Designing and Attacking Remote Software Attestation for Smart Electricity Meters

Vignesh Babu
babu3@illinois.edu

Kartik Palani
palani2@illinois.edu

Introduction

With the evolution of the existing power grid into the modern smart grid, smart electricity meters have become a ubiquitous occurrence in the residential and industrial sectors. The primary purpose of these smart meters is to reliably and efficiently provide real-time statistics about the consumption to the utilities. These smart meters are in essence networked embedded devices, which, increases the security risk, associated with these meters. Attacks on the smart meter are only a small part of the smart grid security issue. Since many smart-grid technologies depend on distributed control and monitoring for data gathering and control, it opens up many avenues for a security attack on the grid. Needless to say, an attack on the power grid will have a huge impact on the socio-economic lifestyle of a nation.

We have tried to solve one of the issues related to data reliability in the grid. In order to extract required information from smart meters, the utilities run various applications on these meters. In order to ascertain that the data they receive from these meters are reliable, the utilities need to trust that the applications running on the smart meters are not tampered with. These smart meters are low cost, low power and low computation devices and hence standard Trusted Platform Modules can't be used to guarantee reliability.

In this paper, we evaluate the method of remote software attestation to guarantee reliability to the utilities. Remote attestation allows authorized users, utilities in our case, to detect changes in their software. We aim to demonstrate the construction of malware rootkits on embedded systems and implement remote software based attestation techniques to allow a remote verifier to ascertain the presence or absence of compromised applications on the embedded device.

In the first part of the paper, we describe a system call hooking attack on a machine running a Linux kernel. The initial attack describes the procedure used to develop a hook on the "write" system call. Later, we describe how the hook function was modified in order to try to nullify the effects of a remote attestation. In the second half of the paper we describe how a remote software attestation model is developed. We further use this model to test if the attack devised in part one still succeeds.

System Call Hooking

A system call is a function that does not run in the user space but in the kernel space. In fact, a system call is how a program running in the user space requests a service from an operating system. It acts as an interface between the user space and the kernel space. The most basic example of a system call is system I/O. To get input or output, the user program needs to request the operating system. This request is passed using a system call.

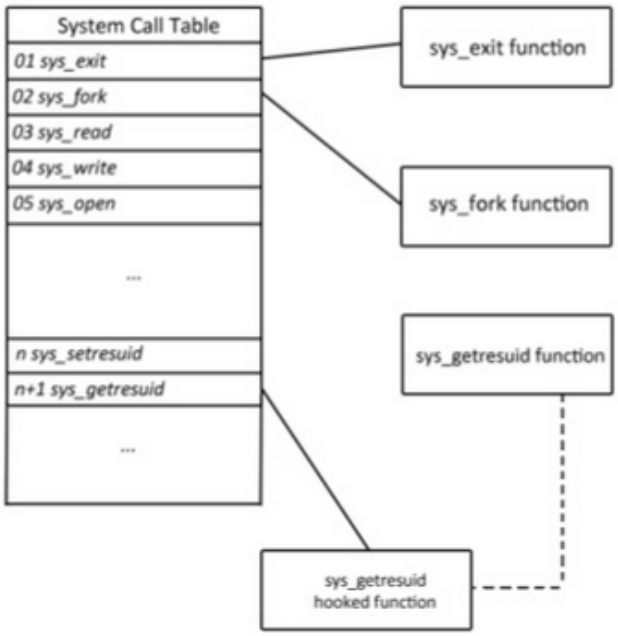
In terms of security, a hook refers to a technique where the original behavior of the system is modified by intercepting functions or messages that get passed between the software components. System call hooking specifically refers to the act of intercepting a system call and performing some activity before actually executing the original functionality of the system call. These system call hooks can also be used maliciously. A rootkit is software is hidden from the user but continues to run in the system with root privileges. The installation of a rootkit requires admin/root privileges.

In this section we describe how we devised a rootkit to steal smart meter data. We do not concern ourselves with how the rootkit was installed. We assume that some kind of social engineering led to the installation of a rootkit. Thus, we only describe how to develop a rootkit and the functionality it provides.

Linux kernel System Call Hooking

The Linux kernel no longer exports the system call table along with it. Thus, the traditional manner of hooking system calls by directly modifying the pointer entries in the sys_call_table no longer works. We however use other methods to locate the sys_call_table in the kernel and intercept the “write” system call.

The reason we choose to hook the *write* system call is because this is the system call that the application would call to write data into the storage. In a smart meter, this data is very sensitive. It contains per second details about the electricity usage patterns of a user. As an attacker if we can capture the information being written to a buffer, we can basically transmit that information to oneself. This gives us immense power. This image delivers the idea of a hooked system call.



