Sixth Framework Programme
Information Society Technology

RE-TRUST
Remote EnTrusting by RUn-time Software authentication

Project Number: FP6 - 021186

Deliverable: D1.5

HW/SW-based method final reference architecture design
### Summary

<table>
<thead>
<tr>
<th>Project Number:</th>
<th>FP6-021186</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project Title:</td>
<td>RE-TRUST: Remote EnTrusting by RUn-time Software auThentication</td>
</tr>
<tr>
<td>Deliverable Type:</td>
<td>RP</td>
</tr>
<tr>
<td>Deliverable Number:</td>
<td>D1.5</td>
</tr>
<tr>
<td>Contractual Date of Delivery:</td>
<td>September 2009</td>
</tr>
<tr>
<td>Actual Date of Delivery:</td>
<td>September 2009</td>
</tr>
<tr>
<td>Title of Deliverable:</td>
<td>HW/SW-based method final reference architecture design</td>
</tr>
<tr>
<td>Workpackage Contributing to the Deliverable:</td>
<td>WP1</td>
</tr>
<tr>
<td>Nature of the Deliverable:</td>
<td>Report</td>
</tr>
<tr>
<td>Author(s):</td>
<td>C. Basile, Y. Ofek (UNITN)</td>
</tr>
<tr>
<td>Reviewer(s):</td>
<td>M. Baldi (POLITO)</td>
</tr>
</tbody>
</table>

**Abstract:** This deliverable presents the final reference architecture for remote entrusting solutions using additional hardware. The final architecture details the entrusting services that were developed during the project, both at trusted entity side and at the trusted hardware side. The entrusting services are associated to the different types of solutions, as shown in deliverable D4.2/D4.4, for instance, dynamic replacement, code splitting and remote attestation solutions. The final architecture provides the necessary capability for developing other entrusted services and solutions.

**Keywords:** Trust analysis, security analysis, remote entrusting, obfuscation, reverse engineering, attack model.

**Classification:**

<table>
<thead>
<tr>
<th>Name of Client:</th>
<th>European Commission</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distribution List:</td>
<td>European Commission Project Partners</td>
</tr>
<tr>
<td>Authorised by:</td>
<td>1.0</td>
</tr>
<tr>
<td>Issue:</td>
<td>RE-TRUST Deliverable</td>
</tr>
<tr>
<td>Reference:</td>
<td>--</td>
</tr>
<tr>
<td>Total Number of Pages:</td>
<td>Prof. Yoram OFEK</td>
</tr>
<tr>
<td>Contact Details:</td>
<td>University of Trento, Dept. of Information and Communication Technology (DIT), Trento, Italy</td>
</tr>
<tr>
<td>Phone:</td>
<td>+39 0461 88 3954 (office)</td>
</tr>
<tr>
<td>Fax:</td>
<td>+39 0461 88 2093 (DIT)</td>
</tr>
<tr>
<td>E-mail:</td>
<td>ofek-at-dit.unitn.it</td>
</tr>
</tbody>
</table>
Table of Contents

1. Introduction .............................................................................................................. 4
2. Architecture requirements ...................................................................................... 5
3. Remote entrusting architecture .............................................................................. 8
4. The trusted entity ................................................................................................... 10
  4.1 Input generator and remote code execution engine ......................................... 10
  4.2 Replacement service and program factory ...................................................... 11
  4.3 Verification service ......................................................................................... 12
  4.4 Intra-T communications ............................................................................... 12
  4.5 Interactions with the untrusted area ............................................................... 13
  4.6 Interactions with the trusted hardware ............................................................ 13
5. The untrusted host ................................................................................................. 14
  5.1 Monitor ........................................................................................................... 14
  5.2 Remote attestation agent and HW remote attestation agent ......................... 15
  5.3 Replacement agent and HW replacement agent ............................................ 15
  5.4 Reaction agent and HW reaction agent .......................................................... 16
  5.5 T’s input manager and HW Input generator and remote code execution engine

6. Trusted hardware devices ..................................................................................... 17
  6.1 Smart card ...................................................................................................... 17
  6.2 High density smart card ............................................................................... 18
  6.3 Trusted platform module .............................................................................. 18
  6.4 Secure co-processors ..................................................................................... 19
7. Conclusions .......................................................................................................... 20
8. References ............................................................................................................. 21
1. Introduction

During the three years of RE-TRUST project, various solutions were developed to introduce the concept of remote entrusting using software running on an untrusted host. This concept of remote entrusting is denoted as a system in which a remote trusted entity is able to force software running on an untrusted host to be executed untampered. Such kind of approach is twofold. First, the trusted (verification) entity should be able to verify assertions on the execution of the software at run-time. Secondly, it should be able to stop the execution of the software when malicious or unexpected behaviour is detected. Stopping execution can be interpreted in a broad sense: immediately halt execution, make the program crash, exit, or run without meaningful behaviour.

