Advanced Software Protection:
Integration, Research and Exploitation

D1.04
Reference Architecture

Project no.:	609734
Funding scheme:	Collaborative project
Start date of the project:	1st November 2013
Duration:	36 months
Work programme topic:	FP7-ICT-2013-10
Deliverable type:	Report
Deliverable reference number:	ICT-609734 / D1.04 / 2.1
WP and tasks contributing:	WP 1 / Tasks 1.4
Due date:	October 2015
Actual submission date:	5 April 2016
Responsible Organization:	NAGRA (v1.0) - ~~GTO (v2.0)~~ - UGent (v2.0-v2.1)
Editor:	Brecht Wyseur (v1.0) - Bjorn De Sutter (v2.0-v2.1)
Dissemination Level:	Public
Revision:	v2.1

Abstract:
The reference architecture describes the architectural solution of applications that have been protected by the ASPIRE techniques. It provides a detailed description of the components that are introduced both at client-side as at server-side by the ASPIRE tool flow, and how these components interact during the execution of the ASPIRE protected application. Additionally, common logic is described that the technique components can use, such as the communication logic that supports the ASPIRE client-server protocol.

Keywords:
Architecture, technique components, API, ASPIRE protocol, communication logic, ACCL.

Editor

Brecht Wyseur (NAGRA) - Bjorn De Sutter (UGent)

Contributors (ordered according to beneficiary numbers)

Andrea Avancini, Mariano Ceccato (FBK)

Jerome d'Annoville (GTO)

Aldo Basile (POLITO)

Andreas Weber (SFNT)

Alessandro Cabutto, Paolo Falcarin (UEL)

Bart Coppens, Stijn Volckaert (UGent)

The ASPIRE Consortium consists of:

Ghent University (UGent)	Coordinator & Beneficiary	Belgium
Politecnico Di Torino (POLITO)	Beneficiary	Italy
Nagravision SA (NAGRA)	Beneficiary	Switzerland
Fondazione Bruno Kessler (FBK)	Beneficiary	Italy
University of East London (UEL)	Beneficiary	UK
SFNT Germany GmbH (SFNT)	Beneficiary	Germany
Gemalto SA (GTO)	Beneficiary	France

Coordinating person:	Prof. Bjorn De Sutter
E-mail:	coordinator@aspire-fp7.eu
Tel:	+32 9 264 3367
Fax:	+32 9 264 3594
Project website:	www.aspire-fp7.eu

Disclaimer
The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement n° 609734.

Executive Summary

This deliverable presents the ASPIRE Reference Architecture. It defines the components of ASPIRE protected applications and their server-side support, and how these interact with each other. In other words, this reference architecture presents which components are introduced by ASPIRE protection techniques and how these techniques operate during the run-time of the protected application.

As a basis for the reference architecture, a multi-tier architecture is defined. This captures an architecture where a multitude of client-applications can connect to the ASPIRE portal infrastructure, which will manage the connections with a multitude of backend services. In Section 1 of this deliverable, a detailed view of this high level architecture is presented and motivated. Within the context of this multi-tier architecture, the ASPIRE protection techniques are defined. The architecture of each of these techniques is detailed in the subsequent sections. Note that this only applies for techniques that introduce additional components to the protected applications. Protection techniques that are solely related to the operational behaviour of the application and as such do not introduce new assets or new dependencies are not included; they have no architectural impact.

Section 2 of this deliverable presents the ASPIRE protocol. This defines how the ASPIRE protected applications communicate with the ASPIRE security server. This is achieved by introducing an ASPIRE portal – a part of the ASPIRE security server that exposes a web service – and a special-purpose component (the 'ACCL') that needs to be integrated into the protected application. This ACCL abstracts the communication link and should make it easier to develop the ASPIRE protection techniques. Two different types of protocols have been identified: a simple request protocol and a protocol based on WebSockets.

The next three sections describe parts of the reference architecture, related to individual protection techniques: the server-side and client-side components that each technique introduces and how they operate during the execution of the ASPIRE protected application.

Section 3 presents the architecture of the ASPIRE anti-reverse engineering protection techniques. These include obfuscation techniques and anti-debugging techniques. Three code obfuscation techniques are presented, which all relate to code splitting. In client-side code splitting, code that has been split from the original application is translated into custom bytecode that will be executed in a virtual machine component that has been embedded into the client-side application. In server-side code execution, the native code is executed server-side, while mobile code is a technique that delivers code chunks (that have been split from the application) to the application at run-time. The data obfuscation techniques that are introduced relate to the obfuscation of cryptographic keys, with techniques such as white-box cryptography and multi-threaded crypto. As anti-debugging technique, a technique is introduced which attaches an internal debugger to the protected application.

Section 4 presents the architecture of the ASPIRE anti-tampering techniques. In contrast to Section 2, this does not present complete solutions as individual techniques in separate subsections. Instead, different types of components are individually presented: tamper detection components (attestator components and verifier components) and tamper response components (delay components and reaction components). A complete anti-tampering solution comprises these different types of components. As tamper detection technique, code guards, CFG tagging, call stacks check, and anti-cloning are presented. As response components, delay data structures and software time bombs are presented. Some examples of compositions thereof are introduced as well: a completely offline combination, and remote attestation techniques.

