BIND: A Fine-grained Attestation Service for Secure Distributed Systems*

Elaine Shi
Carnegie Mellon University
rshi@cmu.edu

Adrian Perrig
Carnegie Mellon University
perrig@cmu.edu

Leendert Van Doorn
IBM T.J. Watson Research Center
leendert@watson.ibm.com

Abstract

In this paper, we propose BIND (Binding Instructions aNd Data),[1] *a fine-grained attestation service for securing distributed systems. Code attestation has recently received considerable attention in trusted computing. However, current code attestation technology is relatively immature. First, due to the great variability in software versions and configurations, verification of the hash is difficult. Second, the time-of-use and time-of-attestation discrepancy remains to be addressed, since the code may be correct at the time of the attestation, but it may be compromised by the time of use. The goal of BIND is to address these issues and make code attestation more usable in securing distributed systems. BIND offers the following properties: 1) BIND performs fine-grained attestation. Instead of attesting to the entire memory content, BIND attests only to the piece of code we are concerned about. This greatly simplifies verification. 2) BIND narrows the gap between time-of-attestation and time-of-use. BIND measures a piece of code immediately before it is executed and uses a sand-boxing mechanism to protect the execution of the attested code. 3) BIND ties the code attestation with the data that the code produces, such that we can pinpoint what code has been run to generate that data. In addition, by incorporating the verification of input data integrity into the attestation, BIND offers transitive integrity verification, i.e., through one signature, we can vouch for the entire chain of processes that have performed transformations over a piece of data. BIND offers a general solution toward establishing a trusted environment for distributed system designers.*

*This research was supported in part by CyLab at Carnegie Mellon under grant DAAD19-02-1-0389 from the Army Research Office, and grant CAREER CNS-0347807 from NSF, and by gifts from IBM and Cisco. The views and conclusions contained here are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either express or implied, of ARO, Carnegie Mellon University, IBM, Cisco, NSF, or the U.S. Government or any of its agencies.

[1]The term BIND is also used in Domain Name Service (DNS) terminology to stand for the Berkeley Internet Name Daemon. Here we use it in a different context.

1 Introduction

Securing distributed systems continues to be an important research challenge. One hard problem in securing a distributed system arises from the fact that a remote software platform may be compromised and running malicious code. In particular, a compromised platform may exhibit arbitrarily malicious behavior. Such attacks are referred to as Byzantine attacks [12] in the literature. The task of remote code attestation then is to identify what software is running on a remote platform and to detect a corrupted participant.

The Trusted Computing Group (TCG, formerly known as TCPA) [40] and the Next Generation Secure Computing Base (NGSCB, formerly known as Palladium) [30] propose to use a secure coprocessor (Trusted Platform Module) to bootstrap trust during system initialization. These approaches compute a hash value of a loaded program before execution starts, the hash value can later be used by a remote party to identify the system's loaded code and configuration. Meanwhile, operating system architectures have been built to incorporate this approach [35, 36].

Previously proposed TCG-style attestation mechanisms have a *coarse granularity*, they verify the entire operating system and loaded applications. However, operating systems often contain numerous modules that depend on the installed hardware, as well as different versions of the same software, or the same version compiled with different compiler settings, or patched with different patches. Even tiny differences in the execution code result in a different hash value. Thus, such coarse-grained attestation makes remote verification very difficult. The Terra Virtual Machine Monitor [16] alleviates this problem by decomposing attestable entities into fixed-sized blocks, and computing a separate hash over each block. Apart from being coarse-grained, TCG-style attestation only provides load-time guarantees, as the attestation in TCG only reflects the memory state right after the program is loaded. However, it may well be that the software gets compromised at run time (e.g., buffer overflows, format string vulnerabilities), which load-time attestation cannot possibly detect.

In another line of work, researchers have proposed Copilot [32], a run-time memory attestation mechanism. Here, extra hardware periodically computes a hash of the memory to detect deviations from the expected contents, which would indicate malicious code. However, Copilot checks memory periodically, which may miss a short-lived intrusion. Meanwhile, Copilot also verifies memory at a coarse granularity much the same way as TCG-style attestation.

In this paper, our motivation is two-fold. First, we seek to answer the question: how can code attestation aid us in designing a distributed system? Second, we make an effort at addressing the above mentioned issues regarding current code attestation technology. We present the following contributions: (1) We propose *fine-grained attestation*, where we attest only to the critical piece of code involved in producing a certain output, instead of computing the checksum across the entire software system. We achieve this through an attestation annotation mechanism. We allow the programmer to identify and annotate the beginning and the end of this critical piece of code; and every time this piece of code is executed, our attestation service will be invoked. (2) We narrow the gap between time-of-attestation and time-of-use. We attest to the critical piece of code immediately before it is executed, and we use a sand-boxing mechanism to protect the execution of the critical code. So even though the rest of the software system may be compromised, it cannot tamper with the execution of the critical code. (3) We propose to tightly *bind code integrity with data integrity*. In BIND (Binding Instructions aNd Data), an integrity proof for a piece of code is cryptographically attached to the data it has produced. This allows us to pinpoint what code has been run to produce a certain piece of data. (4) We design a construction where we incorporate the integrity proof of the input data into the integrity statement of the code and output data. This construction enables us to achieve *transitive integrity verification* with constant overhead, i.e., we only need to verify one signature to guarantee the integrity of the entire chain of processes that transformed the data.

We explain how to build BIND using current TCG and microprocessor technology. To illustrate how BIND can be used as a general solution toward establishing trust in real-world distributed systems, we study a distributed computation application and the BGP routing protocol as examples.

2 Distributed System Security and Role of Attestation

In this section, we seek to answer the question: how is attestation useful in securing real distributed systems? We begin by establishing a conceptual model for distributed systems. We then examine the threat model and based on that, we pinpoint the role of attestation in dealing with these threats.

2.1 A Conceptual Model for Distributed Systems

We first propose a conceptual model for distributed systems. We consider a distributed system to be comprised of *processes*, *data* and *intermediaries*.

Process The process is a producer and consumer of data. It represents protocol logic, i.e., the computations we perform on data. In the context of our discussion, a process is a piece of software code which is to be attested. Note that we borrow the term process from the operating system literature, but use it in a different way.

Data Data represents information exchanged between the processes. In our model, we distinguish between *primitive* and *derived* data. Primitive data are external inputs to the distributed system, whereas derived data are the output of some process, i.e., derived data are generated by applying protocol logic over some input data. In real-world systems, primitive data usually exist in the form of human input, configuration files, external timing, etc.

Intermediary The intermediary represents the medium over which data is communicated from/to a process, including where data is stored outside the process. In reality, the role of the intermediary is acted by the network that forwards the data between hosts, the operating system that dispatches the data to the process concerned. We also model the local storage system as an intermediary, including the hard drive or any part of RAM outside the process.

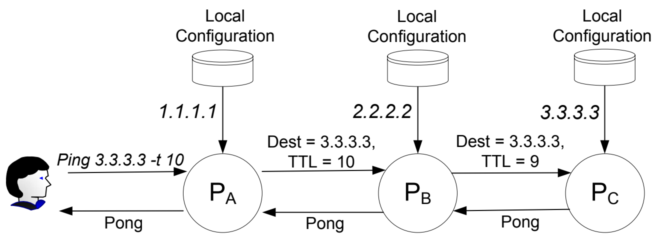

Figure 1. Simple Ping: The Conceptual Model for Distributed Protocols

To demonstrate the generality of this model, we consider a simple ping-pong protocol (ICMP echo request and reply) as an example (see Figure 1). We first illustrate the protocol in descriptive language, then explain how this protocol fits into our model by identifying the processes, data, and intermediaries. In this example, three hosts, A, B, and C, each run a ping-pong process, namely P_A, P_B, and P_C. In reality, the ICMP protocol is part of the TCP/IP stack. However, we are interested only in the critical piece of code that performs transformation on ICMP data packets, not the entire TCP/IP stack. Here our process is defined to be that piece of critical code. At host A, the user issues a ping command with destination=C, and TTL=10. The command is dispatched to process P_A, that sends it onto P_B. P_B decrements the TTL by 1 and forwards it to P_C. P_C discovers that it is the destination, and it replies with an echo message. The echo is then passed by P_B to P_A. Finally the result is returned to the user. Applying our conceptual model to the simple ping-pong protocol, the processes are P_A, P_B, and P_C which run on hosts A, B, and C, respectively. The data includes the user command, the ping and pong messages as well as the local IP address of each host. Among them, the human input command and the local IP addresses are primitive data; the ping and pong messages are the derived data. To reach a process, the data passes through the network and the operating system on the a host. Therefore, though they are not explicitly shown in the figure, the network and the OS constitute the intermediaries in the simple ping-pong protocol.