One major category of solutions make use of an additional hardware on the untrusted host. This hardware is assumed to be completely trusted. This is particularly useful because the trusted verification entity can rely on it when the proof of the correct execution is required.

The goal of this deliverable is to provide the final architecture in which solutions developed throughout the project can be mapped. Additionally, the architecture must be general enough to also cover future research directions.

Although the initial architecture D1.3 sufficiently covered the various HW/SW-based solutions developed in the project, in the following document some refinements are described. These refinements further clarify the novelty of the project.

The document is structured as follows: Section 2 analyses the remote entrusting assumptions and other architectural requirements for HW/SW-based solutions; after introducing the initial architecture, Section 3 presents the final reference architecture for HW/SW-based solutions; in Section 4 the entrusting services of trusted entity are introduced and in Section 5 the program and the its running environment, the untrusted host together with the trusted hardware devices are presented; Section 6 lists the real devices available as trusted hardware; finally Section 7 summarises the document and draws conclusions.
2. Architecture requirements

This section analyzes the main requirements and assumptions to consider in the architecture definition. First of all let us consider the system to model: the project targeted a general distributed system, i.e., a system composed of a set of different computational components connected through “the network”.

Defined the type of system to model, we need to consider the type of software applications that will be executed by the computational nodes composing the system and for which we need to guarantee a trusted execution. A detailed analysis of classes of applications to consider in the framework of the RE-TRUST project was performed in deliverable D1.1 “Analysis of generic classes of applications” whose validity is still high. As discussed in that deliverable, the classes of applications targeted within this project are networked applications.

In preparing the final architecture of the HW/SW-based solutions the natural starting point is the deliverable D1.3 “HW/SW-based method initial reference architecture design” and the deliverable D1.4 “SW-based method final reference architecture design”. The objective of the initial architecture was to define an architecture as general as possible to be the reference point for the development of solutions and to better understand the remote entrusting paradigm. In three years of experience many solutions were proposed by the RE-TRUST project. The initial architecture has been a valid reference point in the development of the solution. However, inside the very general architecture different components and functionalities were highlighted. The role of these parts is at this point of the project very clear. The final architecture will add details to the initial one in order to completely cover all the HW/SW-based solutions developed by the project.

![Figure 1 – Initial architecture of HW/SW-based methods](image)

Figure 1 depicts the general remote entrusting scenario, as was initially introduced: a remote trusted entity $T$ interacting with a program $P$ running on an untrusted host $U$. The interaction is performed in form of message exchanges: $m_P$ identifies a generic message sent by $P$ to $T$, $m_T$ identifies a generic message sent by $T$ to $P$. Additionally, on the untrusted host it is available some trusted hardware device that can be used to strengthen the entrusting process.

For the interaction hypothesis to be satisfied, remote entrusting model relies on having continuous network connectivity. The concept of network here must be considered as an abstract concept (i.e., a communication mechanism between the components) [D1.1]. It can be then easily mapped on real mechanisms such as the Internet or a local/corporate network.

Even if the attacker can do almost everything on $U$, he has neither access, nor visibility into $T$. The attacker may know what operating system and software is running on $T$, he may also know what kind of protection techniques are deployed, but should not be able to defeat them. $T$ is considered completely trusted. Specifically, the attacker can only interact in a black-box manner with $T$. 
Assumption (trusted black-box): In the remote entrusting model, the entity $T$ is assumed to be a black-box and completely trusted.

This assumption limits the adversarial powers of an attacker by stating that the static properties of the code of the trusted entity cannot be inspected by him. The adversary can however inspect dynamic behaviour such as the I/O from $T$. This assumption directly implies next one that we write explicitly for sake of clearness.

Assumption (trusted $T$-output): Every message $m_T$ produced by $T$ is assumed to be correct.

