VM '04 Paper [VM '04 Technical Program]

Semantic Remote Attestation -- A Virtual Machine directed approach to Trusted Computing

Vivek Haldar, Deepak Chandra and Michael Franz

{vhaldar, dchandra, franz}@uci.edu

Department of Computer Science

University of California

Irvine, CA 92697-3425

Abstract:

Remote attestation is one of the core functionalities provided by trusted computing platforms. It holds the promise of enabling a variety of novel applications. However, current techniques for remote attestation are static, inexpressive and fundamentally incompatible with today's heterogeneous distributed computing environments and commodity open systems. Using language-based virtual machines enables the remote attestation of complex, dynamic, and high-level program properties -- in a platform-independent way. We call this *semantic remote attestation*. This enables a number of novel applications that distribute trust dynamically. We have implemented a prototype framework for semantic remote attestation, and present two example applications built on it -- a peer-to-peer network protocol, and a distributed computing application.

Introduction

Two major trends have been immensely influential in today's computing environment. The first is heterogeneity. We compute using everything from small cellphones, to handhelds, to desktop workstations, to rack-mounted servers -- and these must all inter-operate. This has led to the widespread acceptance of open protocols for communication and portable language runtimes (such as the Java and .NET virtual machines) for execution of programs.

The second major trend is mobility. Not only must all these varied computing devices inter-operate seamlessly, we must also be able to use most of our familiar programs and data across all of them. Both of these have

significantly increased the importance of having a so-called ``common platform''. This common platform is increasingly the language runtime, that executes some form of platform-independent mobile code.

As we become dependent on this computing infrastructure, its weaknesses also become painfully apparent. On the one hand, we have periodic waves of network outages caused by worms such as Nimda and SQL Slammer. On the other, content producers want control over the proliferation of their creations. The network is a hostile place -- the default assumption is to assume an adversarial remote machine.

One way to get security assurances is to use *closed systems*. They enforce compliance with a certain security policy by being tightly controlled. They are usually manufactured by a single vendor to rigid specifications. Designers have complete control over the whole system and build it specifically to conform to a given security policy. When one closed system communicates with another, it knows within very tight bounds the expected behavior of the remote party. Common examples of closed systems are automated teller machines (ATMs) and proprietary game consoles

Open systems, on the other hand, have no central arbiter. Commodity personal computers and handhelds are examples of open systems. An open system can be easily changed to behave maliciously towards other systems that communicate with it. Two communicating open systems cannot assume anything about each others' behavior, and must be conservative in their assumptions.

Trusted computing is an effort to bring some of the properties of closed, proprietary systems to open, commodity systems. Trusted computing introduces mechanisms and components in both hardware and software that check and enforce the integrity of a system, and allow it to authenticate itself to remote systems. For example, a trusted booting procedure makes sure that the operating system has not been tampered with. Using a chain of reasoning that starts from a trusted hardware module, we can arrive at a conclusion about the state of a system after boot-up. Similarly, we can deduce for sure what particular program is running on a system. *Remote attestation* is the process by which software authenticates itself to remote parties. This allows the remote party to make certain assumptions about the behavior of the software.

Before trusted computing can reach its full potential, questions such as the following need to be addressed:

- how do programs running on trusted platforms authenticate each other in a manner that ensures that each party satisfies some security criteria, while leaving room for various differing implementations?
- the current client-server network computing model assumes a trusted server, and untrusted (even malicious) clients. Thus, even though a significant fraction of work is done at the clients, all the trust resides at the server. How can we design new network protocols (or adapt existing ones) to work in an environment that allows a more flexible partitioning of trust?
- Moving away from the model of having a fully trusted server, and a fully untrusted client, how do we design models and applications that use them, that can broker trust in more flexible and dynamic ways than is possible today?

As we explain in this paper, current methods of remote attestation suffer from many critical drawbacks. It is a technique well-suited to rigidly controlled closed systems, but completely inadequate for the open systems of today, which embody a great variety of hardware and software platforms.

We propose a way out of the problems of standard remote attestation by using a trusted virtual machine as a basis for doing remote attestation. We call this *semantic remote attestation*, since it can certify high-level, fine-grained, and dynamic properties of platform-independent mobile code. The core idea behind semantic remote attestation is to use a *trusted virtual machine* (TrustedVM) to explicitly derive or enforce, on behalf of a remote party, various properties of applications running within it.

Intuitively, ``authentication'' of an entity should have a broader meaning than it does currently. It should encompass not just verifying the source from which it originated, but also include verifying or proving that its behavior conforms to a required security policy.

To gain experience with semantic remote attestation, we have implemented a prototype TrustedVM, and two example applications on it. The first application is a simplified peer-to-peer networking protocol, and the second is a distributed computing client-server application. Implementing these within our framework achieved two benefits:

- trust relationships between peers, or between clients and servers, were *made explicit, and then checked or enforced* by the TrustedVM. Typically, they are implicit and taken on trust.
- making the trust relationships explicit results in having some knowledge of *degree of trustworthiness* of clients and peers. (for example, knowing which properties were satisfied, and which were not). This allows the applications to make informed decisions about the ``goodness'' of a result, and dynamically adjust its trust relationships.

