Software based remote attestation for OS kernel and user applications

Raghunathan Srinivasan
Arizona State University
Tempe, AZ, USA
Email: raghus@asu.edu

Partha Dasgupta
Arizona State University
Tempe, AZ, USA
Email: partha@asu.edu

Tushar Gohad
Monta Vista Software, LLC

***Abstract*—This paper describes a software based remote attestation scheme for providing a root of trust on an untrusted computing platform. To provide a root of trust, this work focuses on obtaining the integrity of the OS running on the platform, and then leverages the techniques to obtain the integrity of a user application. A trusted external entity issues a challenge to the client platform. The challenge is executable code which the client must execute, and the code generates results which are sent to the external entity. These results provide the external entity an assurance as to whether the client application and the OS at the client end are in pristine condition. This work also presents a technique where it can be verified that the application which was attested, did not get replaced by a different application once the challenge got completed.**

***Index Terms*—Remote attestation, integrity measurement, device drivers, code injection**

I. Introduction

A consumer computing platform can be compromised by malicious logic in many different ways. Preventing compromises requires safe coding, developing secure Operating Systems (OS), and developing secure kernel modules. Fault densities in OS kernels can range from 2 - 75 per 1000 Lines of Code (LOC) [1]. OS kernels are often supplemented by many device drivers or kernel modules which have higher error rates [2]. Buffer overflow is a common vulnerability that exists in many application software which may allow malware to compromise systems. Operating systems do not offer isolations to user space programs from one another [3], in addition a rogue kernel module or a device driver can overwrite any section of kernel memory. The use of anti-malware software is a workaround the problem of protecting the operating system as anti-malware software can only detect known malicious logic. This is because they are primarily signature based scanners, in addition, smart malware can render these detection techniques ineffective. Consequentially, a user (Alice) has to request integrity measurement of the platform from an entity that operates beyond the bounds of the operating system.

Remote attestation is a set of protocols that uses a trusted service to probe the memory of a client computer to determine whether one (or more) application has been tampered with or not. Primarily used for DRM, these techniques can be extended to determine whether the integrity of the entire system has been compromised. Remote attestation has been implemented using Hardware devices, VMM and software based techniques. The Trusted Platform Module (TPM) chip has been used extensively to build hardware based solutions. In most cases, some integrity measurement values are stored in the Platform Configuration Registers (PCR) and anytime a measurement is to be taken a private key stored in the hardware is used to sign the integrity values read from the system software [4], [5]. The TPM based schemes are fairly robust for determining the integrity of applications during system initialization. The TPM based schemes lack the ability to address scenarios where these applications may get compromised in memory [6].

Software based solutions for Remote Attestation vary in their implementation technique. Most methods involve taking a mathematical or a cryptographic checksum over a section of the program in question ($\mathcal{P}$), and reporting the results of verification to a trusted external server (Trent) [7], [8]. TEAS [9] proves mathematically that it is difficult for an attacker to forge integrity results obtained on a client platform, provided the integrity measurement code changes for every attestation instance, however, an implementation framework is not provided in the work.

To provide a secure computing platform a root of trust needs to be established on the client machine. To provide a root of trust on a platform, this work presents two schemes. The first sceme presents methods to obtain the integrity of the OS text section, system call table and the Interrupt Descriptor Table (IDT). Once the attestation is completed for the OS areas, the attestation proceeds with the second scheme which measures the integrity of a particular client application. The second scheme builds on a prior publication [10] and adds a component where an external agent can determine whether the application that was attested earlier is still executing. The OS provides system call interface which is extensively used by the user application scheme, this way the OS serves as a root of trust The user application attestation scheme is summarized as follows. Alice has a user level application $\mathcal{P}$ that Alice wants attested. Trent is a trusted server who generates challenges that determine whether $\mathcal{P}$ is compromised or pristine. The challenge is executable code $\mathcal{C}$ which is injected on $\mathcal{P}$ at Alice's machine (M_{Alice}). For details, refer to the prior publication [10]. The extension provided to the prior publication is explained as follows. It is possible that once *Remote Attestation* is completed for the user application, Mallory may replace the attested software ($\mathcal{P}$) with a corrupted

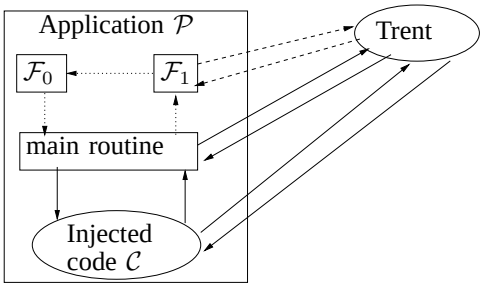

Fig. 1. Overview of verified code execution

version ($\mathcal{P}'$). Alice would have no knowledge of such a change as long as $\mathcal{P}'$ performs all the functionalities of $\mathcal{P}$. To prevent Mallory from achieving such attacks, a framework for *verified code execution* is presented in this paper. This involves server making some changes to the code section of $\mathcal{P}$ during remote attestation. Trent uses $\mathcal{C}$ to change a function call in $\mathcal{P}$ to call a new function $\mathcal{F}_1$ instead of calling $\mathcal{F}_0$. When the changed section of $\mathcal{P}$ executes it communicates back to Trent. This communication tells Trent that the attested program indeed completed execution. All changes made by Trent to the attested program are non-persistent and remain in-core to prevent Mallory from analyzing the changes made to $\mathcal{P}$. In addition, this keeps the binary image of $\mathcal{P}$ unmodified. Fig. 1 shows the overview of the verified code execution process.