Our notion of process and data is related but not equivalent to the concepts of code section and data section at the binary executable level. For the ease of hash verification, we consider memory content that is immutable upon entry to the critical code to be part of the process; this includes the code section[2] and possibly parts of the data section. Memory contents that are mutable upon entry to the process is treated as data.

2.2 Security Problems in a Distributed System

Our goal is to eliminate *software attacks*, i.e., we assume that a process is faulty because it is running some malicious software code. We do not consider faults introduced by malicious hardware such as a malicious processing unit.

In this context, we consider the *Byzantine attacker* model [23] where a corrupted process or intermediary can exhibit arbitrarily malicious behavior.

Malicious Intermediary and the Data Misuse Attack A malicious intermediary may arbitrarily alter and inject protocol data. To prevent such attacks, we can employ cryptographic constructions such as message authentication codes or digital signatures. Through the use of cryptography, the intermediary is rendered an outsider to the distributed protocol, for it does not possess the correct cryptographic keys to modify or inject authenticated protocol data.

In spite of the cryptography mechanisms, a malicious intermediary is capable of the *data misuse* attack, where it uses authenticated protocol data in a malicious way. For instance, a malicious intermediary can perform a data suppression attack by refusing to forward any data. She can perform a replay attack by replaying data that have been authenticated but are outdated. The malicious intermediary can also perform a *substitution attack* where he has two pieces of authenticated data D_A and D_B; yet instead of feeding the correct piece D_A to some process, he feeds D_B instead.

Malicious Process and the Data Falsification Attack In a highly adversarial environment, an attacker may corrupt one or more processes in the system. A malicious process is capable of injecting bogus data into the distributed system. We refer to this attack as the *data falsification* attack. Traditional cryptography does not defend against data falsification attacks by a malicious process, for the malicious process has the correct cryptographic keys to disguise itself as a legitimate participant. For instance, in our simple ping-pong protocol, a corrupted process can arbitrarily modify the TTL field in a ping message. Thus, if the simple ping-pong protocol is intended for estimating network distances, then the estimate will fail due to a TTL modification attack.

2.3 Attestation Design Considerations

Fine-grained v.s. Coarse-grained Attestation We propose the concept of granularity in attestation. At one end of the spectrum, we can do coarse-grained attestation over entire software platforms; on the other hand, we can do fine-grained attestation by attesting to just a critical piece of code, which becomes the notion of a process in our conceptual model. The reason why we make an explicit distinction between the process and the intermediary is to enable the notion of granularity in attestation; whereas in reality, both the process and the intermediary can exist in the form of software code on some computing device. In our conceptual model, the boundary between the process and the intermediary represents the boundary between what code is being attested to and what is not.

The following arguments speak in favor of fine-grained attestation: 1) As we describe in Section 1, fine-grained attestation simplifies hash verification. With fine-grained attestation, we can also perform software upgrades more easily, since the expected hash for each process can be updated independently. 2) Fine-grained attestation allows the distributed system architect to focus on the security of a critical module by singling it out from a potentially complicated system.

On the other hand, with fine-grained attestation, software code omitted from the attestation is called an intermediary in our conceptual model. Fine-grained attestation does not address intermediary attacks. In Section 2.4, we examine potential mechanisms that can be used to deal with intermediary attacks in this framework.

Desired Properties Regardless of the granularity of attestation, the following is a list of properties we would ultimately like to achieve, not saying that BIND achieves all of them. 1) Ideally our attestation service should be free of all software attacks; 2) Apart from making sure that the process is correct at load-time, we would like to achieve instantaneous detection of run-time compromises as well; 3) The attestation service needs to be efficient.

The BIND Endeavor BIND offers a fine-grained attestation service. Though BIND does not strictly guarantee some of the above-mentioned properties, our goal is to see how far we can push our limits under the constraints of currently available microprocessor and trusted computing technology. In Section 3, we detail the properties BIND achieves.

2.4 Using Fine-grained Attestation to Secure Distributed Systems

Before getting into the details of BIND, we address the following question: suppose we were able to build a *per-*

[2]This assumes that no variable data exists in the code section.

fect attestation service satisfying all of the desired properties mentioned above, how would it aid us in designing secure distributed systems?

First, if such an attestation service can be built, a corrupted process immediately gets detected, and legitimate processes can thereupon eliminate it from the distributed system by refusing to accept any data they produce. Thus the system architect no longer has to deal with Byzantine faults on the part on the part of processes. Instead he only needs to consider a simple *fail-stop* failure mode. By contrast, to deal with data falsification attacks by malicious processes, traditional Byzantine fault tolerant algorithms create replicas of each process, and use variants of majority voting algorithms to eliminate corrupted data. However, creating replicas is expensive and hence often impractical. Meanwhile, though researchers have endeavored to improve the performance of majority voting algorithms, they still involve high communication overhead.

On the other hand, a fine-grained attestation service does not deal with data misuse attacks from malicious intermediaries. In fact, similar to malicious delay and replay attacks, probabilistic delays, duplicates, and out-of-order message transmissions may just be inherent properties of the underlying network. The system architect should embed logic in the process itself to deal with such attacks. Many approaches have been proposed to deal with data misuse attacks by malicious intermediaries. For instance, we often use timestamps and sequence numbers to provide resilience to message delay and replay attacks. To counter the substitution attack (See Section 2.2 for the definition of the substitution attack), a process can tag each piece of data with a specification on how it is to be used, before having it signed by the attestation service. Now even if the malicious intermediary can cheat and substitute the input, the process can easily detect it by examining the specification.

In conclusion, we examined the desired properties we would ultimately like to achieve out of attestation. We argue that if we were able to build a perfect attestation service with the desired properties, we would be able to deal with malicious processes in a distributed system by reducing Byzantine faults to fail-stop failures. On the other hand, as with all fine-grained attestation, it is up to the protocol semantics to rule out intermediary attacks.

3 BIND Overview

In this section, we give a design overview of BIND. We begin by defining what it means for a process and data to be genuine under the conceptual model defined in the previous section. Then we describe the techniques BIND uses to ensure process and data integrity. We also explain the interface BIND exposes to the programmer, and state the properties BIND guarantees.

3.1 Process and Data Integrity

Defining Process and Data Integrity Under the conceptual model proposed in Section 2, we define the notion of integrity for process and data. Since BIND does not address malicious intermediary attacks, we are not considering such attacks in the following definition.

The integrity of a process is a relatively simple notion. Since a process is essentially a piece of code, the integrity of a process is defined by the genuineness of the code.

Integrity of data (or that the data is *genuine*) can be inductively defined as:

1. Either the data is primitive and is genuine;
2. Or the data is derived by running a genuine process over genuine data inputs.

Ensuring the Integrity of Primitive Data In practice, the meaning of primitive data integrity is application specific. Therefore, BIND cannot enforce primitive data integrity by itself. Instead we need a mechanism external to BIND to ensure primitive data integrity. Potential techniques include:
Semantic Check: We can embed logic in the process itself to perform semantic checks on primitive data, e.g., a distributed scientific computing application checks whether an input matrix is well-formed.
Certificates: We may have a central trusted authority to sign certificates for primitive data, e.g., in a secure version of Domain Name Service (DNS), hostname to IP address mappings are secured through digital signatures from a trusted DNS authority.
Trusted Path: For user input data, we may use trusted path mechanisms to ensure that the data came from an authenticated user.