Section 5 presents an assessment of the composability of the many protections already supported and foreseen to be supported by the ASPIRE Compiler Tool Chain, i.e., to what extent multiple protections can be applied to protect the same code fragment. It also discusses where synergies exist between individual protections to let them reinforce each other, and where additional design and development work is foreseen to build even stronger protections out of compositions of existing ones. This specifically concerns adaptations to mobile code, remote attestation (and its code guards) and client-side code splitting to support remote attestation of mobile code & data, and of the software components implementing the mobility.

Section 6 details the forms of renewability that will be developed in year 3 of the project on top of the Code Mobility protection. Several strategies are proposed to combine diversity in space with diversity in time, and to make some protections themselves renewable, such as remote attestators and their reaction mechanisms.

Contents

Section 1 Introduction **1**
1.1 Role of this document 1
1.2 Approach 1
1.3 High-level Architecture 2
1.3.1 Motivation 3
1.3.2 Detailed view 4
1.4 Technique-specific architectures 5
1.5 Conventions and Notations 7
1.5.1 Workflow diagrams 7
1.6 Updates of version 2.0 compared to v1.0 7
1.7 Updates of version 2.1 compared to v2.0 7
Section 2 ASPIRE Protocol **8**
2.1 Introduction 8
2.1.1 Application identifier 8
2.1.2 Protocol security 9
2.1.3 External HTTP stack 10
2.2 Simple Request Protocol 11
2.3 WebSocket Protocol 13
2.3.1 Protocol initialization 14
2.3.2 Client initiated communication 15
2.3.3 Server initiated communication 15
2.3.4 Scalability and Performances 15
2.4 ACCL API 16
2.4.1 acclExchange 16
2.4.2 acclSend 17
2.4.3 acclWebSocketInit 17
2.4.4 acclWebSocketExchange 18
2.4.5 acclWebSocketSend 18
2.4.6 acclWebSocketShutdown 19
Section 3 Anti-reverse engineering techniques **20**
3.1 Client-side code splitting 20
3.1.1 Introduction 20
3.1.2 System requirements and assumptions 20

3.1.3 Client-side components 20
3.1.4 Run-time behaviour of client-side code splitting 21
3.2 Anti-debugging 23
3.2.1 Introduction 23
3.2.2 System requirements and assumptions....23
3.2.3 Client-side components 24
3.2.4 Anti-debugging run-time behaviour....25
3.2.5 Impact 26
3.3 Client-server code splitting 26
3.3.1 Introduction 26
3.3.2 System requirements and assumptions....27
3.3.3 Architecture Overview....27
3.3.4 Client-side components 27
3.3.5 Server-side components....29
3.3.6 Messages 29
3.3.7 Client/server code splitting splitting sequence diagram....30
3.3.8 Impact 33
3.3.9 Limitations....33
3.4 Code Mobility.... 33
3.4.1 Introduction 34
3.4.2 System requirements and assumptions....35
3.4.3 Client-side components 35
3.4.4 Server-side components....36
3.4.5 Code Mobility run-time behaviour 36
3.4.6 Impact 37
3.4.7 Error management....37
3.4.8 Composability 37
3.5 White-box cryptography.... 37
3.5.1 Introduction 38
3.5.2 Client-side components 38
3.5.3 Server-side components....39
3.5.4 Offline white-box crypto workflow 39
3.5.5 Online white-box crypto workflow 40
3.5.6 Impact 42
3.5.7 Renewable White-Box Cryptography....42
3.6 Multi-threaded cryptography 44
3.6.1 Introduction 44

3.6.2 System requirements and assumptions 45
3.6.3 Client-side components 45
3.6.4 Server-side components 45
3.6.5 Multithreaded crypto workflow diagram 45

Section 4 Anti-tampering 47

4.1 Overall Anti-Tampering Architecture 47
4.1.1 Tamper detection 49
4.1.2 Delay components and tamper response 54
4.2 Tamper detection technique 1: Code guards 59
4.2.1 System requirements and assumptions 59
4.2.2 Client-side components 60
4.2.3 Server-side components for online code guards 60
4.2.4 Code guards offline techniques workflow diagram 61
4.3 Tamper detection technique 2: Call Stack Checks 62
4.3.1 Introduction 62
4.3.2 System requirements and assumptions 62
4.3.3 Client-side components 62
4.4 Tamper detection technique 3: Static Remote Attestation 64
4.5 Tamper detection technique 4: CFG Tagging 68
4.5.1 Introduction 68
4.5.2 System requirements and assumptions 68
4.5.3 Client-side components 69
4.5.4 Verifier connector 69
4.5.5 Server-side components 69
4.5.6 CFG tagging offline technique run-time behaviour 70
4.5.7 CFG tagging online technique run-time behaviour 70
4.6 Tamper detection technique 5: Anti-cloning 72
4.6.1 Introduction 72
4.6.2 System requirements and assumptions 72
4.6.3 Client-side components 72
4.6.4 Server-side components 73
4.6.5 Anti-cloning workflow diagram 73
4.6.6 Server status report request 74
4.6.7 Error management 75
4.7 Delayed Tamper Response: Delay Data Structures 76
4.7.1 Delay data structures and their API 76
4.8 Reaction: Software Time Bombs 79