Before we dive into how the hook is performed, let us look at interrupts. Interrupts are used, either by software or hardware, to call the processor. In x86 machines, a software interrupt is raised using the INT instruction. The INT instruction takes in as an operand, the interrupt number that needs to be raised.

The following image shows how an interrupt is serviced by the CPU:

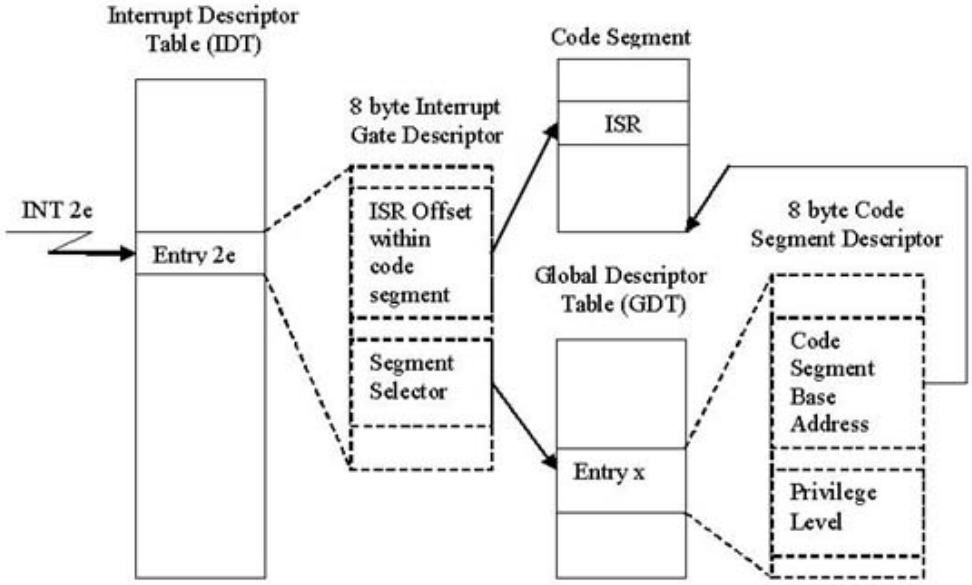


Figure 3. How the CPU finds the Interrupt Service Routine for software interrupt 2e.

The most commonly used interrupts are INT 3 (used for debugging programs) and INT 0x80 (allows user to make system calls). Since INT 0x80 allows users to make system calls, it can used to access the kernel.The following code is an example in assembly of how a INT 0x80 is used to make a system call to write().

```
section .text
 global main

HW db 'Hello World', 0xa

main:
 mov edx, 12
 mov ecx, HW
 mov ebx, 1
 mov eax, 4
 int 0x80

 xor eax,eax
 int 0x80
```

The part that is of import to us is what happens after the interrupt is raised and before the actual write occurs.

As we saw in the figure, when an interrupt is raised, it looks up the offset that is passed to it in the Interrupt Descriptor Table and executes it. In our case, when 0x80 is passed, the 128th entry of IDT is looked up and the action and performs it. What is special about 0x80 is that it allows for a program in the user space to access the kernel API.

The following lines of code describe the INT 0x80 interrupt handler.

```
sysenter_do_call:
    cmpl $(nr_syscalls), %eax
    jae syscall_badsys
    call *sys_call_table(,%eax,4)
    ...
```

Before calling INT 0x80, we saw that eax stores the syscall number. We see in the handler code that, if the eax value passes all validations then the syscall is performed by looking it up in the sys_call_table. The value of eax is used as an offset to identify functionality in the sys_call_table. The sys_call_table contains pointers to all the possible system call functions. Since the sys_call_table is a list of pointers, as an attacker if we can modify the pointers then we have control over the system calls.

The code we wrote to hook the system call achieves this functionality using the following steps:

1. Locate the Interrupt Descriptor Table.
2. Look through the IDT to locate the system call handling routine.
3. Locate the sys_call_table in memory using a scan of known sys_call_table pattern in the system call handling routine.
4. Save the state of the sys_call_table to revert back to when the malicious kernel module is unhooked.
5. Disable memory protection on the sys_call_table.
6. Overwrite the sys_call_table write entry to point to the hooked malicious code.

The Payload

The payload is the malicious function that gets called when the system call is hooked perfectly. For our payload we wanted a function that can sniff the write buffer and store the values in memory.

In order to have our system call hook go undetected by generic remote attestation modules, we did the following:

1. Perform a hook on the write() system call when a /proc/Attest file is found.
2. Get the file descriptor for the file.
3. In the payload, perform malicious code and then regular write() functionality to go undetected.
4. In the payload, when a file matching the file descriptor is found, read in values from the buffer and recreate the memory addresses from which attestation is required.
5. Rewrite the buffer to the /proc/Attest file after modifying the entries.

