandom pattern that was sure to cover the entire 16MB virtual space.

To invoke the test we booted the appropriate kernel on an Entity comprised of a 133MHz Pentium CPU with 32MB of RAM and an RTL-8139B 100Mb/sec NIC. The system was on the same Ethernet segment as the Authority in order to discount the effects of network latency. When the kernel invoked the challenge negotiation code, the Authority recorded that the Entity was able to receive the test, compute the checksum value and random identifier, encrypt the results and return them via the network in 7.93 seconds. Repeated invocations of the same test, either via reboot or via secure initiation by the Authority, consistently produced the same checksum result (and a different random identifier). It is worth noting here that the encryption step takes approximately 900000 CPU cycles, or about 0.007 seconds, so it does not contribute significantly to the response delay.

In order to successfully masquerade as a Pentium 133, an attacker would be required to simulate the testcase execution flow, page tables, PTE update, ITLB and DTLB CAMs, data and instruction caches, as well as keep track of the count of different types of instructions executed.

As an initial test, we compared the native execution time against an instruction simulator. We used Bochs 2.0 [31], a program that simulates an entire x86 system. It also emulates a NE2000 network adaptor and encapsulates traffic through its host. This conveniently allowed us to use it as though it were a real system on the network. As an initial test, we configured Bochs to ignore the WRMSR and RDMSR instructions in the testcase. The resulting test was executed by Bochs in 18.53 seconds on the 2.4GHz host. Clearly, modifying Bochs to support the hardware monitoring facilities necessary for successfully running the real test to produce a correct result would increase its execution time by a considerable amount.

We then built an ideal simulator that relied on the assumption that it would have perfect *a priori* knowledge of how the test code would act. This would be equivalent to a virtualizing simulator that could *instantly* transform the testcase into an accelerated form that could run natively on the simulation host. To build such a simulator, we took each of the 22 testcase nodes and manually encoded the precise effects of their execution on the ITLB, the DTLB, caches and the two instruction counts. We also built page table traversal and update routines that worked with the TLBs. (We allowed the simulator the advantage of not having to generate a random value.) The simulator was run on a 2.4GHz Pentium 4 system that also hosted the Authority software. The execution time of the ideal simulator was 10.72 seconds. While this is not significantly more than the execution time of the native CPU, it is also based on a few unrealistically optimistic assumptions. In particular, the simulator is using *a priori* knowledge of the testcase to build a native representation with no time penalty. A real simulator would need additional time to analyze and convert the instruction stream. Furthermore, our testcase also represents an intentionally-impaired test for which it was relatively easy to construct an efficient simulator. Several improvements for the Pentium genuinity test are possible such as using self-modifying code, reconfiguring the performance counter registers on-the-fly to periodically monitor other events and using CPU registers to inspect partial information about its own instruction decoding. Incorporating this information into the testcase would require a corresponding simulator that was significantly more difficult to build.

After finding the best-case execution time of a simulator, we can use it as the maximum allowable time for the genuinity test for real Entities. We conservatively mandate that all Entities must respond within this time, regardless of their CPU speed. Since we cannot predict or measure the network delay, it is included in the total response time. This means that Entities that experience pathological network delays during test negotiation will fail. If average network delay becomes a significant component of the total turnaround time of the genuinity tests, some modifications can be made to the test generator to compensate. For instance, if the size of the virtual space mapped by the test is doubled, the execution time for the test will approximately double as well, making the network delay a smaller component of the total time.

Newer implementations of the x86 CPUs have advantages not only in terms of speed but also architectural complexity. For instance, a Pentium/MMX is significantly harder to simulate than a Pentium running even at the same speed because of much higher associativity of its TLBs as well as data and instruction caches that are double the size and associativity of its predecessor. Even though most newer CPUs have TLBs and caches larger than earlier models, this quality does not improve their ability to masquerade as earlier models or more easily support simulators that implement the earlier TLBs and caches. As we consider higher-performance x86 microprocessors, because there is a smaller margin between their performance and that of the best available simulation host, the feasibility of constructing a fast simulator is diminished. This, in conjunction with the general difficulty of constructing a correct simulator (at any speed), creates a reasonable certainty of the veracity of the genuinity test.