For the OS attestation part this paper provides a framework where an external entity can determine the state of the OS Text section, System call table, and the IDT on the client machine entirely in software. This is achieved by providing newly generated code to a kernel module which executes the provided code. The response from the code indicates whether the OS is compromised. It may be argued that once the OS is attested, it can be used to attest the application rendering the second scheme redundant. However, many applications execute on a client platform, and each gets updated frequently. If the OS performs integrity measurement on each binary, for security requirements the definitions should reside somewhere in the protected space. These definitions will have to be updated in the protected space frequently as each software gets updated. This can be considered a major overhead in the system. Instead of this, the simpler solution is to have an external agent such as the application vendor, or a network administrator provide integrity measurement for the applications as the definitions need to be updated only at one location.

Remote attestation for both schemes is implemented in this work by downloading and executing new instance of attestation code for every attestation request by the client (Alice). New code makes it difficult for the attacker (Mallory) to forge any results generated by the integrity checking mechanism. This is because Mallory has no prior knowledge of the structure or content of the code, as a result of which, Mallory has no knowledge of the operations performed by the downloaded code. To launch a successful attack, Mallory would have to perform an 'impromptu' analysis of the executable code.

The remainder of this work is organized as follows, section II describes the related work for integrity measurement, section III describes the threat model for each framework, section VI-A describes the software based remote attestation, section VI-B presents the design of the verified code execution component, section V describes the kernel integrity measurement scheme, section VI-C presents the implementation of all the above works and section VIII concludes the paper.

II. Related Work

Code attestation involves determining whether the process executing is in the same state after it was installed or has been modified by any malicious code. This section discusses various schemes implemented previously to perform attestation.

Some hardware based schemes that determine the integrity of a client platform operate off the TPM chip provided by the Trusted Computing Group [4], [5]. These schemes may involve the kernel or an application executing on the client obtaining memory reads, 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 TPM scheme may be capable of providing secure bootstrap, but subsequent deployment of malware can go undetected based on the implementation of the protocol. TPM based solutions have the stigma of Digital Rights Management (DRM), can be difficult to reprogram, and are not well suited for mass deployment.

Co-processor schemes that are installed on the PCI slot of the PC have been used to measure the integrity of the kernel as mentioned in section 2.1. One scheme [3] computes the integrity of the kernel at installation time and stores this value for future comparisons. The core of the system lies in a co-processor (SecCore) that performs integrity measurement of a kernel module during system boot. The kernel interrupt service routine (SecISR) performs integrity checks on a kernel checker and a user application checker. The kernel checker proceeds with attesting the entire kernel .TEXT section and modules. The system determines that during installation for the machine used for building the prototype, the .TEXT section began at virtual address 0xC0100000 which corresponded to the physical address 0x00100000, and begin measurements at this address. The Copilot [11] is a hardware coprocessor that constantly monitors the host kernel integrity. It cannot handle dynamic kernel modules and user-level applications and it does not have a mechanism for a kernel patch.

Integrity Measurement Architecture (IMA) [12] is a software based integrity measurement scheme that utilizes the underlying TPM on the platform to measure the integrity of applications that are loaded on the client machine. IMA maintains a list of integrity values of many files in the system. When an executable, library, or kernel module is loaded, IMA performs an integrity check prior to executing it. IMA does not provide means to determine whether any program that is in execution got tampered in memory. Terra uses a trusted virtual machine monitor (TVMM) and partitions the hardware platform into multiple virtual machines that are isolated from one another [13]. Terra is installed in one of the VMs (TVMM)

and is not exposed to external applications like mail, gaming, and so on. The TVMM takes measurements on the VMs prior to loading them. The TVMM acts as a root of trust on the machine.

VIS [14] is a hardware assisted (Intel VT-x) virtualization scheme which determines the integrity of client programs that connect to a remote server. VIS contains an Integrity Measurement Module which reads the cryptographically signed reference measurement (manifest) of a client process. VIS requires that the pages of the client programs are pinned in memory (not paged out), VIS restricts network access during the verification phase to prevent any malicious program from avoiding registration.