This leads us to create a payload that defeats the purpose of a generic protection like remote attestation. The idea is to illustrate that a specific security remote attestation module need to be created to secure smart meters and other embedded devices. In this section we saw the attack, in the next section we show a method we devised to protect against this attack.

Remote Software Attestation

This section gives a brief overview of the Attestation process in general. Remote software attestation is the process of verifying the software running on a remote system. The attestation system typically consists of a Verifier (Client) and the remote system (Server). The Client initiates the attestation process by sending an attestation routine to the Server. The Server receives the attestation routine and promises to execute it on behalf of the client. The attestation routine which is executed, checks the static memory locations of the system and computes a result based on the content of these locations. The computed result R_1 is then transmitted by the Server back to the client. The Client usually holds a local image of the static memory content of the remote workstation/system. It then simply uses this local image to compute another result R_2 and checks whether R_1 equals R_2. The attestation is said to have succeeded if both the results are indeed equal. Otherwise the client assumes that the verification has failed which means that the software running on the remote system is different than the one expected.

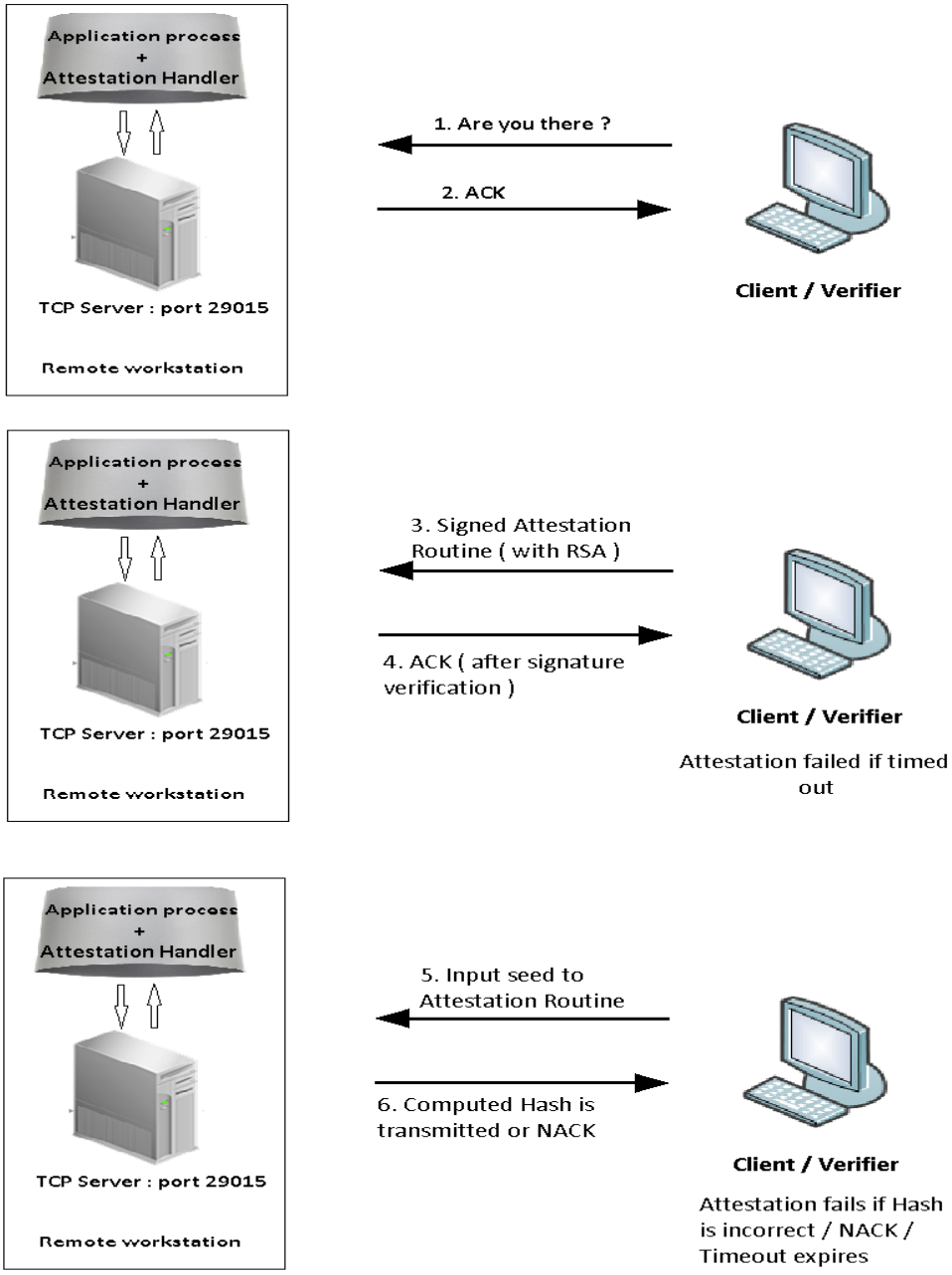
There are some issues which were not discussed in the overview presented in the previous paragraph. Since the goal of the verification process is to check the contents of the static memory locations or the program memory, it becomes necessary to assume an attack model where the attacker has physical access to the remote system and he can alter the memory contents to inject malicious software. The focus of this project is not concerned with how the software running on the remote workstation is changed by the attacker. Rather it is only concerned with the detection of the change. Since the attacker is assumed to have physical access to the system, he has access to the incoming attestation routine and can examine it to predict the result that would be computed by the routine. If it were easy to statically analyse the attestation routine, then the attacker could easily forge the computed result and make the attestation process succeed. To avoid this situation, some important counter measures have to be implemented.

