Authentication and secured execution for the Infrastructure-as-a-Service layer of the Cloud Computing model

Laurent Hubert  
CEA, LIST, Embedded real Time Systems Laboratory  
Email: laurent.hubert@cea.fr

Renaud Sirdey  
CEA, LIST, Embedded real Time Systems Laboratory  
Email: renaud.sirdey@cea.fr

Abstract—This paper presents a full system architecture allowing the authentication and secured execution of binary files using hardware-assisted on-the-fly memory encryption/decryption. In a context of general blurring of the physical relationship between a user and the computer which which it eventually interacts, this architecture has been thought so as to achieve a certain degree of robustness against corruptions in a cloud computing IaaS software stack, including when the corruptions may come from the lowest hypervision layer.

I. INTRODUCTION

The goal of Cloud Computing is to provide a scalable access to IT resources and services in a simple and efficient way. The user, we also mean companies, who is no longer the manager of these IT servers, can use them without knowing the infrastructure or their locations. These servers are interconnected through a network and host virtual machines (VM). We can find three main layers in cloud computing: SaaS (Software as a Service), PaaS (Platform as a Service) and IaaS (Infrastructure as a Service). In SaaS, application-level software is hosted by the cloud and used by clients, the PaaS model makes available an execution environment to clients and the IaaS layer provides a full infrastructure (ie., an OS-level abstraction). This paper focuses on IaaS: In IaaS, the provider handles the virtualization (generally relying on hypervision technology), the storage, the network and the hardware. The subscriber needs a login/password or a smartcard to be connected and to start his sessions.

Indeed, a user can establish a connection between his computer and the cloud through the Internet (unsafe network) thanks to an interface and can access his distant applications and data anywhere with an Internet access. Providers of Cloud Computing must ensure that users applications and storage of all data are protected against attacks from the outside (hacking, piracy) or the inside (compromise of the hypervision layer). Data (applications and files) can be encrypted but it raises several questions: where is the key material generated? How is it used? How is data integrity ensured? And, more generally, how to protect the data while they are manipulated by a possibly compromised host?

Cloud computing is a tangible reality offering many solutions in various domains for traditional users and for companies but this new way of thinking raises a lot of questions and challenges.

Indeed, although the cloud computing model is developing massively, its development concerns mainly the general consumer market which (perhaps unfortunately) has relatively low security requirements. In order, to allow the penetration of the cloud model in the more traditional industry sector (beyond the use of private clouds), the security of the model needs to be improved. From a cost effectiveness point of view, the general goal is to achieve a level of security comparable to that of industry-grade telecom technologies (VPN or cellular networks, most notably), which are de facto secure enough for intellectual property-sensitive contexts. In particular, the Achille’s heel of the IaaS model is the hypervision layer which is in charge of enforcing the confidentiality, integrity and availability of VM data until it may end up compromised in which case all these properties are lost. Thus, in this paper, we propose an architecture that allows to safeguard VM data confidentiality against any software compromise of the IaaS stack, relying on both tamper proof hardware on the subscriber side as well as ad hoc hardware components and key distribution protocols on the cloud side.

We organized this paper as follows: after a state of the art in Section II, Section III provides a description of the different components of our proposed infrastructure, in section IV, we explain how to start a session for the first time and the subsequent ones. All the functional specifications are then detailed in section V. Lastly, Section VI presents some experimental work which functionally validate the approach.

II. RELATED WORKS

The Cloud Computing model brings a number of security issues as shown in [4][13] and several papers have been proposed to execute encrypted programs or VM [10] [7] under a non-secure infrastructure [11] in order to raise the Cloud Computing security level.

A framework has been developed to enable the execution of encrypted programs on a virtual machine in [3] but this solution is totally based on software and has no hardware part. Furthermore, the decryption can be carried in parallel with memory access as shown in [14] to reduce the overhead due to the cryptographic calculations.