Detecting kernel level rootkits from within the Operating System requires scanning the kernel memory to determine whether or not the system call table and other sections in the high memory has been modified [15]. Many rootkits change the pointers of the system calls present in the system call table. This causes execution to jump to rootkit code instead of OS code during system call invocation. To detect these changes an initial measurement is stored which contains the state of the kernel memory in pristine state. During attestation the runtime memory is scanned using the '/dev/kmem' file. The runtime memory is compared against the stored pristine state to determine whether a rootkit has modified the kernel image.

In *Pioneer* [7], the integrity measurement is done entirely in software. 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 performs attestation on an entity within the machine and transfers control to it. *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 a copy of the client code in a sandbox and obtain integrity measurements from it. In *TEAS* [9], 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.

Genuinity [8] implements a remote attestation system in which the client kernel initializes the attestation for a program. It receives executable code and maps it as directed by the trusted authority. The executable code performs various checks on the client program, returns the results to 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. A problem with this scheme is that the results are not communicated to the server by the downloaded code. This may allow a malicious OS to analyze and modify certain values that the code computes. Genuinity has been shown to have weaknesses by two works [16], [17]. The first work suggests placing attack code on the same physical page as the checksum code such that the attack code resides on the zero-filled locations in the page. Authors of Genuinity countered these findings by stating that the attack scenario does not take into account the time required to extract test cases from the network, analyze it, find appropriate places to hide code and finally produce code to forge the checksum operations [18]. The attacks would require complex re-engineering to succeed against all possible test cases. The second work suggests that Genuinity reads 32 bit words for performing a checksum and hence will be vulnerable if the attack is constructed to avoid the lower 32 bits of memory regions. This claim is countered by the authors of Genuinity where they state that genuinity reads 32 bits at a time, and not the lower 32 bits of an address.

Every attack scenario has its limitations as countered by the authors of Genuinity. In this paper, remote attestation is implemented by downloading new (randomized and obfuscated) attestation code for every instance of the operation. This operation makes it difficult for the attacker to forge any results that are produced by the attestation code. 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.

Program analysis requires disassembly of code and the control flow graph (CFG) generation. CFG generation involves identifying blocks of code such that they have one entry point and only one branch instruction with target addresses. The execution time of these algorithms is non-linear (n^2) [19]. The linux tool 'objdump' is one of the simplest linear sweep tools that perform disassembly. It moves through the entire code once, disassembling each instruction as and when encountered. This method suffers from a weakness that it misinterprets data embedded inside instructions hence carefully constructed branch statements induce errors [20]. Linear sweep is also susceptible to insertion of dummy instructions and self modifying code. Recursive Traversal involves decoding executable code at the target of a branch before analyzing the next executable code in the current location. This technique can also be defeated by opaque predicates [21], where one target of a branch contains complex instructions which never execute.

III. Threat Model and Attack Scenarios

This paper presents two schemes to attest the integrity of applications on an untrusted platform. The first scheme presents a method to perform remote attestation of the OS text

section, system call table, and the Interrupt descriptor table (IDT). The second scheme presents a method to attest the integrity of a user application and to means to check whether the verified application continued execution or got replaced by an attacker. For the sake of simplicity, the user application scheme is discussed prior to discussing the kernel attestation scheme.

For the kernel attestation it is assumed that the kernel may be compromised, system call tables may be corrupted, and malware may change interrupt descriptors. The kernel attestation scheme measures the integrity of the OS text section, Interrupt descriptor table and system call table. It is assumed for the kernel attestation scheme that the trusted server (Trent′) is the OS vendor or a corporate network administrator who has clear knowledge about the mappings in the clean OS image.

For the user application attestation we assume that the client machine may be infected with user level malware, but not infected with a kernel level rootkit as these infections would be revealed by the kernel attestation scheme. We assume that Mallory (the attacker) can start a clean copy of Alice's installed application $\mathcal{P}$ to execute it in a controlled environment. Mallory can also attempt to re-direct the challenge to another machine which runs a clean copy of $\mathcal{P}$. We also assume that the client machine M_{Alice} is not running behind a NAT. This assumption is made as $\mathcal{C}$ takes measurements on M_{Alice} to determine if it is the same machine that contacted Trent. If M_{Alice} is behind a NAT the connections would appear as coming from a router and not the machine, while $\mathcal{C}$ would respond with the machine IP. It can also be noted that in case Trent is a network administrator, the NAT does not come into play at all.

In both schemes we assume that Alice will trust Trent to provide non malicious code since Trent is a trusted server, and Alice has means such as digital signatures that verify that $\mathcal{C}$ was generated by Trent.

IV. Generating code for attestation

Trent is a trusted server that provides integrity measurement code $\mathcal{C}$ to Alice. Alice injects the code on the user application $\mathcal{P}$. $\mathcal{P}$ transfers control to $\mathcal{C}$ and allows it to report measurements to Trent. Trent must prevent Mallory from analyzing the operations performed by $\mathcal{C}$. To achieve this, Trent can utilize a combination of obfuscation techniques which are summarized in this section. They are described in detail in our previous publication [10].