4.8.1 Introduction 79
4.8.2 System requirements and assumptions 79
4.8.3 Client-side components 80
4.8.4 Software Time Bombs run-time behaviour 81

Section 5 Composability 84

5.1 Composability of different protections in the ACTC 84
5.1.1 Code mobility combined with binary obfuscations 84
5.1.2 Code mobility combined with the SoftVM 84
5.1.3 Code mobility combined with anti-debugging 85
5.1.4 Code mobility combined with WBC 85
5.1.5 Code mobility combined with binary attestation techniques 85
5.1.6 Code guards and remote attestation combined with the SoftVM and WBC 85
5.1.7 CFG Tagging combined with attestation techniques, SoftVM, anti-debugging and mobile code 85
5.1.8 Remote attestation combined with anti-debugging 85
5.1.9 Call stack checks combined with binary obfuscations 85
5.1.10 Invariant Monitoring combined with all Diablo-implemented techniques 85
5.1.11 Invariant Monitoring combined with Client-Server code splitting 86
5.1.12 Invariant Monitoring combined with data obfuscations and WBC 86
5.1.13 Multi-threaded crypto combined with client-server code splitting 86
5.2 Custom support for specific protection compositions 86
5.2.1 Composability challenges 86
5.2.2 Solutions 87
5.3 Server-generated bytecode 88
5.4 Synergies of protections 89

Section 6 Renewability techniques 94

Section 7 List of technique identifiers 97

Section 8 List of Abbreviations 98

Bibliography 99

List of Figures

Figure 1 – ASPIRE multi-tier architecture: high-level view. 3
Figure 2 – Architecture view on a single client-server 4
Figure 3 – The ASPIRE architecture with an external HTTP stack 11
Figure 4 – The Simple Request Protocol 12
Figure 5 – WebSocket Protocol Initialization 14
Figure 6 – Client-side code splitting run-time behaviour 22
Figure 7 – Anti-debugging workflow diagram 25
Figure 8 – Reference Architecture for client-server code splitting 27
Figure 9 – Structure of a message 30
Figure 10 – Sequence Diagram for Code Splitting 30
Figure 11 – Code Mobility High-Level Architecture 35
Figure 12 – Code Mobility workflow diagram 36
Figure 13 – Client-side white-box workflow diagram 40
Figure 14 – Online dynamic-key workflow diagram 41
Figure 15 - Renewable WBC workflow 42
Figure 16 – Multi-threaded Crypto Encryption Processing 45
Figure 17 – Anti-tamper components 48
Figure 18 – Code guards workflow diagram 50
Figure 20 – Remote Attestation Architecture (from D3.04) 51
Figure 21 – Remote attestation workflow. 52
Figure 22 – Architecture of the RA Manager (from D3.04). 53
Figure 23 – Architecture of the Verifier (from D3.04). 54
Figure 24 – Reaction enforcement workflow diagram 56
Figure 25 – Offline code guards workflow diagram 61
Figure 27 – Call stack check workflow diagram 63
Figure 28 – Static remote attestation reference architecture 64
Figure 29 – Static remote attestation workflow. 65
Figure 30 – Attestation request format 66
Figure 31 – Attestation response format 67
Figure 32 – Offline CFG Tagging workflow diagram 70
Figure 33 – Online CFG Tagging workflow diagram 71
Figure 34 – Anti-cloning workflow diagram 73
Figure 35 – Request for trustworthiness status report 75
Figure 36 – STB workflow diagram for Passive Operations 81

Figure 37 – STB workflow diagram for Active Operations..82
Figure 38 – Extended Reference Architecture for Renewability..95

List of Tables

Table 1 – Code Splitting Manager API 28
Table 2 – Server-side internal code splitting API 29
Table 3 – Summary of the synergies between ASPIRE protections. 90
Table 4 – List of technique identifiers 97

Section 1 Introduction

Section Author:

Brecht Wyseur (NAGRA)

1.1 Role of this document

The goal of this deliverable (see GA Annex II Description of Work (DoW) Part A p. 8-9) is to present a reference architecture that defines the structure of ASPIRE protected applications. More specifically, this deliverable presents all additional components that have been introduced by the ASPIRE protection techniques in WP2 and WP3 – components both at client-side as server-side support logic – and their run-time behaviour.

In other words, this deliverable aims to present how the ASPIRE protection techniques operate once they have been integrated into an application. It does not present how the integration itself proceeds – that will be described in the tool flow architecture deliverable (Deliverable D5.01). The ensemble of the introduced components results into a reference architecture that allows meeting the requirements elicited in Deliverable D1.03 ("Security Requirements").

The reference architecture aims to mitigate the attacks described in Deliverable D1.02 ("Attack Model") on generic applications, by complementing the architecture of the original (unprotected) application with additional components that come from the ASPIRE protection techniques that are introduced. This will be validated on the use-cases that have been presented in Deliverable D1.01 ("Use-Case Specifications"). Hence, given any software application, the reference architecture presents what components will be added and how they will operate with the original application logic.

Therefore, the role of this document is the following:

- To establish an unambiguous understanding in the ASPIRE consortium of the run-time behaviour of the software protection techniques that are developed in WP2 and WP3.
- To support the development of the protection techniques by identifying common components and specifying their APIs.
- To present a view on what ASPIRE-protected applications will look like, on the basis of which the development in WP5 and WP6 can then be fine-tuned.