Ensuring the Integrity of Process and Derived Data BIND offers a mechanism to ensure the integrity of process and derived data. BIND produces an *authenticator* for every piece of data a process generates. The authenticator states the fact that the data is generated by running a legitimate process on genuine data inputs. We assume that the authenticator is attached to the data throughout its lifetime, i.e., when data is sent over the network, or stored and fetched from local untrusted storage, etc.

3.2 BIND Interface

We assume the process, i.e., the critical code, is stored in contiguous memory regions. Figure 2 depicts the interface BIND offers to the programmer. At the beginning of the process is an ATTESTATION_INIT call to initiate an attestation phase. The parameters passed along with ATTESTATION_INIT include the memory addresses of the process's input data and the size of the process code. On receiving the ATTESTATION_INIT request, BIND first verifies the authenticator on the input data. If there are multiple instances of input data, BIND verifies each one of them. BIND then hashes the process code along with the input data addresses. To make sure that what is hashed is what is executed, BIND sets up an isolated environment for the process to execute. If the above steps are successful, BIND

```
1. interface ATTESTATION_INIT
        in input data memory addresses,
        in size of process code,
        out success indicator;
2. interface ATTESTATION_COMPLETE
        in output data memory addresses,
        out authenticator.
```

Figure 2. Interface of the attestation service to the process

yields control to the process with a success indicator, and from the this point on until an ATTESTATION_COMPLETE command is issued, the process is ensured to execute in a protected environment safe from tampering. At the end of the process is an ATTESTATION_COMPLETE command with output data addresses as parameters. In response to the ATTESTATION_COMPLETE command, BIND computes an authentication tag over both the output data and a hash of the process code. This authentication tag binds the output data with the code that has generated it. Then BIND undoes the protections it has set up for the process and returns the authenticator to the process.

3.3 BIND Properties

We now discuss the salient ideas underlying BIND:

Fine-grained Attestation When designing a distributed protocol, we are often concerned about the trustworthiness of a remote participant. In particular, we care about the process, the critical piece of code that performs transformations over protocol data.

As Figure 2 shows, BIND allows a programmer to identify the process and annotate the beginning and end of the process with an ATTESTATION_INIT and ATTESTATION_COMPLETE call. In this way, every time the process executes, BIND will be invoked to attest to its integrity. Through this attestation annotation mechanism, BIND offers fine-grained attestation by attesting only to the process but not any uncritical code. This simplifies hash verification.

Binding Process and Data Integrity From the definition of data integrity, it is evident that the integrity of derived data builds on the integrity of its generating process, since a corrupted process may have performed arbitrary operations on the data it produces. On the other hand, it is not meaningful to speak of the integrity of a process unless we actually use the process. A process contributes to the system by operating on input data and thus generating new data. Ultimately, our concern over the integrity of a process stems from the concern whether a piece of data has been generated by a genuine process. Therefore, in a distributed system, the integrity of processes and that of data are inseparable from each other, and it means little to speak of either alone.

BIND embeds the integrity proof of a process in the integrity proof of the data it has generated. This binding also arises from our endeavor to narrow the time-of-use and time-of-attestation discrepancy which current code attestation technology has not resolved. We seek to prove what process code has been run to generate a piece of data, instead of what the process code is at an arbitrary point of time throughout the life span of the system.

BIND hashes the process code immediately before it is going to be executed. Then it sets up protection mechanisms so that the process will be executed in a sandbox safe from tampering. After the process has completed execution, BIND signs the hash of the code along with the output data the process has produced. In this way, BIND proves what code has been executed to generate the data.

Transitive Integrity Verification The inductive definition of data integrity naturally reflects the existence of a chain-of-trust. To ensure the integrity of some derived data, we need to ensure the integrity of its generating process as well as the integrity of the input data to the process; and to ensure the integrity of the input data, it is necessary to ensure the integrity of the previous hop process (i.e., the process that generated the input data), as well as the input to the previous hop process, etc.

BIND achieves transitive integrity verification with $O(1)$ overhead. A BIND authenticator not only vouches for the most recent process that has operated on the data, but the entire chain of prior processes as well. To achieve this, BIND always verifies the authenticator on the input data when attesting to the process and its output data. A similar inductive construction is used by Arbaugh et al. [8, 9] to achieve a secure bootstrap mechanism. While they verify the integrity of the next layer software before loading it, BIND verifies the integrity of previous hop process/data before using it; and we assume an external mechanism exists to verify primitive data integrity too.

Efficient BIND utilizes TPM's hardware-based cryptography engine to enable fast cryptography computations needed for attestation. Meanwhile, because BIND supports transitive integrity verification, it is efficient to verify the integrity of a piece of data, even when the data has traversed multiple processes.

Discussion BIND does not deal with software vulnerabilities in the attested code itself. To exploit a software vulnerability in the process code, a malicious attacker can supply the process with malicious input data, so that the process may be compromised during its own execution. At first, it may seem that if we hash the process both before and after its execution, we can eliminate attacks that exploit bugs inside the process. In fact, this does not provide a fundamental solution to the problem, for a sophisticated attacker can escape detection by restoring the correct code right before reaching ATTESTATION_COMPLETE. Though BIND fails to detect a software vulnerability inside the process, the adversary has a restricted attack interface. First, since the process code is small in size, it is easier to manage and verify; hence it arguably has few vulnerabilities. It may even be possible to perform security evaluation on such a small piece of code to prove that it is free of vulnerabilities. Moreover, to exploit a vulnerability in the process code, the attacker has to feed it with carefully-constructed malicious inputs. However, BIND verifies the integrity of any derived data input to a process, which drastically reduces the vulnerability surface.

The microprocessor and TCG technology we use for BIND offers a hybrid hardware and software solution toward overcoming software-based attacks. BIND runs in the core privilege ring of a modern TCG-aware processor. An adversary may attempt to break BIND through exploiting a vulnerability in the BIND code. However, such an attacker has a restricted attack interface. First, BIND relies heavily on protection mechanisms provided by the hardware microarchitecture. Second, the BIND code is small in size and complexity such that a security evaluation may be possible to prove that it is free of bugs. Third, BIND offers a minimal interface to the operating system and applications.

Denial-of-Service (DoS) attackers may be able to compromise the availability of BIND by repeatedly sending attestation requests to BIND. However, such DoS attacks can only hurt the adversary, for unless BIND is available, data produced by that host will not be accepted by legitimate processes in the system. The compromised host not only fails to participate in the system, but also is going to be detected. Therefore we do not consider DoS attacks in the design of BIND.

4 Detailed Design of BIND

This section explores how to instantiate the trusted entity and the attestation service. First we discuss the various implications of placing trust on different software/hardware components; next we present a design based on state-of-the-art processor isolation technology with TCG/TPM support.

4.1 Trust Assumptions

Placement of Trust and Implications We consider how to instantiate a trusted entity on a host. Our options include: trusting the operating system, trusting the hardware, or trusting the Secure Kernel (SK). This section discusses the security implications of each solution.

Operating System: One option is to integrate the attestation service into the operating system or run it as an application. Here we must assume that we trust the protection and fault isolation mechanisms offered by the OS. This is a relatively weak security practice, since numerous kernel vulnerabilities exist, and once an attacker has successfully exploited a buffer overflow [13] or a format string [4] vulnerability, it can inject arbitrary code to be run at the kernel privilege level. Even though researchers have developed sophisticated software verification techniques, e.g., static analysis [15], to detect software vulnerabilities, these techniques are not perfect and may miss unknown vulnerabilities.