Another solution is proposed in [6], [5] with fully homomorphic encrypted circuits that can execute secret programs in the Cloud by solving the problem of encrypted storage access with encrypted addresses and encrypted branching. But this
The connection between the cloud and the user is made by a smartcard (tamper resistant), although some HW support is required, the processor core that performs calculations does not need to be modified. The on-the-fly encryption/decryption module is the only part in the infrastructure that can access the encrypted memory. The program code is opaque to the hypervisor or the operating system, a preamble precedes each encrypted program code to set up the encryption/decryption key and to load it into the cryptographic module, encrypted blocks format are resistant against relocations attacks and known plaintext attacks (which occur when two identical clear blocks lead to the same encrypted block).

III. SYSTEM COMPONENTS

Our infrastructure is composed of several components grouped into two main domains: the subscriber side and the cloud side.

A. The subscriber side

The subscriber side involves a client machine hereafter referred to as the host as well as a smartcard. The host is the terminal from which the subscriber accesses the cloud. Cloud access is assumed to be performed over an insecure network such as the public Internet. The smartcard is used to perform a number of security-related functions: secure storage of a Long Term Secret (LTS) unique to the card e.g., a 128 bits random number as well as algorithmic support for challenge-response authentication and key establishment (which mainly involves a keyed one-way hash function). Additionally, the smartcard can be assumed to embed a crypto-grade Pseudo Random Number Generator (PRNG). The LTS never gets out of the card and cannot be modified.

To be more precise, given an arbitrary value $X$, the legitimate user of the smartcard can only get $f_{LTS}(X)$, where $f$ is a suitable function which successive results reveal no information on $LTS$. User legitimacy is verified by the smartcard using an ad hoc mechanism such as a PIN code.

As first approach, the reader can assume that the smartcard is analogous in principle to the SIM cards used in most mobile networks for both subscriber authentication and key establishment [12]. Still, its use will be extended in the present context.

As such, the smartcards are most likely managed and owned by the cloud operator. In terms of assumptions, the smartcard is assumed to guarantee LTS secrecy as well as to prevent LTS modification and the subscriber’s host is assumed secured (memory and execution privacy and integrity). In terms of responsibility, the cloud operator is responsible for smartcard security and the subscriber or, more likely, its IS organization is in charge of host security. Note that cloud model or not, host security must be achieved if the subscriber is to securely perform any function. Our goal, here, is to ensure that the introduction of the cloud computing model does not worsen the security level.

B. The cloud side

The cloud side involves one (or more) gateway, one (or more) Authentication Server (AuS) as well as a number of machines, hereafter referred to as servers. Each server can be seen as a (possibly multi-core) computer, each core of which being (at least functionally) associated a Hardware Security Module (HSM). Each server also runs hypervision software which allows effectively sharing the server resources among a number of virtual machines.

The gateway defines the interface between the cloud and the insecure network via which the subscriber connects to the cloud. It is this node which defines the frontier of the cloud, insecure on the outside, secured on the inside. Its securization is clearly under the responsibility of the cloud operator and is out of the scope of this paper as it relies on classical telecom security know-how.

The AuS is a security-critical database which stores, possibly among other data, the LTS associated to a subscriber cannot be modified from the outside. As a first approach, the AuS can be thought of as the analogous of the Authentication Center (AuC) presents in mobile telecommunication networks[12]. Its securization (memory and execution privacy and integrity) is out of the scope of this report as it is also assumed to rely on telecom infrastructure security know-how and can be assumed to be a reasonably well solved problem: such a setting is currently operated in present mobile communication networks.

The servers are the machines on which the subscribers execute their virtual machines. For simplicity sake, we assume that the servers are mono-core machines. Each server thus host a Hardware Security Module which contents and functions are physically inaccessible to the processor, the HSM acts as a AuS proxy on each of the server. In a nutshell, as we shall later see, the HSM is an auxiliary unit which main responsibility is to encrypt/decrypt on-the-fly all the data (instructions as well as data) going in and out of the server processor. The HSM has a privileged relationship with the AuS, in the sense that it is the only device in the server which is allowed to make requests to the AuS.