The presented ASPIRE reference architecture should not be considered as the final version. At this early phase in the project, some choices have been made towards the definition of the architecture, taking the known constraints and assumptions in mind. These may however change during the course of the project as the research on the different protection techniques proceeds, and as a result impact the definition of the reference architecture. Therefore, a revised version of the reference architecture is envisioned at M24 (Deliverable D1.05 – "Intermediate Validation, Requirements & Architecture Update"). Additionally, for some techniques the design phase still needs to start. This in particular applies for renewability techniques (Task 3.3), whose conception and design only starts in M19. The architecture of these techniques will be specified in the Reference Architecture revision deliverable.

1.2 Approach

The ASPIRE reference architecture is obviously a client-server architecture. This has already been presented in the Annex I DoW, where at client side additional components would be integrated into the ASPIRE protected application, and server-side logic needs to support the network-based protection techniques such as remote attestation and mobile code. Defining more fine-grained details to support the design and development of protection techniques is subject to the activity conducted in Task 1.4. This activity has been organised as follows:

- At the kick-off of this activity, a preliminary high-level architecture has been presented to the consortium. This preliminary architecture consolidates the expertise and concerns expressed by the industrial partners (e.g., on the need of scalability and server infrastructure constraints) and the expertise of academic and industrial partners on the design of protection techniques. This preliminary high-level view is described in Section 1.3 of this document and has been accepted by the consortium as a basis for the reference architecture definition.
- Based on the presented architecture, a first evaluation round on the protection techniques has been organised. The main objective of this round was to evaluate if the reference architecture is able to support the protection techniques that are envisioned in WP2 and WP3. Additionally, this evaluation allowed to further fine-tune the architecture and to present a first set of common components. For example, the client-side communication logic has been identified in this phase.
- The major part in the reference architecture definition was subsequently conducted: the detailed definition of each of the protection techniques based on the presented high-level architecture and common components. To support this activity and ensure that the definition for each of the techniques would proceed in a uniform format, a template was presented by the task leader. Based on this template, each partner has described the details of their techniques: each component that is introduced, and the run-time behaviour of the protection technique. The latter was a very important step, because the constraints that were imposed by the high-level architecture description often imposed additional reflections on the design of the protection techniques. Prior to this phase, protection techniques were described within their own context and with their proper architecture assumptions – this activity was the first step towards unifying the different approaches. The result of this part is described in Section 3 and Section 4 of this document.
- In the last phase, given the technique-specific descriptions, the overall reference architecture has been further fine-tuned and the common logic specified, as well as the identification of anti-tampering support blocks in Section 4.

1.3 High-level Architecture

As a basis for the ASPIRE reference architecture, a multi-tier architecture structure was selected, as depicted in Figure 1. This captures an architecture where a multitude of client applications connect to a portal infrastructure, which manages the connection to a multitude of backend servers. The ensemble of the portal infrastructure and backend servers, one per security service (i.e., implemented online protection) we denote as the *ASPIRE security server*. We adopt such infrastructure for the client-server communication of the ASPIRE network-based protection techniques, and deploy this in parallel to the client-server communication that the original application might already use.

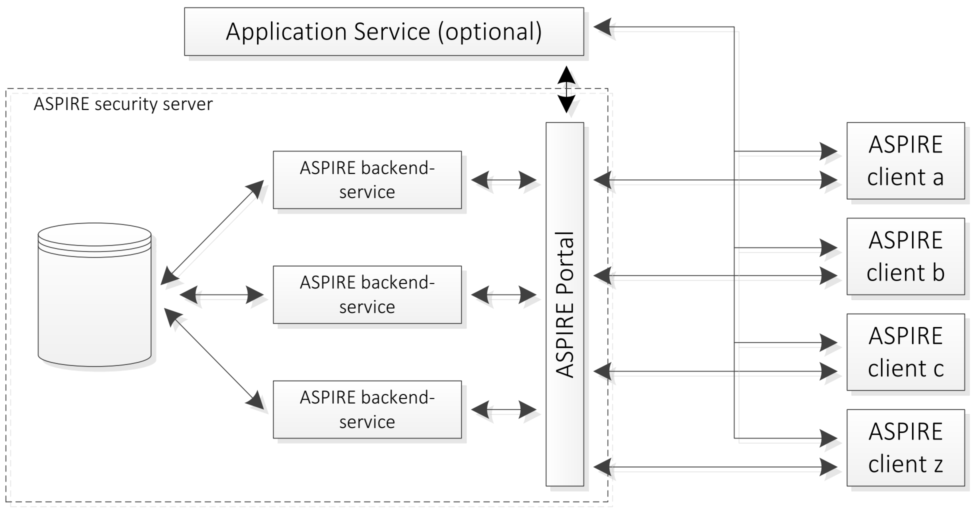

Figure 1 – ASPIRE multi-tier architecture: high-level view.

1.3.1 Motivation

This approach was selected for many practical reasons.