Fig. 2 shows a sample snippet of the $\mathcal{C}$ mathematical checksum code. The send function used in the checksum snippet is implemented using inline ASM. The checksum is dependent on the constants defined in the source code of $\mathcal{C}$. These constants It is evident that in order to forge any results, Mallory must determine the value of checksum2 being returned to Trent. This requires that Mallory identify all the instructions modifying checksum2 and the locations on stack that it uses for computation. Since constants like x are changed during compilation, the checksum being sent out differs in every instance.

```
{
  ....
  N = memory range
  x,y,z = <random values placed
         during compilation>

  a = 0;
  while (a<N){
      checksum1 += Mem[a];
      if ((a  % y) == 0)
            checksum2 += checksum1/x;
       a = a -z;
       a++;
  }
  send checksum2;
  ....
 }
```

Fig. 2. Snippet from the checksum code

Similarly the other two constants 'y' and 'z' defines the subregions and the overlap on which the checksum are taken. These constants effect the overall output of the checksum. For MD5 of the application memory, x is not generated, however y and z are generated to change the regions on which the MD5 values are obtained.

To prevent Mallory from analyzing the injected code, certain obfuscations are placed in $\mathcal{C}$ such as changing the flow of execution, changing locations of variables on the stack, inserting dummy instructions, and placing incorrect executable code in C. The incorrect code is fixed just prior to execution at M_{Alice} during attestation. Further details on these obfuscations are present in our prior publication.

Trent also maintains a time threshold (T) by which the response from M_{Alice} is expected. If $\mathcal{C}$ does not respond in a stipulated period of time (allowing for network delays), Trent will know that something went wrong at M_{Alice}. This includes denial of service based attacks where Trent will inform Alice that $\mathcal{C}$ is not communicating back.

V. Remote Kernel attestation

To measure the integrity of the kernel, this work implements a scheme which executes newly injected code that determines and provides results back to the trusted server. The difference between the user application attestation and the kernel attestation lies in its implementation. $Trent'$ is a trusted server who provides kernel integrity measuement code ($\mathcal{C}_{kernel}$) to Alice. The code is received on M_{Alice} by a user level application $\mathcal{P}_{user}$ which has been provided by $Trent'$ previously. $\mathcal{P}_{user}$ provides the code to the Kernel on Alice's machine using a series of *ioctl* calls. $\mathcal{C}_{kernel}$ is received into a kernel module $\mathcal{P}_{kernel}$. It is assumed that Alice has means such as digital signature verification scheme to verify that the code was sent by $Trent'$. On execution, $\mathcal{C}_{kernel}$ obtains integrity measurements (H_{kernel}) on the OS Text section,

system call table, and the interrupt descriptors table. These results are passed to the userland application which forwards them to $Trent'$. These results can be encrypted using a simple one time pad scheme if required, however as the executable code is generated for each instance, this is not a required operation.

$Trent'$ is assumed to be a corporate network administrator, or an OS vendor; hence $Trent'$ has access to a pristine copy of the kernel executing on M_{Alice} on which the expected integrity values to be generated by $\mathcal{C}_{kernel}$ are obtained. Although this seems like the trusted server would need infinite memory requirements to keep track of every client, most OS installations are identical as they are installed 'off the shelf'.

A. Implementation

The framework for this paper was implemented on an x86 based Ubuntu 8.04 machine executing a 2.6.24-28-generic kernel. This section describes the integrity measurement of the OS text section (which contains the code for system calls and other kernel routines), the system call table and the interrupt descriptor table. The overall scheme is similar to the techniques described in [15].

In Linux, the exact identical copy of the kernel is mapped to every process in the system. The Text section was found at virtual memory 0xC0100000. The end of kernel text section was located to be at 0xC03219CA. The system call table was located at 0xC0326520 and was found to be 1564 bytes. The Interrupt descriptor table was located at 0xC0410000 to be of size 2048 bytes. At the server side the $\mathcal{C}_{kernel}$ is compiled as part of a device driver (kernel module). The driver has an ioctl which on being called by a userland application provides the executable code of $\mathcal{C}_{kernel}$ as a character buffer. This buffer is sent to persistant storage along with the integrity measurement value expected from it for a particular kernel. For the sake of simplicity, the server and the client had the same OS image during our tests. This process is repeated to generate a large number of $\mathcal{C}_{kernel}$ routines. On an attestation instance, any one of the generated $\mathcal{C}_{kernel}$ routines is read and loaded in memory. This process is essential as the loader removes certain instructions specific to 'ftrace' and replaces them with multi byte no-ops during module insertion.