Additionally, the cloud operator is assumed to operate two independent networks within the cloud: a user network and a control network. The control network interconnects the HSM and the AuS, via a dedicated network interface. The user network interconnects the server main network interface to the rest of the cloud infrastructure.
IV. USE CASES

For simplicity sake but without much loss of generality, we assume virtual machines with 32-bit addressing and 32-bit instruction set sharing a microprocessor with the same characteristics.

The memory footprint of a VM is assumed to be statically dimensioned upon commissioning.

Basically, from an abstract viewpoint, a virtual machine can be considered to be an executable memory image.

A. Virtual machine commissioning

From a subscriber perspective, virtual machine commissioning involves the following steps:

1) Open a private, bilaterally authenticated connection with the cloud (smartcard interaction required).
2) Download a blank virtual machine memory image from the cloud.
3) Encrypt the memory image (smartcard interaction required to generate an encryption key).
4) Upload the encrypted memory image back to the cloud.

Step 1 above is optional in case it has already been performed in preamble of a previous operation.

Alternatively, if certain conditions are met (see Section IV.B), VM encryption can be performed on the cloud side.

Note that it is assumed that the blank virtual machine cannot be compromised. This can be enforced via standard electronic signature schemes.

The virtual machine encryption key is unique to each VM. At that point, the new virtual machine is known to the cloud and ready for use.

B. Use of an existing virtual machine

Again, still from a subscriber perspective, using an existing virtual machine requires going through the following steps:

1) Open a private, bilaterally authenticated connection with the cloud (smartcard interaction required).
2) Select and connect to an existing virtual machine (smartcard interaction required to get the VM key).
3) Interact with the virtual machine.

Again, step 1 above is optional in case it has already been performed in preamble of another operation.

V. FUNCTIONAL SPECIFICATIONS

A. Connecting to the cloud

Bilateral authentication aims at providing both parties, the subscriber and the cloud, with the guarantee that they communicate with a genuine entity. It also provides, as a by-product, a session key which allows creating a private communication channel using symmetric algorithms.

Bilateral authentication, rather that unilateral one (namely subscriber authentication by the cloud), is required to prevent cloud impersonation.

Authentication is performed in a classical challenge-response fashion. The cloud (respectively the subscriber) sends a challenge, say a 128-bit random number $RANDC$ (respectively $RANDS$), to the subscriber (respectively the cloud), the subscriber (respectively the cloud) computes $RES_S = f_{LTS}(RANDS)$ (respectively $RES_C = f_{LTS}(RANDC)$) and sends $RES_S$ (respectively $RES_C$) to the cloud (respectively the subscriber) which privately performs the same calculation, getting $RES'_S$ (respectively $RES'_C$), and verifies that $RES_S = RES'_S$ (respectively $RES_S = RES'_S$).

Note that all calculations on the subscriber side are performed within the smartcard. As already stated in the subscriber side section, only the $RES$ and never the $LTS$ gets out of the card.

By construction, challenge-response authentication can be performed over an insecure channel and then used to derive session keys. For example, $f_{LTS}(RANDC \oplus RANDS)$ would provide a suitable session key.

At that point, the subscriber and the cloud are able to communicate privately and have mutually convinced each other that they know the $LTS$.

B. VM memory image encryption

Memory image encryption occurs, as stated in Section 2.2, during VM commissioning where the cloud sends the memory image of a blank, though bootable, VM to the subscriber.

Upon reception of this memory image, both the subscriber and the cloud generate a (e.g., 128-bit) random numbers, say $RANDS$ and $RANDC$ (respectively), and combine them into a token $TVM = RANDS \oplus RANDC$ (for example), which is unique and will remain forever attached to the VM. $TVM$ is not, in principle, secret but is used to derive a secret encryption key associated to the VM (each time it is needed): $KVM = f_{LTS}(TVM)$.