Firstly, the co-existence of the ASPIRE client-server communication and the original client-server communication was selected to avoid too much impact on existing application services. This was expressed by the industrial partners in the project, who seek to deploy ASPIRE protection techniques, but cannot do so when it would impact existing services too much. Additionally, given that ASPIRE aims to be as generic as possible; it cannot make too many assumptions on the client-server communication that may already be in place. That communication may come in a too large variety to leverage it as a building block for generic protection services. In some cases it can even be impossible to exploit that communication. For example, one-directional satellite communication serving live video cannot be exploited for remote attestation. Last but not least, the application service and the ASPIRE protection service may be the responsibility of different entities and may be running in different server infrastructure facilities.

While we strive for minimal impact on the original client-server communication, we nevertheless allow *some* impact on the original client-server communication. For example, instead of sending keys from the server to the client, the server might first ask some ASPIRE backend service to obfuscate the keys for a particular protected client instance, and send that as payload instead. Or the original client-server communication might be exploited to signal some request from the client to an ASPIRE backend service. For this purpose, we also allow communication directly between the application service, and the ASPIRE portal. This can then also be exploited by the application server for other means, e.g., to request a trustworthiness status on particular clients, upon which the application service can decide if it wants to proceed or not.

Secondly, a multi-tier architecture to support the ASPIRE protection techniques was selected because its flexibility, scalability, and reusability. A portal service is in place as a terminator for the secure link between protected applications and the server-side – as such the individual protection services do not need to take the communication protocol details into account. This portal would be a lightweight service that re-directs messages between protected applications and the relevant protection services. As such, protection services can be scaled onto different devices. This supports the adoption of the ASPIRE results in an industrial context. Additionally, this also facilitates concurrent development of protection techniques within the ASPIRE consortium, as protection services can run completely independently and even be embodied in any given form, such as a script, a local process, or a service on a different physical machine – as long as the ASPIRE portal knows how to communicate with them.

Alternative options, such as for example a monolithic server infrastructure, were considered. But given the advantages of multi-tier architecture that we just presented, we did not opt for them. Nevertheless, a review of the architecture and alternative options will be executed for the reference architecture review that will be reported in M24.

In conclusion, this basis for the reference architecture design has been selected for both practical reasons to support the development during the project as well as for potential future adoption in an industrial context. This addresses the requirements that were elicited in Deliverable D1.03 ("Security Requirements") – in particular on the impact of network-based protection techniques – and extends beyond that with respect to additional architecture recommendations elicited by the industrial partners.

1.3.2 Detailed view

For the sake of clarity, in the remainder of the document we will represent the reference architecture with a single client and assume that the server-side logic of each protection technique is implemented as independent services. This is depicted in Figure 2, where at the server-side the ASPIRE portal interfaces with individual protections services, each service depicted by a dashed box and potentially comprising several components. Additionally, a database is present at the server-side that is shared by the protection services. In the remainder of this deliverable, we shall refer this database (-infrastructure) as *the ASPIRE database or ASPIRE DB*. At the client side, we depict the different components of protection techniques. Components that correspond to the same protection techniques are depicted together in a single dashed box.

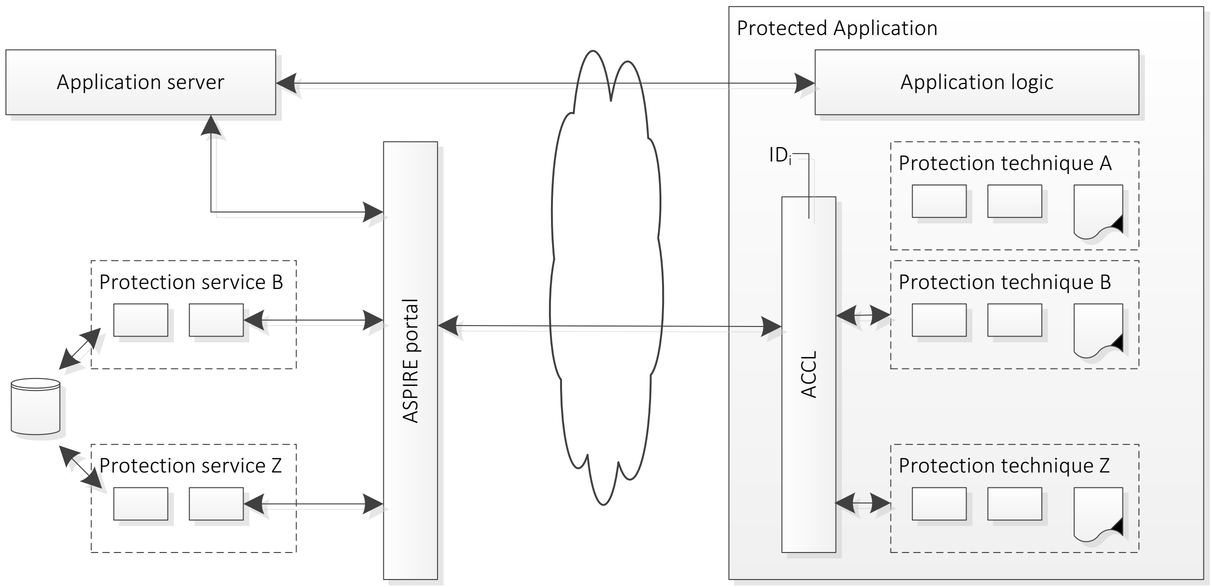

Figure 2 – Architecture view on a single client-server

To facilitate the communication between the client-side protection technique components and the corresponding server-side support components via the ASPIRE portal, we introduce a special-purpose communication logic: the ASPIRE Client-side Communication Logic (ACCL). This logic abstracts the communication for the protection techniques. A more detailed description is provided in Section 2. This abstraction facilitates easier development of the individual protection techniques. Additionally, it also allows us to reduce the focus of the protocol details. As we stated in the DoW and repeated in the Attack Model, man-in-the-middle attacks are out of scope in the ASPIRE project, as these can be solved by implementing state-of-the-art cryptographic protocols.