The trusted server Trent$'$ communicates to a user level application $\mathcal{P}_{user}$. $\mathcal{P}_{user}$ can be assumed to be an application provided by Trent$'$. $\mathcal{P}_{user}$ communicates to a kernel module $\mathcal{P}_{kernel}$ (also provided by Trent$'$) using a character device called 'remote_attestation_device' which is created using a command 'mknod /dev/remote_attestation_device c 100 0'. The last two numbers in the command provide the MAJOR_NUM and MINOR_NUM for the device. At the client end, kernel modules can be relocated, hence the target to function calls can be potentially different during each boot instance. To execute any function in the kernel module $\mathcal{P}_{kernel}$ which are essential, $Trent'$ obtains their addresses using various ioctl calls through $\mathcal{P}_{user}$, one for each such address. Once $\mathcal{C}_{kernel}$ is brought into memory at the server end during attestation initiation, the call instructions are identified in the code and the correct target address is patched on the call instruction. The patching routine essentially performs function as shown in figure 3. The address of routine 2 is the routine which $\mathcal{C}_{kernel}$ will call to perform any function. The 'code_in_mem_buffer' is an array that holds the $\mathcal{C}_{kernel}$ routine prior to send to the client, and 'call_locations' is an array which holds the offsets in $\mathcal{C}_{kernel}$ where the call instructions are present.

```
jump_target = - ((address_C_kernel
                  + jump_locations[1]
                  + length_ofcall)
                  - address_routine2);
code_in_mem_buffer[call_locations[index]
                                      +1]
                              = jump_target;
```

Fig. 3. Fixing call targets in $\mathcal{C}_{kernel}$

The patched $\mathcal{C}_{kernel}$ is then sent to $\mathcal{P}_{user}$. $\mathcal{P}_{user}$ executes another ioctl which receives $\mathcal{C}_{kernel}$ and places it in the location specified. After the code is injected, $Trent'$ issues a message to $\mathcal{P}_{user}$ requesting the kernel integrity measurements. $\mathcal{P}_{user}$ executes an ioctl which causes the $\mathcal{P}_{kernel}$ to execute the injected code. $\mathcal{C}_{kernel}$ reads various memory locations in the kernel and passes the data to the MD5 code, $\mathcal{C}_{kernel}$ also computes arithmetic checksums on the read data. The MD5 code returns the MD5 checksum value to $\mathcal{C}_{kernel}$ which in turn returns the value to the ioctl handler in the $\mathcal{P}_{kernel}$. $\mathcal{P}_{kernel}$ then passes the MD5 and arithmetic checksum computations back to $\mathcal{P}_{user}$ which forwards the results to the $Trent'$.

The measurement code also obtains the location of the system call table in memory by processing the 'SIDT' instruction. The SIDT instruction contains the runtime location of the system call table to be processed by the software interrupt (INT 80 instruction). The code to process the runtime system call table was obtained from [22] is shown in figure 4 and is explained as follows. The IDTR structure is used to store the base address of the IDT. The Idt_descriptor structure is used to store the contents at a particular IDT descrptor entry. The 'SIDT' instruction is parsed to obtain the location of the system call. The address of the system call is a 32 bit value. The high descriptor value is left shifted by 16 bits to place in bits 31:16, then the low descriptor value is slotted in bits 15:0 by doing a bitwise OR operation.

If required the disable interrupt instruction can be issued by $\mathcal{C}_{kernel}$ to prevent any other process from obtaining hold of the processor. It must be noted that in multi processor systems, the disable interrupt instruction (CLI) may not prevent a second processor from smashing kernel integrity measurement values. However, as the test cases are different for every attestation instance, Mallory may not gain anything by smashing integrity measurement values.

```
struct Idt_Descriptor {
    unsigned short offset_low;
    unsigned short selector;
    unsigned char zero;
    unsigned char type_flags;
    unsigned short offset_high;
} __attribute__ ((packed));

struct IDTR {
    unsigned short limit;
    void *base;
} __attribute__ ((packed));

struct IDTR idtr;
struct Idt_Descriptor idtd;
void *system_call;
asm volatile("sidt %0" : "=m"(idtr));
memcpy(&idtd, idtr.base +
         0x80*sizeof(idtd), sizeof(idtd));
system_call_table_location = (void*)
((idtd.offset_high<<16) | idtd.offset_low);
```

Fig. 4. Determining runtime location of system call table

It can be noted that even if the client does not communicate the address of the functions, $\mathcal{P}_{kernel}$ can be designed such that the MD5 driver provided by $Trent'$ and the MD5 code reside on the same page. This means that the higher 20 bits of the address of the MD5 code and the downloaded code will be the same and only the lower 12 bits would be different. This allows the $Trent'$ to determine where $\mathcal{C}_{kernel}$ will reside on the client machine, and automatically calculate the target address for the MD5 code.