The rest of the paper is organized as follows: Section 2 briefly covers basic concepts in trusted computing and remote attestation, and explains the shortcomings of remote attestation as it is today; Section 3 explains how virtual machines can be used for flexible, expressive remote attestation -- this forms the core of the paper; Section 4 discusses the implementation issues involved in this and presents two example applications; Section 5 surveys related work; Section 6 discusses avenues for future work; and Section 7 concludes.

Background: Trusted Computing and Remote Attestation

The broad goal of trusted computing is to add components and mechanisms to open, commodity systems to bestow on them some of the properties of high-assurance closed systems. This is done using a combination of hardware and software support. It requires three core mechanisms:

- **Secure boot:** to make sure that the system is booted into a trusted operating system that adheres to some given security policy. If such a mechanism is not provided, an adversary could boot the system into a modified operating system that bypasses the security policy.
- **Strong isolation:** to prevent the system from being compromised after it has been booted, and to prevent applications from tampering with each other.
- **Remote attestation:** to certify the authenticity of software being run by a remote party.

The first two guarantee integrity -- that the system was not tampered with since it was last turned off (secure boot), and that execution of programs will not be tampered with (strong isolation). The third has to do with authenticity -- to be sure of the identity of a remote party or program. The focus of this paper is remote attestation.

At the root of the attestation process is a hardware device called a *trusted module* (TM). This has embedded in it the private key of a public-private key-pair. This key-pair is certified by a certification authority (CA). The TM also has a small amount of non-volatile storage. A hash of the BIOS is signed using the TM's private key and stored here. At boot-up, control is first passed to the TM. It recomputes the hash of the BIOS, and compares it with its stored hash. If they are the same, then we know that the BIOS has not been tampered with, and control is passed to the BIOS. The BIOS does a similar check before passing control to the boot-loader. And the boot-loader does a similar check before passing control to the operating system. All layers along this chain have their own public-private key-pair, which is certified (signed using the private key) by the layer immediately preceding it in the chain. This in turn is used to certify the layer immediately above it. Precisely, each layer signs two things of the layer above it: a hash of its executable image, and its public key. This binds the public key to the software.

Note that the private key of every layer along this chain must be kept secret from the layer immediately succeeding it. Thus, the BIOS must be unable to read the TM's private key, the boot-loader must be unable to

read the BIOS's private key and so on. At the successful completion of this chain of checks, the system is guaranteed to have booted into a trusted, untampered operating system.

Thus, a combination of hardware (the trusted module) and software (used for secure booting and isolation) is needed to guarantee both integrity and authenticity of a trusted system. The trusted computing base of this setup consists of the hardware trusted module, and a trusted operating system that handles booting, and enforces strict isolation between applications.

Remote Attestation -- and its problems

Remote attestation is the process by which an application authenticates itself to a remote party. When asked to authenticate itself, an application asks the operating system for an endorsement. The operating system signs a hash of the executable of the application. The entire certificate chain, starting from the trusted module all the way up to the application, is sent to the remote party. The remote party verifies each certificate of this chain, and also checks that the corresponding hashes are of software it approves.

The attestation process must result in the client and server sharing a secret, or else the session can be easily hijacked (e.g. by doing the attestation using one program, and further communication using another).

This method of remote attestation suffers from several critical drawbacks. Briefly, they are:

- It says nothing about program behavior
- It is static, inflexible and inexpressive
- Upgrades and patches to programs are hard to deal with
- It is fundamentally incompatible with a widely varying, heterogeneous computing environment
- Revocation is a problem

We discuss each of these in more detail below.

The most critical shortcoming of remote attestation is that it is not based on program behavior. Even though what is fundamentally sought is some assurance of program behavior (with respect to some security policy), remote attestation certifies something completely different. It simply certifies what exact executable is running. Any assurances about the behavior of the program are taken on trust. It is possible for an attested program to have bugs, or otherwise behave maliciously.

Remote attestation defined in this way is completely static and inflexible. It can convey no dynamic information about the program, such as its runtime state, or the properties of the input it is acting upon. It is a one-time operation done at the beginning of a network protocol.

Another problem is that upgrades and patches are hard to deal with. Linear upgrades from one version to the next can be dealt with by simply updating the list of ``approved'' software that a remote party uses. In closed and tightly controlled systems such as ATMs, this is tractable.

The situation with widely available commodity software is completely different. As is increasingly common today, upgrades and patches are released very frequently. Also, software is patched more often than it is upgraded. There are usually multiple patches for multiple bugs and insecurities for a given program. Any subset of these patches may be applied in any order. This results in an exponential blowup in the space of possible binaries for a program.

In such a scenario, remote attestation faces problems at both ends of the network. Servers have to manage the growing intractability of maintaining a very large list of ``approved software'', which is likely to always be behind the current state. Clients, on the other hand, may have to hold off on applying patches or on upgrading, simply to be able to work within a remote attestation framework.

Today's computing ecosystem is extremely varied and accommodates a spectrum of heterogeneous systems with widely varying capabilities. These systems range from high-end supercomputers, to consumer devices such personal computers, handhelds, cellphones and watches, and even ubiquitously embedded microprocessors. Thus, a high premium is placed on portability and interoperability. This is one reason why cross-platform portable solutions such as Java are so popular. Remote attestation, however, with its stress on certifying particular platform-specific binaries, is fundamentally incompatible with this reality. Just as with managing upgraded and patched versions of software, certifying programs that run on a variety of platforms and that must inter-operate with each other, quickly becomes intractable.