What is of importance in the definition of the protocol, however, is its high-level behaviour, and in particular how different communications and services are initiated and invoked. From a practical point of view, the best approach would be one in which the individual protected applications take the initiative to query the portal, and where the server-side response then depends on a stateless computation. Indeed, this approach can support many business models as it easily scales (due to its statelessness) and is independent of the client-side network infrastructure (e.g., clients can easily communicate with an HTTP portal while being behind a firewall or while hopping between different networks such as 3G and different Wi-Fi networks). Up to the extent possible, we do not favour any techniques where the ASPIRE portal solicits clients.

Nevertheless, allowing an active bi-directional communication channel rather than a stateless query-response channel can in some cases be an enabler for some novel and effective protection techniques, or greatly improve the performance and security of other protection techniques. Therefore, in some cases, we shall also allow active bi-directional communication between the ASPIRE portal and the communication logic in protected client applications. At this phase in the project, we are investigating the option of using WebSockets [RFC_WS] for that purpose. Using this technology, clients can initiate a channel with the ASPIRE portal, which the portal can use at any time during the lifetime of that channel to invoke client-side operations when it wants.

For example, while we would favour that the protected application initiates a request to the ASPIRE portal such as "could you give me a new piece of mobile code?" we would allow nevertheless that the ASPIRE portal can push some mobile code to clients at any given moment. We see in particular a big advantage in this approach if this can mitigate significant modifications that would else be required on the original client-server communication. We elaborate on this in Section 2 of this deliverable.

Finally, we remark that obviously individual clients need to be identified. For that purpose, each protected application instance will be associated with an ID. This ID will be shared with the protection services and the application server. We shall assume that within the ASPIRE project, an ID is fixed for each protected application instance (e.g., as a static variable) and that the ASPIRE database comprises a list of valid protected application identifiers. This pre-condition is set because account management and the establishment of such identifier is out of scope of the ASPIRE project; they are an engineering task for the application vendor.

1.4 Technique-specific architectures

In the subsequent sections of this deliverable, we shall describe the individual ASPIRE protection techniques that have been envisioned in the ASPIRE DoW. Each protection technique shall be described as a part of the architecture view presented in Figure 2. Covering each of these technique-specific architectures, we describe the full reference architecture of the ASPIRE protected application.

The reference architecture of each of the protection techniques will comprise the following content:

- An introduction to the protection techniques: the high-level objectives and concepts introduced. This also covers architecture-related assumptions and constraints that need to be taken into account.
- Details of each of the components that are introduced. These are distinctive components introduced both at client side (which will be integrated into the protected application by the ASPIRE tool flow), and server-side components that support the operational aspects of the protection techniques.
- A detailed description of the run-time behaviour of all the different ways in which the protection technique may operate.

Organized according to their objective, we elaborate on the following techniques:

- Anti-reverse engineering techniques
    - **Client-side code splitting**: an obfuscation technique developed in T2.3, where virtual machine components are introduced to execute bytecode that is functionally equivalent to native code that has been split from the original application.
    - **Anti-debugging**: an anti-tampering technique developed in T2.5, which specifies how a debugger component serves as a debugger of the protected application, preventing an attacker from attaching his own debugger.
    - **White-box cryptography**: an obfuscation technique developed in T2.2, dedicated to the protection of cryptographic keys in software. We elaborate on fixed-key implementations as well as on dynamic key implementations.
    - **Server-side code execution:** a technique to split code from the original application and execute it server-side.
    - **Code Mobility**: an obfuscation technique developed in T3.1, where a binary/library is incomplete. Missing code is then downloaded at run-time before it is executed.
    - **Multi-threaded crypto:** a source-level obfuscation technique developed in T2.4, which introduces a multi-threaded protocol to hide a cryptographic key.
- Anti-tampering techniques
    - **Code guards**: a tamper detection technique developed in T2.5, which introduces special-purpose integrity verification code into the client application.
    - **CFG tagging**: an anti-tampering technique developed in T3.2, which aims to detect when the execution flow graph is modified.
    - **Temporal remote attestation**: a technique that further extends the code guards approach by detecting tampering via execution time measurements.
    - **Call stack checks**: a technique developed in T2.5 that mitigates callback attacks. These are attacks where an attacker aims to inject malicious code into the protected application or library in the form of additional libraries.
    - **Anti-cloning**: a technique developed in T3.2 that introduces a method to enable the detection of clones via remote unique client identification.
    - **Delay data structures:** a technique that introduces a component which allows tamper verification and tamper response components to communicate the trustworthiness status of the protected application.
    - **Software time-bombs**: a technique developed in T3.2 that embodies a different type of delayed tamper response component.
    - **Combinations** of tamper detection and tamper response that result into new techniques

Note that this is only a subset of the techniques that have been presented in the DoW. Techniques that do not introduce any new components, but merely transform application code (such as local obfuscation techniques) are not included here as they do not have an impact on the overall reference architecture. Additionally, the multi-threaded crypto technique that is included in the list was not explicitly described in the DoW, but it instantiates the domain-specific implementation that is mentioned in Task 2.4.