A timeout is first incorporated into the attestation protocol. The client does not wait indefinitely for the server to return the computed result. If the client times out, it subsequently assumes that the attestation has failed and that the system is compromised. Incorporation of the time-out does not allow the attacker to take arbitrarily long time to perform static analysis of the attestation routine and predict the result. Thus the attacker is forced to execute the attestation routine. Additionally, other code obfuscation counter-measures which are often used by malware writers, are used to design the attestation routine to make static analysis even harder. Examples of these code-obfuscation techniques include dead code insertion at random locations, encryption of the payload, randomly generated decryption routines etc. These techniques not only hinder the static analysis, they substantially differentiate any two attestation routines so that the attacker cannot use the knowledge gained by analysing one attestation routine at a particular system, to fool the attestation process at another remote workstation.

However, introduction of time-outs has its own drawbacks. It is possible that the result transmission is simply delayed because of network congestion. In this case, it would still be ruled out as a failure. Nevertheless, the implementation is primarily concerned with minimizing the false negatives. False positives can always be reduced by retrying the attestation process several times and confirming a failure only when all the retry attempts prove to be futile. Following the brief overview of the implementation, the overall setup is

described in the following figure. The messages transmitted between the verifier and the workstation use TCP to ensure that there are no message losses.

Setup:



Signature verification is performed at Step 3 to ensure that the attestation routine indeed came from the verifier and not from a client impersonating to be the actual verifier. The input seed to the attestation routine is also signed before transmission and the signature is verified at the server/workstation. The implementation of the entire protocol was done in python and the signature algorithms were implemented using the pyCrypto library. RSA was used for signature verification. The public and private keys used were generated once and stored in two files: Public_key.pem and Private_key.pem. The workstation was given access only to

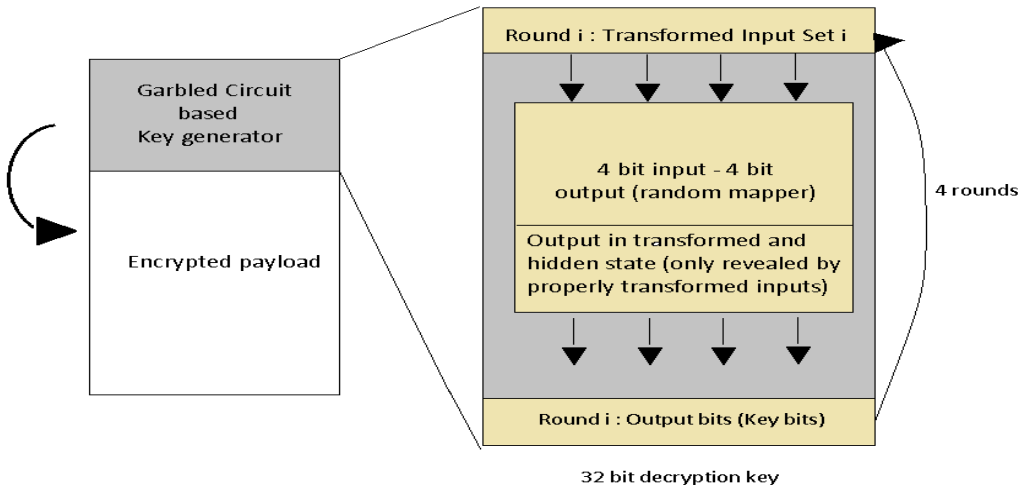
Public_key.pem and it was allowed to read this file at run time and obtain the public key of the verifier. The verifier and the remote workstation were simulated using different VMs. There were three VMs in total which could exchange messages with each other. The client VM ran Ubuntu 12.04 with python2.7. The other two VMs were used to simulate two different workstations. One of them ran Ubuntu 12.04 with python2.7 and the other VM ran debian OS with python 2.6. The attestation system implementation was tested by verifying whether the remote workstation runs the same version of python as the Client/verifier.