VI. Attestation of user application

A. Software based remote attestation

This section contains a brief overview of the user application attestation scheme, further details are present in our prior publication [10]. Alice (the user) has previously installed an application ($\mathcal{P}$). Alice wants to get the application attested to determine whether it has been compromised. Alice contacts a Trusted authority Trent who provides remote attestation services. When Alice contacts Trent using $\mathcal{P}$, the remote attestation starts. The trusted server (Trent) has prior knowledge of the code structure and process image of $\mathcal{P}$ of $\mathcal{P}$. Trent provides code ($\mathcal{C}$) generated on the fly to perform an integrity check on it. To prevent a replay based attack, Trent changes the operations performed by $\mathcal{C}$ in every attestation instance. Generating code on the fly makes it difficult for Mallory (the attacker) to determine the results of the computations performed by $\mathcal{C}$ beforehand. The attestation code $\mathcal{C}$ is injected by $\mathcal{P}$ on itself. This allows $\mathcal{C}$ to execute within the process space of $\mathcal{P}$, utilizing all descriptors of $\mathcal{P}$ on M_{Alice} without creating new ones. The advantage of this is that $\mathcal{C}$ can determine whether more than one set of descriptors are present for $\mathcal{P}$.

When $\mathcal{P}$ runs $\mathcal{C}$, $\mathcal{C}$ hashes the code section of $\mathcal{P}$ in random overlapping sections and sends the measurement value M_1 to Trent. The overlapping subregions are defined in the code of $\mathcal{C}$ and is changed in every attestation instance to make it difficult for a compromised $\mathcal{P}$ to mimic $\mathcal{C}$'s behavior and forge the results that would be generated by $\mathcal{C}$ on a clean instance of $\mathcal{P}$. Some additional information in M_1 allows Trent to determine whether the challenge was replayed, or performed inside a controlled environment. Trent has a local copy of $\mathcal{P}$ on which the same sets of tests are executed as above to produce a value M_0. When Trent gets the results from $\mathcal{C}$, Trent compares M_1 and M_0; if the two values are the same then Alice is informed that $\mathcal{P}$ has not been tampered.

To determine that $\mathcal{P}$ was not executed in a sandbox environment, $\mathcal{C}$ determines the number of processes having an open connection to Trent on the client machine. This is obtained by scanning the remote address and remote port combinations on each of the port descriptors in the system. This is further explained in the previous publication. To determine whether $\mathcal{C}$ was bounced to another machine, Trent obtains the address of the machine that $\mathcal{C}$ is executing on. Trent had received an attestation request from Alice, hence has access to the IP address of M_{Alice}. If $\mathcal{C}$ returns the IP address of the machine it is executing on, Trent can determine if both values are the same. It may be argued that IP addresses may be spoofed, but additional safeguards like scanning the port descriptors on the machine and reusing the existing connection to Trent prevents mitigates spoofing based attacks. Communication with Trent is achieved by executing the software interrupt with the interrupt number for the OS call socketcall as presented in our prior work [10].

B. 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 (the attacker) may choose to wait till the attestation process is complete, and substitute the client program $\mathcal{P}$ with a corrupted program $\mathcal{P}_c$. To prevent Mallory from doing 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, hence Trent has to change the flow of execution in the client process such that the sequence of events reported will allow Trent to determine whether the 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 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 $\mathcal{F}_0$ 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

```
__asm__  ("mov %ebp, %esp \n"
          "pop       %ebp \n"
          "jmp       0x8048bff \n");
```

Fig. 5. Tail portion of $\mathcal{F}_1$

its execution, $\mathcal{F}_1$ undoes all its stack operation and jumps to the address where $\mathcal{F}_0$ is located.

If $\mathcal{F}_1$ executes a return instruction then control would move back to $\mathcal{P}$ and the functionality of $\mathcal{F}_0$ would be lost. It cannot do a function call to $\mathcal{F}_0$ as this may cause loss of some parameters passed by $\mathcal{P}$. In the Intel x86 implementation of Linux, stack is defined during compilation time and allocated only during runtime.The compiler generates code which first saves the base pointer on the stack; this saves the frame for the previous function. It then moves the current stack pointer (which points to the location beyond the last push), into the base pointer. This is done so because locations on the local stack are addressed relative to the current base pointer. The generated code subtracts some memory from the current stack pointer; this allocates the memory for the local variables. The last two instructions in a function are usually *leave* and *ret*. The leave instruction reverses all net stack operations performed by the function by moving the base pointer value into the stack pointer and pops the value stored in stack location for the base pointer of the previous frame. The ret instruction resumes execution at the return address stored on the stack.

$\mathcal{F}_1$ allocates some stack for its local memory during execution. However, instead of letting it return to $\mathcal{P}$, a jump instruction is executed after the stack operations are reversed. $\mathcal{F}_1$ can either execute the MOV instruction to place the value of the base register on the stack pointer (this reverses stack allocation), pop the current stack value into the base pointer, and then jump to the start of $\mathcal{F}_0$. Another approach would be to execute the leave instruction and jump to $\mathcal{F}_0$. Both the methods allow $\mathcal{F}_0$ to receive all parameters passed on the stack by $\mathcal{P}$ and resume normal execution. When $\mathcal{F}_0$ executes a return instruction, the control moves back to $\mathcal{P}$.