Finally, although we can estimate how well a virtualizing simulator might perform, it is important to remember that, presently, no such simulator exists. The certainty of whether a Pentium 133 can be unambiguously verified as genuine will depend on monitoring the state of the art in simulators.

5.4 Effectiveness of Random Number Generation

The random identifier generated during the genuinity test must be reasonably unguessable by an attacker to prevent replay attacks and interposition attacks against the protocol. To assess the effectiveness of the random value generation, we ran a captive testcase on a single machine repeatedly and collected and analyzed the results. (In each repetition, the same checksum result was generated.) The particular testcase interrogated the timestamp counter (TSC) only 40 times. We further constrained the testcase to clear the TSC at the beginning of each repetition and limited the entropy gatherer to incorporate only the lower two bits of each of the 40 samples. Each sample was incorporated into the random value by rotating the random value by one bit and XORing the lower two least-significant bits of the TSC. This meant that each bit of the resulting 32-bit random value was the result of either one or two samples of the TSC.

We collected 99778 consecutive random values. In the entire set, there were only two 32-bit values duplicated. This roughly corresponds to the expected collisions in a linear distribution (i.e. $2/99778 = 0.000020$ and $99778/2^{32} = 0.000023$). Furthermore, a two-dimensional visual analysis of the generated values did not reveal any apparent clustering.

Because the random value is not correlated to the checksum result and does not even reveal clues about the likely outcome of the next repetition of the contrived testcase, we believe it is reasonably unguessable. In normal testcases, the random value generation is much less constrained and provides an even greater degree of unpredictability.

Some systems provide hardware-based random number generators on the motherboard chipset. For instance, the Intel i810 and AMD 768 chipsets use sampled thermal data and constrained electrical noise to generate "truly-random" numbers. Interestingly, use of these devices is not widespread [22]. One particular disadvantage of chipset-based random number generators is that an attacker could conceivably watch bus transactions to read the random number in transit from the chipset to the CPU. Our genuinity test keeps generated random values only in CPU registers and is immune to such a potential attack. Recently, a microprocessor vendor has added support for random number generation on the CPU itself [22, 24].

6 Related Work

Our work is the first to deal specifically with the problem of determining whether a remote computer is a real computer. Nevertheless, there are several related projects that propose alternative methods for creating trusted remote systems.

6.1 Execution Verification

We are not the first to leverage the time delay inherent in the computational complexity of a problem to prove characteristics of an operating environment. Much work has been done in constructing programs that can check their work to ensure their integrity [9, 47] as well as making sure that they have not taken shortcuts in their execution [10, 20]. In particular, Jakobsson and Juels characterized and articulated the concept of a *proof of work* [27] that is similar to the rationale by which our checksum works. In their taxonomy, the result of our memory checksum during directed exercising of the CPU constitutes a *bread pudding protocol*—a way of reusing a hard-to-compute result for another purpose.

Monrose, Wyckoff and Rubin illustrate a system similar to ours that verified the correct execution of code on a remote Java Virtual Machine [34] to detect the possibility that the JVM had *cheated* on a calculation. An important distinction between their work and ours is that a JVM, being a simulator, is always susceptible to potential eavesdropping by its controller and its data could be manipulated at any time. Execution verification must be performed at all times for a JVM rather than once at boot time in order to confirm determinism of computational results.

Hohl [23] presented a thorough overview of the problem of malicious hosts and potential ways of addressing it. This research recognized the apparent insolubility of the authenticity problem and concentrated on establishing an environment whereby work could be sent to (possibly malicious) remote Entities in such a manner that the integrity of computation could be proven. Secrecy was maintained by using software obfuscation techniques. This system was also sensitive to the execution time of the distributed computation in order to detect potential cheating. The cost of using malicious hosts for computation was realized by the additional network bandwidth and increased execution time.