Remote attestation inherits a problem from public-key cryptography -- revocation. Once a certification authority issues a certificate, it is very hard to revoke. One method is to have publicly available certificate revocation lists (CRLs) which are looked up at regular intervals. Thus, there may be some time lapse between a certificate being revoked, and access being denied to it. Checking with some revocation infrastructure (such as a CRL) at every attestation would be very inefficient.

Semantic Remote Attestation

The shortcomings of traditional ways of remote attestation can be traced back to one root cause -- *what is desired is attestation of the behavior of software running on a remote machine, but what actually gets attested is the fact that a particular binary is being run.*

Whether a remote party is running, say, Outlook 5.1, is not per se the information that is sought. What is sought is whether that particular program abides by some security policy. However, all that traditional remote attestation is able to certify is simply what exact binary code is running on a remote platform. From this, an indirect implication is drawn about the program's behavior. It is very important to realize that such an assurance gained about a program's behavior is based purely on trust, not on verification. Given the knowledge of what exact program is running on a remote platform, we trust it to behave according to a given policy because, essentially, the vendor of the software claims so. The chain of cryptographic certificates binds this claim to the vendor, and the program.

This is a direct consequence of using purely cryptographic methods for remote attestation. Cryptography is good at certifying entities -- it is not suitable for certifying behavior.

The solution we propose leverages the techniques of language-based security and virtual machines. Language-based security [21,25] has been variously defined as ``a set of techniques based on programming language theory and implementation, including semantics, types, optimization and verification, brought to bear on the security question'' and ``leveraging program analysis and program rewriting to enforce security policies''. It derives assurances about the behavior of the code by looking at the code itself.

Recent and very promising examples of this approach include proof-carrying code[24], typed assembly language (TAL), inlined execution monitors[14], and information flow type systems[23]. These techniques can be thought of as falling into two major categories -- program rewriting and program analysis. Program analysis covers a variety of techniques that statically try to check a program's conformance to a security policy. The primary example of this is type-safe programming and types-based approaches to security such as TAL. Program rewriting is a complementary set of techniques which aim to enforce security by rewriting programs to conform to a security policy. Inlining security monitors is an example of this class of techniques. The primary advantage of the language-based approach to security is that it is flexible and can easily express fine-grained security policies.

Using a Trusted Virtual Machine for Attestation

We propose making remote attestation more flexible and expressive by leveraging language-based techniques and virtual machines. The goal is to *attest program behavior, not a particular binary*. Our domain is platform-independent mobile code that runs on a virtual machine. Instead of a trusted operating system, we use a *trusted virtual machine*, or TrustedVM, to attest to remote parties various properties of code running within it. Software up to, and including, the virtual machine, still has to be trusted, and attested using the standard ``signed-hash'' method.

In this paper, we use the term virtual machine (VM) to mean a language-based virtual machine that executes some form of platform-independent code. The most widespread example of such a virtual machine is the Java Virtual Machine. It is important to differentiate this from virtual machine monitors (VMM) that virtualize a hardware architecture, and present an interface exactly like, or very similar to, raw hardware. Examples of VMMs are VMWare, and Disco [9].

Requests for remote attestation are now made to the TrustedVM. These requests ask for the enforcement or checking of specific security policies on code that is being run by the TrustedVM. Thus, what is enforced is not the execution of a particular binary, but security policies and other constraints specified by a remote party.

There are two key observations that enable our TrustedVM approach:

- Firstly, code expressed in a portable representation (e.g. Java bytecode) contains high-level information about the code -- such as its class hierarchy, method signatures, typing information etc. The presence of this high-level information, as well as the fact that code is expressed in a platform independent format, makes such code very amenable to program analysis. A trusted virtual machine can attest to high-level meta-information about a program, as well as the results of some code analysis done on a program.

- Secondly, code runs completely under the control of a virtual machine, and so its execution can be explicitly monitored. Thus, a trusted virtual machine can attest to dynamic properties regarding the execution of a program, or its inputs.

Some examples of properties that a trusted virtual machine can attest are:

- **TrustedVM attests properties of classes:** the remote party may require class A to subclass a well-known class B, or some interface C. This may be because extending B or C constraints the behavior of A in some way. For example, C may be a restricted interface for input-output operations, that disallows arbitrary network connections.

- **Attesting dynamic properties:** the program being attested runs under complete control of a TrustedVM. Thus, a TrustedVM can attest to dynamic properties. This includes the runtime state of the program and properties of the input of the program.

- **Attesting arbitrary properties:** A TrustedVM has the ability to run arbitrary analysis code(within the limits set by the security policy of the local host) on the program being attested on behalf of the remote party. Thus the remote party can test for a wide variety of properties by sending across code that does the appropriate analysis.

- **Attesting system properties:** a remote party can send across code that tests certain relevant system and virtual machine properties, and the TrustedVM can attest its results. For example, before running a distributed computing program (such as SETI@Home, or Folding@Home), the server may want to test the floating point behavior of the system and virtual machine by having the TrustedVM run a test suite of floating point programs.

Note that this sort of attestation is a much more fine-grained and semantically richer operation than signing the hash of an executable image -- we call this *semantic remote attestation*. What is now attested is not the presence of a particular binary executable, but relevant properties of a program. This has the effect of explicitly separating

two concerns that were earlier merged into one -- identity and behavior. Claims about code behavior are now made by the trusted virtual machine explicitly checking or deriving them.