Hardware: Since our goal is to overcome software-based attacks, we would like to push the trust onto hardware. The general belief is that compromising hardware is much more difficult than compromising software for the following reasons. First, subverting a piece of hardware usually requires physical vicinity to the targeted hardware; second, tamper resistant hardware technology continues to mature. The drawbacks of using trusted hardware include its relatively high cost for design, manufacture, and deployment, and inflexibility when it comes to customization or upgrading.

Secure Kernel: The Secure Kernel (SK) is a new mode on modern processors such as AMD's Secure Execution Mode (SEM) [5]. In Section 6, we provide more information on SEM-like technology. The SK can be viewed as a middle ground between pure software and pure hardware solutions. First the SK is by nature a piece of software code, and suffers the same software vulnerability problem as the OS kernel. Yet, since the SK is usually a compact piece of code, and offers a minimal interface to the OS, it is far more manageable than the OS kernel, and it may even be practical to run software verification on the SK code. One of the greatest challenges of software verification is complexity. With a small piece of code such as the SK, it may be possible to enumerate all possible states and verify their correctness. The SK runs at the core of the privilege rings. It can utilize several new hardware protection features to protect itself and other sensitive system resources, including memory, IO, DMA and system registers. SEM also provides a secure kernel intercept mechanism to serve as the single interface between the OS and SK. With the reasons stated above, it is reasonable to assume the security of the SK code, and that it remains intact and trustworthy throughout its lifetime. In Section 6, we also review other microprocessor technologies that can be used in place of SEM.

BIND Trust Assumptions In designing BIND, we assume that every participating platform is equipped with a TPM chip and a modern secure processor similar to AMD's SEM chip. Trust thus builds on the TPM as well as the Secure Kernel. In Section 4.1, we discuss the security implications of trusting the SK. Meanwhile, since the TPM is by nature a passive chip in the TCG context, the SK serves to bridge the gap between the TPM and the untrusted OS/application code.

We rely on secure boot and load-time attestation [36] to establish trust on the integrity of the SK.[3] One responsibility of secure boot is to set up appropriate hardware protection mechanisms on the processor. Here we need to enforce write protection on the SK memory space, so that it cannot be altered by OS/application code nor through DMA; we also need to establish a secure channel between the SK and the TPM. To this end, we can allocate an exclusive TPM

[3]For the purpose of this paper we assume a static root of trust. Trust in the SK can also be established through the dynamic root of trust mechanism [40].

locality for the SK and protect the corresponding memory-mapped I/O space from OS access. Through load-time attestation, we can prove the integrity of the SK and hardware protection environment at boot time. And henceforth we shall rely on the security of the SK and the protection environment during run-time.

One design consideration for the SK is the trade-off between its simplicity and functionality. At one end of the spectrum, we can build an SK as sophisticated as a Virtual Machine Monitor (VMM) [10, 16] that has partial control over OS resources such as page-tables and descriptor tables, and build the attestation service inside the VMM. Yet as the SK code grows in complexity and size, its manageability and verifiability diminish, and hence it becomes prone to software vulnerability exploits. By contrast, one may seek to minimize functionality at the SK to trade for security and this is what we do in our design that follows.

4.2 BIND Design

Attestation Overview Figure 3 depicts the sequence of operations that happen during one attestation phase.

After the process code is invoked, the first operation is to call the BIND by raising a Secure Kernel Intercept (SKI). Upon receiving an attestation request, the Secure Kernel first verifies the authenticator on the input data to the process. The address range as to where this data is stored is provided by the requester, and to ensure that the requester does not lie about the input data, these memory addresses will be incorporated into the measurement. The signature on the input data is validated through the public signing key of the TPM that signed the input data. The SK also needs to verify the hash value. Later in this section, we discuss how to enable different software versions and upgrades. If either the signature or the hash verification fails, the SK returns to the process with a failure indicator. Else the SK hashes the process code as well as the input data addresses, and before yielding control to the process, the SK sets up certain protections to ensure that the process code is executed in a safe and untampered environment. We shall what specific protections we need later in this section.

After control is handed back to the process code, the process performs computation on the incoming data, and at the very end, requests the output data to be signed by the SK along with a hash of the code itself. On receiving the the ATTESTATION_COMPLETE request, the SK first ensures that the ATTESTATION_COMPLETE call comes from a process being attested, i.e., one that performed ATTESTATION_INIT. This is to prevent a malicious OS kernel from issuing a ATTESTATION_COMPLETE call outside any attested process in an attempt to thwart the attestation. After computing a signature over the output data and the measurement result (if there are many instances of output data, the SK should compute a signature for each instance), the SK disables the protection mechanisms established for the attested process and yields control.Here we do not explicitly measure the output data addresses as we do for the input data, since the process code being measured contains the ATTESTATION_COMPLETE call as well as references to the output data.

In Figure 3, the SK utilizes the TPM's hashing and digital signature functionalities. The TPM provides protected registers called Platform Configuration Registers (PCR) for storing integrity measurements. The PCR_Reset function clears the PCR, and the PCR_Extend(n) function takes a 160bit number n, and updates a PCR value through PCR $\leftarrow$ $\text{SHA1}(\text{PCR}||n)$. After sending the input data addresses and the process code to be hashed by the TPM, the SK calls PCR_Read to retrieve the measurement result. The measurement result is stored in the SK's memory space until an ATTESTATION_COMPLETE call is issued. Then the SK sends the output data along with the measurement result to the TPM to be signed. The signature is computed by the TPM using a private signing key that has been loaded into the TPM's memory prior to the Sign call. A signing pair (K^{-1}, K) is created inside the TPM, and the private key K^{-1} is known only to the TPM and will never be exposed outside the TPM. A TPM can sign a certificate for its public signing key using its identity private key K_{ID}^{-1}. The certificate also states that the key is bound to the SK's locality and cannot be used by the OS kernel. The identity public key is vouched for by a CA.

BIND should support concurrent attestations of multiple processes. For this purpose, the SK maintains a data structure that records the information of all processes under attestation.

Verifying the Hash Two steps are required to verify an authenticator on a piece of input data: 1) verify the signature, 2) verify the hash. Since verifying the signature is straightforward, we now explain how to verify the hash and how to enable different software versions and software upgrades. BIND allows the application to register one or more legal hash values. We assume that for each application, there is a trusted authority that signs certificates for legal hash values. When an application registers a hash value, it has to show a correct certificate. The public key of an application's trusted authority is included whenever BIND is signing an authenticator for this application. Therefore, BIND supports various software versions and software upgrades.

Ensuring the Untampered Execution of the Process Code We argued that one of the unaddressed problems with current code-attestation technology is the time-of-use and time-of-attestation discrepancy. Even though the code attested may be legitimate at the time of attestation, it could have been compromised by the time of use. In our case, we also need to ensure that what is executed is exactly the code that is hashed. We address this problem by having the SK set up a safe environment for the process to execute. In this safe environment, the process code is "locked" from outside interference. The protection mechanisms introduced in this paragraph are hidden from the programmer's point of view.

- *Memory protection* One requirement of a safe execution environment states that no malicious intermediary

PROCESS $\rightarrow$ SK:	**ATTESTATION_INIT**(input data addresses, size of process code)
SK:	disable interrupt
	verify authenticator on input data
SK $\rightarrow$ TPM:	PCR_Reset
SK $\rightarrow$ TPM:	PCR_Extend(input data addresses, process code)
SK $\rightarrow$ TPM:	$h \leftarrow$ PCR_Read
SK:	set up secure execution environment for the process
	enable interrupt
SK $\rightarrow$ PROCESS:	yield control
PROCESS:	perform transformation on input data
PROCESS $\rightarrow$ SK:	**ATTESTATION_COMPLETE**(output data addresses)
SK:	disable interrupt
	verify that the call comes from the process being attested
SK $\rightarrow$ TPM:	Sign(output data, h)
TPM $\rightarrow$ SK:	$\{$output data, $h\}_{K^{-1}}$
SK:	clear protection
SK:	enable interrupt
SK $\rightarrow$ PROCESS:	$\{$output data, $h\}_{K^{-1}}$

Figure 3. Attestation Service Based on TPM and SEM

(including malicious software code, malicious I/O devices) can modify the process memory space during its execution. This we achieve through memory protection. With AMD's new SEM mode, memory can be protected on a per-page basis from access by OS kernel and peripheral devices. The attestation service running at the SK privilege level can utilize these features, and set up corresponding protection data structures to ensure that the process executes in a safe environment.