Renewability techniques typically build on several individual protections and extensions thereof. The will be discussed in a separate Section 6.

Furthermore, a separate Section 5 is devoted to the composability of the individual protections.

1.5 Conventions and Notations

1.5.1 Workflow diagrams

In this document, we will support the description of individual techniques with workflow descriptions that detail the sequence of operations of the technique. This will additionally come with a figure to give a comprehensive overview.

The boxes in the figure represent components of the architecture, such as individual libraries or individual routines, both of which can be statically linked into the protected application or library. Arrows between those components represent a transition from one component to another. This could be a jump during the program execution, or a call with some parameters. In this sense, arrows represent some data passing between those components too.

1.6 Updates of version 2.0 compared to v1.0

This document features a major revision of the original deliverable D1.04. The most important changes are the following:

- Section 2.1.3: the decision not to spend engineering resources on implementing encryption in the ACCL has been documented.
- Sections 2.3 & 2.4: the WebSocket-based protocol support in the ACCL is documented and specified.
- Section 3.2.2: The fact has been added that we experimentally verified that the anti-debugging technique works on (unrooted) Android 4.0, 4.4, and 5.0.
- Section 3.3: Some vocabulary has been updated and minor design changes were made to the protection of Client-server code splitting, now that the technique has matured.
- Section 3.4: Code Mobility has undergone a major revision: the original in-place storage of mobile code blocks has been replaced by heap-based (and hence randomized and therefore more protected) storage.
- Section 3.5: The section has been updated, most importantly by adding time-limited WBC.
- Section 3.6: The vocabulary in this section has been updated and the discussion has been revised lightly now that the technique's design has matured.
- Section 4 on the anti-tampering tecniques has been restructured and has undergone a major revision, as the designs of several techniques have matured, and as it has become clearer which techniques will be supported within the limited time frame of the project.
- Section 5 on the topic of composability has been added.
- Section 6 on the topic of renewability has been added.

1.7 Updates of version 2.1 compared to v2.0

In response to the requests for updates in the review report of the second year technical review, several paragraphs have been added at the end of Section 2.1.2 regarding the security of the ASPIRE servers that provide the online protection support. The newly added text starts on top of page 10. Furthermore, a discussion of the use of WebSockets and the potential impact on security has been added in the introduction to Section 2.3 (i.e., before Section 2.3.1). This inserted discussion starts with the last paragragh on page 13.

Section 2 ASPIRE Protocol

Section Author:

Brecht Wyseur (NAGRA), Alessandro Cabutto (UEL)

2.1 Introduction

In this section, we present more details on the protocol and the logic to support the communication between ASPIRE-protected client applications and the ASPIRE server-side infrastructure.

Network-based protection techniques that are developed in this project use this logic, which ensures that messages between the protected applications and the server-side support are correctly transferred. Additionally, this logic makes sure that the ASPIRE protection techniques can be protocol-agnostic: the logic abstracts the transport stream. The advantage is that it becomes trivial to mount ASPIRE protection techniques in other protocols and adapt it to other scenarios without impacting the design of the individual techniques.

At the client side, the ASPIRE protocol is supported by a component that we denote as ACCL (ASPIRE Client-side Communication Logic). This component is implemented as a C library that is statically linked into the protected application or protected library. It exposes a C API that the ASPIRE protection techniques can use.

For practical purposes, the communication is over HTTP, which is supported at server side with an ASPIRE portal that is implemented as a web service.

Two different types of protocols are supported:

- A **Simple Request Protocol**, where a protected application takes the initiative to query the ASPIRE portal. This is the most natural protocol and is the main protocol for our software protection techniques.
- The **WebSocket Protocol**, where a session between the protected application and the server remains in place, and allows the server to take the initiative to query the protected application. This protocol is less favoured than the Simple Request Protocol because it is more complex. However, in some technique use-cases, a protocol where the server invokes a client-side function is inevitable or may make the protected application more efficient. This is for example the case with the remote attestation technique, described in Section 4.1.1.2.1. which is less secure if the client has to start the attestation process.

2.1.1 Application identifier

Each protected application is associated with a unique client identifier. This identifier is used by the server-side support of the protection techniques to keep track of different application instances. A list of legitimate identifiers will be stored in the ASPIRE database.

When protected applications communicate with the server-side support, the identifier needs to be communicated. The ACCL will ensure this. The ACCL has access to the unique application identifier that is stored at the client side, and includes this identifier in the payload that it sends to the ASPIRE portal.