Additionally, the program $P$ (and every successive replacement) executing on host $U$ is assumed not to be malicious (such as malware) at the time of its “production” in $T$. But, an adversary (quite likely a human) in control of the operating environment of $U$ and possessing static and dynamic program analysis skills (and tools) may completely reverse engineer $P$ and compromise its behavioural integrity (which loosely means that the attacker, in addition to changing the static properties of $P$’s code, may also change runtime parameters such as buffer length and message characteristics such that $P$ behaves unexpectedly). The adversary can also monitor the message exchanges between $P$ and $T$ and has full control of the message buffers of $U$.

$T$ will interact with $P$ and with the trusted HW to be continuously assured that the functionalities of $P$ are not altered nor modified.

Principle (limited execution): The program $P$ cannot execute without exchanging messages with $T$.

In this setting, an attacker succeeds when he is able to tamper with the application running on the untrusted machine (client) without being detected by the trusted server. In order to mount a successful attack, the malicious user needs to understand the application that he/she is trying to tamper with. An attacker has at his/her disposal all kinds of automatic static and dynamic technique for reverse engineering the application. A successful attack usually needs program comprehension information obtained by the effort of human beings using such program analysis tools.

Principle (black-box trusted hardware): In the remote entrusting model, the trusted hardware is assumed to be a black-box and completely trusted.

The trusted hardware environment consists of hardware components directly connected to the untrusted platform. In this model, we assume that these hardware components behave as black boxes. This means that any execution of code on these hardware devices cannot be tampered with, and that an adversary is not able to read its content (code or data, such as secret keys) unless it is intended to. Hence, we assume no side-channel attacks take place. A model, in which side-channel attacks on hardware devices are incorporated, is examined in Task 3.4: “Physically observable cryptography”. In other cases, this assumption need to be justified, as for the FPGA solution [Di Carlo].

Additionally, we assume that the client program $P$ requires a service from an application server (e.g., a network game client that needs input from a game server, such as location information of other players on the game map).

We start from the original program $P$, which we want to entrust. In order to list the necessary requirements for the initial architecture, we describe what functionalities need to be
added. For remote attestation purposes, we envisage adding functionality to the program with the purpose of monitoring its execution from the trusted entity. We denote this extra functionality as the monitor or entrusting agent. However, because of the extended control of the attacker over the untrusted host, the trustworthiness verdict is reached on a trusted entity, not on the client.

The trusted HW also provides entrusting services. It behaves essentially as an appendix of $T$ inside the untrusted host that is available to have a better control on program execution. For this reason, the service it provides have always a counterpart in $T$. However, the decision about the trustworthiness in the remote entrusting paradigm should be reached on $T$. This is a main distinction from the standard local techniques such as software guards [Auchsmith].

Another important consideration holds at this point: the detection is effective only if the trusted entity is able to react to attacks. Reaction is distinctive of the remote entrusting paradigm. Nevertheless the type of reaction is of classes of applications (D1.1) and specific of RE-TRUST solutions. The detection assumption imposes some limits in the attacker model. In fact, she/he is discouraged from tampering with the program by the fact that $T$ will interrupt the stream of input if it detects misbehaviours (i.e., anomaly in the sequence $m_T$ messages). In fact, the $m_T$ and $m_P$ message exchange can be viewed as an implicit continuous challenge-response operation.

The categories of RE-TRUST solutions introduce additional requirements. In fact, they need to be covered by the final architecture. In the deliverable D4.2/D4.4 (D4.x) the techniques are divided in:

- **dynamic replacement**: the basic idea of the solutions using dynamic replacement is to periodically replace the copy of the program running on the untrusted machine with the goal of limiting the amount of time that the attacker has at hand to reverse engineer the application (e.g., the available time);
- **remote attestation**: remote attestation allows changes to the entrusted program $P$ to be detected by the trusted entity $T$.
- **code splitting/execution on the trusted entity**: this technique moves some functional part of the program $P$ is executed on the trusted entity $T$. The “process” of selecting the part to execute on $T$ is called “code splitting”.

3. Remote entrusting architecture

Based on the architecture requirements discussed in the previous section, we propose the following conceptual architecture.