As noted before, the Client requires a local copy of the contents of the static memory region of the remote system. To satisfy this requirement, we run an instance of the application process on the client as well with its IP address set to the local host. Thus there is an instance of the application process running as a localhost server in the client itself. The client simply replays the same attestation routine to the localhost server to obtain the locally computed result.

The overall structure of the system is as follows. The client/verifier runs a Client process which generates the random attestation routine and sends it to the remote workstation for performing the verification. The client process also sends the same attestation routine to an instance of the application running on the local host. It then obtains results from the remote server and the local host server and compares both results. The remote workstation runs an instance of the application which is capable of running the received attestation routine. The attestation routine passes on some information to a kernel module running in the workstation. The kernel module uses this information to compute the desired result. The application then subsequently transmits this result back to the Client. Let us now discuss the main components of the attestation system one by one.

Attestation Routine:

The attestation routine was implemented entirely in python. The general structure is represented by figure 2.



It consists of a garbled circuit based key generator which takes an input seed and outputs a decryption key. The decryption key is used to decrypt the payload and the payload is subsequently executed.

Garbled circuit based key generator:

Garbled circuits were proposed first in Yao's secure two party computation protocol in []. They were used primarily to two parties to communicate with each other and solve a common problem by providing their private inputs. The protocol is designed to allow both parties to learn the output of the common problem they were trying to solve without revealing their private inputs to the other party. A garbled circuit as its name suggests can represent any digital circuit such that the function implemented by the circuit is hidden and not understandable by any entity other than the entity that created the circuit.

A garbled circuit is considered as a black box with some input wires and output wires. Each input wire is used to represent an input bit and each output wire is used to represent an output bit. The black box can match any input combination to any output combination. However the mapping remains hidden and in a transformed state. In a garbled circuit representation, each bit value i.e either a 1 or 0 is represented by a different random number depending on the wire in which that particular bit value is observed. For example a bit value of 1 in a wire i is represented by a random number K_i^1 and similarly a bit 0 in wire i is represented by another random number K_i^0. Following this representation, a different random number is assigned to represent bit values of 1 and 0 on different input and output wires. The circuit in itself is nothing but a table containing encrypted values of the output wire representations. The representation for each output wire is encrypted with the set of representations of the input wires that would yield that particular output wire representation. Let us understand the concept with an example of a simple 2 input single output OR gate. The OR gate has two input wires i_1 and i_2 and one output wire o_1. The corresponding representations for these wires are K_{i1}^1 , K_{i1}^0 , K_{i2}^1 , K_{i2}^0 , K_{o1}^1 , K_{o1}^0 respectively. The following is the truth table of a simple OR gate.

Input bit 1	Input bit 2	Actual input 1	Actual input 2	Output bit	Actual output
1	0	K_{i1}^1	K_{i2}^0	1	K_{o1}^1
0	1	K_{i1}^0	K_{i2}^1	1	K_{o1}^1
1	1	K_{i1}^1	K_{i2}^1	1	K_{o1}^1
0	0	K_{i1}^0	K_{i2}^0	0	K_{o1}^0

The input bit columns and the output bit columns are for reference.

Construction Phase:

In the black box garbled circuit implementation, only the values present the Actual input columns are fed as input to the circuit. It should in turn output the values in the Actual output column. For the above 2 input OR gate, the garbled circuit construction would be as follows:

$E(K_{i1}^1, E(K_{i2}^0, K_{o1}^1))$
$E(K_{i1}^0, E(K_{i2}^1, K_{o1}^1))$
$E(K_{i1}^1, E(K_{i2}^1, K_{o1}^1))$
$E(K_{i1}^0, E(K_{i2}^0, K_{o1}^0))$

Let us understand how this garbled circuit table was constructed. From the truth table, one can observe that K_{o1}^1 was outputted for the following input combinations: {(K_{i1}^1, K_{i2}^0), (K_{i1}^0, K_{i2}^1), (K_{i1}^1, K_{i2}^1)}. The garbled circuit table contains the encryption of K_{o1}^1 for each of these input combinations. Similarly if contains the encryption of K_{o1}^0 for the combination (K_{i1}^0, K_{i2}^0). These encrypted values are not stored in any particular order. This garbled circuit table is given to the entity which wants to evaluate the circuit by feeding inputs to it. For evaluation purposes, the output values representations are made public to the evaluating entity. Thus evaluator has access to the garbled circuit table and the set of output value representations. It is easy to see that the evaluator cannot obtain any information about the actual circuit that is implemented with just these two objects. This is the main goal of the garbled circuit implementation: to keep the implemented circuit hidden from the evaluator.