6.2 Secure Coprocessors

Several projects in industry [17, 44, 48] and academia [50] have focused on the development and use of *secure coprocessors* that guarantee their trustworthiness in a hostile environment by employing physical tamper-proofing measures as well as incorporating mechanisms that guard against loading untrusted software. Any attempt at unauthorized access to the system would destroy its contents, so as long as it still held a usable secret, it could be assumed to be secure. Such a system could then authenticate itself to another remote system by means of public-key cryptography based on an internal secret. Here, the system demonstrates its genuinity not by proving that it is a genuine computer but by proving its *identity*. That identity must be known in advance to other systems for the secure coprocessor's security to be meaningful to a remote party. In contrast, our work allows previously unknown systems to authenticate themselves to a central authority.

6.3 Secure Bootloaders

Several projects have been dedicated to the development of *secure bootloaders* that allow a computer system to authenticate the software that it loads [6, 11, 26, 50] by using a cryptographic secret stored in a secure coprocessor or on a smartcard. These systems require the integration of a secure BIOS or special loader to guarantee that no hostile code becomes operational on the machine in question. By comparison, our work requires no firmware modifications. It is also not sensitive to the *means* by which the loaded kernel became operational. As long as the genuinity test can verify that the kernel is running and in control of the system, a secure bootloader for the kernel offers no additional benefit.

6.4 TCPA, Palladium and LaGrande

Recently, a great deal of media attention has been focussed on the Trusted Computing Platform Alliance (TCPA) [1], Microsoft's Palladium Initiative [14], and Intel's LaGrande [35]. Nevertheless, it is not always clear how the respective systems work or what they are intended for. Many publications have been made available in an attempt to correct these misunderstandings [2, 4, 5, 18, 41, 43], and we encourage the curious reader to survey them carefully. We can point out two primary differences between these systems and our work:

- None of these systems appear to provide a means of demonstrating an anonymous system's genuinity to a central authority in the way our method does. Instead, they specifically aid in the generation and manipulation of cryptographic secrets for the purpose of identity management and access control.

- Each of the aforementioned systems require the addition of some form of hardware to a participating computer in order to manipulate and hide cryptographic secrets. We are not, at this time, advocating the addition of any security or cryptography hardware to the system, nor do we rely on any static secrets that must be integrated into the computer's firmware to support evidence of its identity.

What remains to be seen is whether these technologies will provide a useful mechanism for increasing our ability to resolve the genuinity of computer systems.

7 Summary

We have implemented a method by which a remote computer system can demonstrate the genuinity of its hardware and running software to a certifying authority without the need for a trusted human intermediary. We believe such a method will be useful for enabling the aggregation of arbitrary and, potentially, anonymous systems into distributed computational clusters. To revisit our introductory example, Alice could employ such a method to allow Mallory to add trusted hosts to her network environment with an acceptably low risk of Mallory gaining unauthorized access to resources. In addition, this could be done with no other demands on Alice's time other than the initial configuration and deployment of the certifying authority.

Our implementation shows a reasonable safety margin against potential simulation attacks even when using a very slow target microprocessor. Other more direct attacks are possible, but the complexity of their undertaking may allow our implementation to be acceptable in a number of environments. We continue to develop our implementation to broaden the number of applicable microprocessors by improving the types of meta-information that can be incorporated into the checksum algorithm. To our knowledge, all high-performance microprocessors presently in production have enough measurable side-effects to support a genuinity test. Lower-performance embedded CPUs may also be valid targets of our technique—especially if their instruction sets are not common to higher-performing CPUs since this would certainly force a potential attacker to use an instruction simulator.

Acknowledgments

We thank Mike Franzen for the numerous lengthy conversations that greatly influenced the early versions of this research. Mike Shuey also provided timely criticism in the development of the network protocol and timely loan of test machines. We thank the anonymous reviewers for their many detailed and helpful remarks.

