Determining the Integrity of Application Binaries on Unsecure Legacy Machines Using Software Based Remote Attestation

Raghunathan Srinivasan[1], Partha Dasgupta[1],
Tushar Gohad[2], and Amiya Bhattacharya[1]

[1] Arizona State University, Tempe AZ 85281, USA
{raghus,partha,amiya}@asu.edu
[2] MontaVista Software, LLC
tusharsg@gmail.com

Abstract. Integrity of computing platforms is paramount. A platform is as secure as the applications executing on it. All applications are created with some inherent vulnerability or loophole. Attackers can analyze the presence of flaws in a particular binary and exploit them. Traditional virus scanners are also binaries which can be attacked by malware. This paper implements a method known as Remote Attestation entirely in software to attest the integrity of a process using a trusted external server. The trusted external server issues a challenge to the client machine which responds to the challenge. The response determines the integrity of the application.

Keywords: remote attestation; integrity measurement; code injection.

1 Introduction

An untrusted computing platform poses many risks for its user (Alice). Execution of a security sensitive program ($\mathcal{P}$) can be tampered in many ways by an attacker (Mallory). Mallory can modify the binary image of $\mathcal{P}$ on the secondary storage media, Mallory can modify its executing in-core image, or Mallory can execute another program $\mathcal{P}'$ which mimics $\mathcal{P}$'s behavior. These changes can be made so that the untrusted platform reveals some secrets about Alice to Mallory. These attacks occur as Alice may execute many unverified or unsecure applications on the platform along with $\mathcal{P}$. It is estimated that on average programs can contain between 6 - 16 bugs per 1000 lines of code (LOC) in an application binary [1]. It is estimated that fault density in Operating System (OS) kernels can range from 2 - 75 per 1000 LOC [8]. In addition an OS consists of many device drivers, which have error rates much higher than kernel code [2], hence it can be concluded that it is difficult to eliminate all vulnerabilities from a system.

All copies of an application are identical; this gives Mallory the opportunity to analyze the presence and locations of flaws in the application, and develop means to exploit these flaws. Operating Systems offer little or no fault isolation;

S. Jha and A. Maturia (Eds.): ICISS 2010, LNCS 6503, pp. 66–80, 2010.

this can lead to a malware rapidly obtaining control of a computing platform [17]. Detection of malicious logic is known to be difficult [3] and smart malware can render detection schemes ineffective; this is due to the fact that traditional detection mechanisms operate off application binaries which can be disabled or patched to escape detection [15]. Consequentially a user (Alice) has to request integrity measurement of the platform from an external agent, or an entity that operates beyond the bounds of the operating system.

Hardware and virtualization based attestation schemes have been researched intensively. Hardware based schemes involve taking integrity measurements by using the Trusted Platform Module (TPM) chip provided by the Trusted Computing Group [16], [11], [6], or with the use of a secure coprocessor that can be placed on the PCI slot [17], [9]. Virtualization schemes involve the hypervisor or a Domain 0 trusted OS obtaining integrity measurements on an untrusted guest OS [5], [10]. Hardware schemes have significant drawbacks: they require specialized hardware, cannot be re-programmed easily, are difficult to manage, and have the stigma of Digital Rights Management (DRM) attached to them. Virtualization requires greater resources than a system that operates on only one native OS. Software based solutions to measure the integrity of a platform involve the use of a set of techniques known as Remote Attestation, which can allow a remote agent to determine whether a particular application executing on a platform has been tampered. These set of techniques can be extended to determine the integrity of all applications on the platform.

In this paper a software based solution is implemented to measure the integrity of an application using a trusted external server (Trent). Trent issues a challenge to the client application $\boldsymbol{\mathcal{P}}$. The response provided by $\boldsymbol{\mathcal{P}}$ allows Trent to determine whether its integrity is compromised. The challenge should have inherent characteristics that prevent Mallory from forging any section of the results generated. A software protocol allows Mallory to perform various attacks. If the challenge is not significantly different for every attestation instance, Mallory can construct a replay of a response from a previous instance of the attestation. If the challenge is not complicated enough, Mallory can compute the response without executing the requested challenge and send the results to Trent. In addition, Mallory may bounce the challenge to another machine which contains a clean copy of the program to obtain results of the challenge. Mallory may also execute the challenge in a sandbox to determine its results.