![Diagram of HW/SW-based methods: detailed initial architecture](image)

There are several actors that use the system. The defender is the person who deploys the remote entrusting mechanism and seeks to protect some aspects of the system. All trusted relationships described in the remainder of this document are in terms of what the defender trusts. The other two actors in the system are usually the end-user and the attacker. According to the deliverable D2.1/3.1, the attacker has no restriction on the tools and techniques to use to reverse-engineer and then to tamper with the application. That is, it can install any software, read and modify any part of the hardware (memory, CPU register, etc.) as well as network communications. Furthermore, the attacker can mount environmental, static and dynamic attacks. It is clear that, the assumption of Man-In-The-Middle does not depict correctly the power of the attacker in the remote entrusting paradigm. For this reason we introduced the definition of Man-In-The-End (MITE). For this reason, in the remote entrusting paradigm, there the end-user and the attacker correspond to the same actor. It is worth noting that, by assumption, the attacker cannot tamper with the trusted hardware.

The entire system is divided in two environments:

- the **trusted area**, divided in two disjoint parts, separated by an untrusted area: the trusted entity \( T \), which provides services to the program running on the untrusted host and where the trust result is achieved and where the verdict about trustworthiness of the program is reached, and the trusted hardware, corresponding to the trusted hardware devices available at the untrusted host that provide a set of HW-based entrusting services helpful to formulate the trust result;

- the **untrusted area**, corresponding to the part of untrusted host \( U \) with the only exception of the untrusted host. This is considered as potentially under full control of the adversary. This is where (part of) the original application we want to entrust is executed as well as (part of) the entrusting agent.

The arrows depicted in the architecture represent the communication channels between the different components. Between the trusted environment and the untrusted host, the communication may be transmitted over a normal network (i.e., TCP/IP). The fact that the trusted area is composed by two disjoint parts implies that the communications between the trusted entity and the trusted area pass through the untrusted area. This channel must be explicitly protected if integrity, authentication or confidentiality are needed against Man-In-The-Middle attacks.
Below, we provide an overview of the elements composing the architecture. For each component, we describe its desired behaviour, although we do not specify how this behaviour can be established.

As presented before, the trusted entity and the trusted hardware are completely trusted. Both present different component corresponding to the different entrusting services. In particular, the architecture highlights the services offered to entrust the program (entrusting services), both from the trusted entity side from the trusted hardware.

Denote with $P$ the original program, that is the source code/binaries implementing the functionalities for which it was originally written and deployed. In order to be protected, the original program needs to be extended with extra functionalities, which we capture with the concept of “monitor” (see Section 5.1). This includes code to compute and send the attestations, code that manages the software updates, code that could force the application to stop when a trigger arrives from a trusted entity, and so forth.
4. The trusted entity

In Figure 3 it is presented the architecture of the trusted entity.

![Figure 3: HW/SW-based method architecture: the trusted entity](image)

The trusted entity includes the following components:

- **verification service**, able to evaluate if the program actually runs as expected
- **replacement service**, able to replace the versions of the program
- input generator and code execution service, manages the
- **application host**, is the same as in the original program setup, before the protecting techniques are deployed, and still provides the same service to the client program \( P \). Examples of these application host can be found in the deliverable D1.1;
- **other services**, the final architecture must cover all the classes of solutions developed by RE-TRUST but it should also able to cover future research directions. As soon as new categories of entrusting services are developed they may be available to the defender to entrust the program. This makes the final architecture and the trusted entity very flexible and expansible.

4.1 Input generator and remote code execution engine

One of the distinctive characteristics of the remote entrusting paradigm is the “limited execution”. The limited execution is possible if only a part of the entire program is sent to the untrusted host for execution. There are different ways that the trusted entity may use to limit the execution.

First, some (security sensitive) part of the program are never sent to the untrusted host and are always executed on \( T \) or inside the trusted hardware. When the missing functionalities are required for the correct program execution, a call to the trusted entity or to the trusted devices is made. After the execution of remote code, a result is returned to the program. In this case we talk of execution on the trusted entity, if the code is executed on \( T \), or execution on the trusted hardware in the other case. A part of \( P \) can be moved from the untrusted host to the trusted entity in order to ensure its integrity and often confidentiality.
Define this part as $P_T$. Another part of $P$ can be moved from the untrusted host to the trusted hardware. Define this part as $P_H$.

The remaining program is $P_U$. $P_U$ is merged with the monitor $M_U$ such that an adversary is not able to distinguish code from $P_s$ and $M_s$. Define $P'$ as the merged program. In the rest of the document, the term program will be often referred to the portion $P_U$ of program sent to and executed on the untrusted host.