A direct consequence of this is that now a variety of different implementations of some functionality will be able to function within our remote attestation framework -- as long as they satisfy the properties required of them. Cryptography now plays the part of binding this claim about code behavior to an entity which is qualified to make such claims -- a trusted virtual machine.

Advantages of Semantic Remote Attestation

Semantic remote attestation has a number of advantages over traditional remote attestation:

- It overcomes the most critical shortcoming of traditional remote attestation -- semantic attestation reasons about the behavior of the code, without tying that reasoning to a particular executable binary.
- It is dynamic in nature -- it can attest to various dynamic properties of a program, such as its runtime state at interesting program points, or input to the program. As opposed to traditional remote attestation, it is not one-time.
- It is flexible -- a TrustedVM can carry out arbitrary code analyses and attest its results.

Semantic remote attestation done in virtual machines with platform-independent code enables truly new functionality that was not possible before, because this sort of high-level attestation of program properties cannot be done using native code. Firstly, native code is too low-level, and does not have enough high-level structure and information to drive the sort of analysis our TrustedVM does. Secondly, some of the functionality of a TrustedVM (such as a server sending code to analyze or monitor the program being run) requires platform-independent code.

Semantic attestation can allow for the possibility that some of the properties required of a program may not be satisfied. In that case, the remote party can lower its expectations of how trustworthy the behavior of the program is likely to be, and proceed accordingly, rather than terminate the whole protocol altogether. Thus, properties that make an application trustworthy can now be thought of as *falling within a range, rather than one all-or-nothing attestation*. This has an important consequence for backward compatibility. Most common network protocols today (TCP, HTTP) assume a completely untrusted client. Using a flexible approach to attestation allows these untrusted protocols to be gradually endowed with trusted capabilities, and the ability to deal with clients of varying trustworthiness.

Semantic remote attestation also allows attestation without locking the client into a particular platform, or binary. By far the most scathing critiques of trusted computing have focused on the idea of remote attestation being used to lock consumers into a particular platform, thus extending monopoly control[6]. Semantic remote attestation, however, explicitly separates identity from behavior, and allows flexible attestation of code properties, while allowing inter-operation of a variety of implementations that satisfy these properties.

Implementation and Results

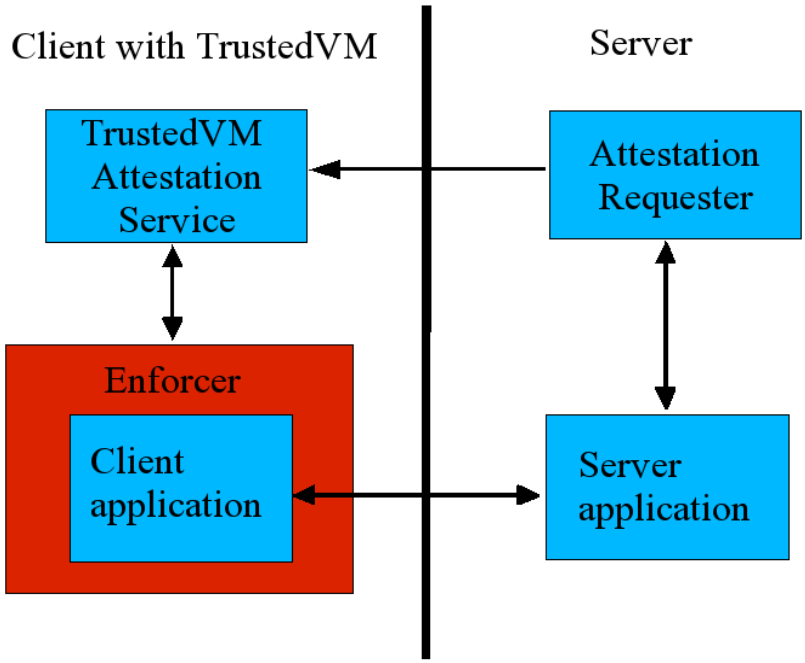

Figure 1: Top-level architecture for dynamic checking in a TrustedVM

Semantic remote attestation is a dynamic process, and attestation of properties can be done at various points during the lifetime of a distributed application. We refer to the machine running the TrustedVM as the client, and the machine which is interested in attesting programs on it as the server. The TrustedVM on a client machine runs an attestation service, whose job is to check or analyze the behavior of applications running under the TrustedVM, and answer attestation requests from other machines. The server has two channels of communication with the client: one with the client application, and another with the attestation service, with which the server communicates to find out about the behavior of the client application. This is shown in Figure 1. The communication between the server and attestation service must be secure and must guarantee integrity and authenticity. This can be done using various cryptographic protocols, such as SSL.

Semantic attestation certifies properties of programs running in a virtual machine. The virtual machine itself, however, runs on top of an operating system. To certify the layers of software up to, and including the virtual machine, still requires traditional remote attestation. This is done in the standard way using signed hashes.

The trusted base of a virtual machine also includes its standard libraries. Even if the virtual machine itself is trustworthy, an adversary could modify the standard system libraries for malicious ends -- for example, by substituting the standard security manager class with a weaker one. Thus, the operating system certifies both the virtual machine binary, and the libraries it uses.

At the time of this writing, we did not have access to trusted hardware. Thus, the certificate chains we emit are not rooted in a trusted hardware module, but in the operating system. We do not foresee any conceptual difficulties in interfacing to real trusted hardware, since it is a matter of extending the certificate chain.