Assuming a 32 bits machine, that key is then used to encrypt the memory image in the following fashion: let $w_0, w_1, \ldots, w_{2N−1}$ denote the sequence of $2N \leq 2^{48}$ 32-bit words making up the VM memory image then each (64-bit) pair $(w_{2i}, w_{2i+1})(i = 0 \ldots N − 1)$ is replaced by a 128-bit block, $B_i$, computed as follows:

$$B_i = AES_{KVM}(w_{2i} || w_{2i+1} || 0z/0000 || R_{16} || 2i)$$

where $||$ denotes the concatenation operator and where $R_{16}$ is a 16-bit random number.

In the above, $(w_{2i} \ || \ w_{2i+1})$ defines the payload of the block and the remaining 64 bits provides a number of desirable security properties (which are straightforward consequences of the avalanche property of good block ciphers):

1) The probability for an adversary to forge a valid block without knowledge of $KVM$ is at most $2^{-48}$ (since decrypting an arbitrary block, not encrypted with $KVM$, results in 128 random bits, matching the 16-bit of zeros and the 32-bit of offset is a event with $2^{-48}$ probability).

2) The presence of the (possibly optional) $R_{16}$ field prevents the same payload to be encrypted into the same 128-block at different times, at the same location.

3) The presence of the field $2i$, which represents the address of the block relatively to the beginning of the VM memory image, guarantees that, for a given VM, two blocks identical in payload lead to two completely different encryptions (thereby mitigating
the use of AES or any 128-bit block size block cipher in ECB mode, which is required to enable random access.

4) The presence of the field $2i$, furthermore, guarantees that a valid encrypted block for a given VM cannot be re-injected elsewhere in the VM memory image (since the $2i$ field of the decrypted block would not match the relative address at which it was fetched).

However, it should be emphasized that the above scheme does not protect against temporal replay i.e., an adversary can re-inject at a given memory location an encrypted block which occurred in the past at that (and only at that) memory location. This limitation, however, is a matter of integrity and has no impact with respect to confidentiality.

In terms of efficiency, the memory footprint of the VM is increased by a factor of 2. This can be reduced to a factor 1.33 by increasing the block payload to 3 32-bit words, but this, as a consequence, increases the probability of valid block forging to 2-32 without furthermore including any $R_i$ field, yet another matter of trade-off. Conversely, when criticality justifies it, it is possible to decrease memory efficiency to increase the security level by having a payload of only one 32-bit word per 128 bits encrypted block (i.e., with a padding 32-bit of zeros, 32 random bits and 32 bits of offset). In that latter case the probability of forging by chance a valid block becomes 2-64 and the probability that two identical payloads successively written at the same location lead to identical cipher texts is $2^{-32}$.

Also note that, in order to avoid transferring huge volumes of data between the cloud and the subscriber at machine commissioning, only the initialized part of the VM memory image can be transferred to the subscriber for encryption. The uninitialized payload blocks present in the VM memory image could then be expanded to a pre-set 128-bit block (e.g., $0xffe\ldots fd$) allowing to identify uninitialized memory (the probability of a genuine encrypted block having that value being remote). That way, explicit transfer of the full image is avoided, but the part of the image which contains useful data is given away to a potential adversary (which may open hidden channels, via data access patterns, in case of an initially compromised blank VM).

Alternatively, it is possible to have the cloud preparing the encrypted VM. But this must be done in a trusted environment, for example, on the AuS.

C. Hardware-assisted execution of an encrypted VM

As stated in section II.B we consider the setting in which a standard processor is paired, without any modifications, with an HSM. The HSM sits in between the processor and the external memory. The processor runs hypervision software which job is mainly to schedule and enable the execution of the VMs on the processor. The hypervisor has no physical access to the HSM internal memory, only the HSM has the ability to send queries to the AuS via the control network to which the processor (and a fortiori the hypervisor) has no access.