- *Securely restoring execution environment after interrupts* If the process code were simple enough to be able to execute in a single pass without going through the OS scheduler interrupts, then no code could have intercepted its execution, and no software attack would be able to change the processor environment during its execution.

  In reality, however, the process code may take a long time to execute and due to the OS scheduler, it may be suspended and resumed several times before completion. While performing these context switches, a malicious OS can cheat and not restore the correct execution environment. For instance, the OS may resume the code not at the instruction where it has left off, but at a different instruction address. A malicious OS can also modify its kernel data structure for the process in between two scheduler events, so that when the process is resumed, the register contents are changed. As a counter-measure, we propose the following approach. Before yielding to the process code, the SK loads a shadow Interrupt Descriptor Table (IDT) and a shadow interrupt handler that overrides the OS interrupt handling mechanisms. In this way, every time an interrupt is raised during the execution of the process code, the SK takes over. The SK then makes a copy of the run-time environment of the process, it also inserts in the process code an Secure Kernel Intercept instruction exactly where execution is going to resume. Then the SK dispatches the interrupt to the OS. And next time the process is resumed, it will trap to the SK first so that the SK can check if all run-time environment has been faithfully restored by the OS before resuming the process. The run-time environment to be checked comprises of 1) register values including general purpose registers, system/control registers, etc. 2) virtual to physical address mapping for the process.

A Symmetric Key Alternative So far we have used TPM's digital signature functionality to sign the measurement results. The drawback of asymmetric key cryptography is its high computational overhead. In some situations where efficiency is crucial, we would like to use symmetric key cryptography instead. In this paragraph, we propose a symmetric key alternative. To do this, we need to securely establish a secret MAC key between two TPMs; we also consider key management issues; and since the TPM does not support a MAC function by itself, we explain how to efficiently instantiate a MAC using the TPM's SHA-1 function. Our guidelines for key agreement and management are as follows:

- Since we assume untrusted intermediaries, the key ex-

$$\begin{aligned}
\text{SK A}:&\quad \text{generate } a, g^a \; mod\; p\\
\text{SK A} \rightarrow \text{TPM A}:&\quad \text{Seal}(a)\\
\text{SK A}:&\quad \text{destroy } a\\
\text{SK A} \rightarrow \text{TPM A}:&\quad \text{Sign}(g^a \; mod\; p)\\
\text{TPM A} \rightarrow \text{SK A}:&\quad \{g^a \; mod\; p\}_{K_A^{-1}}\\
\text{SK A} \rightsquigarrow \text{SK B}:&\quad \{g^a \; mod\; p\}_{K_A^{-1}}\\
\text{SK B} \rightsquigarrow \text{SK A}:&\quad \{g^b \; mod\; p\}_{K_B^{-1}}\\
\text{SK A}:&\quad \text{verify signature of } \{g^b \; mod\; p\}_{K_B^{-1}}\\
\text{SK A} \rightarrow \text{TPM A}:&\quad \text{Unseal}\\
\text{TPM A} \rightarrow \text{SK A}:&\quad a\\
\text{SK A}:&\quad \text{compute } SK_{AB} \leftarrow g^{ab} \; mod\; p\\
\text{SK A} \rightarrow \text{TPM A}:&\quad \text{Seal}(SK_{AB})\\
\text{SK A}:&\quad \text{destroy } a, SK_{AB}
\end{aligned}$$

Figure 4. Diffie-Hellman Key Exchange between TPM A and TPM B. $\rightsquigarrow$ denotes that the message has gone through an untrusted intermediary, whereas $\rightarrow$ denotes a secure channel, i.e., the channel between the SK and a local TPM is secure.

change protocol needs to be resilient to man-in-the-middle attacks.

- The keys should be sealed in TPM's memory and should remain invisible to any untrusted party, including the OS kernel, application code, peripheral devices, etc.

- The keys are unsealed to the SK upon time of use. The SK will use them to 1) verify the MAC on input data; 2) compute a MAC over the output data and the hash of the process code.

- The secret keys remain in the SK's memory space for a controlled period of time. To minimize the chance of leakage, the SK should destroy the keys immediately after usage. To prevent the untrusted OS kernel from reading off the secret key information, the SK should be executed in a globally uninterrupted manner. And since we have securely set up the DMA Exclusion Vectors (DEV) during secure boot, we can also prevent peripheral devices from reading the SK memory space.

For key exchange, we may use Diffie-Hellman [14]. In Figure 4, two participating hosts A and B try to establish a secret MAC key between their two TPMs. Here K_A^{-1} is the private signing key of TPM A. The signing pair (K_A^{-1}, K_A) is created inside the TPM, and the private key K_A^{-1} is known only to TPM A and will never be exposed outside the TPM. TPM A can sign a certificate for K_A using its identity private key $K_{ID(A)}^{-1}$. The identity public key is vouched for by the CA.

After a secret MAC key is established between TPM A and TPM B, for SK A to verify the MAC on some data incoming from B, SK A first requests TPM A to unseal SK_{AB}, then it verifies the MAC using SK_{AB}, and immediately destroys SK_{AB} from memory.

We now explain how to efficiently and securely implement a MAC function using the TPM. The TPM provides a SHA-1 functionality. So we can instantiate an HMAC [11] using SHA1, i.e., $MAC(msg, K) =$ SHA-1($K \oplus opad$, SHA-1($K \oplus ipad, msg$)). The SK has to facilitate the generation of the HMAC, since the TPM does not support an HMAC function by itself. Therefore the SK retrieves the MAC key from TPM to perform the XOR and concatenation functions; the result will then be handed to the TPM to be hashed.

BIND should provide both asymmetric and symmetric key options, so that the process can choose whichever version to use based on need. In addition, for the symmetric key version, we need a small modification to its interface: 1) the ATTESTATION_INIT call has to specify where the input data came from; 2) the ATTESTATION_COMPLETE call has to specify the intended recipients, so that BIND can create a MAC for each recipient.

5 Case Study

To demonstrate the use of BIND in real-world distributed systems, we present two case studies: securing distributed computation applications and securing the BGP routing protocol.

5.1 Securing Distributed Computation Applications with BIND

Introduction to Distributed Computation Applications Recently, there has been a surge of interest in using large-scale distributed computation to solve difficult computational tasks. By utilizing the spare processor cycles of many personal computers, we can obtain the computational power of one or more super-computers. For instance, the well-know SETI@Home [3] project uses the free cycles of Internet-connected computers to analyze radio telescope data in the Search for Extraterrestrial Intelligence. The GIMPS [2] project is intended for the search of new Mersenne primes and encryption keys; and the Folding@Home [1] project uses distributed computing to study protein folding, misfolding, aggregation, and related diseases.

A distributed computation application is made up of a supervisor and many participants. The supervisor splits a job into tasks and assigns a task to each participant. One of the greatest security concerns for distributed computation applications is the honesty of participants. In many cases a participant may be motivated to cheat and inject incorrect results, either to disrupt the computation or to gain monetary remuneration.

Therefore researchers have endeavored to secure distributed computation applications against dishonest participants. Golle and Mironov [17] present a security based administrative framework for commercial distributed computations. Their solution relies on redundancy to guarantee probabilistic detection of cheating behavior. While their work is restricted to the class of distributed computing applications that try to invert a one-way function, Szajda, Lawson, and Owen [39] extend Golle and Mironov's work to optimization and Monte Carlo simulations. Meanwhile, Monrose et al. propose to generate execution trace on the participants that can be later used to verify the integrity of program execution [29].