The encryption process described in the construction of the garbled circuit table, depends on the order in which the representations of the input wires are applied to encrypt the output wire representation. In order to make it independent of that order, the encryption function chosen should be commutative. In other words $E(K_1,E(K_2,M))$ should be the same as $E(K_2,E(K_1,M))$. To achieve this property, we chose the encryption function in our implementation to be a simple XOR function.

Evaluation phase:

Now let us move on to the evaluation phase. The evaluator is given a set of input wire representations which represent a particular input combination. The evaluator decrypts each entry in the garbled circuit table with the given input wire representations. Only one of these decryptions yields a value which belongs to the set of output wire representations. The evaluator can then deduce the corresponding output bit value. For this particular example, suppose the input combination (K_{i1}^1, K_{i2}^0) is given to the evaluator, it decrypts each entry of the 4 entries in the garbled circuit table. Only one of these entries: $E(K_{i1}^1, E(K_{i2}^0, K_{o1}^1))$ would infact be correctly decrypted to yield the value K_{o1}^1. The evaluator then easily deduces that the output bit value is 1 for this input combination. It must also be noted that the evaluator remains unaware of what the actual input bit values are.

Garbled circuit construction and implementation:

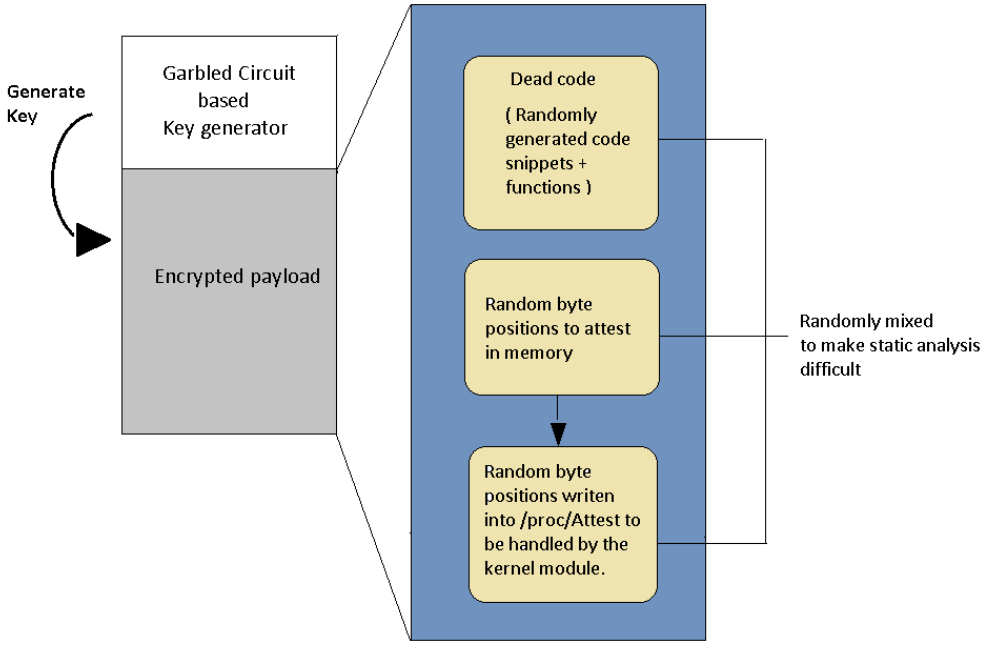
The above section gave an overview of the general theory behind garbled circuits and their general construction and evaluation phases. In our implementation, we implemented a 4 input 4 output garbled circuit with each input combination randomly mapped to an output combination. The random mapping varies with each attestation routine to make sure that the attacker could not accumulate knowledge about the mapping over time and use it for performing static analysis. The garbled circuit table is constructed according to the generated mapping. The input and the output wire representations are also chosen randomly for each attestation routine. The generated garbled circuit table and the output wire representations are included in the attestation routine as a part of the key generator. The input representation values are not sent along with the attestation routine. From the previous description, it is clear that an attacker could not gain any information about the underlying matching function without knowledge of the combination of input value representations.