To mitigate these situations Trent generates a new instance of attestation code $\boldsymbol{\mathcal{C}}$ which is sent to Alice for execution. $\boldsymbol{\mathcal{C}}$ is binary code which is injected by the application $\boldsymbol{\mathcal{P}}$ on itself, in addition, $\boldsymbol{\mathcal{C}}$ does not require the system library support as it executes any required system call by executing software interrupts. This prevents any user level malware from tampering with the results of integrity measurements, the kernel should not be compromised for this process to work. It may be argued that since the kernel is not compromised, it can be used to perform the entire attestation. This scenario would require that the OS keep track of every possible vendor's binaries, which can be a difficult task. The attestation is performed by an external agent to free up such complex requirements at

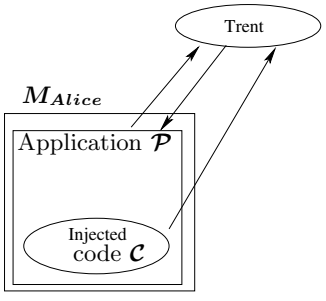

Fig. 1. Overview of Remote Attestation

the OS end. Injection of code on a client machine ($\boldsymbol{M_{Alice}}$) to obtain integrity measurements is a novel aspect of the solution provided in this work. This reduces the window of opportunity that Mallory may have to analyze operations being performed on $\boldsymbol{M_{Alice}}$.

The operations performed by $\mathcal{C}$ in each attestation instance are changed to prevent Mallory from performing a *replay* attack. There are many operations performed during attestation that make determining the response difficult for Mallory without executing $\mathcal{C}$. In addition, $\mathcal{C}$ measures some machine and process identifiers which are determined through the system interrupt interface to make forging of results difficult. $\mathcal{C}$ has inherent programming constraints which ensure that if $\mathcal{C}$ executes, it sends the results back to Trent.

Fig. 1 provides an overview of remote attestation. Trent is a trusted entity who has knowledge of the structure of a clean copy of the process ($\mathcal{P}$) to be verified. Trent has to be a trusted server, as Alice executes code received from Trent. Trent provides executable code ($\mathcal{C}$) to Alice which is injected on $\mathcal{P}$. $\mathcal{C}$ takes overlapping MD5 hashes and overlapping arithmetic checksums on subregions of $\mathcal{P}$ and returns the results to Trent. This prototype determines the integrity of a binary executing at ($\boldsymbol{M_{Alice}}$). This protocol is robust against user mode viruses that can modify system libraries, but not against rootkits. The remote attestation implemented as part of this work is more robust and works under harder constraints moredifficult than those implemented in previous works *Pioneer* [13], *Genuinity*, [7].

The remainder of this work is organized as follows, section describes the related work for integrity measurement, section 3 describes the threat model and attack scenarios and their fixes in detail, section 4 describes the design of the software based remote attestation, section 5 presents the implementation details, section 6 describes work that is proposed as an extension of completing the remote attestation, and section 7 concludes the paper.

2 Related Work

Integrity measurement involves checking if the program code executing within a process, or multiple processes, is legitimate or has been tampered. It has been

implemented using hardware, virtual machine monitors, and software based detection schemes. Some hardware based schemes operate off the TPM chip provided by the Trusted Computing Group [16], [11], [6], while others use a hardware coprocessor which can be placed into the PCI slot of the platform [17], [9]. The schemes using the TPM chip involve the kernel or an application executing on the client obtaining integrity measurements, and providing it to the TPM, the TPM signs the values with its private key and may forward it to an external agent for verification. The coprocessor based schemes read measurements on the machine without any assistance from the OS or the CPU on the platform, and compare measurements to previously stored values.

Terra uses a trusted virtual machine monitor (TVMM) and partitions the hardware platform into multiple virtual machines that are isolated from one another [5]. Hardware dependent isolation and virtualization are used by Terra to isolate the TVMM from the other VMs. Terra relies on the underlying TPM to take some measurements and hence is unsuitable for legacy systems. In *Pioneer* [13], the integrity measurement is done without the help of hardware modules or a VMM. The verification code for the application resides on the client machine. The verifier (server) sends a random number (nonce) as a challenge to the client machine. The response to the challenge determines if the verification code has been tampered or not. The verification code then performs attestation on some entity within the machine and transfers control to it. This forms a dynamic root of trust in the client machine. *Pioneer* assumes that the challenge cannot be redirected to another machine on a network, however, in many real world scenarios a malicious program can attempt to redirect challenges to another machine which has a clean copy of the attestation code. *Pioneer* incorporates the values of Program Counter and Data Pointer, in its checksum procedure; both the registers hold virtual memory addresses. An adversary can load another copy of the client code to be executed in a sandbox like environment and provide it the challenge. This way an adversary can obtain results of the computation that the challenge produces and return it to the verifier. *Pioneer* also assumes that the server knows the exact hardware configuration of the client for performing a timing analysis; this places a restriction on the client to not upgrade or change hardware components.