References

[1] Trusted Computing Platform Aliance. TCPA main specification. http://www.trustedcomputing.org/.

[2] Ross Anderson. TCPA / Palladium Frequently Asked Questions. http://www.cl.cam.ac.uk/users/rja14/tcpa-faq.html.

[3] AOL. The America Online Instant Messenger Application. http://www.aol.com/, 2002.

[4] William A. Arbaugh. Improving the TCPA. *IEEE Computer*, 35:77–79, August 2002.

[5] William A. Arbaugh. The TCPA; what's wrong; what's right and what to do about it. http://www.cs.umd.edu/˜waa/TCPA/TCPA-goodnbad.pdf, July 2002.

[6] William A. Arbaugh, David J. Farber, and Jonathan M. Smith. A Secure and Reliable Bootstrap Architecture. In *IEEE Symposium on Security and Privacy*, pages 65–71, May 1997.

[7] R. Bedichek. Some efficient architecture simulation techniques. In *Proceedings of the USENIX Winter 1990 Technical Conference*, pages 53–64, Berkeley, CA, 1990. USENIX Association.

[8] S. M. Bellovin and M. Merritt. An attack on the interlock protocol when used for authentication. *IEEE Transactions on Information Theory*, 40(1):273–275, January 1994.

[9] Manuel Blum and Sampath Kannan. Designing programs that check their work. In *ACM Symposium on Theory of Computing*, pages 86–97, 1989.

[10] Jin-Yi Cai, Richard J. Lipton, Robert Sedgewick, and Andrew Chi-Chih Yao. Towards uncheatable benchmarks. In *Structure in Complexity Theory Conference*, pages 2–11, 1993.

[11] Paul C. Clark and Lance J. Hoffman. Bits: A smartcard protected operating system. *Communications of the ACM*, 37(11):66–94, November 1994.

[12] Christian Collberg and Clark Thomborson. On the limits of software watermarking. Technical Report 164, University of Auckland Dept. of Computer Science, August 1998.

[13] Christian Collberg and Clark Thomborson. Watermarking, tamper-proofing, and obfuscation—tools for software protection. Technical Report 170, University of Auckland Dept. of Computer Science, February 2000.

[14] Microsoft Corporation. Microsoft "Palladium": A business overview. http://www.microsoft.com/presspass/features/2002/jul02/0724palladiumwp.asp.

[15] Donald W. Davies and Wyn L. Price. *Security for Computer Networks, second ed.* John Wiley & Sons, second edition, 1989.

[16] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-11:644–654, November 1976.

[17] Joan G. Dyer, Mark Lindemann, Ronald Perez, Reiner Sailer, Leendert van Doorn, Sean Smith, and Steve Weingart. Building the IBM 4758 secure coprocessor. *IEEE Computer*, 34(10):57–66, October 2001.

[18] Paul England and Marcus Peinado. Authenticated operation of open computing devices. In *Proceedings of the 7th Australasian Conference on Information Security and Privacy*, pages 346–361. Springer-Verlag, 2002.

[19] Free Software Foundation. GNU GRUB. http://www.gnu.org/software/grub/grub.html, 2003.

[20] Philippe Golle and Ilya Mironov. Uncheatable distributed computations. In *CT-RSA*, pages 425–440, 2001.

[21] D. Grover. The protection of computer software: Its technology and applications. In *The British Computer Society Monographs in Informatics*, chapter Program Identification. Cambridge University Press, second edition, 1992.

[22] Mark Hachman. Via adds crypto to "Nehemiah", plans mobile launch. http://www.extremetech.com/article2/0,3973,838362,00.asp, 2003.

[23] Fritz Hohl. Time limited blackbox security: Protecting mobile agents from malicious hosts. *Lecture Notes in Computer Science*, 1419:92–113, 1998.

