An analysis of proposed attacks against genuinity tests

Rick Kennell and Leah H. Jamieson
Purdue University, School of Electrical and Computer Engineering

August 9, 2004

Abstract

A number of attacks have been proposed against the idea of a *genuinity test*. The rationale for these attacks is based on misinterpretation of published details about this system. We correct these misunderstandings by providing a detailed analysis and contradictory evidence for each claim.

1 Introduction

We proposed the notion of a *genuinity test* [3] to provide a means of discerning whether or not a distant computer system consists of real hardware running an expected software environment. The basic mechanism used in such a test involves a software hash that evaluates memory contents while it incorporates architectural meta-information expected to be found for a real microprocessor of a specific type. A separate evaluator, called an *Authority*, can generate unique testcases and use each one to challenge a remote system, called an *Entity*, to ascertain its genuinity. Upon passing this test in a timely manner, the Entity remains under the supervisory control of the Authority.

While the general idea of a genuinity test is applicable to many types of high-performance microprocessors, we illustrated the development of a testcase for a specific type of microprocessor as an example. We deliberately chose a low-performance processor—specifically, a non-MMX Intel Pentium—in order to demonstrate the resilience of the method to brute-force simulation and similar attacks. The genuinity test that we illustrated was randomly generated by an Authority system. It resulted from a number of guidelines used in the construction of the Authority's test generator. This example was not necessarily the only form of genuinity test possible for the target microprocessor, nor was it optimal.

A sufficient amount of architectural meta-information must be incorporated into the genuinity test in order to it to provide a meaningful result. Central to the idea of testcase development is the necessity for concrete evidence that the code being evaluated is presently the code being run. Without such an indication, no guarantees can be made about the state of the system since proper termination of the testcase would not mandate the subsequent execution of trustworthy software. The need for this type of evidence is the first reason to rely on microarchitectural meta-information. A second reason is to ensure that the entire system is not simply running wholly within the context of some form of simulator. Because of the intrinsic parallelism of many microarchitectural features, simulation is necessarily slower than native execution.

A number of attacks against the example implementation which we described have been proposed. It is clear that since our implementation is not the sole way to construct a genuinity test for the architecture in question some improvements may eventually be needed. Nevertheless, we find that the attacks as described for the example implementation cannot succeed, and additional negative claims made about the security of our proposed system, in general, are demonstrably false.

This report is organized as follows: In Section 2, we consider the deficiencies of the example implementation proposed by Seshadri, Perrig, van Doorn and Khosla related to their development of SWATT [4]. In Section 3, we examine the attacks proposed by Shankar, Chew and Tygar [5]. We summarize this report in Section 4.

2 Deficiencies proposed in the description of SWATT

Seshadri et al proposed a technique useful for software-based attestation (SWATT) of the correct contents of a region of memory found in a device with an embedded processor. Their method is similar to ours in the sense that a pseudo-random traversal of memory is used to build a hash to indicate the contents of memory. Because this method is intended for embedded devices, where the availability of meta-information is rare, it is applicable to more types of systems than ours. They employ tight timing bounds to ensure that the program running to perform the attestation has not been tampered with. However, they do not consider the use of a network for transport of the attestation results.

In comparison to their system, the authors point out two deficiencies with our approach:

- The act of sending a challenge testcase in the form of executable code "introduces vulnerabilities due to the threats of mobile code."
- The technique "suffers from a security vulnerability that enables an attacker to change an arbitrary number of memory locations and remain undetected with a 50% vulnerability."

As we describe, at the end of Section 3 of [3], the first vulnerability is mitigated by the use of a public key embedded in the kernel of the Entity. Since we assume that the initial kernel is obtained from the Authority, the Authority can use the corresponding private key to sign a challenge before sending it to the Entity. This allows the Entity to discriminate against invalid challenges by checking that the test code is properly signed by the Authority. We give more information about detecting and overcoming denial of service attacks in Section 3.3.

The second criticism is overstated since the attack they describe involves only the manipulation of the most-significant bit of 32-bit words in the evaluated memory. The authors claim this deficiency comes as a result of our checksum algorithm reading 32-bit words. However, we state that our checksum algorithm reads *bytes* from memory and incorporates them into a 32-bit checksum value (Section 5.3, point 7 of [3]). Therefore, it is not subject to the described vulnerability.