Securing Distributed Computation Applications with BIND We model the distributed computing application as comprised of a supervisor process and many participant processes. The supervisor process splits the job into tasks. A participant solves a task and reports the outcome to the supervisor. Under our conceptual framework, a job is modeled as primitive data; and each task is derived data output from the supervisor process, and input to a participant process. The computation result is derived data output from a participant process and input to the supervisor process.

Assume now we have the BIND attestation service in place. Then the supervisor process can ask BIND to sign a task description before dispatching it to a participant. The participant can ask the local BIND service to verify the integrity of the task description; and when the participant process computes over the input data, its execution is protected and vouched for by BIND. The computation result will be signed by BIND along with an integrity proof of the participant process that has produced the result. In this way, when the result is reported to the supervisor, he can easily check whether the result is trustworthy.

When compared with the literature on securing distributed computation applications, BIND offers the following properties (here we are assuming that the attacker performs software attacks only):

- **Deterministic Guarantee:** BIND offers deterministic guarantee on the integrity of the computational result. In particular, BIND guarantees what code has been run in generating the result.

- **General:** While Golle et al. [17] and Szajda et al. [39]. suggest different approaches for securing different functions, BIND is applicable to all types of distributed computation applications regardless of what function we want to compute. Meanwhile, existing works consider a centralized distributed computation model with one supervisor that distributes tasks to participants. If, however, we are to change to a distributed coordination model where participants coordinate themselves, BIND can still be readily applied.

- **Efficient:** With BIND, the integrity proof for the computational result is efficient to generate and verify. In comparison, the execution trace approach by Monrose et al. is more expensive, since the trace size is linear in terms of the number of instructions actually executed [29]. For BIND, the entire code is measured only once regardless of how many times each instruction is executed, and the verification takes constant overhead.

5.2 Securing BGP with BIND

The Border Gateway routing Protocol (BGP) enables routing between administrative domains, and is thus one of the most important protocols in the current Internet [33].

In this section, we give a brief overview of BGP and its security requirements, then we show how we can apply the transitive trust model in this paper to design a highly efficient and secure BGP protocol.

BGP Primer An Autonomous System (AS) is a collection of routers under one administrative domain, for simplicity assume that an AS corresponds to an Internet Service Provider (ISP). In practice, an ISP may have multiple ASes. An AS is identified through an AS Number (ASN). The current Internet has about 19,000 ASes.

A prefix is a destination that the routing protocol needs to establish a route to. A prefix is characterized by an IP address and the number of bits that are fixed in it, for example, the prefix `209.166.161/24` denotes one of Akamai's class C networks; `/24` means that the first 24 bits of this address are significant, all combinations of the remaining 8 bits belong to that network.

BGP stands for Border Gateway Protocol; it is a path vector routing protocol, establishing a path to each existing prefix. In BGP, neighboring ASes exchange prefix information through BGP Update messages. In a simplified

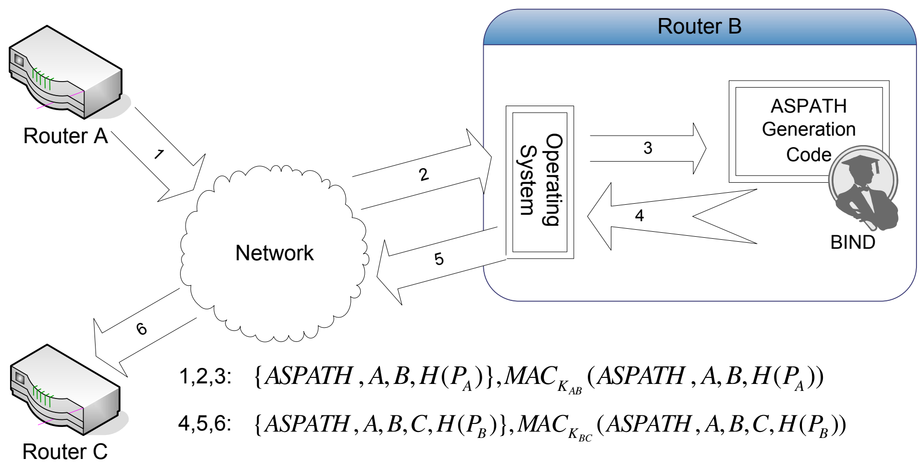
Router B
Router A
1
2
Operating System
3
ASPATH Generation Code
4
BIND
Network
5
6
Router C
1,2,3: $\{ASPATH, A, B, H(P_A)\}, MAC_{K_{AB}}(ASPATH, A, B, H(P_A))$
4,5,6: $\{ASPATH, A, B, C, H(P_B)\}, MAC_{K_{BC}}(ASPATH, A, B, C, H(P_B))$

Figure 5. Securing BGP using BIND, the numbers on the arrows represent the temporal order of the messages.

description, a BGP Update message consists of a prefix along with an ASPATH, i.e., a list of ASes that need to be traversed to reach that prefix. For example, our simplified update may look as follows: `209.166.161/24, <701, 3356>`, which means that AS 3356 owns the prefix, and AS 701 will forward its traffic destined for `209.166.161/24` to AS 3356. If an AS uses that route, it sends a BGP Update to all of its neighboring ASes, prepending its ASN to the ASPATH. In our example, if AS 209 want to use this path, it send the following BGP Update to its neighbors: `209.166.161/24, <209, 701, 3356>`. More comprehensive descriptions of BGP are available [38, 44].

BGP Security Requirements and Approaches The current version of BGP was designed for a benign environment, where ASes could trust each other. Unfortunately, BGP is so brittle that even misconfigurations can cause network-wide connectivity outages [25, 28]. BGP security vulnerabilities are thoroughly studied, for example in the Routing Protocol Security IETF working group [34], or by research groups [21, 31].

The two main classes of attack against BGP are: unauthorized prefix announcements (prefix theft), and ASPATH falsifications in BGP update messages. Both attacks are in the category of attracting traffic to a point in the network that would otherwise not receive the traffic. This allows an attacker to control packets that it would otherwise have no control over. These attacks are sometimes referred to as *blackhole attacks*, since they attract traffic to a point in the network.

For example, an attacker may want to control packets destined for the Google search engine. Using unauthorized prefix announcement, the attacker (who we assume owns an AS) injects BGP update messages claiming ownership of the Google prefix. Its neighboring ASes will then start forwarding traffic destined to Google towards the attacker. Several researchers have designed mechanisms to defend against this attack. For example, Kent et al. proposed S-BGP, where they attach an *address attestation* to a BGP update, which is a statement signed with the prefix's private key, allowing the first AS to announce that prefix [21, 22]. Other researchers use a similar approach, requiring the prefix owner to obtain a certificate proving ownership [7, 18, 20, 43].

The second major class of BGP attacks is ASPATH falsification, where an attacker not only adds its ASN to the ASPATH, but also removes or alters earlier ASNs on the path. Since the number of ASes on the path are used for routing decisions, removing ASes from the path results in a shorter router, which makes the path through the attacker more attractive (thus also attracts traffic to the attacker). To prevent unauthorized ASPATH modifications, S-BGP utilizes *route attestations*, which are essentially delegation messages where one AS delegates the right to announce a prefix to its neighboring AS. An S-BGP update is valid if it the ASPATH matches the address attestation and the chain of route attestations, and if all the signatures of the attestations are valid [21, 22]. As an alternative mechanism, researchers proposed symmetric cryptographic primitives to prevent unauthorized ASPATH modifications, e.g., SPV [20].

Securing BGP using BIND The two main mechanisms we need to secure BGP is to verify the correctness of the origin of the prefix (to prevent prefix theft), and to prevent a malicious AS from altering the ASPATH in any other way than appending its own ASN to the path.[4]

Applying our conceptual model to the BGP case, we can categorize the prefix as primitive data. Therefore the integrity of a prefix has to be ensured external to our attesta-

[4] In SPV, to achieve the delegation property, they slightly modify the protocol such that each router signs in the ASN of the next hop. We adopt this approach here for the same purpose.

tion service. We may adopt the known certificate approach, where a prefix owner obtains a prefix certificate from a trusted authority.