The verifier first sends the attestation routine and once it has been acknowledged, it choses 8 random input combinations each containing 4 input wire representation values and sends these as an input seed to the remote work station. The workstation, feeds this input seed to the garbled circuit in 8 rounds. In each round, one input combination is fed to the circuit to obtain a 4 bit output. After 8 rounds, a 32 bit decryption key is obtained which is used for decrypting the payload. The encrypted payload is stored as a string which was encrypted with AES during the creation of the attestation routine. The encrypted payload is then decrypted and executed as a subprocess using the **subprocess.call()** function in python.

Payload

The payload contains some important steps which need to be executed in the attestation process. In a typical embedded system, the attestation routine would usually compute the hash of some randomly pre-chosen static memory locations or program memory locations. In our implementation, we simulate this process in VMs running the linux operating system. So we opted for computing the hash of some randomly pre-chosen memory locations in the code section of the application process. Thus the payload would include a set of pre-chosen random offsets from the start of the code-section of the application process, which indicate the memory locations to be subsequently read and concatenated and hashed. These random offsets are chosen by the verifier when the attestation routine was first created.

The following figure shows the general structure of the payload.



The random offsets are stored in a python list. In addition to this some randomly generated code snippets are inserted at random locations in the payload before it was encrypted. These code snippets do not affect the outcome of the payload. They have just been inserted to obfuscate the payload and make it harder to analyse it.

Dead code snippet generation:

A general procedure was followed to generate random code snippets. The code generation process was written in high level languages like python. The generated code was stored as a string and python allows execution of strings containing valid python code content.

Two variables were chosen and the code generator randomly initialises them. It then produces a random number of lines of code. At each line of code one of the two variables is assigned the result of a random operation (addition, subtraction, multiplication, division) performed on the two variables. An example of one such snip pet is as follows:

```
x = 8916.80813789
y = 596.078526106
y = x-y
x = x+y
x = x+y
print x,y
```

Similarly random functions were also generated using the same procedure. The function names were chosen randomly. The code snippet defines such a function with a random name. The body of the function contains a random code snippet like the one presented above. A call statement is finally made to that function. An example is shown here:

```
def func0L4ZUYJ11E() :
        x = 2270.57288442
        y = 5940.37262355
        x = x-y
        x = x*y
        y = x+y
        x = x-y
        y = x+y
        print x,y


func0L4ZUYJ11E()
```

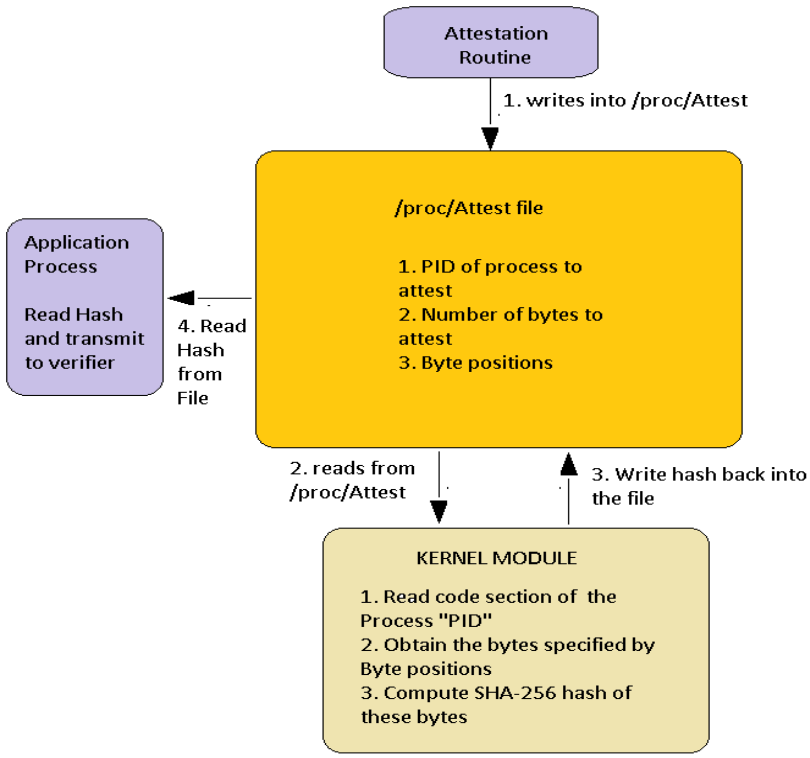
Such random code snippets and function snippets are generated and inserted into the payload string at random positions before the payload string is encrypted. The file Dead_code_pool.py is used for generating these dead code snippets.