We now explain the implementation of two applications that we implemented on our prototype TrustedVM.

A peer-to-peer protocol

To gain experience with building and using a TrustedVM for semantic attestation, we chose peer-to-peer networks as one example domain. In particular, we considered the Gnutella P2P protocol[2]. Peer-to-peer communication is particularly interesting as a trial application for remote attestation. It is inherently distributed in nature, and current protocols vest a tremendous amount of trust in clients for correct operation of the network. This has led to various security and policy violations, and has even been used to stage denial-of-service attacks

[26,13,8]. In the case of the Gnutella protocol, the reason for these security weaknesses is that peers believe each other without verification. For example, when given the result of a query(a *query hit* in Gnutella terminology), there is no guarantee that the given machine will actually have the content asked for -- the query result can be easily faked and is not checked. This can be exploited to mount a denial-of-service attack against a particular machine by simply flooding the network with query results all pointing to that one machine. The same holds for routing messages that announce which machines are part of the P2P network. These routing messages (called *pong* messages in Gnutella, because they are sent out as replies to ping messages to find out network topology) can also easily be faked.

Our implementation is based on the Java Virtual Machine. We used the Java Development Kit version 1.4.1 from Sun, running on an Intel Pentium-M 1.3 Ghz machine with 384 MB of RAM.

We modified the standard Java network socket class so that network communication over sockets could be captured and monitored. An API is exposed to do this. This captured traffic is sent to a protocol watcher which checks the protocol messages for conformance with a security policy, and informs the server accordingly. The Java bytecode for protocol watcher itself is sent by the server to the attestation service on the client, since it embodies the policy the server is interested in enforcing on the client.

Here we see the utility of using a machine-independent code representation -- the protocol watcher can be sent to and executed on various platforms, as long as they have a Java TrustedVM. Note that the protocol watcher is independent of the application. Various implementations of a protocol can be checked by same protocol manager.

We implemented a protocol very similar to Gnutella, though somewhat simpler in its wire format (to simplify debugging and parsing of network messages). The protocol watcher examined all outgoing messages from the client, and checked two properties:

- for *pong* messages, make sure that the IP address in the message is the same as that of the client.
- for *query hit* messages, make sure that the file that is mentioned in the query actually exists on the client.

These two properties were checked in particular because they can exploited to mount denial-of-service attacks, and can be easily spoofed -- they are completely unchecked.

We have not yet implemented a protocol checker for multi-hop messages, but this can be done using transitive certificate chains.

At the time of this writing, we are not aware of any applications that use functionality provided by trusted hardware, so we do not have a meaningful comparison benchmark. However, we can measure the overhead that protocol checkers impose on network latency. To measure this, we measured the time it took for 5000 subsequent ping/pong, and query/queryhit messages to travel between two machines. Both were on a 100 Mbps Ethernet network. This was compared with the same benchmark run without a protocol checker. Within experimental error, the timings were the same -- between 9.5 and 10.2 seconds for both cases. Increasing the number of iterations to 10,000 yielded similar results -- both cases took between 18.5 and 20.8 seconds.

This result is not a surprise since the checks are simple, and network round-trip-time dominates computation time. Even if the checks are more complex, in realistic situations they are only done periodically, and not in a tight loop. Thus, end-to-end performance of applications is still likely to show very little overhead.

We can distinguish between two kinds of applications that run on a TrustedVM. Legacy applications are those that do not explicitly make use of trusted functionality such as remote attestation. We just presented an example of a legacy untrusted application being attested by a TrustedVM, completely transparently. New applications that are written with trusted functionality in mind can use a much broader range of a TrustedVM's facilities.

We also unsuccessfully tried writing protocol watchers by using bytecode rewriting[12]. This turned out to be unsuitable for the task because load-time rewriting of system classes is not allowed in the JVM. Rewriting

bytecode at load-time requires a specialized class loader, that transforms the bytecode before handing it off to the virtual machine. However, all system classes (`java.lang.*`, `java.net.*` etc) must be loaded by the ``primitive'' system classloader. Systems such as SASI[14] and Naccio[15] inline reference monitors by rewriting Java bytecode. However, they transform the classfiles off-line, and not at load-time.

A distributed computing application

The previous application was an example of dynamic enforcement of security properties within a TrustedVM. A TrustedVM can also attest static properties, of both the system it is running on, as well as the code that it runs. The attestation requester sends across code for testing various properties, which the TrustedVM then executes. The results are signed, and sent back to the attestation requester (see Figure 2). The results of these tests can then affect further computation and communication between the two parties.

This is useful for distributed computing. There are a number of popular distributed computing projects, such as SETI@Home [20] and Folding@Home[27], that distribute work units out to a large number of clients. They face a common problem -- getting some assurances about the behavior and capabilities of their numerous clients. There is a complex trust relationship between the server and client, since the server expects the client to use a particular algorithm, compute answers within certain bounds, and not return maliciously crafted or incorrect answers. Currently, this problem is solved to some extent by measures such as redundancy and keeping track of client behavior over a period of time[22].

Using a semantic remote attestation framework can benefit a distributed computing application in the following ways:

- The server can test various properties of both the system, as well as the client, by having the TrustedVM execute tests for it. This would give the server a better idea of the capabilities of the platform as well as the client. This knowledge can be used both for giving the client suitable work units, and estimating how ``good'' its answers are likely to be.
- Testing for properties in this way makes the trust relationships between the client and server explicit. Now, instead of being implicit and being taken on trust, they are explicitly enforced or checked by the TrustedVM.