Another viable approach consists in initially sending a version of the program that is not correct (and in general non-executable). Before the execution, the part to be executed is corrected using code-modifying blocks of code sent by $T$ and opportunely managed by the entrusting monitor. After the correction, the attacker achieved the correct version of the part of the program executed as presented by Collberg et al. [Collberg].

In other cases, the program sent to $U$ misses control flow information that must be continuously asked to the trusted area, for example, jump locations better if associated with some code flattening technique. In this way, any time that the program execution reaches some of the missing pointers, a request to $T$ or to the trusted hardware is sent. Even if the functional code of the program is available to the attacker, he has to understand all the code in order. Even in this case, after an information is sent, the attacker can try to reconstruct the original program.

For the last two cases we say that the trusted entity plays the role of input generator for the program. In both cases, dynamic replacement techniques should be associated in order to make impossible to the attacker to store and use the received information (see section Replacement service and program factory).

### 4.2 Replacement service and program factory

The replacement service replaces (parts of) the program with a fresh program generated by the program factory. The program factory should generate highly independent versions of the program, such that information gathered with reverse-engineering attacks on expired agents cannot be used to speed up an attack on a fresh agent, that is, it makes impossible the learning. Even if this hypothesis can be considered quite strong, by means of obfuscation techniques, it is achievable practically (see the paper from Ceccato et al. [ORTHO]).

The program factory may be considered independent from the on-line entrusting process. The different versions of the program can be obtained offline, or before the program execution or on-line, just before they have to be sent. However, it is possible to assume the it does not use computational resources on $T$.

It is a role of the program factory splitting the program in the two or three parts $P_U$, $P_T$, and $P_H$.

The updating process should be constructed as such that it does not reveal useful information, which allows the adversary to attack the client program. If an adversary blocks the program updating to keep the previous one, this should be detected as malicious behaviour. This can be achieved remote attestation procedures deployed on $P$.

The replacement can be done either by entirely substituting the security sensitive part of the program or by performing the “continuous replacement”. In the second case, the replacement service obtains from the program factory a set of blocks whose objective is to modify the program executable and whose final result is to “induce” another version of the program (see the already referenced paper from Collberg et al.).

Moreover, only authentic program/blocks should be accepted by the replacement agent. This can be established by the means of a signature verification built into the replacement engine. This verification process may be delegated to the trusted hardware.

The reason to introduce a program factory and a replacement service in the remote entrusting architecture is to narrow attack window for an adversary, the program have a
limited validity. Once its validity has been elapsed, the program (or part of it) is replaced by a fresh one generated by the factory. The replacement strategy is based upon the assumption that compromising a program requires time if humans are involved in the process. For this reason, the duration of program validity is based on an estimation of the difficulty of breaking its integrity (attack complexity according to D4.x).

4.3 Verification service

The basis of the architecture is a remote attestation scheme. Attestations are pieces of code that intend to provide some “proof” of authenticity. In this case, they are sent from the (remote) untrusted execution platform to trusted entity. Attestations are some signature of the binary code that is loaded. Since they are computed on an untrusted computing platform, it is a hard problem to guarantee that the correctness of attestations corresponds to the authenticity of the software. An adversary could for example compute signatures on binary code of an untampered copy, while executing a tampered version of the application. To strengthen the attestation process, the trusted hardware plays a major role. In fact the attestation can be generated inside the trusted hardware (using information from the untrusted host) and sent to the trusted entity through a “secure” channel.

In the remote entrusting paradigm, the attestations must be verified continuously. or this, it is necessary to have a verification server. The verification server is the core of trust in the architecture. It is able to verify certain assertions on the execution of the client program, and enforce the program to run as intended. The verification server must have a counterpart in the program able to generate the attestations.

The attestation may be:

• unsolicited, the monitor in the program decides independently when to send the attestations;
• solicited by the verification service.

Another functionality to provide randomness and freshness to the attestation procedure in order to avoid reply attacks.

Once a tampering or malicious behaviour has been detected the verification service must force the client application to stop its execution. However, in some cases (depending on the application class), this can also be achieved when the application server stops providing inputs. For example, in an online game, if the game server does not provide updated locations of opponents, it does not make sense to continue playing the game.

4.4 Intra-T communications

Inside the trusted area, different communications may happen. In Figure 3 they have been presented as a bus communication to simplify the picture, nevertheless, the inter-components communication can be point to point.

Application host – verification service. The application can be informed if the client program is running as expected. The application server and the verification server are depicted as two different entities; however they can be one single entity as well.