Genuinity [7] implements a remote attestation system in which the client kernel initializes the attestation for a program. It receives executable code and maps it into the execution environment as directed by the trusted authority. The executable code performs various checks on the client program, returns the results to a verified location in the kernel on the remote machine, which returns the results back to the server. The server checks if the results are in accordance with the checks performed, if so the client is verified. This protocol requires operating system (OS) support on the remote machine for many operations including loading the attestation code into the correct area in memory, obtaining hardware values such as TLB. It also requires the client OS to disable interrupts in order to have confidence that the attestation code actually executed. However, if a client OS is corrupted then it may choose to not disable interrupts in which case

various meta-information about the process incorporated into the checksum will not be correct. Another problem with this scheme is that the results are communicated to the server by the kernel and not the downloaded code. This may allow a malicious OS to analyze and modify certain values that the code computes. In *TEAS* [4], the authors propose a remote attestation scheme in which the verifier generates program code to be executed by the client machine. Randomized code is incorporated in the attestation code to make analysis difficult for the attacker. The analysis provided by them proves that it is very unlikely that an attacker can clearly determine the actions performed by the verification code; however implementation is not described as part of TEAS and certain implementation details often determine the effectiveness of a particular solution.

SWATT [12] implements remote attestation scheme for embedded devices where the attestation code resides on the node to be attested. The code contains a pseudorandom number generator (PRG) which receives a seed from the verifier. The attestation code includes memory areas which correspond to the random numbers generated by PRG as part of the measurement to be returned to the verifier. The obtained measurements are passed through a keyed MAC function, the key for the instance of MAC operation is provided by the verifier. The problem with this scheme is that if an adversary obtains the seed and the key to the MAC function, the integrity measurements can be spoofed as the attacker would have access to the MAC function and the PRG code. Remote Attestation for Wireless Sensors has also been implemented by sending executable code to the sensor node. The node executes the code which returns measurements to the Base station [14]. This method appears similar to the one presented in this work, however, in this paper we provide remote attestation for a x86 based computer which has many nuances like protected code sections while sensor nodes do not. Injecting executable code into the process running on a x86 based computer is a harder task than injecting executable code in a sensor node. In addition, this paper also attempts to detect impersonation of a client machine by a machine controlled by an attacker which is not covered in the above work.

In this paper remote attestation is implemented by downloading new (randomized and obfuscated) attestation code for every instance of the operation. Executing attestation code that is recently downloaded from an external machine makes it difficult for the attacker to fake any results that are produced by the attestation code. This is because Mallory has no prior knowledge of the structure or content of the code, as a result of which the attacker has no knowledge of the operations performed by the downloaded code. This means that to launch a successful attack, Mallory would have to perform an 'impromptu' analysis of the operations performed and report the forged results to Trent within a specific time frame. This can be considered difficult to achieve.

3 Threat Model and Attack Scenarios

Alice executes a security sensitive application $\mathcal{P}$. Alice wants to know whether $\mathcal{P}$ has been tampered in any way prior to entering sensitive information in it.

Alice contacts Trent (who is a Trusted externel agent) to attest $\mathcal{P}$. In order to determine the integrity of $\mathcal{P}$, Trent needs to obtain measurements from $\boldsymbol{M_{Alice}}$. To achieve this, Trent provides executable code $\mathcal{C}$ to Alice to execute, Alice in turn injects the code on the application $\mathcal{P}$ to be attested. It is assumed that Trent has prior knowledge of the binary image of $\mathcal{P}$. This is a reasonable assumption as Trent can take these measurements locally on a known pristine copy of $\mathcal{P}$ even if Trent was not the vendor who generated $\mathcal{P}$. $\mathcal{C}$ performs certain checks on $\boldsymbol{M_{Alice}}$ to determine whether $\mathcal{P}$ has been tampered. It must be noted that denial of service based attacks, or process termination attacks which do not allow $\mathcal{C}$ to measure and communicate results to Trent will also be flagged as a tampered system at Trent's end.

3.1 Threat Model and Assumptions

It is assumed that Mallory may have installed a backdoor at $\boldsymbol{M_{Alice}}$ which can inform Mallory that an attestation process has been initiated. The backdoor may divert the challenge to another machine inside Mallory's control which can provide the response for the challenge. The backdoor can also utilize dis-assembly tools to determine the operations performed by the challenge. In addition the backdoor may attempt to execute the challenge inside a sandbox to determine the results of the response.

Since Trent is a trusted server, it is assumed that Alice will execute the code provided by Trent. Trent may be the vendor of the binary or a commercial provider of remote attestation service for many binaries. It is also assumed that Alice has a digital signature scheme, which can identified that the executable code was generated by Trent. The attestation code determines the IP address of the client which serves as its machine identifier, due to this it is assumed that Alice is not executing the programs behind a NAT. $\mathcal{C}$ executes OS calls by utilizing software interrupts, due to this it is assumed that the OS on $\boldsymbol{M_{Alice}}$ is not compromised by a *rootkit*. The presence of a *rootkit* would require the use of a VMM or a hardware based checker to determine integrity. It can also be noted that there are many software interrupts in the Linux operating system, due to which it can be assumed that a user level malware will find it difficult to intercept the operations of software interrupts.

It is assumed that $\mathcal{P}$ is not self-modifying code. Any integrity measurement technique cannot obtain measurements on self modifying code as the state of the code section changes with time and execution. Moreover, on computing platforms based on the Intel x86 architecture, the code section is 'write protected' by default, which reduces the scenario of self modifying code existing in common applications.