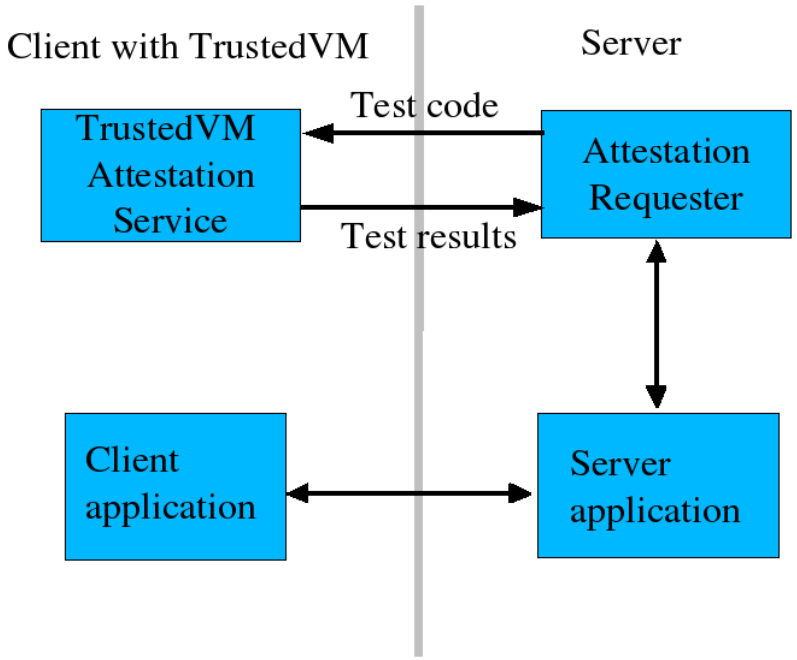

Figure 2: Top-level architecture for using test suites in a TrustedVM

To experiment with these concepts, we took an existing distributed computing project, examined its client-server trust relationships, and re-implemented it to run on our TrustedVM. We chose a distributed computing project that tries to find large Mersenne Primes -- the Great Internet Mersenne Prime Project, or GIMPS[4]. Mersenne primes are prime numbers of the form $2^n - 1$. Just like SETI@Home[20], GIMPS distributes its computation over a large number of clients on the Internet. We chose GIMPS because it divides the problem into three subproblems, each with different computational needs. They are:

1. First time primality check: This is the most computationally intensive of the three problems and is assigned to the fastest clients. It also requires double precision floating point support for doing a fast Fourier transform (FFT).
2. Double check assignment: It verifies the results of the first time primality check. The workload is smaller in this case and hence is assigned to slower clients. It also requires double precision floating point support.
3. Factoring work: This tries to eliminate a few test candidates by finding small factors using some common factoring algorithms. This reduces candidates for more expensive primality checks. Since this least expensive of the three this gets assigned to the slowest clients.

For a full treatment of the mathematical background of this problem see [4].

Many different implementations of the client exist [1]. The various clients differ from each other in the following ways

- The clients have subtle differences in the algorithms used.
- Some of the clients are highly specialized for a particular architecture and hence lose portability. Others have to be compiled for various architecture/OS combinations.
- The performance of different clients differ, even on the same platform due to implementation differences.
- The accuracy of the clients also differ hence results slightly vary. This is mainly because of the different algorithms used.
- More surprisingly the results differ for the same client on different platforms because of the differences in the underlying hardware.

Thus, GIMPS suffers from the same problems of distributed computing as pointed out above: its client-server trust relationships are implicit, and the server has very little information about the capabilities of the clients. A client is expected to behave reasonably because it is specific to a particular platform, and its behavior has been tested on that platform.

These problems can be solved by running this application on a TrustedVM. The server now explicitly tests the relevant capabilities of the client-side by asking the TrustedVM to execute a *test suite*, and return the attested results. This solution also has the added advantage of being portable across any range of architectures that implement a TrustedVM.

The two capabilities that are relevant for the GIMPS project are: floating point precision and the computational power. Our test suite has a compoenent for each.

The most computationally intensive routine in the prime factorization problem is computing a fast Fourier transform and its inverse. Hence to test for computation speed we execute a one-dimensional fast Fourier transform over small but typical data and time it. The result of this test helps the server to give the client an appropriate work unit.

For floating point precision we are interested in testing if the platform implements a double precision floating point operations and also if it complies with the IEEE 754 standard for floating point. In particular we used the Java port of the Elefunt test suite[3]. Elefunt[11] is a test suite to check for the compliance of floating point implementation of the various functions with the IEEE floating point standards. Since the FFT and the inverse FFT use the sine and the cosine functions the two tests that we are interested in are the ones that determine the

accuracy of these functions. Depending on results we determine whether the client is accurate enough to run the computations.

Thus, the server now has reliable information about the clients it is communicating with. Using this information, it can both vary the work units given out to clients, as well as make estimates about the accuracy of their answers. Moreover, the whole application is now portable, and does not depend on specific clients.

Related Work

To the best of our knowledge, there is no prior work that aims to make the mechanism of remote attestation more fine-grained, dynamic and platform-independent. However, the field of trusted computing has attracted a great deal of attention recently.