Replacement service – verification service. Since the replacement may (and should) change the entrusting monitor, the verification server must be informed about the new attestation methods.

Replacement service – input generator and remote code execution. When different (orthogonal) versions of the program are sent the input generator and remote code execution service must know the version of the program currently in use.
Verification service – input generator and remote code execution. If a misbehaviour is detected by the verification server, the input generator and remote code execution will suspend the interaction with the program, thus blocking its execution.

4.5 Interactions with the untrusted area

To avoid easy classification of messages that can give an advantage to the attacker, the trusted entity must make indistinguishable the messages also hiding the component that produced it. For this the communication channel between $T$ and $P$ is managed by an I/O unit whose objective is to prepare the messages to $P$.

4.6 Interactions with the trusted hardware

There are no direct interactions between the trusted entity and the trusted hardware. The two trusted areas are separated by an untrusted region. For this reason, all the security properties required by the communication channel must be explicitly enforced using ad techniques (see D4.7).
5. The untrusted host

In Figure 4 it is presented the architecture and the components of the on the untrusted host: the program and the trusted hardware. The functionalities summarized by the concept of monitor in previous sections have been isolated and put in relation with the entrusting services to which they are related. Moreover, the trusted HW represents a trusted area inside the untrusted host that is exploited by the trusted entity in order to entrust the program.

Figure 4 – HW/SW-based method final architecture: the untrusted host, including the program and the trusted HW

5.1 Monitor

The monitor or entrusting agent $M$ is the collection of all functionality that is added to the program $P$ to enable a verification server to monitor the client's execution. The agent provides the following four components:

- the remote attestation agent facilitates the monitor to report to the verification service. Based on the generated tags, the verification server is able to reach a trustworthiness verdict of the application. This agent may be helped in generating the attestation by the HW RA agent;
- the replacement agent manages all the functionalities related to the substitution of different versions of the program. Some of the replacement functionalities may be moved to the HW replacement agent;
- the reaction agent enforces the reaction of the trusted entity especially when misbehaviors are detected. Even in this case, some reactive function may be enforced by the HW reaction agent;
- the $T$’s input manager interacts with the trusted entity in order to obtain the information required to continue its execution (limited execution principle). This software entity may also interact with the HW input generator and code execution component located inside the trusted HW;
- other agents, as soon as new categories of entrusting services are developed at the level of the trusted entity, the monitor may have other components to help them in implementing their functionalities. This makes the final architecture and the monitor very flexible and expansible. This agents may have a counterpart inside the trusted hardware.
5.2 Remote attestation agent and HW remote attestation agent

The remote attestation agent is the engine that reports to the attestations server. It generates attestations based on the state and/or code of the program $P$ (the part delivered to $U$) which the verification server can use to reach a trustworthiness verdict.

An adversary should not be able to forge attestations, hence these attestations should be unpredictable, and its computation should not be tampered with. Therefore, this attestation procedure should be protected using strong obfuscation when it is computed on the untrusted host. Moreover, attestations may be “signed” to prove their authenticity to the trusted entity. They can also be encrypted to prevent an adversary to gain knowledge from their content. Important here is the freshness to prevent replay attacks. These nonce are produced and sent in $T$ by the verification services. There are many ways to achieve this: challenge-response protocols, timestamps, counters, etc. Additionally, the remote attestation agent may be solicited by $T$ or may produce the attestations when particular part of the code have been executed (e.g., at the end of a function) or

In order to help in generating stronger attestations of the correct program execution, the trusted hardware may provide a very significant service. The hardware can provide randomness and freshness to the attestation procedure in order to avoid reply attacks. Moreover, attestations may be completely generated inside the trusted hardware and sent through a secure channel to the verification service.

Additionally, it can store secrets that are shared with the trusted entity, it may generate “signatures” based on code and/or states of the entrusting program. To calculate these signatures, the trusted hardware may have direct access to system buses, memory and other resources (without relying on the operating system on which the program executes).

5.3 Replacement agent and HW replacement agent

The replacement is the counterpart on the untrusted host of the replacement service. It verifies the authenticity of the request of replacement before updating the program. It leverages the power of the trusted entity such that the program can be forced to stop its execution and it switches the control to the new monitor after a replacement has been completed. The replacement can be done either by entirely substituting the security sensitive part of the program or by performing the “continuous replacement”. In the second case, the replacement service obtains from the program factory a set of blocks whose objective is to modify the portion of program to be executed inside the trusted hardware (see paper from Collberg et al.). Only authentic program/blocks should be accepted by the replacement agent. This can be established by the means of a signature verification built into the replacement engine.