The authors acknowledge in the description of their own technique that the final word has not been written about the best way to structure hash algorithms for purposes that we have each considered. Our own technique is not wedded to one particular implementation of checksum algorithm, and we look forward to seeing more defining work in this area.

3 Attacks proposed by Shankar, Chew and Tygar

Shankar et al (referred to hereafter as simply, "the authors") have proposed attacks [5] against the example implementation of a genuinity test for the Pentium that we described. They make a number of claims which we summarize as follows:

1. A *substitution attack* is possible against our described implementation as well as a broad class of genuinity tests in general.
2. Testing the genuinity of a system is practically infeasible due to the lack of precise architectural meta-information.
3. Denial of service attacks can be efficiently launched against either an Entity or an Authority.
4. Our technique is not sufficient to prevent adversaries from attacking protocols such as NFS.
5. We are promoting the use of genuinity tests for software-only systems (independently of hardware).

We show each of these claims to be false. The basis for each claim is a result of several misunderstandings of the authors including:

- Neglecting to follow development guidelines in construction of a genuinity test.
- Invalid attempts to apply specific nuances of an example testcase that was generated for a specific microprocessor to a general class of microprocessors.
- Misunderstandings with respect to crucial microarchitectural concepts.

The authors acknowledge that our described example was generated for a non-MMX Pentium processor. However, since they did not have ready access to such a system on which to evaluate their attacks, they used a Pentium-4—a very different type of microprocessor—to evaluate such a test. In so doing, they discounted the architectural artifacts that were different without taking advantage of the newer features that would have made the test more resilient to attack.

3.1 Substitution attacks do not work

A substitution attack involves replacing the genuinity test code so as to compute the correct result while hiding the indications that the code has been modified. The goal is such that, at the conclusion of the test, the Authority will believe that the Entity is genuine although it will be running the imposter code. Although the authors did not indicate how the imposter code was to be inserted into the testcase, we can assume that it would have to be either implanted while in transit over the network, or instrumented *in situ* on the target Entity.

The authors point out that it is important for the attack code to avoid loading values from or storing values to memory. Doing so would affect the state of hard-to-simulate meta-information sources such as the DTLB. Furthermore, since the ITLB will detect large-scale program deviations involving multiple pages, it is necessary to constrain the attack code within the page allocated to the testcase. To do so they suggest a clever scheme that replaces the code that normally interrogates the state of the instruction cache with expanded code (within the same virtual page) that covers over its own existence with only a small runtime expansion. When the code is invoked, if an inquiry is being made about the region it resides in, it detects and corrects for the situation using a series of branches and immediate loads.

Five problems exist with this approach:

1. Although the quoted time expansion, alone, may still fall within the Authority's deadline, it does not include the time necessary to extract the testcase from the network, analyze the testcase, find the appropriate place(s) to insert the attack code, repackage the testcase in such a manner so as to forge the Authority's signature (discussed previously in Section 2), and re-send the test to the Entity. Even if the attack was performed *in situ* on the target Entity, the analysis required to determine the multiple points to re-instrument the code would require a long period of time. Because every genuinity test consists of a unique arrangement of code, these steps are always necessary to initiate a substitution test. These delays were not considered by the authors.

2. The fact that the genuinity test illustrated in our example had only 22 nodes, only one of which interrogated the caches, was simply an artifact of the random generation process of the Authority. Testcases are readily generated with fewer or many more nodes. A test may also have multiple cache interrogation nodes. Generally, it is desirable to generate tests in such a manner that they completely fill the pages that they reside in. Doing so complicates the necessary analysis to find insertion points for the imposter code. It also makes it more difficult for an attacker to determine how the imposter code can repair the damage created by its own presence.

3. Both the signed copy of the test sent over the network and the in-kernel public key can be exposed to the memory checksum, resulting in a further broadening of necessary attack code exclusions. For this reason alone, it is doubtful that a remote substitution attack is possible due to the limited memory constraints of the test environment.

4. The ability for an attacker to augment or diminish the core genuinity test would violate our stated principle that there must be some evidence that the code being tested was actually being run. Because this attack was implemented on a Pentium-4, the appropriate meta-information would have been different than the ones described in our example. However, the authors chose to limit the utilized meta-information sources to only the ITLB miss count and branch count, thereby allowing an imposter to be inserted and run anywhere on the memory page used for the test. The code that the authors attacked did not constitute a complete genuinity test.