Functionally speaking, when swapping a given VM into the processor, the hypervisor starts by configuring the HSM, providing it with the VM token, $T_{VM}$. The HSM then sends the $T_{VM}$ to the AuS and gets $K_{VM}$ in return. The HSM is thus able to proceed with on-the-fly encryption/decryption of the stream of instruction/data for this VM.

Per se, the processor (hence the hypervisor) can make only two configurations requests to the HSM:

- $cipher_{mode}(base_{VM}, T_{VM})$: results in configuring the HSM to perform ciphered exchanges with the memory with key $K_{VM} = f_{LT}(T_{VM})$, as explained above, all addressing being performed relatively to $base_{VM}$.

- $clear_{mode}$: results in configuring the HSM to perform unciphered exchanges with the memory (reserved for non-user code).

Thus when the processor performs a read operation (instruction and data alike) of the 32-bit word at (clear) address $ptr$ (addressing is necessarily performed relatively to the beginning of the VM), the HSM fetches the 128-bit encrypted block at physical address

$$base_{VM} + (ptr \& 0xffffffff) \ll 1$$

and, provided the decrypted 128-bit block passes the integrity verifications (5th 16-bit word must be identically 0 and the fourth 32-bit block must be equal to $ptr \& 0xffffffff$), serves the processor with either the first decrypted payload word (case of an even address) or the second one (case of an odd address).

Conversely, when the processor performs a write operation of the 32-bit word at (clear) address $ptr$, the HSM first fetches the 128-bit encrypted block at (relative) address

$$base_{VM} + (ptr \& 0xffffffff) \ll 1,$$

decrypts it with $K_{VM}$, checks it (see above), replace either the first (even address) or second (odd address) payload word with the 32-bit word to be written, encrypts the new clear 128-bit block and finally stores it in external memory at the above (relative) address.

In terms of implementation, caching can of course be used (privately into the HSM) to avoid, as much as possible, spurious fetches.

When a VM is swapped out of the processor (as a result of an hypervisor-triggered interruption) and before the hypervisor code kicks in, a ROM-stored interrupt handler preamble must systematically be executed so as to partially reset the processor, notably storing the execution context at a fixed (relative) address in (encrypted) VM memory space as well as clearing the register file and the processor cache, and configure the HSM back in clear mode. This is in order to avoid the hypervisor having access to any unencrypted data from VM space.

D. Subscriber interaction with an encrypted VM

In order to interact with an encrypted VM, the subscriber is provided, on its host, with a remote read/remote write service in encrypted memory.

For example, the standard input and standard output can be mapped to statically dimensioned buffers at fixed (relative) addresses in VM space.

Of course, memory encryption and decryption must be performed on the subscriber side, which is straightforward since, given $T_{VM}$, the subscriber can interact with its smartcard to get $K_{VM}$ and, thus, maintain a local memory image of the
input/output buffers which is encrypted/decrypted on-the-fly by its host when entering/leaving from/to the cloud. 

Input/output are performed under interruption (which means that the ROM preamble of the previous section is also involved before switching from payload code to interrupt handling by the hypervision kernel).

An interruption starts and stops in the few following steps.

The encrypted program or VM loads the registers which defines the interruption parameters ( interruption opcodes, reading or writing address) and jumps to the ROM.

The VM context then is saved in the encrypted memory image and all the registers are set to zero (except the registers further used for interruption handling) and the processor cache is emptied. The HSM is then switched to clear_mode (still as part of the ROM, as in a “normal” context switch).

The hypervisor then sends/receives encrypted data to/from the subscriber.