Some of the functionalities of the replacement service can be moved to the trusted HW. The HW replacement service provides two services to the program and to the trusted entity, according to the security requirements needed by the solution. It check the correctness of the replacement request by authenticating the trusted entity (e.g., via shared secrets). Additionally, it may replace the parts of the code to be executed inside the trusted hardware or it may manage the storage of information necessary to the correct program execution.

A second reason is to leverage the power of the trusted entity to force the client application to stop its execution to complete the replacement. However, in some cases (depending on the application class), this can also be achieved when the application server stops providing inputs. For example, in an online game, if the game server does not provide updated locations of opponents, it does not make sense to continue playing the game.
5.4 Reaction agent and HW reaction agent

The reaction agent enforces the decisions of the trusted entity when an attempt to tamper with the program is detected. This may include techniques of graceful degradation [Randell], deletion/modification of the correct program code, etc.

As for other agents, even in this case, some reactive functionality can be moved to the trusted hardware.

5.5 \( T \)'s input manager and HW Input generator and remote code execution engine

The \( T \)'s input manager is the component that is able to interact with the trusted entity and the trusted hardware in order to request the execution of the part of the code left on the trusted entity or to receive the information to continue the program’s execution.

As explained in Section 4.1, the program may be split in two or three parts to be executed in the different areas. In particular, the trusted hardware may have computational capabilities or simply storage capabilities. In some cases, the program misses control flow information: the missing information may be stored in the trusted hardware in order to avoid frequent remote requests. Additionally, as for the barrier slicing with trusted hardware solution, some (security sensitive) parts of the program may be executed inside hardware devices (e.g., smart cards).
6. Trusted hardware devices

Trusted hardware devices can provide a protected execution environment that can not be manipulated nor observed by an adversary. They are immune to software attacks, and usually provide some form of physical security, making them resistant against some hardware attacks. Trusted hardware exists in various form factors, but typically the devices provide similar basic functionality. They contain a microcontroller, some non-volatile memory (possibly rewritable) to store private data (e.g., cryptographic keys) and/or program code, and optionally a cryptographic coprocessor (e.g., modular exponentiation). The communication interface with the device is normally limited in bandwidth; the software to talk to the trusted hardware needs to run in the untrusted host environment and can thus be tampered with.

In this section, we describe a number of high assurance trusted hardware components. You can image the usage of other more general purpose devices to achieve a secure execution environment, but they provide a much lower level of assurance. You could for instance connect a PDA or a smartphone to the untrusted host (e.g., through Bluetooth), but these devices are prone to the same software attacks (i.e., reverse engineering and software tampering) as the untrusted host. Another option could be to connect a FPGA to the untrusted host, but the configuration bitstream of the FPGA can also be analysed and even modified with very simple hardware means; some high end FPGAs do have bitstream encryption however.

6.1 Smart card

A smart card has a full operating system and is a programmable device. Therefore, applications can be managed (load, activate, remove) and nowadays the preferred language to implement the application, is Java Card, a subset of the Java language. Another important property is its inherently nomadic feature. A smart card typically belongs to a certain user, contrary to a trusted platform module that physically bound to a computing platform. It has to be evaluated how far this feature is meaningful for the purpose of the project.

Smart cards are mainly used for its authentication feature. Important examples are the (U)SIM ((UMTS) Subscriber Identity Module) of mobile phones or EMV credit cards. Electronic identity cards (eID) are used to identify citizens, but in some countries eID cards can also be used to produce non-reputation signatures, i.e., legally binding digital signatures.

Traditionally the bandwidth of the communication interface is quite limited. Contact smart cards (ISO 7816) used to have a serial I/O interface with a data rate of 9.6 kbit/s, while contactless cards typically have a communication interface of around 100 kbit/s. Newer wireless interfaces, like RFID and NFC (Near Field Communication), provide communication speeds of up to 848 kbit/s; e.g., biometric passports contain a RFID smartcard, that prevents unauthorized access to the biometric data with a cryptographic challenge-response protocol.

The main issues with smart card based solutions are related to the difficulty of the deployment of the smart card software and reader installations. The cost of the reader can also be an issue. Classical smart cards require the installation of a smart card reader. The well-known form factor is the standard plastic card. The new generation of smart cards is moving away from the plastic card form factor to USB tokens. This eliminates the requirement of a dedicated smart card reader.