3.2 Fixing Attack Scenarios

Trent has to prevent Mallory from performing attacks that can compromise the attestation protocol. To achieve this, Trent must incorporate a degree of randomness and obfuscation in the system, so that it is difficult for Mallory to gain

any knowledge about the computed response. Trent randomizes the operations performed by the attestation routine, inserts operations that are not executed, and inserts operations that do not impact the calculation of the response. Trent also checks the state of the open network connections on $\boldsymbol{M_{Alice}}$ to determine whether there are multiple copies of $\boldsymbol{\mathcal{P}}$ executing on $\boldsymbol{M_{Alice}}$. If Mallory executes a clean copy of $\boldsymbol{\mathcal{P}}$ to perform the challenge-response sequence with the server, $\boldsymbol{\mathcal{P}}$ would open a network port to communicate with Trent. In such scenarios $\boldsymbol{\mathcal{C}}$ would observe two ports with the remote connection to Trent, and would flag it as an error. It can be argued that Mallory may force both versions of $\boldsymbol{\mathcal{P}}$ to share the network port. However such sharing would not occur normally on a clean copy of $\boldsymbol{\mathcal{P}}$, or without the explicit intervention of the OS. For the latter to happen, the OS must be corrupted by a *rootkit*, a scenario that is not considered in this work. To determine whether $\boldsymbol{\mathcal{C}}$ was replayed to another machine, Trent obtains machine identifiers of $\boldsymbol{M_{Alice}}$. As a last step Trent utilizes the technique of self-modifying code where some instructions in the challenge are modified post compilation, a separate section is placed in the challenge which fixes these instructions back to their original value during runtime.

4 Performing Remote Attestation

Fig. 2 shows the detailed steps in performing Remote Attestation. Alice makes a verification request and Trent sends a challenge which is the attestation routine. As part of the response Alice has to inject $\boldsymbol{\mathcal{C}}$ at a specified location and execute it. $\boldsymbol{\mathcal{C}}$ determines the machine identifier, socket and port state on the machine, process identifiers, arithmetic checksum on the code section of $\boldsymbol{\mathcal{P}}$, and MD5 hash of the code section of $\boldsymbol{\mathcal{P}}$. It must be noted that these measurements need not be performed in order, for randomization purposes they can be moved around.

4.1 Injection of Code on $\mathcal{P}$

The attestation code $\boldsymbol{\mathcal{C}}$ is injected by $\boldsymbol{\mathcal{P}}$ on itself. This allows $\boldsymbol{\mathcal{C}}$ to execute within the process space of $\boldsymbol{\mathcal{P}}$. This way $\boldsymbol{\mathcal{C}}$ can utilize all descriptors of $\boldsymbol{\mathcal{P}}$ on $\boldsymbol{M_{Alice}}$ without creating new descriptors. The advantage of this is that $\boldsymbol{\mathcal{C}}$ cannot be executed in a sandbox easily and $\boldsymbol{\mathcal{C}}$ can determine whether more than one set of descriptors are present for $\boldsymbol{\mathcal{P}}$.

Implementation. At the client side $\boldsymbol{\mathcal{P}}$ makes a connection request to Trent. Trent responds by providing the size of attestation routine $\boldsymbol{\mathcal{C}}$ followed by the actual executable code to determine the integrity of $\boldsymbol{\mathcal{P}}$. Trent also sends the information on the location inside $\boldsymbol{\mathcal{P}}$ where $\boldsymbol{\mathcal{C}}$ should be placed. $\boldsymbol{\mathcal{P}}$ receives the code and prepares the area for injection by executing the OS call *mprotect* on the area. Once injection is complete, $\boldsymbol{\mathcal{P}}$ creates a function pointer which points to the address of the location and calls $\boldsymbol{\mathcal{C}}$ using the pointer.

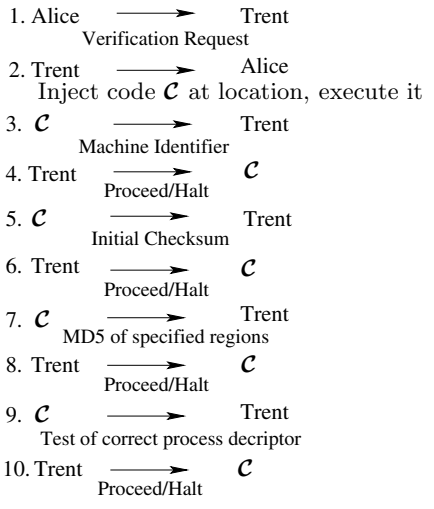

Fig. 2. Detailed steps in Remote Attestation process

4.2 Communication with Trent

The attestation routine does not have any calls to system libraries. This is because libraries may get compromised by an attacker to return incorrect results. In addition, the references to libraries are present at different location in every machine. It is easier to generate interrupts to execute the required functionality instead of placing the correct references to libraries in C. Moreover, a call to a system library may expose the functionality of the code to Mallory. Communication with Trent is achieved by executing the software interrupt with the interrupt number for the OS call *socketcall*.