The subscriber receives the interruption opcodes (read or write), the clear address inside the encrypted memory image and the encrypted block. The subscriber decrypts the 128-bit encrypted block with $K_{VM}$, tests its integrity and acts accordingly:

If the integrity check fails, the subscriber must send back an error message, else he acts as following:

- If it is a read then the subscriber prints the first word (even address) or the second word (odd address).
- If it is a write:
  - The subscriber modifies the first 32-bit word (even clear address) or the second word (odd clear address) of the 128-bit clear block.
  - The subscriber encrypts the new 128-bit clear block with $K_{VM}$.
  - The subscriber sends the 128-bit encrypted block to the cloud.

After the interruption, the system turns back into nominal mode.

Lastly, the encrypted payload context is restored by switching back to cipher_mode(base_VM, $T_{VM}$) thanks to the payload preamble, restoring the program execution context and jumping to $PC+4$ (the next instruction after the program interruption) adding the offset base_VM.

### E. Software prototyping

We prototyped our proposed infrastructure using SystemC as a modelling language.

Our initial prototype involved a simple abstract microprocessor with a limited ISA (below) and 16 registers:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOP</td>
<td>0x0f</td>
<td>Halt</td>
</tr>
<tr>
<td>DUP</td>
<td>0x0c</td>
<td>DUP_dest_reg, source_reg</td>
</tr>
<tr>
<td>JRL</td>
<td>0x0d</td>
<td>JRL_addr_reg (relative jump)</td>
</tr>
<tr>
<td>JMP</td>
<td>0x0e</td>
<td>JMP_addr_reg</td>
</tr>
<tr>
<td>STO</td>
<td>0x0f</td>
<td>STO_sour_reg_addr</td>
</tr>
<tr>
<td>ADD</td>
<td>0x10</td>
<td>ADD_dest_reg, sour_reg1, sour_reg2</td>
</tr>
<tr>
<td>SUB</td>
<td>0x11</td>
<td>SUB_dest_reg, sour_reg1, sour_reg2</td>
</tr>
<tr>
<td>MUL</td>
<td>0x12</td>
<td>MUL_dest_reg, sour_reg1, sour_reg2</td>
</tr>
<tr>
<td>DIV</td>
<td>0x13</td>
<td>DIV_dest_reg, sour_reg1, sour_reg2</td>
</tr>
</tbody>
</table>

Then we consider the following program which computes $b * b - 4 * a * c$ with $a=3, b=10$ and $c=5$. The result will be stored at the address 00000006.

```
00000000 - 10010004 LIL R1,0004
00000001 - 11010000 LIL R1,0000
00000002 - 0d010000 JRL R1
00000003 - 00000003
00000004 - 0000000a
00000005 - 00000005
00000006 - fdfdfdfd
00000007 - 10010003 LIL R1,0003
00000008 - 11010000 LIL R1,0000
00000009 - 12010100 LOA R1,R1
0000000a - 10020004 LIL R2,0004
0000000b - 11020000 LIL R2,0000
0000000c - 12020200 LOA R2,r2
0000000d - 10030005 LIL R3,0005
0000000e - 11030000 LIL R3,0000
0000000f - 12030300 LOA R3,R3
00000010 - 16020202 MUL R2,R2,R2
00000011 - 10040004 LIL R4,4
00000012 - 11040000 LIL R4,0000
00000013 - 16040001 MUL R4,R4,R1
00000014 - 16040003 MUL R4,R4,R3
00000015 - 15010204 SUB R1,R2,R4
00000016 - 10020006 LIL R2,0006
00000017 - 11020000 LIL R2,0000
00000018 - 13010200 STO R1,R2
00000019 - 0f000000 HLT
```

The goal of the first instructions is to set the register $R1$ to 0x4, this allows to make a relative jump at the address 0x7. Data are stored between 0x3 and 0x6.