[24] VIA Technologies Inc. The VIA Padlock Data Encryption Engine. http://www.via.com.tw/en/viac3/padlock.jsp, 2003.

[25] VMware, Inc. The VMware workstation simulator. http://www.vmware.com/, 2002.

[26] N. Itoi, W. A. Arbaugh, J. McHugh, and W. L. Fithen. Personal secure booting. In *Proceedings of the Sixth Australian Conference on Information Security and Privacy*, pages 130–144, July 2001.

[27] Markus Jakobsson and Ari Juels. Proofs of work and bread pudding protocols. In *IFIP TC6 and TC11 Joint Working Conference on Communications and Multimedia Security (CMS '99), Leuven, Belgium*. Kluwer, September 1999.

[28] Steven Kent and Randall Atkinson. Security architecture for the internet protocol. IETF RFC 2401, November 1998.

[29] John B. Lacy, Donald P. Mitchell, and William M. Schell. CryptoLib: Cryptography in Software. In *UNIX Security Symposium IV Proceedings*. USENIX Association, 1993.

[30] Kevin Lawton. The Plex86 simulator. http://plex86.org/, 2002.

[31] Kevin Lawton. Bochs: The Open Source IA-32 Emulation Project. http://bochs.sourceforge.net/, 2003.

[32] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Adreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, February 2002.

[33] Peter S. Magnusson and Bengt Werner. Some efficient techniques for simulating memory. Technical Report R94-16, Swedish Institute of Computer Science, August 1994.

[34] Fabian Monrose, Peter Wyckoff, and Aviel D. Rubin. Distributed execution with remote audit. In *ISOC Network and Distributed System Security Symposium*, pages 103–113, 1999.

[35] Paul Otellini. Intel developer forum, fall 2002, keynote speech. http://www.intel.com/pressroom/archive/speeches/otellini20020909.htm.

[36] Etherboot Project. Etherboot. http://www.etherboot.org/, 2003.

[37] Niels Provos. Encrypted virtual memory. In *Proceedings of the 9th USENIX Security Symposium*. USENIX Association, 2000.

[38] PyxisSystemsTechnologies. AIM/oscar protocol specification: Section 3: Connection management. http://aimdoc.sourceforge.net/faim/protocol/section3.html, 2002.

[39] Ronald L. Rivest and Adi Shamir. How to expose an eavesdropper. *Communications of the ACM*, 27(4):393–395, April 1984.

[40] John Scott Robin and Cynthia E Irvine. Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor. In *Proceedings of the 9th USENIX Security Symposium*. USENIX Association, 2000.

[41] David Safford. The need for TCPA. http://www.research.ibm.com/gsal/tcpa/why_tcpa.pdf, October 2002.

[42] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun Network Filesystem. In *Summer 1985 USENIX Conference*, 1985.

[43] Seth Schoen. Palladium details. http://www.activewin.com/articles/2002/pd.shtml.

[44] Sean W. Smith, Elaine R. Palmer, and Steve Weingart. Using a high-performance, programmable secure coprocessor. In *Financial Cryptography*, pages 73–89, 1998.

[45] A.M Turing. On computable numbers: With an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265, 1936.

[46] L. van Doorn, G. Ballintijn, and W. A. Arbaugh. Design and implementation of signed executables for linux. Technical Report HPL-2001-227, University of Maryland, College Park, June 2001.

[47] Hal Wasserman and Manuel Blum. Software reliability via run-time result-checking. *Journal of the ACM*, 44(6):826–849, 1997.

[48] Steve R. White and Liam Comerford. ABYSS: A trusted architecture for software protection. In *IEEE Symposium on Security and Privacy*, pages 38–51, 1987.

[49] Emmett Witchel and Mendel Rosenblum. Embra: Fast and flexible machine simulation. In *Measurement and Modeling of Computer Systems*, pages 68–79, 1996.

[50] Bennet Yee. *Using Secure Coprocessors*. PhD thesis, Carnegie Mellon University, May 1994.