Implementation. Communication to Trent is achieved by utilizing the socket connection that $\mathcal{P}$ created for an attestation request. All messages are sent to Trent using the *socketcall* system call. ASM code for a network send using *socketcall* is shown in Fig. 3. The routine allocates space on the stack for the parameter, followed by placing the parameters on the stack. The system call number for *socketcall* is 102, which is moved into the A register. The call number for a send in *socketcall* is 9, this value is moved to the B register, the location of the parameters are then moved to the C register and the system call is executed using the interrupt instruction (INT 80). Once the interrupt returns the stack is restored to the original value and the result is obtained in the A register. The functions provided inside *socketcall* is present in the Linux source code in the file $< \boldsymbol{include/linux/net.h} >$.

4.3 Determining Machine Identifiers

To determine that $\mathcal{C}$ was not re-directed to another machine, Trent obtains the machine identifier on which $\mathcal{C}$ executes. Trent had received the request for

```
_asm_ (
        "sub  $16,%%esp\n"
        "movl   %%ebx,(%%esp)\n"
        "movl   %%ecx,4(%%esp)\n"
        "movl   %%edx,8(%%esp)\n"
        "movl   $0,12(%%esp)\n"
        "movl   $102,%%eax\n"
        "movl   $9,%%ebx\n"
        "movl   %%esp,%%ecx\n"
        "int    $0x80\n"
        "add    $16,%%esp\n"
        : "=a" (res)
        :"b" (send_sock), "c" (p_MD5Buf), "d" (len)
       );
```

Fig. 3. *send* routine through *socketcall* in ASM

attestation from Alice, hence has access to the machine IP address from which the request came. $\mathcal{C}$ obtains the IP address of the platform on which it is executing and communicates the result to Trent. Trent compares the two values to determine if the platform in which $\mathcal{C}$ is obtaining results is the same as the platform from which the request came. It can be argued that IP addresses are dynamic; however there is little possibility that any machine will change its IP address in the small time window between Alice requesting a challenge, to measurements being provided to Trent. $\boldsymbol{M_{Alice}}$ is not behind a NAT; hence Trent observes the IP address of $\boldsymbol{M_{Alice}}$ and $\mathcal{C}$ reports the same address. It can be argued that Mallory may have redirected the challenge to another machine ($\boldsymbol{M_{Mallory}}$), and changed the address of the network interface on $\boldsymbol{M_{Mallory}}$ to match that of $\boldsymbol{M_{Alice}}$, but as $\boldsymbol{M_{Alice}}$ is not behind a NAT it would be difficult for Mallory to provide the address to another machine on an external network and achieve successful communication.

Implementation. $\mathcal{C}$ determines the IP address of $\mathrm{M}_{\boldsymbol{Alice}}$ using system interrupts. The interrupt ensures that the address present on the network interface is correctly reported to Trent. This involves loading the stack with the correct operands for the system call, placing the system call number in the A register and the other registers and executing the interrupt instruction. Reading the IP address involves creating a socket on the network interface and obtaining the address from the socket by means another system call *ioctl*. The obtained address is in the form of an integer which is converted to the standard A.B.C.D representation.

4.4 Determining MD5 and Arithmetic Checksum

To determine whether the code section of $\mathcal{P}$ has been tampered, $\mathcal{C}$ computes an MD5 hash on the code section of $\mathcal{P}$. It is possible that since the code section of the binary is available, Mallory may compute the MD5 hash of every possible

boundary region prior to Trent sending a challenge. To prevent this attack, Trent defines sub-regions in the binary and also defines overlaps on the sub-regions before measuring the MD5 hash of the overlapping regions. Overlapping checksums ensure that if by accident the sub-regions are defined identically in two different versions of $\mathcal{C}$, the overlap provides a second set of randomization and ensures that the results of computation produced by $\mathcal{C}$ are different. This also ensures that some random sections of $\mathcal{P}$ are present more than once in the checksum making it more difficult for Mallory to hide any modifications to such regions.

To increase the complexity of the attestation procedure, Trent changes the MD5 measurement to a two phase protocol. MD5 code cannot be randomized, only changes that can be made in it are the changes to the overlapping sub-regions To prevent possible attacks on this protocol, Trent also obtains an arithmetic checksum of the code section of $\mathcal{P}$. The checksum is taken on overlapping sub-regions as described above. The sub-regions defined for the arithmetic checksum are different from the sub-regions defined for obtaining the MD5 hash.