File operations:

As explained before, the payload contains the random offsets stored as a list. It writes these byte positions into a file "**/proc/Attest**" which is a proc file belonging to the proc virtual file system. This file was created on the remote workstation by a kernel module which runs in the background. The kernel module takes care of the actual computation of the hash and it is explained in the subsequent section. The payload also writes the process ID of the process whose code section has to be verified. Since the code section of the application process has to be verified, the payload obtains the process ID of the application process by using the system call **os.getppid()** . This function call is invoked by the payload and it returns the process ID of the parent process which is the application process. Both the parent process ID and the random byte offsets are written into the "**/proc/Attest**" file.

Attestation kernel module:

The kernel module forms the final component of the attestation system and it is the most important component. It creates a "**/proc/Attest**" file as soon as it is launched. It then handles the read and write operations directed towards this file. Data written into the file is copied into a buffer stored in memory in the kernel space. When the file is read, the data is transferred from this buffer to the user space. The following figure illustrates the general sequence of steps followed by the attestation module:



As explained in the previous step, the payload writes the process ID and the random byte locations into the "**/proc/Attest**" file. The kernel module obtains both of these components

separately. It sorts through the list of processes running in the system and checks for a match with the specified process ID. It then uses the **task_struct** of the corresponding process to obtain the process memory map. From the memory map, it identifies the start and end points of the code section. The kernel module then subsequently reads contents from the specified offsets (start of the code section + offset) and stores them in an array. The array is then converted to a string by concatenation and SHA-1 hash is computed. The computed hash is then written back into the "**/proc/Attest**" file after flushing the original contents. This hash value is then read by the application process and transmitted back to the Client.

Description of the Source Files:

Client_process.py:

Implements the Client functions by generating the random attestation routine and transmits it to the desired destination as well as the localhost server. It then obtains the computed hash and compares them.

Garbled_circuits.py:

It is used as a library for the construction of N-input N-output garbled circuits. Its functions are used by Garbled_circuits_loader.py

Dead_code_pool.py:

It implements functions which are used to generate random code and function snippets. It is used by Garbled_circuits_loader.py

Garbled_circuits_loader.py:

It uses the Garbled_circuits library to construct the garbled circuit based key generator. The input seed is also chosen and remembered. Subsequently, the garbled circuit is stored as a string. The payload is also initialised as a string. Random dead code snippets are then inserted into the payload. The payload is then encrypted using the generated key and the encrypted string is added to the original string containing the garbled circuit based key generator along with a decryption routine. A set of commands to run the payload are also included.

Public_key.pem:

Export of the public key which was generated using the pyCrypto Library

Private_key.pem:

Contains information to extract both the public key and private key using the pyCrypto Library.

Application_process.py:

It includes two threads, one which performs a dummy application. The second thread handles any received attestation routines and performs signature verification. It receives the attestation routine as a string and it executes it by using subprocess.call() function in python.

Read_process_memory_map.c:

This file contains a set of functions used by the attestation module to obtain details about the process which has to be verified

Attestation_module.c:

Contains the implementation of the kernel module whose functionality was explained in the previous section.

rootkit/

This folder contains a kernel hack that works with systems running linux kernel version 2.6 with a python2.6 installation. This kernel hack makes it look like python2.7 is actually running on the machine.

Testing and Experimentation

In our implementation, we were able to verify the version of python running on a VM. In our experiment, the client VM had python2.7 installed in it and ran Ubuntu 12.04. Two other VMs were spinned up to simulate two remote workstations. One of those VMs was running python2.7 with an Ubuntu 12.04 OS whereas the other VM ran python2.6 in a debian OS. The attestation kernel modules were installed in all VMs and initialised. Application processes were started in the remote VMs with the IP address set to the IP address of the VMs. Another instance of the Application process was started in the Client to act as a localhost server (IP address set to localhost). Since the application_process.py file is given as an argument to the python executable, the python process is started. When the attestation routine tries to obtain the parent process ID, the processID of python is returned. Thus the kernel module will examine the code section of the python executable and compute the hash. Since one of the remote VMs runs python2.7, the hash computed by this VM would be the same as the hash computed by the localhost server running in the Client. On the contrary, the hash computed by the other VM running python2.6 executable would be completely different. This is because the code section of the python2.6 executable is different from that of the python2.7 executable. We were able to observe this in our experiment. Attestation succeeded on the VM running python2.7 whereas it failed for the other VM running python2.6.

We tested the kernel hack that we developed to spoof the attestation process. With a large time out (10 sec), the rootkit was able to spoof the attestation process thus making the verifier VM running python2.7 believe that the remote workstation is running python 2.7 too although the installed python version on it was 2.6. However, with a smaller time out values which were computed after taking into account the time required for remote attestation, we observed that prescence of rootkit was detected and the attestation process failed as expected.