5. Had this Pentium-specific example been run and attacked on a Pentium, the instruction cache, by virtue of the fact that it is a complex indicator of what is being run, would have still exposed the presence of introduced attack code.

We expand on the last item. To do so we must explain the function of the Pentium instruction cache in detail.

The Pentium icache is arranged as sets of two 32-byte lines. When an instruction is to be fetched, its address is divided by the line size, and the lower bits of the result are used as an index into one of the sets. Each line includes an extra bit to indicate whether it is in a valid or invalid state as well as a *tag* that contains the remaining upper bits of the address that were not used for the set index. If the tag bits of one line match those of the fetched address, the cache access is a *hit*, and the fetch completes quickly. Otherwise, the fetch must be satisfied with a memory request, and one of the two lines of the appropriate set must be replaced with a valid entry. One additional bit per set keeps track of which of the two lines is the newest so that, in the case that both lines are valid when a replacement is needed, the older of the two will be replaced. We referred to this bit as the Least Recently Used (LRU) indicator.

The *testability registers* for the Pentium [1] allow a program to issue a request to directly examine the contents of a given line. To do so, the set index, the 1-bit line selector, and the 3-bit buffer offset are placed into the appropriate testability register to instruct the processor to interrogate the cache. Thereafter, a second testability register contains a 32-bit portion of the line at the selected offset. A third testability register contains the address tag for the selected line as well as the one bit LRU indicator and valid flags for the set.

We specified that the appropriate testcase node of our example implementation extracted the tag and replacement data from the icache[1]. The icache set and line were selected using the running checksum value at the time. This raw meta-information source was, like all other sources, incorporated into the checksum using a 32-bit XOR operation. This information changes dynamically; it is not invariant, as the authors suggest.

The authors of the substitution attack discounted this fact and instead described an attack where the memory data of an icache line was incorporated into the checksum. It is important to recognize that such an operation provides no more information to the genuinity test than does the customary summing of bytes. Nevertheless, an inserted attack would still be required to maintain enough state to keep track of the LRU bit as well as which lines were currently present in the cache set corresponding to the replaced cache set.

To describe this situation, consider a testcase consisting of a number of nodes which branched to each other in a pseudo-random fashion. If the testcase were arranged so that each node mapped to the same icache set, there would always be a question of which two cache blocks were present in the cache as well as which way the LRU indicator was set. In the case of limited cache associativity, it is also possible to occasionally jump to trampolines embedded in kernel pages to create additional contention for a given icache set. At minimum, this would require a substitution attack to keep track of enough bits to represent the LRU and encode all the possible variations of two-line combinations in the icache set.

[1]Actually, the node in question also interrogated the dcache in a similar manner. This does not affect the nature of the attack.

By requiring an attacker to keep track of state, it means that all code sections that affect that state must also be instrumented with imposter code. As the problem unfolds the required solution approaches simulation. Simulation of implicitly parallel microarchitectural features is inevitably slow. Furthermore, it raises the question of where to save the state needed for the simulation. We consider this further in Section 3.1.2.

It is worth noting here that a genuinity test where all nodes caused contention in one icache block would be slow—possibly slow enough that brute-force simulation on a faster system would become feasible. Because of this, it is important that the arrangement of the testcase only *occasionally* execute certain nodes in order to limit the number of cache misses to a reasonable level but still cause enough runtime ambiguity regarding the current contents of the icache. A pseudo-randomly directed testcase can use biases to avoid the excessive invocation of certain testcase nodes.

3.1.1 Two-page substitution attacks do not work

Another attack similar to the basic substitution attack involves replacing a previously zero-filled page with imposter code that is invoked as needed. Since this attack will execute code found on an entirely different page, an assumption is made that the ITLB misses will be corrected by the imposter code.

A number of technical errors exist in the description of this attack:

1. The authors incorrectly state that the Pentium has a 48-entry, fully-associative ITLB. We are not aware of any x86 microprocessor with such an ITLB configuration. The Pentium has a 32-entry, 4-way set-associative ITLB [1].