C. Implementation

The implementation details of the user application attestation is described in detail in our previous publication. The implementation of verified code execution is described as follows. As part of sending messages to $\mathcal{C}$, Trent provides the code of $\mathcal{F}_1$, the location where $\mathcal{F}_1$ should be placed, and a particular address inside $\mathcal{P}$ which corresponds to a function call to $\mathcal{F}_0$. $\mathcal{C}$ places the code $\mathcal{F}_1$ at the specified location, and changes the target of the call instruction inside $\mathcal{P}$ to point to $\mathcal{F}_1$. When $\mathcal{F}_1$ executes it communicates to Trent and informs Trent that it executed. It can be noted here that $\mathcal{F}_1$ can use the existing connection to Trent or open a new connection. Since Trent generated $\mathcal{F}_1$, Trent can place a random secret inside $\mathcal{F}_1$ which gets communicated to Trent. On receiving the secret, Trent knows that $\mathcal{F}_1$ executed.

```
int fix_address(void){
  int length_ofF1= 0xbd;
  int location_F0 = 0x08048bff;
  int location_F1 = 0x08048c92;
  int offset_of_jmp_in_F1 = 0xa3;
  int eip_offset_for_jmp = 5;
  ....
  T_address = location_F0 -
              (location_F1
                + offset_of_jmp_in_F1
                + eip_offset_for_jmp );
  Write_back [0xa4] =  T_address &
                           0x000000FF;
  Write_back [0xa5] = (T_address &
                           0x0000FF00) >>8;
  Write_back [0xa6] = (T_address &
                           0x00FF0000) >>16;
  Write_back [0xa7] = (T_address &
                           0xFF000000) >>24;
  ....
}
```

Fig. 6. Fixing Jump target

The tail portion of $\mathcal{F}_1$ is provided with code similar in functionality as shown in fig. 5. $\mathcal{F}_1$ clears its stack by moving the base pointer into the stack pointer. It then pops the base pointer value of the previous routine into EBP. Then finally executes a jump to $\mathcal{F}_0$. As $\mathcal{F}_1$ is compiled as a stand alone function, the gcc complier generates an incorrect target address for the Jump instruction. This is fixed by the server side program by correcting the target of the jump instruction as seen in fig. 6. The code calculates the actual 4 byte address for the JMP instruction and then writes it back in the binary in the little-endian format.

VII. Results

Table I provides cumulative results for various operations on an Intel Pentium 4 with 1 GB of RAM and an Intel Core 2 Quad machine that had 3 GB of RAM. As expected, the Pentium 4 machine has slightly lower performance than a platform with 4 Intel Core 2 processors. The application attestation code takes lower time than the kernel attestation scheme as the size of the application (11 Kilobytes) is small compared to the size of the OS text section, system call tables, and IDT. The network delay shown in the table is for each send/receive operation occurring between the client and server machines. Hence if the two machines perform 10 sends/receive, the network delay value will be greater than other components. The code generation time does not include the time required to issue the make command for the kernel module. This is because the make command had variable response rates and took order of a few seconds. Due to this aspect, code generation can be seen as a major overhead for

TABLE I
TIME TO COMPUTE MEASUREMENTS

Time (ms)	Pentium 4	Core 2 Quad
Generation of challenge	12.8	5.5
Execution of $\mathcal{C}$	0.6	0.4
Execution of $\mathcal{C}_{kernel}$	175	54.3
Network delay	21	15

the server for each attestation. To alleviate the load on the server, code generation can occur for a number of test cases beforehand, and use any one of them in a random fashion during attestation.

The application attestation scheme can detect modifications made to the text section of the application. It cannot detect changes made in the runtime on the heap and the stack section. The stack and the heap contain the current program execution state and are highly variable. On the Intel x86 architecture for Linux, the OS enforces that these two sections cannot contain executable pages. Hence the modifications made to the application would reside only on the code section of the application.

The kernel attestation scheme can detect rootkits that make changes to the system call table and the IDT as described in [15]. This scheme cannot determine rootkits that do not change the system call table or the IDT.

VIII. CONCLUSION AND FUTURE WORK

This paper presents a technique to obtain the integrity of a user application entirely in software. A trusted external entity provides Alice with generated code that when executed on the client side provides guarantee that the client side application is not compromised. This paper also extends remote attestation by verifying whether the application attested continued executing or was replaced by the attacker. The check involves placing new code in the in-core image of the application and replacing a function call inside the application to point to the new code. The execution of the new code provides an attesting server the guarantee that the application was not replaced. A series of such changes made inside the attested application can dramatically reduce the opportunities that an attacker may have to hijack authenticated sessions by tampering with the client end software. This paper also presented a technique to obtain the integrity measurement of the OS text section, system call table and Interrupt descriptor table. These measurements are important as the remote attestation scheme for the user application requires the assistance of system calls and the interrupt interface to obtain its measurements. This scheme was implemented on Intel x86 architecture using Linux and its performance was measured. As future work the Remote Attestation scheme can be implemented with hardware assisted virtualization as every consumer x86 computing platform is currently manufactured with this ability.