Now we direct our attention toward the attestation of the derived ASPATHs. For this purpose we shall apply BIND. We assume the presence of a trusted attestation service on every router. Here the process to be attested is the ASPATH generation code, and the untrusted intermediaries include the router's operating system, the network and the router's local storage.

Figure 5 shows what happens in Router B when it receives a BGP update from Router A, performs transformations on the ASPATH attribute, and forwards the resulting update to Router C. Here we assume symmetric key cryptography is used.

First, Router B receives a route update $\{\text{ASPATH}, A, B, H(P_A)\}, MAC_{K_{AB}}(\text{ASPATH}, A, B, H(P_A))$ from Router A, here $H(P_A)$ denotes the hash of ASPATH generation process on Router A. The ASPATH along with the hash value is cryptographically protected by a message authentication code using key K_{AB}, the secret key shared between the trusted attestation services on Router A and B. Now Router B's ASPATH generation process issues a ATTESTATION_INIT request; the request includes the memory addresses as to where the incoming update is stored. In reply to the request, the trusted entity on Router B first verifies the MAC on the incoming routing update. Then it hashes the process code along with the memory addresses of the input data. Finally it sets up a secure execution environment and resumes the execution of the ASPATH generation process. Now the Router B's ASPATH generation process appends the ASN[5] of the next hop router to the ASPATH.[6] This explicitly delegates the right to propagate the ASPATH to Router C. At the end of the process is a ATTESTATION_COMPLETE request whose parameters include the memory addresses of the resulting ASPATH, and the next-hop router where this update is intended. The trusted attestation service then computes a message authentication tag over the output data and the hash, i.e., $MAC_{K_{BC}}(\text{ASPATH}, A, B, C, H(P_B))$. Finally after clearing the protection mechanisms, it resumes the execution of the process where it has left off.

When compared with existing approaches, here are the desirable properties BIND has to offer:

- **Simple design** To ensure that a router performs correct operations on an ASPATH, S-BGP and SPV both require complex cryptographic constructions to use as authenticators of data. In particular, SPV uses a complicated hash tree to compute a one-time signature for each suffix of an ASPATH. The cryptographic construction is problem specific, i.e., one needs to come up with different cryptographic constructions for different protocols; and proving the security of different cryptographic constructions requires a lot of expertise.

  By contrast, through the use of BIND, we do not have to reply on these complicated cryptographic constructions, and the task of securing BGP is much simpler.

- **Efficient** Previous approaches [20, 21, 22] of route attestation incur an $O(n)$ signature overhead where n is the ASPATH length. We note that this problem is ideally suited for our transitive trust approach. By applying the transitive trust approach to securing BGP, we can reduce the $O(n)$ signature overhead to $O(1)$. Since simply by verifying the previous hop process, we can guarantee that the entire chain of routers have performed legal transformations on the ASPATH.

By applying BIND, we can achieve at least the same properties as S-BGP [21, 22] and SPV [20] assuming full deployment. First, it provides protection against modification and truncation of the ASPATH, since with the attestation service, we can guarantee that a router has performed the only legal operation to an ASPATH, i.e., appending the ASN of the next-hop router. Meanwhile we are now also secure against the ASPATH lengthening attack, which SPV is unable to prevent. Another important property which S-BGP and SPV achieve is the delegation property: a malicious router M cannot propagate an ASPATH without the permission of the last hop router on the ASPATH. Here we ensure this property partly through protocol semantics: the ASPATH generation process explicitly appends the next-hop router to the ASPATH to delegate the right to forward the ASPATH to the next-hop router. And since the delegation code is inside the process attested, the correctness of delegation can be verified under the transitive trust properties of BIND.

However, as we have pointed out, the attestation service does not prevent data misuse attacks by a malicious intermediary. For the BGP case, a typical data misuse attack is the data *replay attack*, where the malicious intermediary replays a route that has already been withdrawn. As we have mentioned, under our model, it is up to the protocol semantics to address such attacks. For instance, to defend against the replay attack, the ASPATH generation process can attach a timeout field with each ASPATH before sending them to the attestation service to be MACed. Meanwhile we should embed the logic for checking the timestamp in the attested process code. Meanwhile, the BIND approach does not work well for incremental deployment.

6 Related Work

In this section, we review related work on verifying correct code execution, Virtual Machine Monitors (VMM), and the TPM and SEM technology that we extensively used in this paper. Other related work, including securing BGP and distributed computation applications have been discussed in Section 5.

Verifying Code Execution Wasserman and Blum [42] review the line of theoretical work that enables us to check

[5]The real-world BGP protocol performs ASN prepending, but for simplicity of explanation, we assume ASNs are appended to the ASPATH.

[6]The ASN of the next-hop router is a primitive data input to the ASPATH generation process. Its integrity has to be enforced using a mechanism external to BIND.

the result of a program. While of theoretic interest, their methodology is restricted to specific functions and thus of limited use in reality.

Vigna [41] proposes to use cryptographic traces to enable mobile code to be securely executed on an untrusted host. Basically this approach requires that the untrusted host store cryptographic traces for the execution, so that a trusted host can request the trace and verify it by executing the code again and comparing the execution with the stored trace. However, their approach is expensive, since the verifier has to execute the entire code again, and the size of trace grows linearly with the size of the code.

Malkhi et al. [26] build Fairplay, a secure two-party computation system. In fact there has been a huge amount of theoretic work in secure multi-party computation, however, this line of work is chiefly concerned about ensuring the secrecy of input data instead of the integrity of the computed outcome.

To verify the integrity of the booting process, we can use secure boot mechanisms [8, 9]. Starting from an initial trusted state, each layer verifies the digital signature of the next layer before executing it. This ensures that the software stack has not been altered. Their mechanism is also similar to our transitive integrity verification mechanism. While they verify the integrity of the next layer software before loading it, we verify the integrity of previous hop process/data before using it.

In another line of work, Seshadri et al. use timing properties to perform SoftWare-based ATTestation (SWATT) for embedded devices [37]. In SWATT, the embedded device computes a checksum of its memory whenever it receives a challenge. SWATT is designed in a way such that a malicious attack that has modified the memory contents of an embedded device would have to take a longer time to come up with the correct checksum.

Virtual Machine Monitors In the area of Virtual Machine Monitors (VMM), Garfinkel et al. build Terra [16], a virtual machine-based platform for trusted computing. They partition a tamper-resistant hardware platform into multiple, isolated Virtual Machines(VM). The VMM and the trusted hardware can attest the software running on each VM to a remote verifier. Their approach assumes that the VMM cannot be compromised at runtime, and they partly address the efficiency and usability of attestation. In particular, though they perform coarse-grained attestation, they propose to split attestable entities into smaller blocks and compute a hash over each block.

Trustworthy Computing Platform One chief standard developed by TCG [40] is the Trusted Platform Module (TPM). The TPM is a coprocessor intended to serve as the hardware root of trust of a trusted platform. The TPM provides several functional components, including fast cryptographic engines, protected storage, key generation, etc. Several researchers have examined how to use a TPM to perform load-time attestations of the software system [27, 36].

The SEM architecture is part of AMD's drive toward enabling a trustworthy computing environment. SEM was designed with a primary goal to counter software attacks, and it offers a hybrid hardware and software solution. Built on top of the x86 architecture, SEM provides a new mode, the Trusted Execution Mode (TX = 1), where the TX Mode bit is a new CPU state bit. The Secure Kernel runs in the TX Mode which offers several hardware protection mechanisms including memory, I/O, DMA and system/control register protection. Meanwhile SEM offers a single entry-point into the SK called a Secure Kernel Intercept (SKI). It also supports secure initialization through the SKINIT instruction which works together with the TPM to securely record the measurements of the software thus loaded.