2. The authors assume that it will be possible to run the test in advance to determine how and when its ITLB miss count will stabilize. This assumes that 22-nodes of a genuinity test (plus 22 imposter nodes) will be used with a 48-entry ITLB. This implies that, in addition to all of the delays involved with analyzing and modifying the uniquely-generated test,

this attack also requires a first run to characterize the ITLB fill pattern. Even using a much faster execution host, this will certainly miss the Authority's deadline. In reality a 22-node testcase would not be appropriate for a 48-entry, fully-associative ITLB. There must always be more nodes in the genuinity test than the associativity of the ITLB in order to avoid reaching a steady-state condition. If not, the miss count is worthless as an execution meta-information source. Simulation of the ITLB (including full 4-way pseudo-LRU evaluation) is necessary for a two-page substitution attack.

3. The authors assume that the testability registers can be used to insert values into the icache in order to mimic the natural effects. Artificial insertion of values into the icache of a running system would be likely to eventually replace a line that was currently being executed. This could cause the flow of execution to be changed in a manner that would undo the imposter's intent. It might also cause the processor to halt because of illegal instructions if it fetched a new cache line while that line was being updated. Furthermore, this approach also does not obviate the need for the same icache replacement simulation as would be required for the single-page version. There must be some stored state to indicate what configuration the icache should be in under natural circumstances.

3.1.2 The hopelessness of other substitution attacks

As the complexity of a proposed substitution attack increases, so does the need for simulation. Although the problem of simulation latency may be mitigated by using a faster processor implementation, it does not address how or where to store the additional simulated state. For instance, full simulation of a single Pentium cache set requires the storage of two 'valid' bits, one LRU bit and two 20-bit address tags. Although the additional required state could be reduced by symbolically encoding all possible line states for the set, it would still require storage and manipulation of a few bits. Even more bits would be required for simulation of a mechanism such as a TLB set because of the higher associativity.

Storage of the values in memory would lead to either corruption of (or full simulation of) the DTLB which has already been identified as a difficult problem. The x86 architecture has a very constrained register set, so finding space to store and manipulate values there requires a careful, efficient encoding. Even for an architecture with more registers, a mechanism to detect such an attack would be easily incorporated into a genuinity test by simply initializing all extra registers, using them for temporary storage of intermediate values, and occasionally incorporating them into the memory checksum.

We note that even normal genuinity tests use a hidden bit of storage (for instance, in one of the x86 debug registers) to indicate whether the running test should be used only to generate results for a test without subsequently jumping into secure operation. It is possible to preserve some such storage without allowing its illicit use to proliferate.

The authors claim that the problem of augmenting their attacks to be able to work against additional measures such as self-modifying code and dynamically-varying performance counters does not seem insurmountable. However, these attacks would also require additional state, much more aggressive analysis, and further instrumentation of code unrelated to the imposter. Simulation becomes the only mechanism likely to succeed.

Finally, although the authors show how to cleverly implement imposter code that avoids the use of memory accesses, additional branches are introduced into the flow of execution. Such branches will have an impact on the branch predictor and this will have a subsequent impact on other processor subsystems due to the entanglement of meta-information discussed in the next section.

3.2 Obtaining meta-information is possible

The authors of the attacks make several claims that the meta-information sources found on the Pentium, as well as other processors, do not produce deterministic values and are therefore unacceptable for incorporation into a genuinity test. Intuitively, this is a difficult notion to accept since the internal state of a microprocessor is finite. At some level, its entire operation must be deterministic. While we readily grant that certain operations of the processor may vary according to unpredictable delays in execution (e.g. dynamic memory refresh contention), as long as processor decisions are not based on these values, operation will remain predictable. A case in point is our use of a timestamp counter to generate a random, or at least unpredictable, value due to timing variations introduced by the memory subsystem. As long as no decisions are made based on this value, it should not affect the operation of the processor.

Ostensibly, internal race-conditions that exist within a complex speculative processor are similarly deterministic. Given two identical processors with equivalent state, running the same code, with the same memory contents, one would have every expectation that they will produce the same values. Often there is some difficulty in forcing a particular processor into a deterministic state. This situation requires some investigation in order to find the correct manner of doing so. For instance, for the Pentium, simply invalidating the caches and TLBs does not necessarily force all of their LRU information into the same state each time. However, some post-processing instruction sequences can be executed reasonably quickly that will produce a definite start state.