Garfinkel et. al.[17] have proposed the TerraVM[16] virtual machine monitor architecture to interface with underlying trusted hardware. Their architecture provides two VMM abstractions to software -- an open box VMM, and a closed box VMM. The open box VMM simply provides a legacy, untrusted interface. This allows old operating systems and software to run unmodified on it. The closed box VMM, however, provides an interface to underlying trusted hardware that new software can use. A number of such VMMs can execute on bare hardware. They are strongly isolated from each other, and have their own encrypted storage.

There are many important differences between TerraVM and our TrustedVM architecture. While TerraVM provides an interface just like real hardware, the TrustedVM exposes a much higher-level interface. It executes platform-independent code. Their goal is providing strong isolation between applications running in different VMMs, ours is to provide a better technique for remote attestation. Most importantly, TerraVM uses the standard ``signed-hash'' method of remote attestation. The authors acknowledge some of the shortcomings of standard remote attestation, and call for ``appropriate technical and legal protections ... to minimize abuse''.

The goal of TerraVM is similar to Microsoft's Palladium architecture. Palladium is said to have a high-assurance trusted microkernel running on hardware (called the *nexus*) that provides strong isolation between legacy untrusted and newer trusted applications, as well as among trusted applications. Unfortunately, to our knowledge there is no published technical documentation about Palladium, which makes it hard to make an in-depth comparison.

The Cerium system[10] aims to provide tamper-evident execution. The goal is to know if the remote execution of a program was carried out in an untampered manner. They use a physically tamper-resistant CPU block. This executes all code -- it is not a co-processor. Tamper-resistance is provided both at the hardware level, by physical means, and at the software-level, by encrypting all data (including cache data that is written back to main memory). Semantic remote attestation is orthogonal to this -- a TrustedVM could use Cerium as an underlying hardware architecture.

The Digital Distributed System Security Architecture[18] had many of the features of today's TCPA specification[5], including secure bootstrapping, and remote attestation of system software using signed hashes. The Aegis system[7] provided secure bootstrapping. Every layer of software from the hardware up was checked (using stored hashes) and control was passed to it only if it was untampered. Both these systems did not focus on improving remote attestation.

The Trusted Computing Group (TCG), has begun producing specifications of a hardware trusted module to be used in personal computers[5]. They call it a trusted platform module (TPM). Some models of IBM ThinkPad laptops contain a similar module. A commodity trusted computer will couple this trusted module with software that provides secure booting and strong isolation.

Discussion and Future Work

There are many avenues for further investigation. Protocol watchers and test suites, as presented here, are only two of many kinds of expressive attestation a TrustedVM is capable of.

A TrustedVM is capable of attesting the results of some static analysis done on code. However, there are not many static analyses of code for properties of interest to a remote server. Most static analyses and runtime enforcement policies so far have been geared towards protecting a host from malicious mobile code. Thus, the emphasis has been on type-safety, information-flow, and resource control and other safety issues. The emphasis is different for remote attestation. Servers want to know if the application is obeying some high-level semantic rules. One candidate for an analysis that may be of interest to servers is information flow[23]. Such an analysis would convince the server that a client is not leaking the results of some confidential computation, or data.

In our current implementation, the policy a server wants enforced is embodied in the protocol watcher or test suite. We would like to develop a systematic language for expressing remote attestation requests. With ``signed-hash'' attestation, this was not an issue. But a TrustedVM provides a broad range of fine-grained attestation capabilities, and a language is probably the right tool to make full use of them.

We would also like to gain experience with developing more applications that use the functionality provided by a TrustedVM. Distributed firewalls[19] implement a network traffic policy at the end-points of a network, rather than at one single point. This way of distributing trust seems like a good match for implementation on a TrustedVM.

The ability to communicate to a server what particular property of a program could not be certified can be very useful. Using TrustedVMs, this information can be communicated, and the server can get detailed information about what desired properties are not present in a client program. It can then make an informed decision about either decreasing its trust in this particular instantiation of a protocol, or stopping altogether. Thus, the server gains some dynamic feedback about the trustworthiness of its clients. We believe this property can be fruitfully exploited to "port" a variety of untrusted network protocols (TCP, HTTP etc) to a trusted computing framework in a gradual manner, and yet have various implementation of them inter-operate. This is in stark contrast to the all-or-nothing model that standard ``signed-hash'' remote attestation provides -- attestation either passes or fails -- there is no gradation. This would also provide a gentler upgrade path for applications as trusted hardware becomes increasingly available in the market.

Conclusion

Standard ways of doing remote attestation are based purely on cryptography, and suffer from many critical shortcomings -- they are static, inexpressive, inflexible and do not scale. Most importantly, they do not speak about program behavior -- they can only attest to the presence of a particular binary. It is possible for an attested binary to have bugs and not obey the security policy a server was expecting it to. Remote attestation is hard to scale up to a flood of software patches and upgrades. It also does not accommodate a varied, homogeneous computing environment very well.

We have introduced a novel technique for remote attestation based on language-based virtual machines. The core idea behind our technique, called semantic remote attestation, is to use a language-based virtual machine that executes a form of platform-independent code. Software up to and including this virtual machine is trusted. However, the virtual machine can certify various properties of code running under it by explicitly deriving or enforcing them. This can be done in many ways, such as observing the execution of programs running in a VM, or analyzing the code before execution. This is particularly easy to do with high-level platform-independent code that has a lot of information about the structure and properties of code.