Apart from AMD's SEM technology, Intel's Vanderpool [6] and Lagrande [19] processors provide similar TCG/TPM and isolation features BIND requires. Meanwhile, AMD's new generation of processor virtualization technology Presidio [24] works for BIND too.

7 Conclusion and Future Work

As code attestation technology receives increasing attention in the research community, we are interested in addressing the following questions: 1) What are the desired properties we would ultimately like to achieve out of attestation? 2) Suppose we were able to build a perfect attestation service with all of the desired properties, and make it available on every platform, how can it aid us in designing secure distributed systems in general? 3) How far are we from the perfect attestation service and how far can we push our limits toward this goal using currently available TCG and microprocessor technology?

We propose BIND, a fine-grained attestation service that ties the proof of what code has executed to the data the code has produced. By attesting to the critical code immediately before it executes, we narrow the gap between time-of-use and time-of-attestation. BIND is useful for establishing a trusted environment for distributed systems, and greatly simplifies the design of secure distributed systems.

For future work, we want to investigate the feasibility of a hardware based design for BIND. The current version of BIND runs in the Secure Kernel and assumes that the Secure Kernel is trustworthy, which is a hybrid hardware and software solution. However, it will be desirable to place trust only on hardware and no software components at all.

8 Acknowledgments

We gratefully acknowledge support, feedback, and fruitful discussions with Dawn Song and James Hendricks. We would especially like to thank Virgil D. Gligor for his suggestions on how to improve the paper, and the anonymous reviewers for their insightful comments.

References

[1] The folding@home project. Stanford University, `http://www.stanford.edu/group/pandegroup/cosm/`.

[2] The great internet mersenne prime search. `http://www.mersenne.org/prime.htm`.

[3] The search for extraterrestrial intelligence project. University of California Berkeley, `http://setiathome.berkeley.edu/`.

[4] Exploiting format string vulnerabilities. TESO Security Group, `http://www.team-teso.net/articles/verbformatstring`, Sept. 2001.

[5] AMD platform for trustworthy computing. WinHEC 2003, `http://www.microsoft.com/whdc/winhec/papers03.mspx`, Sept. 2003.

[6] Intel Vanderpool Technology for IA-32 processors (VT-x) preliminary specification. Intel C97063-001, `ftp://download.intel.com/technology/computing/vptech/C97063.pdf`, Jan. 2005.

[7] W. Aiello, J. Ioannidis, and P. McDaniel. Origin authentication in interdomain routing. In *Proceedings of ACM Conference on Computer and Communications Security (CCS 2003)*, pages 165–178, Oct. 2003.

[8] W. A. Arbaugh, D. J. Farber, and J. M. Smith. A reliable bootstrap architecture. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 65–71, May 1997.

[9] W. A. Arbaugh, A. D. Keromytis, D. J. Farber, and J. M. Smith. Automated recovery in a secure bootstrap process. In *Proceedings of Symposium on Network and Distributed Systems Security (NDSS)*, pages 155–167, Mar. 1998.

[10] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebar, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2003.

[11] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. In *Advances in Cryptology - Crypto '96*, pages 1–15, 1996.

[12] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *Proceedings of Symposium on Operating Systems Design and Implementation (OSDI)*, Feb. 1999.

[13] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *Proceedings of DARPA Information Survivability Conference and Exposition (DISCEX 2000)*, pages 119–129, Jan. 2000.

[14] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Trans. Inform. Theory*, IT-22:644–654, Nov. 1976.

[15] J. S. Foster, M. Fahndrich, and A. Aiken. A theory of type qualiers. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'99)*, May 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 Symposium on Operating System Principles (SOSP)*, Oct. 2003.

[17] P. Golle and I. Mironov. Uncheatable distributed computations. In *Proceedings of the RSA Conference (CT-RSA 2001)*, 2001.

[18] G. Goodell, W. Aiello, T. Griffin, J. Ioannidis, P. McDaniel, and A. Rubin. Working around BGP: An incremental approach to improving security and accuracy in interdomain routing. In *Proceedings of Symposium on Network and Distributed Systems Security (NDSS)*, Feb. 2003.

[19] D. Grawrock. Lagrande architecture SCMS-18. `http://www.intel.com/technology/security/downloads/scms18-LT_arch.pdf`, Sept. 2003.

[20] Y.-C. Hu, A. Perrig, and M. Sirbu. SPV: Secure path vector routing for securing BGP. In *Proceedings of ACM SIGCOMM*, Sept. 2004.

[21] S. Kent, C. Lynn, J. Mikkelson, and K. Seo. Secure border gateway protocol (S-BGP) — real world performance and deployment issues. In *Proceedings of Symposium on Network and Distributed Systems Security (NDSS)*, pages 103–116, Feb. 2000.

[22] S. Kent, C. Lynn, and K. Seo. Secure border gateway protocol (S-BGP). *IEEE Journal on Selected Areas in Communications*, 18(4):582–592, Apr. 2000.

[23] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, July 1982.

[24] M. LaPedus. Amd tips 'Pacifica' and 'Presidio' processors for '06. `http://www.eetimes.com/semi/news/showArticle.jhtml?articleID=52601317`, Nov. 2004.

[25] R. Mahajan, D. Wetherall, and T. Anderson. Understanding BGP misconfiguration. In *Proceedings of ACM SIGCOMM*, Aug. 2002.

[26] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay - a secure two-party computation system. In *Proceedings of USENIX Security Symposium*, pages 287–302, Aug. 2004.

[27] H. Maruyama, S. Munetoh, S. Yoshihama, and T. Ebringer. Trusted platform on demand. IPSJ SIGNotes Computer SECurity Abstract No. 024 - 032.

[28] S. A. Misel. Wow, AS7007! NANOG mail archives, `http://www. merit. edu/mail.archives/nanog/1997-04/msg00340.html`, 1997.

[29] F. Monrose, P. Wyckoff, and A. Rubin. Distributed execution with remote audit. In *Proceedings of ISOC Network and Distributed System Security Symposium (NDSS '99)*, Feb. 1999.

[30] Next-Generation Secure Computing Base (NGSCB). `http://www.microsoft.com/resources/ngscb/default.mspx`, 2003.

[31] O. Nordström and C. Dovrolis. Beware of BGP attacks. *ACM Computer Communications Review*, 34(2):1–8, Apr. 2004.

[32] N. L. Petroni, T. Fraser, J. Molina, and W. A. Arbaugh. Copilot – a coprocessor-based kernel runtime integrity monitor. In *Proceedings of USENIX Security Symposium*, pages 179–194, Aug. 2004.

[33] Y. Rekhter and T. Li. A border gateway protocol 4 (BGP-4). RFC 1771, Mar. 1995.

[34] Routing protocol security requirements (rpsec). IETF working group, `http://www.ietf.org/html.charters/ rpsec-charter.html`, 2004.

[35] R. Sailer, T. Jaeger, X. Zhang, and L. van Doorn. Attestation-based policy enforcement for remote access. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)*, Oct. 2004.

[36] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *Proceedings of USENIX Security Symposium*, pages 223–238, Aug. 2004.

[37] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla. SWAtt: SoftWare-based Attestation for embedded devices. In *Proceedings of IEEE Symposium on Security and Privacy*, May 2004.

[38] J. W. Stewart. *BGP4: inter-domain routing in the Internet*. Addison-Wesley, 1999.

[39] D. Szajda, B. Lawson, and J. Owen. Hardening functions for large-scale distributed computations. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 216–224, May 2003.

[40] Trusted Computing Group (TCG). `https://www.trustedcomputinggroup.org/`, 2003.

[41] G. Vigna. Cryptographic traces for mobile agents. In *Mobile Agents and Security*, volume 1419 of *LNCS State-of-the-Art Survey*, pages 137–153. Springer-Verlag, June 1998.

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

[43] R. White. Securing BGP through secure origin BGP. *Internet Protocol Journal*, 6(3):15–22, Sept. 2003.

[44] R. White, D. McPherson, and S. Sangli. *Practical BGP*. Addison-Wesley, 2004.