Another difficulty in using these meta-information sources lies in the possibility of predicting their values using a simulator. Certainly, the simulator must match the full functionality of the processor, but this is tedious, time consuming, and, very often, difficult because of undocumented corner cases and processor implementation bugs. Because some subsystems affect others, the necessary complexity of the simulator often grows to an unmanageable level. For instance, the branch predictor of a processor indirectly affects the ITLB since mispredicted branches cause the wrong line (from the wrong page of memory) to be fetched. Even though the ITLB miss count may be preserved, low-level access to the ITLB may still show the effects. In the case of the Pentium, it is easy to initialize the branch predictor to a certain state. However, incorporating branch prediction functionality into a simulator proved to be harder than it was worth. In order to be able to legitimately compare an example of such a test to a simulator, we instead used a genuinity test for which branch prediction was disabled. For most purposes, in order to avoid the need to create precise, high-performance simulators, we advocate using native calculation of the testcase results.

Finally, we must also consider trends in future microprocessors in order to anticipate different sources of meta-information. Although it is possible to envision a microprocessor for which no execution meta-information is made available, in reality it will always be obtainable. For instance, a mispredicted branch will always take longer than a correctly predicted branch. Furthermore, as architectural complexity increases, there is a growing need to use internal information sources simply as an indication of whether or not a processor is functioning correctly. This was the case even with the Pentium and was the motivation for creation of the testability registers. There are many undisclosed and undocumented sources of meta-information hidden in other microprocessors as well.

3.3 Avoiding denial of service attacks

The authors claim that denial of service attacks are easily mounted against the unchecked Entity as well as the Authority. We contend that such attacks are not as efficient as the authors claim and that the ones which may be successful occur in pathological circumstances where most other network services would fail as well. In any case, behavior of our system in the presence of denial of service is always fail-stop.

3.3.1 The Entity can resist DoS attacks

The proposed DoS attack against the Entity involves an attacker that waits for an Entity to request a challenge from an Authority. While the Entity computes the result, the attacker sends invalid responses to the Authority disguised to look as though they were sent by the Entity. The goal is to get the Authority to send refusals to the real Entity in order to prevent it from making forward progress in establishing a relationship with the Authority. However, such an attack is easily detected by the Authority since multiple responses from the same Entity indicate the presence of an adversary. The Authority must record the time that each response packet is received. However, it can accumulate the responses and simply defer evaluation until some reasonable time has elapsed. Each response can be evaluated to determine if *any one of them* is correct and received before the appropriate deadline. The Authority sends a "qualification" packet to the Entity (and, likely, the adversary as well) that carries no information other than an indication that the Authority is ready to negotiate a key exchange. Only the Entity that has passed the genuinity test will be able to respond to the Authority correctly. Subsequent attempts by the attacker to send invalid data for the key exchange will also be detected.

3.3.2 DoS attacks against the Authority are hard

The proposed DoS attack against the Authority rests on the assumption that computation of the correct checksum value for a particular uniquely-generated genuinity test must always be performed by the Authority using a simulator. However, we did describe (in Section 5.2 of [3]) a way of precomputing testcase results natively using systems under the direction of the Authority that are already known to be genuine. Without the availability of a simulator there is naturally some challenge in getting the Authority's testcase generator bootstrapped for each of the supported architectures. We can accomplish this by forcing the Authority to trust a known-good system that is physically secure in order to generate initial testcase results. Once other Entities are known by the Authority to be genuine, they can each be instructed not only to compute testcase results but to generate the testcases as well, allowing the Authority to off-load much of its work. While testcases can be constructed during regular operation of the Entity, the evaluation of that testcase must be done with interrupts temporarily turned off. From the perspective of a user, this would appear to be a short pause. It is best to avoid doing this to non-idle Entities with interactive users. Evaluation of testcases on either idle Entities or non-idle Entities with non-interactive use would not cause a perceptible problem. Groups of known-genuine Entities can thereby generate new testcases much faster than a single Entity can use them.

A number of policies can be constructed to prevent the Authority from depleting its supply of testcases for a particular architecture. First, it is reasonable to expect that an Entity that fails a genuinity test be denied additional attempts for a progressively longer period of time. Indeed, at some point, multiple failures from a given IP address (or group of IP addresses) are more likely an indicator of either an attempted attack or a general failure of the system. Second, it would be reasonable for an Authority to maintain jurisdiction over zones of IP addresses from which it might expect requests for genuinity test challenges rather than serving as a global Authority.