After the execution on our abstract processor, we obtain the following memory dump:

```
00000000 - 10010004 LIL R1,0004
00000001 - 11010000 LIL R1,0000
00000002 - 0d010000 JRL R1
00000003 - 00000003
00000004 - 0000000a
00000005 - 00000005
00000006 - 00000028 <- b*b-4*a*c
```

Here, we can see the result of $10*10-4*3*5 = 40$ which is
28 in hexadecimal. This example was executed in clear mode. We then tested the execution of this program with on-the-fly encryption/decryption activated. Assuming the following (dummy) values for the \textit{LTS} (recall that the \textit{LTS} is sealed in the smartcard) and the \textit{RAND},
\[\text{\textit{RAND}} = c6\ a1\ 3b\ 37\ 87\ 8f\ 5b\ 82\ 6f\ 4f\ 81\ 62\ a1\ c8\ d8\ 79\]
\[\text{\textit{LTS}} = 00\ 01\ 02\ 03\ 04\ 05\ 06\ 07\ 08\ 09\ 0a\ 0b\ 0c\ 0d\ 0e\ 0f\]
we obtain the \(K_V M\)
\[K_V M = 69\ 3c\ a2\ 11\ 70\ 55\ 93\ f3\ fd\ fe\ 45\ 76\ 9b\ 11\ 51\ 21\]
With this data, we were able to generate both a preamble (for configuring the HSM) as well as an encrypted payload (hereafter).

\begin{verbatim}
00000000 - 10010000 LIL R1,0000
00000001 - 11010000 LIH R1,0000
00000002 - 10020001 LIL R2,0001
00000003 - 11020000 LIH R2,0000
00000004 - 13020100 STO R2,R1
00000005 - 11020000 LIH R2,c6a1
00000006 - 10020001 LIL R2,3b37
00000007 - 13020100 STO R2,R1
00000008 - 11020000 LIH R2,878f
00000009 - 10020001 LIL R2,5b82
0000000a - 13020100 STO R2,R1
0000000b - 11020000 LIH R2,6f4f
0000000c - 10020001 LIL R2,8182
0000000d - 13020100 STO R2,R1
[...]
0000019 - 130a0200 STO R10,R2
000001a - 13020100 STO R2,R1
000001b - 0e030000 JMP R3
000001c - ee39561e beginning of payload
000001d - a38ccc7a
000001e - 9f4c013d
000001f - fcd45119
0000020 - 51903853
0000021 - e1c7240a
0000022 - bc2ea05c
0000023 - 0b48e6de
0000024 - 3b7ff2fa
0000025 - e699b9c2
0000026 - 1d3a1f9d
0000027 - 1c5a0c0c
[...]
000004c - d715860d
000004d - daf20c66
000004e - d9c75102
000004f - dc624ee0
\end{verbatim}

The first instructions load the (public) \textit{RAND} in the \textit{HSM} and turn it into \textit{crypto_mode}. The last instruction jumps at the beginning of the encrypted payload. After the execution, we can decrypt the payload and we will be able to see that the clear payload was modified as seen in the previous example.

However for this first functional test, the processor used was unrealistic because of the size of ISA, thus we moved on to consider a real processor to illustrate the functioning of this secure system.

VI. EXPERIMENTAL WORK

A. Tools

The Open Virtual Platform (OVP) [1] is a full-system simulator developed by Imperas, in others words, it is a multi-processor platform SystemC-based simulator. It is used to run binary files for the target hardware. It has APIs allowing users to create their own processors, peripheral and platform models. OVP is the platform simulator we used for the prototyping of our infrastructure. This simulator emulates a lot of processors like the MIPS32 RISC processor.

B. Platform

To execute a sample code on an emulated MIPS processor, we have to set up the execution environment. So, the processor used is a MIPS32 RISC, buses are 32-bits sized, the memory range is from 0 to \texttt{0xffffffff} and for the \textit{HSM}, we decide to use a MMC (Memory Model Component) which is integrated between the processor and the memory.