A future evolution would be to rely on Ethernet for TCP/IP compliant cards but the drivers are not yet natively available on common operating systems. The issue with the driver installation is that it might not be authorised in a corporate environment or when the user is logged as a guest in a cyber cafe. In those contexts he namely has low administrative privileges.
6.2 High density smart card

Classic smart cards use different technologies for volatile and non-volatile memory: ROM (Read-Only Memory) for fixed data and code (e.g., smart card operating system), EEPROM (Electrically Erasable Programmable Read-Only Memory) to store Java Card applets and secret user data, and RAM (Random Access Memory) as working memory. With the arrival of cheap and abundant (NOR and NAND) Flash memory, there is an evolution to high capacity smart cards. These cards offer a reasonably large amount of persistent storage to store user data and applications and use faster communication interfaces, like USB and MMC (MultiMediaCard). High density smart cards can be used to store user applications, much like traditional memory sticks. The smart card, however, can enforce a more fine grained access control to the user data.

There is also a movement to more powerful processors, like 32 bit ARM instead of 8/16 bit microcontrollers (e.g., 8051). Therefore, these new generation smart cards, with their high storage capacity and increased computational power, start to resemble secure co-processors, described below.

6.3 Trusted platform module

The Trusted Computing Group (TCG) specifications define three components that form a trusted platform. The core is called the trusted platform module (TPM), which usually is implemented by a smartcard-like chip bound to the platform. The second component is called core root of trust for measurement (CRTM), and is the first code the TCG compliant platform executes when it is booted; in a personal computer, the CRTM is the first part of the BIOS, which cannot be flashed or otherwise be modified. To compensate for the lack of functionality in the TPM, the TCG specifies a software stack (TSS), which facilitates some of the complex, but non-critical functionality and provides standard interfaces for high level applications.

The TPM offers a physical true random number generator, cryptographic functions (i.e., SHA-1, HMAC, RSA encryption/decryption, signatures and key generation), and tamper resistant non-volatile memory (mainly used for persistent key storage). Remark that no symmetric encryption algorithm is provided. The TPM provides a set of platform configuration registers (PCRs) that are used to store measurements (i.e., hash values) about the platform configuration. The content of these registers can only be modified using the extending operation: PCR\(_{i+1}\) ← SHA-1(PCR\(_i\)||M\) with PCR\(_i\) the previous register value, PCR\(_{i+1}\) the new value, M a new measurement and || denoting the concatenation of values.

The initial platform state is measured by computing cryptographic hashes of all software components loaded during the boot process. The task of the CRTM is to measure (i.e., compute a hash of) the code and parameters of the BIOS and extend the first PCR register with this measurement. Next the BIOS will measure the binary image of the bootloader before transferring control to the bootloader which in its turn measures the operating system. In this way a chain of trust is established from the CRTM to the operating system and potentially even to individual applications.

The functionality of a TPM is fixed and hence can it can not be programmed freely. The TPM can sign arbitrary messages, the content of the PCRs representing the platform configuration (this is called attestation), monotonic counters and an internal clock (i.e., tick counter). It is also possible to use TPM for key storage: key and other data will be encrypted by a secret key only known to the TPM, and the sensitive data can be bound to certain platform configuration (this is called sealed storage).
6.4 Secure co-processors

Secure co-processors are the final option to create a trustworthy execution environment. This is an extra processor integrated inside the untrusted host platform. Like a smart card, it runs a secure operating system and can run arbitrary programs. The only commercial available high assurance secure co-processors are the IBM 4758 and 4764 PCI(-X) crypto cards. They include a general purpose processor (486 and PowerPC respectively), battery backed RAM as non-volatile storage and a high performance cryptographic coprocessor.

These devices however are very expensive, especially because of their very high physical security. Therefore this is not an attractive solution within this project.
7. Conclusions

This deliverable presented the final reference architecture for remote entrusting solutions using additional hardware. The final architecture details the entrusting services that were developed during the project, both at the trusted entity side and at the trusted hardware side. The entrusting services are associated to the different types of solutions, as shown in deliverable D4.2/D4.4, for instance, dynamic replacement, code splitting and remote attestation solutions. The final architecture provides the necessary capability for developing other entrusted services and solutions.
8. References