3.4 Appropriateness for securing network protocols

The authors suggest that our proposed system would not be suitable for a number of network-based applications. AIM clients are mentioned in this category; we address this matter in Section 3.5. Our primary practical example was the discussion of how to allow remote systems to act as NFS clients. We now repeat this example in order to eliminate misunderstandings of how the system is intended to work.

3.4.1 Genuine entities can act as reliable NFS clients

Our example involves Alice, a scrupulous system administrator, who tends to the needs of a number of adversarial client users. Among them are Bob, a hard-working NFS user, and Mallory, a thief. Bob would like to use a collection of remote computer systems (which Alice does not maintain) for the purpose of performing a large, distributed computation. Bob requires that these systems have access to Alice's NFS server. Mallory would like to subvert one or more of the NFS clients in order to gain access to Bob's data. For the sake of example, we might even assume that the machines are physically accessible to Mallory.

Alice begins by setting up an Authority system that will create and dispense genuinity tests. The remote Entities, running without the use of their local disks (as we describe in Section 5 of [3]), will not be subject to either the configuration of their local administrator, nor are they expected to be modifiable by Mallory. Each Entity requests, evaluates and passes a genuinity test. They remain under the administrative control of the Authority (and, transitively, Alice). Thereafter, each Entity negotiates with the Authority to perform a key exchange after which they can communicate securely with each other. In particular, they also negotiate IPsec keys for transparent encrypted and authenticated encapsulation of network packets. If we assume the Authority is the NFSv3 server, the Entities can then be trusted to mount its NFS exports. The Entities are known to be trustworthy, the server is assumed to be trustworthy, and the network transport is secure. Some additional negotiations are required to allow the Authority to enable a peer NFS server to use IPsec encapsulation between itself and the Entities which the Authority has found to be genuine.

Bob is allowed to remotely log in to the systems which are running under the administrative control of Alice's Authority. This may be done by either manually creating local accounts or using a network authentication mechanism such as LDAP. Mallory might even be allowed to log in as well, either remotely or on the console. The usual Unix file permission mechanism applies to the genuine Entities as well as it does for any known physically-secure system. In order for Mallory to subvert a known-genuine Entity, it would be necessary to physically attack the system via its memory bus. We discuss weaknesses such as this in Section 4.3 of [3].

To further clarify the situation, we correct some of the authors' misunderstandings. First, neither Bob nor Mallory act as administrators of the remote Entities and cannot misconfigure the systems. They are all under the administrative control of Alice's Authority. This means that the Authority will instruct the Entity as to what filesystems to mount at boot, what peripherals it should use and what daemons it will run. We also reasonably assume that Alice knows what she's doing. Second, using NFSv4 instead of NFSv3 does nothing to augment (or diminish) the security of the system. The negotiated IPsec encapsulation ensures that the file system transport is secure as well as any distributed user authentication system that Alice puts in place.

The authors correctly point out that our system was not designed to address user authentication. Genuinity of computer systems is an orthogonal issue with respect to authentication in the same sense that secure network routing is orthogonal to user authentication. However, it is possible for one to leverage the other to provide augmented services.

The authors also astutely note that our system does not ensure globally-unique identification of systems. Generally, this is unnecessary, so long as the Authority has some reliable means of interacting with a known-genuine Entity. For instance, a genuinity test and its subsequent negotiations should be capable of transiting a firewall.

Finally, we note that NFSv4, although it is a desirable extension to the NFS suite, would not serve as a singular solution to the particular problem we posed. If Alice exported an NFSv4 share to a remote system for which Bob had remote access and for which Mallory had root access, all that Mallory would be required to do is wait for Bob to log in, "change user" to Bob, and read and modify anything. Even for modern credential systems that we are aware of, user processes are generally equivalent in capability in order to allow systems like "cron" to function without requiring a password for filesystem access. In any case, if Mallory is able to gain root access, the system's credential policies could be easily modified as well.

3.4.2 Other network applications are possible

One application that we did not suggest that the authors of the attacks did was the situation of a set-top box used for brokered or distributed gaming. Since we show that substitution attacks and several other forms of attack are unlikely to be achievable, this scenario presents an ideal opportunity for the use of a genuinity test. The reasons for this are as follows:

- The full specifications of the hardware are known in detail, thereby allowing the development of a genuinity test that uses as many execution meta-information sources as possible.
- The owner of such a system could select which Authority would be of greatest use, enabling the development of a market structure whereby game providers (or other service providers) could compete for clients.
- Because the system would not rely on internal hardware-based trusted secrets, it could still be used for general purpose tasks when the original service provider eventually drops support. Meanwhile, other systems that rely on an internal hardware-enforced trusted computing base (such as the Xbox) are doomed to extinction once their support ends since there will be no one to sign software that will run on them.