Implementation. The sub-regions on the MD5 hash are defined by Trent in the un-compiled code of the attestation routine using constants. Prior to compilation, Trent runs a pre-processor which generates random numbers to change these constants. The checksum operations are randomized by creating a basic arithmetic operation for a memory location and modifying the basic arithmetic operation to create alternate operations. This provides a pool of operations that can be performed on each memory location. During code generation, one operation is randomly selected for each memory location and placed in the attestation routine. This changes the arithmetic operations performed for every attestation request. The results of these operations are stored temporarily on the stack. Trent changes the pointers on the stack for all the local variables inside $\mathcal{C}$ for every instance.Trent places many instructions that never execute and inserts some operations that are performed on $\boldsymbol{M_{Alice}}$, but not included as part of the results sent back to Trent. Trent also places a time limit (T) within which the response for these computations must be received. The addition of these operations is aimed at making analysis of operations difficult for Mallory within the time frame.

4.5 Determining Process Identifiers

To determine that the attestation routine was not bounced to execute inside a second copy of $\mathcal{P}$, Trent obtains the state of the machine by comparing the open descriptors on $\boldsymbol{M_{Alice}}$ against a known state of a clean machine. Trent knows that in a clean machine there must be only one set of file descriptors that are used by $\mathcal{P}$. If there are multiple copies of the descriptors used by $\mathcal{P}$, then an error is reported to Trent. $\mathcal{C}$ identifies descriptors that match the known descriptors used by $\mathcal{P}$ and determines the process utilizing these descriptors in the system. If the process utilizing these descriptors are the same as the process inside which $\mathcal{C}$ executes then an OK state is sent to Trent.

Implementation. $\mathcal{C}$ obtains the pid of the process ($\mathcal{P}_0$) under which it is executing using the system interrupt for *getpid*. It locates all the remote connections established to Trent from M_{Alice}. This is done by reading the contents of the '/proc/net/tcp' file. The file has a structure shown in Fig. 4. This file has some more fields which are omitted from the figure. Once all the connections are identified, $\mathcal{C}$ utilizes the *inode* of each of the socket descriptor to locate any process utilizing it. This is done by scanning the '/proc/$<$ *pid* $>$/fd' folder for all the running processes on M_{Alice}. In the situation that $\mathcal{P}$ is not corrupted, there should be only one process id ($\mathcal{P}_0$) utilizing the identified inode. If it encounters more than one such process, then it sends an error message back to Trent.

sl	local_address	rem_address	st	tx_queue	rx_queue	tr	tm−>when	retrnsmt	uid	timeout	inode
0:	0100007F:1F40	00000000:0000	0A	00000000:	00000000	00:	00000000	00000000	0	0	5456
1:	00000000:C3A9	00000000:0000	0A	00000000:	00000000	00:	00000000	00000000	0	0	4533
2:	00000000:006F	00000000:0000	0A	00000000:	00000000	00:	00000000	00000000	0	0	4473
3:	0100007F:0277	00000000:0000	0A	00000000:	00000000	00:	00000000	00000000	0	0	5690
4:	0100007F:0019	00000000:0000	0A	00000000:	00000000	00:	00000000	00000000	0	0	5358
5:	0100007F:743A	00000000:0000	0A	00000000:	00000000	00:	00000000	00000000	0	0	5411

Fig. 4. Contents of /proc/net/tcp file

4.6 Generation of Attestation Code

To prevent Mallory from analyzing operations performed by $\mathcal{C}$, Trent places some obfuscations and randomizations inside the generated code. In addition, a time threshold (T) is created, if $\mathcal{C}$ does not respond back in a stipulated period of time (allowing for network delays), Trent informs Alice of a possible compromise. To prevent Mallory from performing any dis-assembly based analysis on $\mathcal{C}$, Trent can also choose to remove some instructions in $\mathcal{C}$ while sending the code to M_{Alice}. A code restore section is placed inside $\mathcal{C}$ such that during execution this section changes the modified instructions to the correct values. This makes it difficult for Mallory to determine the exact contents of $\mathcal{C}$ without executing it.

Implementation. Trent generates $\mathcal{C}$ for every instance of verification request. The source code of $\mathcal{C}$ is divided into four blocks, independent of each other. Trent assigns randomly generated sequence numbers to the four blocks and places them accordingly inside $\mathcal{C}$ source code. $\mathcal{C}$ allocates space on the local stack to store computational values. Instead of utilizing fixed locations on the stack, Trent replaces all variables inside $\mathcal{C}$ with pointers to locations on the stack. To allocate space on the stack Trent declares a large array of type char of size N, which has enough space to hold contents of all the other variables simultaneously. Trent executes a pre-processor which assigns locations to the pointers. Trent may also choose to change the instructions inside the executable code such that they cause analysis tools to produce incorrect results. $\mathcal{C}$ contains a section ($\mathcal{C}_{restore}$) which changes these modified instructions back to their original contents when it executes. $\mathcal{C}_{restore}$ contains the offset from the current location and the value to be

Table 1. Average code generation time at server end

Machine	Test generation time (ms)	Compilation time (ms)	Total time (ms)
Pentium 4	12.3	320	332
Quad Core	5.2	100	105

placed inside the offset. Trent places information to correct the modified instructions inside $\mathcal{C}_{restore}$. $\mathcal{C}_{restore}$ is executed prior to executing other instructions inside $\mathcal{C}$ and $\mathcal{C}_{restore}$ corrects the values inside the modified instructions.