Various models are available in OVP and we decide to use a MIPS32 RISC processor. MIPS (Microprocessor without Interlocked Pipeline Stages) is a RISC (Reduced Instruction Set Computer) processor developed by MIPS Technologies. Getting started is easy because of a simplified ISA (Instruction Set Architecture).

![Fig. 1. Full infrastructure](image-url)

Figure 1 presents the infrastructure we have prototyped. The processor is an unmodified 32-bits MIPS, the Hardware Security Module \textit{HSM} decrypts the encrypted memory of 4 Gbytes, the network interface (or gateway) is the only component of this system open on the Internet and the terminal models a personal computer. The development was made in C with the OVP Platform. The various steps involved in the execution of an encrypted program were detailed in Section IV.C. When the encrypted program requires an input or an output, the context is saved in encrypted memory using the \textit{ROM} (coded in assembly) and the terminal (user) has to do something (read or write). No matter if it is a read or a write operation, the processor (and of course the hypervisor) does not have access to the data themselves.
C. Test cases

We got started with raw binary files. Such a file would typically contain a few line of MIPS hexadecimal code. This code was then executable our OVP platform. This binary was then encrypted to be executed on a modified version of the OVP platform in which a MMC was added between the processor and the memory so as to perform the decryption/encryption on-the-fly.

After this first test scenario, we decided to move on by considering a basic code performing the preemptive execution of two tasks with different cryptographic contexts. In this stage, the ROM section was used to perform the context saving in the encrypted payload.

At this point, we were able to consider executing simple programs coded in C. Such a program would typically be compiled using GCC (cross-compiled for the generation of MIPS code) giving an executable file. In order to avoid the intricacies of developing a full ELF loader, we slightly cheated by loading this file in memory with the loader of our simulator and then output a memory dump of 32-bits words binary file. We then encrypted this flat executable file by using the encryption program we prototyped. Thus, words are reorganized to fit with our encryption format and a block cipher algorithm (namely the AES) was used to encrypt the payload. A configuration preamble for the crypto module is added at the beginning of this file. Finally, the binary file is loaded into the memory and the simulation can start. During program execution, a task can be preempted, so the ROM was used again for context switching (between the VM and the system software). In the process, all the registers are saved (at a relative address) in the running encrypted payload and the processor is duly reset, thus leaving no window for any system software (hypervisor or not) to access any encrypted payload data.

From there, there remained to address the IO problem, thus we started by adding the possibility to read and write an integer in the encrypted memory and then further developed a printf-like function so as to output any-size strings to a preset memory location in encrypted payload (declared in the C file). This printf raises an interruption which provokes the execution of ROM code for context switching to system software which can then manipulate (send or receive) these (encrypted) data in an oblivious manner for (respectively) decryption or encryption at the user terminal.

VII. Conclusion

We have presented the functional specification of a complete security infrastructure allowing both strong authentication as well as encrypted program execution. This infrastructure provides a solution to the confidentiality issues related to the Cloud Computing IaaS layer by recreating the link between system and physical security. With respect to confidentiality, this allows protecting the subscriber payload (instructions and data) against any software corruption down to the hypervision layer (without exempting one to appropriately secure this layer as well [8]).

The fielding of such infrastructures can be expected to bring the Cloud Computing model to a level of security comparable to that of present day large telecom infrastructures (mobile or not). Security level which is, de facto, sufficient for most applications including those sensitive with respect to industrial property.

In terms of experiments, this work has mainly focused on the functional validation of our proposed infrastructure.

In terms of further research, a prototype implementation of the HSM (and its integration with a standard processor core) on FPGA is in progress. This will allow to duly characterize the performance impact of the solution, when caching and speculation are used at the hardware level to minimize this impact.

An other interesting line of research (this time for the crypto community) would be to design new encryption algorithms with performances (extremely) lightweight enough to cope with the high bandwidth of the memory interface of modern processors (contrary to primitives such as the AES which have been developed for bandwidths typical of the telecom realm).

REFERENCES