The authors propose various attacks against such hardware involving several forms of direct interrogation of hardware to discover a negotiated key. We also mentioned this in our description of attacks as being the most likely means of breaking in to a known-genuine Entity. The authors suggest that an attack of this nature would be easily mounted by use of an ordinary bus-mastering I/O card. However, it is unlikely that a secret would be stored in a location mapped into available I/O space. A memory bus attack is a more viable approach. Nevertheless, such an attack would be complicated because the hardware needed to snoop the memory bus is not readily available, hard to build, and difficult to use. Furthermore, the location of the key could be obfuscated by the system, and active techniques could be employed to avoid leaking secret cache values to the external memory bus.

One illustrated attack on a memory bus [2] involved a system that used trusted hardware to hold a secret key.

Once the complexity of building the memory snooping system was overcome, this attack was somewhat simpler than an attack on a general purpose system because the bus transactions involving the key were easily identified. After the secret key was discovered by snooping the memory bus on one system, it (and all other systems like it) could be modified and exploited relatively easily since they all used the same secret key.

By contrast, since our method does not involve static keys in hardware, compromise of one system does not imply a compromise of all systems. Furthermore, the only way to leverage one compromised system to exploit others would be to set up an infrastructure as the authors suggest with their economic attack. A primary recommendation of the work presented in [2] was that all chip-to-chip busses of the system should be dynamically encrypted. In a set-top box scenario, such a mechanism could actually be implemented without risk of backwards-incompatibility with other systems, and this threat, as well as the remaining possible hardware attacks we illustrated in Section 4.3 of [3], are eliminated. Note that memory bus encryption can and should be done without the need for static stored secrets in the hardware in order to avoid the problems of vendor lock-in and obsolescence pointed out above.

3.5 Software-only systems are not the subject of genuinity tests

The authors claim that we described our system to fill a need in authenticating software systems. Indeed, the title of their paper seems to promulgate this misunderstanding. In particular, they claim that we proposed our system to be used for authenticating AOL Instant Messenger (AIM) clients. We did refer to AIM as an example of a failed form of software-only attestation. We did not and do not claim that a genuinity test can serve as a discriminator of software alone. Although a known-genuine system (hardware and software) could be used to ensure that an arbitrary user did not invoke an illegitimate form of software, this is not the type of problem we are attempting to solve.

4 Summary

We have shown how the attacks and deficiencies described in [4] and [5] are not sufficient to defeat systems that use genuinity tests. Our proposal stands as a potential method of creating remote trusted computing systems in a variety of applications, although much work remains to be done in evaluating different architectures and testing deployments of the system.

Because our system differs substantially from previously proposed solutions for trusted computing environments, some misunderstandings are not unexpected. In particular, a typical argument for or against using a genuinity test comes down to a basic question:

- When a trusted remote system is required for an application, which is easiest to do? Add a hardware-based mechanism for sealed storage of secrets and software attestation that can be useful for establishing identity of a system? Or exploit the nuances of existing microprocessors to determine whether or not they are real and running an expected software environment?

Because we find evidence indicating that trusted systems can be built using existing microprocessors, we are motivated to pursue the latter course as being the more general solution.

References

[1] Intel Corporation. *Embedded Pentium Processor Family Developer's Manual*, volume 1. 1998.

[2] Andrew Huang. Keeping Secrets in Hardware: The Microsoft Xbox Case Study. In *Proceedings of CHES2002*. Springer-Verlag, August 2002.

[3] R. Kennell and L. H. Jamieson. Establishing the Genuinity of Remote Computer Systems. In *Proceedings of the 12th USENIX Security Symposium*, pages 295–310. USENIX Association, August 2003.

[4] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla. Swatt: Software-based attestation for embedded devices. In *IEEE Symposium on Security and Privacy*. IEEE, 2004.

[5] U. Shankar, M. Chew, and J. D. Tygar. Side effects are not sufficient to authenticate software. In *Proceedings of the 13th USENIX Security Symposium*. USENIX Association, August 2004.