5 Results

The remote attestation scheme was implemented on Ubuntu 8.04 (Linux 32 bit) operating system using the gcc compiler; the application $\mathcal{P}$ and attestation code $\mathcal{C}$ were written in the C language. The time threshold (T) is an important parameter in this implementation. The value of T must take into account network delays. Network delays between cities in IP networks are of the order of a few milliseconds [18]. Measuring the overall time required for one instance of Remote Attestation and adding a few seconds to the execution time can suffice for the value of T. The performance of the system was measured by executing the integrity checks on the source code for VLC media player interface [19]. Some sections of the program were removed for compilation purposes. The performance of the system was measured on two pairs of systems. One pair of machines were legacy machines executing on an Intel Pentium 4 processor with 1 GB of ram, and the second pair of machines were Intel Core 2 Quad machine with 3 GB of ram. The tests measured were the time taken to generate code including compile time, time taken by the server to do a local integrity check on a clean copy of the binary and time taken by the client to perform the integrity measurement and send a response back to the server. The time taken for compiling the freshly generated code is reported in Table 1. As expected, the Pentium 4 machine has slightly lower performance than a platform with 4 Intel Core 2 processors.

The integrity measurement code $\mathcal{C}$ was executed locally on the server and sent to the client for injection and execution. The time taken on the server to execute is the time the code will take to generate integrity measurement on the client as both machines were kept with the same configuration in each case. These times are reported in Table 2. As the code takes only in the order of milliseconds to execute on the client platform, the value for T can be set in the order of a few seconds to allow for network delays.

It can be observed from Table 1 that it takes an order of a few hundred milliseconds for the server to generate code, while from Table 2, it can be observed that the integrity measurement is very light weight and returns results in the order of a few milliseconds. Due to this, the code generation process can be viewed as a huge overhead. However, the server need not generate new code for every instance of a client connection. It can generate the measurement code

Table 2. Time to compute measurements

Machine	Server side execution time (ms)	Client side execution time (ms)
Pentium 4	0.6	22
Quad Core	0.4	16

periodically every second and ship out the same integrity measurement code to all clients connecting within that second. This can alleviate the workload on the server.

6 Extended Verified Code Execution

Once Remote Attestation determines the integrity of a program, the server begins communication and sharing of sensitive data with the client program. However, Mallory may choose to wait till the attestation process is completed, then substitute the client program $\mathcal{P}$ with a corrupted program $\mathcal{P}_c$. To prevent Mallory from achieving this, Trent has to obtain some guarantee that the process that was attested earlier is the same process performing the rest of the communication. Trent cannot make any persistent changes to the binary as Mallory would detect these changes under the current threat model. Trent has to change the flow of execution from normal in the client process such that the sequence of events reported will allow Trent to determine whether the previously attested process is executing.

As discussed before, Trent knows the layout of the program $\mathcal{P}$. At the end of Remote Attestation, Trent sends a new group of messages to $\mathcal{C}$. The message contains some code executable code $\mathcal{F}_1$ that Trent instructs $\mathcal{C}$ to place at a particular location in $\mathcal{P}$. Trent also instructs $\mathcal{C}$ to modify a future function call

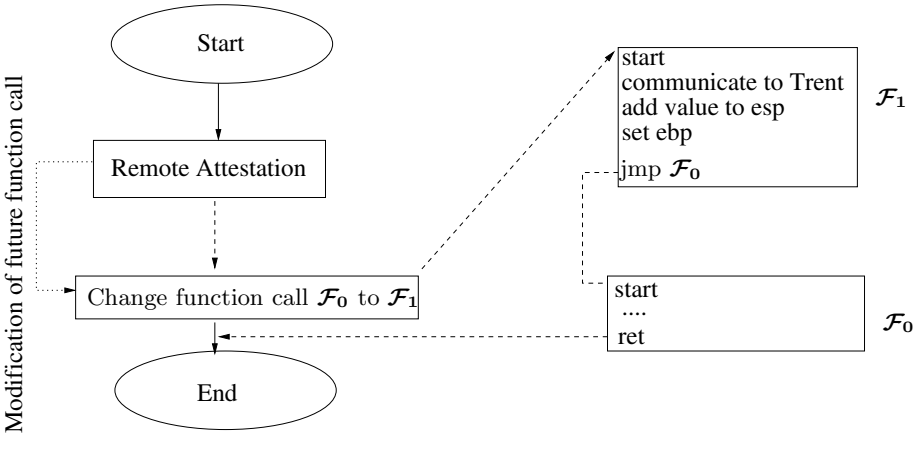

Fig. 5. Change of flow of execution

$\mathcal{F}_0$ in $\mathcal{P}$ such that instead of calling $\mathcal{F}_0$, $\mathcal{P}$ calls $\mathcal{F}_1$. $\mathcal{F}_1$ communicates back to Trent, this way Trent knows that the copy of $\mathcal{P}$ which was attested in the previous step is still executing. At the end of its execution, $\mathcal{F}_1$ undoes all its stack operation and jumps to the address where $\mathcal{F}_0$ is located. This would constitute extended verified code execution and can be seen in Fig. 5. This work can be pursued as a future work of remote attestation.