REFERENCES

[1] T. Ostrand and E. Weyuker, "The distribution of faults in a large industrial software system," in *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*. ACM, 2002, p. 64.

[2] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An empirical study of operating systems errors," in *Proceedings of the eighteenth ACM symposium on Operating systems principles*. ACM, 2001, pp. 73–88.

[3] L. Wang and P. Dasgupta, "Coprocessor-based hierarchical trust management for software integrity and digital identity protection," *Journal of Computer Security*, vol. 16, no. 3, pp. 311–339, 2008.

[4] F. Stumpf, O. Tafreschi, P. Röder, and C. Eckert, "A robust integrity reporting protocol for remote attestation," in *Second Workshop on Advances in Trusted Computing (WATC 06 Fall)*, November 2006.

[5] R. Sailer, X. Zhang, T. Jaeger, and L. Van Doorn, "Design and implementation of a TCG-based integrity measurement architecture," in *Proceedings of the 13th USENIX Security Symposium*, 2004, pp. 223–238.

[6] P. Loscocco, P. Wilson, J. Pendergrass, and C. McDonell, "Linux kernel integrity measurement using contextual inspection," in *Proceedings of the 2007 ACM workshop on Scalable trusted computing*. ACM, 2007, pp. 21–29.

[7] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla, "Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems," *ACM SIGOPS Operating Systems Review*, vol. 39, no. 5, pp. 1–16, 2005.

[8] R. Kennell and L. Jamieson, "Establishing the genuinity of remote computer systems," in *Proceedings of the 12th USENIX Security Symposium*, 2003, pp. 295–308.

[9] J. Garay and L. Huelsbergen, "Software integrity protection using timed executable agents," in *Proceedings of the 2006 ACM Symposium on Information, computer and communications security*. ACM New York, NY, USA, 2006, pp. 189–200.

[10] R. Srinivasan, P. Dasgupta, T. Gohad, and A. Bhattacharya, "Determining the integrity of application binaries on unsecure legacy machines using software based remote attestation," in *Information Systems Security*, ser. LNCS. Springer Berlin/Heidelberg, 2011, vol. 6503, pp. 66–80.

[11] N. Petroni Jr, T. Fraser, J. Molina, and W. Arbaugh, "Copilot-a coprocessor-based kernel runtime integrity monitor," in *Proceedings of the 13th conference on USENIX Security Symposium-Volume 13*. USENIX Association, 2004, p. 13.

[12] R. Sailer, "I.B.M. research - integrity measurement architecture," Retrieved November 3, 2010: http://domino.research.ibm.com/comm/research_people.nsf/pages/sailer.ima.html, 2008.

[13] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, "Terra: A virtual machine-based platform for trusted computing," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, p. 206, 2003.

[14] R. Sahita, U. Savagaonkar, P. Dewan, and D. Durham, "Mitigating the lying-endpoint problem in virtualized network access frameworks," in *Managing Virtualization of Networks and Services*, ser. Lecture Notes in Computer Science, A. Clemm, L. Granville, and R. Stadler, Eds. Springer Berlin - Heidelberg, 2007, vol. 4785, pp. 135–146.

[15] J. Levine, J. Grizzard, and H. Owen, "Detecting and categorizing kernel-level rootkits to aid future detection," *Security & Privacy, IEEE*, vol. 4, no. 1, pp. 24–32, 2006.

[16] U. Shankar, M. Chew, and J. Tygar, "Side effects are not sufficient to authenticate software," in *Proceedings of the 13th USENIX Security Symposium*, 2004, pp. 89–102.

[17] A. Seshadri, A. Perrig, L. Van Doorn, and P. Khosla, "Swatt: Software-based attestation for embedded devices," in *Security and Privacy, 2004. Proceedings. 2004 IEEE Symposium on*. IEEE, 2004, pp. 272–282.

[18] R. Kennell and L. Jamieson, "An analysis of proposed attacks against genuinity tests," CERIAS Technical Report, Purdue University, Tech. Rep., 2004.

[19] K. Cooper, T. Harvey, and T. Waterman, "Building a control-flow graph from scheduled assembly code," Dept. of Computer Science, Rice University, Tech. Rep., 2002.

[20] B. Schwarz, S. Debray, and G. Andrews, "Disassembly of executable code revisited," in *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on*. IEEE, 2003, pp. 45–54.

[21] C. Collberg, C. Thomborson, and D. Low, "Manufacturing cheap, resilient, and stealthy opaque constructs," in *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1998, pp. 184–196.

[22] Weblink, "Kernel mode hooking," Retrieved May 5, 2011: http://codenull.net/articles/kmh_en.html.