The fact that ``trusted computing'', and its core technique, standard remote attestation, can lock consumers into a particular program or platform has been a very widely expressed fear. A key advantage of our approach is that reasoning about the behavior of a program is now not tied to a particular binary. Semantic remote attestation

checks for program properties, and works with different implementation of the same program as long as they satisfy the properties required of them.

To validate our ideas we have implemented a prototype TrustedVM by modifying a Java virtual machine to observe the behavior of network protocols. We have used this prototype to check the untrusted behavior of a Gnutella-like peer-to-peer protocol. Specifically, we check that messages about network topology and query results are not spoofed. Our measurements show that the overheads of checking are negligible, at least for the simple checking this particular application needs. However, even the few simple checks we do are sufficient to overcome some of the most well-known weaknesses in peer-to-peer protocols.

Trusted computing introduced the concept of remotely supervised execution - the idea that a remote server will be able to monitor as well as change the execution of a program on a client machine. Remote attestation is the key to doing this. However, current architectures for attestation are able to implement this idea in only a very limited way. Semantic remote attestation takes this idea to its logical conclusion -- to have fine-grained, dynamic control over the monitoring and execution of an application.

Bibliography

1

GIMPS source code timings page.
https://hogranch.com/mayer/gimps_timings.html.

2

The gnutella protocol specification.
https://www9.limewire.com/developer/gnutella_protocol_0.4.pdf.

3

Java port of elefunt.
https://www.math.utah.edu/ beebe/software/java/.

4

The Great Internet Prime Mersenne Search: GIMPS.
https://www.mersenne.org/.

5

T. C. P. Alliance.
TCPA PC-specific implementation specification (https://www.trustedcomputing.org), May 2001.

6

R. Anderson.
Cryptography and competition policy -- issues with trusted computing.
In *Workshop on Economics and Information Security*, May 2003.

7

W. Arbaugh, D. Farber, and J. Smith.
A secure and reliable bootstrap architecture.
In *IEEE Symposium on Security and Privacy*, 1997.

8

S. Bellovin.
Security aspects of napster and gnutella.
In *USENIX Security Symposium*, Aug. 2000.

9

E. Bugnion, S. Devine, K. Govil, and M. Rosenblum.
Disco: Running commodity operating systems on scalable multiprocessors.
ACM Transactions on Computer Systems, 15(4):412-447, 1997.

10

B. Chen and R. Morris.
Certifying program execution with secure processors.
In *USENIX HotOS Workshop*, May 2003.

11

W. Cody and W. Waite.
Software manual for the elementary functions.
Prentice Hall, 1980.

12

G. Cohen, J. Chase, and D. Kaminsky.
Automatic program transformation with JOIE.
In *1998 USENIX Annual Technical Symposium*, pages 167-178, 1998.

13

Z.-Y. Demetris.
Exploiting the security weakneses of the gnutella protocol, Mar. 2002.

14

Erlingsson and Schneider.
SASI enforcement of security policies: A retrospective.
In *NSPW: New Security Paradigms Workshop*. ACM Press, 2000.

15

D. Evans and A. Twyman.
Flexible policy-directed code safety.
In *IEEE Symposium on Security and Privacy*, pages 32-45, 1999.

16

T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh.
Terra: A virtual machine-based platform for trusted computing.
In *Proceedings of the 19th Symposium on Operating System Principles(SOSP 2003)*, October 2003.

17

T. Garfinkel, M. Rosenblum, and D. Boneh.
Flexible os support and applications for trusted computing.
In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, May 2003.

18

M. Gasser, A. Goldstein, C. Kaufman, and B. Lampson.
The digital distributed system security architecture.
In *Proc. 12th NIST-NCSC National Computer Security Conference*, pages 305-319, 1989.

19

S. Ioannidis, A. D. Keromytis, S. M. Bellovin, and J. M. Smith.
Implementing a distributed firewall.
In *ACM Conference on Computer and Communications Security*, pages 190-199, 2000.

20

E. Korpela, D. Werthimer, D. Anderson, J. Cobb, and M. Lebofsky.
Seti@home -- massively distributed computing for SETI.
Computing Science and Engineering, 3(1), 2001.

21

D. Kozen.
Language-based security.
In *Mathematical Foundations of Computer Science*, pages 284-298, 1999.

22

D. Molnar.
The SETI@home problem.
https://www.acm.org/crossroads/columns/
onpatrol/september2000.html, Sept. 2000.

23

A. C. Myers.
JFlow: Practical mostly-static information flow control.
In *Symposium on Principles of Programming Languages*, pages 228-241, 1999.

24

G. C. Necula.
A scalable architecture for proof-carrying code.
Lecture Notes in Computer Science, 2024:21+, 2001.

25

F. B. Schneider, G. Morrisett, and R. Harper.
A language-based approach to security.
Lecture Notes in Computer Science, 2000:86-??, 2001.

26

D. Wallach.
A survey of peer-to-peer security issues.
In *International Symposium on Software Security*, Nov. 2002.

27

B. Zagrovic, E. Sorin, and V. Pande.
Beta hairpin folding simulations in atomistic detail using an implicit solvent model.
Journal of Molecular Biology, 317(4), 2002.

This paper was originally published in the Proceedings of the 3rd Virtual Machine Research and Technology Symposium,
May 6-7, 2004, San Jose, CA, USA
Last changed: 29 Aprl 2004 aw

VM '04 Technical Program
VM '04 Home
USENIX home