7 Conclusion and Future Work

This paper implements a method for implementing *Remote Attestation* entirely in software. A number of other schemes in literature that address the problem of program integrity checking were presented as related work. This work reduces the window of opportunity for the attacker Mallory to provide fake results to the trusted authority Trent by implementing various forms of obfuscation and providing new executable code for every run. This scheme was implemented on Intel x86 architecture using Linux and its performance was measured. This paper also presented future work to be pursued to detect compromises that occur after the completion of remote attestation.

Acknowledgements. This material is based upon work supported in part by the National Science Foundation under Grant No. CNS-1011931. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF.

References

1. Basili, V., Perricone, B.: Software errors and complexity: an empirical investigation0. Communications of the ACM 27(1), 52 (1984)
2. Chou, A., Yang, J., Chelf, B., Hallem, S., Engler, D.: An empirical study of operating systems errors. In: Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles, pp. 73–88. ACM, New York (2001)
3. Cohen, F.: Operating system protection through program evolution* 1. Computers & Security 12(6), 565–584 (1993)
4. Garay, J., Huelsbergen, L.: Software integrity protection using timed executable agents. In: Proceedings of the 2006 ACM Symposium on Information, Computer and Communications Security, pp. 189–200. ACM, New York (2006)
5. Garfinkel, T., Pfaff, B., Chow, J., Rosenblum, M., Boneh, D.: Terra: A virtual machine-based platform for trusted computing. ACM SIGOPS Operating Systems Review 37(5), 206 (2003)
6. Goldman, K., Perez, R., Sailer, R.: Linking remote attestation to secure tunnel endpoints. In: Proceedings of the First ACM Workshop on Scalable Trusted Computing, p. 24. ACM, New York (2006)
7. Kennell, R., Jamieson, L.: Establishing the genuinity of remote computer systems. In: Proceedings of the 12th USENIX Security Symposium. pp. 295–308 (2003)
8. Ostrand, T., Weyuker, E.: The distribution of faults in a large industrial software system. In: Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis, p. 64. ACM, New York (2002)

9. Petroni Jr., N., Fraser, T., Molina, J., Arbaugh, W.: Copilot-a coprocessor-based kernel runtime integrity monitor. In: Proceedings of the 13th Conference on USENIX Security Symposium, vol. 13, p. 13. USENIX Association (2004)
10. Sahita, R., Savagaonkar, U., Dewan, P., Durham, D.: Mitigating the Lying-Endpoint Problem in Virtualized Network Access Frameworks. In: Clemm, A., Granville, L.Z., Stadler, R. (eds.) DSOM 2007. LNCS, vol. 4785, pp. 135–146. Springer, Heidelberg (2007)
11. Sailer, R., Zhang, X., Jaeger, T., Van Doorn, L.: Design and implementation of a TCG-based integrity measurement architecture. In: Proceedings of the 13th USENIX Security Symposium, pp. 223–238 (2004)
12. Seshadri, A., Perrig, A., Van Doorn, L., Khosla, P.: Swatt: Software-based attestation for embedded devices. In: Proceedings of 2004 IEEE Symposium on Security and Privacy, pp. 272–282. IEEE, Los Alamitos (2004)
13. Seshadri, A., Luk, M., Shi, E., Perrig, A., van Doorn, L., Khosla, P.: Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. ACM SIGOPS Operating Systems Review 39(5), 1–16 (2005)
14. Shaneck, M., Mahadevan, K., Kher, V., Kim, Y.: Remote software-based attestation for wireless sensors. In: Molva, R., Tsudik, G., Westhoff, D. (eds.) ESAS 2005. LNCS, vol. 3813, pp. 27–41. Springer, Heidelberg (2005)
15. Srinivasan, R., Dasgupta, P.: Towards more effective virus detectors. Communications of the Computer Society of India 31(5), 21–23 (2007)
16. Stumpf, F., Tafreschi, O., Röder, P., Eckert, C.: A robust integrity reporting protocol for remote attestation. In: Second Workshop on Advances in Trusted Computing, WATC 2006 Fall, Citeseer (2006)
17. Wang, L., Dasgupta, P.: Coprocessor-based hierarchical trust management for software integrity and digital identity protection. Journal of Computer Security 16(3), 311–339 (2008)
18. Web-link: Global ip network latency, `http://ipnetwork.bgtmo.ip.att.net/pws/network_delay.html` (retrieved on January 17, 2010)
19. Web-link: Vlc media player source code ftp repository, `http://download.videolan.org/pub/videolan/vlc/` (retrieved on February 24, 2010)