The method by which this unique identifier is defined and integrated into the protected application is out of scope of the ASPIRE project. That is, personalisation of ASPIRE protected applications and server-side account management is out of scope. Instead, the protection techniques should assume that this is available, and in our prototypes we shall fix some identifiers into the applications and maintain the corresponding list of identifiers in the ASPIRE database. This approach demonstrates the use of the application identifier (even when fixed), without limiting the exploitation opportunity of the ASPIRE protection techniques. Putting such a personalisation operation in place is a pure engineering task; software personalisation is common practice for the industrial partners of the project, which have already existing solutions upon which they can build this.

2.1.2 Protocol security

The goal of the ASPIRE protocol is related to both functionality and security. The functional goal is to support the communication between the protected application and the server-side support of the individual protection techniques. The security goal is to protect the ASPIRE protection techniques against man-in-the-middle attacks. This includes (but is not limited to) the following attacks:

- Reverse engineering attack that extracts information from the client-server communication. This may be to extract confidential information, or to extract information that can be used to improve other attacks. For example, when an attacker is able to distinguish traffic that relates to different protection techniques, he may use this information to improve dynamic analysis of the protected application.
- Tampering attacks, in which an attacker attempts to modify the communication in a way that would render certain protection techniques obsolete, or that would render the server's tamper verification verdict incorrect.
- Replay attacks, where an attacker replays obsolete messages. For example, the attacker may attempt to replay messages that contain attestation reports that have been gathered at a moment before tampering of the protected application took place. In that case, he can lure the server-side remote attestation support into the perception that the application is still trustworthy.
- Impersonation attacks, in which an attacker attempts to falsify the identity of the application. For example, to execute remote attestation techniques on an un-tampered protected application, while executing another tampered application. Or he could attempt to mislead the verification server to avoid that the service that relates to his account would be terminated as a response to tamper detection.
- Proxy attacks, in which an attacker attempts to install a special-purpose service in between the ASPIRE portal and its protected application that interacts with the communication in a way that circumvents some of the software protection techniques. He could aim to do that to (for example) run multiple copies of the protected application.

In the case where the communication end points (the protected application and the ASPIRE portal) are secure (against Man-At-The-End – MATE attacks), solutions exist to mitigate these attacks. For example, HTTPS has been designed as an authenticated secure channel and aims to mitigate most of the attacks described above. Therefore, given that solutions against Man-In-The-Middle attacks exist, and given that it is not the main challenge of the ASPIRE project (see the ASPIRE DoW and Deliverable D1.02 “Attack Model”), we consider the implementation of such protocol out of scope. Thus, authentication, session key agreement, and covert communication are out of scope.

Since the ACCL abstracts the underlying protocol, we can safely assume that excluding the implementation of a secure protocol does not impact the practical exploitation of the ASPIRE results. Instead, to focus our resources to protecting against MATE attacks, we shall use a plain HTTP protocol. When the techniques need to be deployed in practice, the ACCL will need to implement a secure protocol, and then the ACCL library itself needs to be protected by the ASPIRE protection techniques to protect the protocol end-points against attacks.

In addition, we reiterate from the DoW and the accepted deliverable D1.02 Attack Model that in the scope of this project, the server side is considered secure. As we stated in D1.02:

> *"The same holds for Denial of Service (DoS) attacks. Since those do not explicitly recover any assets, and since no software protection techniques are capable of protecting against DoS attacks, they are clearly out of the scope of the ASPIRE DoW. Preventing such attacks is rather a network infrastructure or system security challenge.*
>
> *In our survey and attack model, we also do not consider attacks on the server as a valid attack class. In the DoW, it was made clear (e.g., in Figure 5 of DoW Part B) that the server is considered trusted, and that ASPIRE aims to protect client-side applications. Such attacks are rather a server-side system security challenge."*

So we do not consider attacks on the server platform, such as server denial of service attacks or sensitive data exposure or code injection on the server side.

Finally, we want to point out that vulnerabilities are out of the scope of the project. As was written in D1.02:

> *"Similarly, ASPIRE does not focus on preventing attacks based on software vulnerabilities (i.e., bugs). Mitigating vulnerabilities is a process that should be deployed during the software development and testing process."*

This implies that when we opt to reuse existing third-party software for implementing the communication between client applications and the ASPIRE security servers, we reuse them as is, without worrying about vulnerabilities in their client-side or server-side implementations.

2.1.3 External HTTP stack

The communication protocol that we will use in ASPIRE is mounted on HTTP. We opted for HTTP because this facilitates easy deployment of a portal infrastructure (based on a simple web service infrastructure such as NGINX [Nginx]) and because there is a vast amount of client-side support that we can use. We chose to use the cURL C library [Curl]. This library can be embedded into the protected application (statically linked and protected like any of the other libraries that co-exist in the protected application) such that the protected application can open sockets and communicate directly with the ASPIRE portal.

The ACCL interfaces with the HTTP stack, as depicted in Figure 3. The motivation for keeping the HTTP stack external from the protected application is mainly for simplicity. This should not pose any problems from a security point of view: the HTTP stack has no security sensitive role. It only opens connections and transfers packages between the ACCL and the ASPIRE portal. The packages contain payload that is properly secured by the ACCL. We finally decided to use a standard C library instead of the Android HTTP stack for ease of integration with all the protection techniques which are coded in C as well.

With regards to payload encryption we considered the performance overhead, and estimated it as insignificant compared to the network. We then decided to design but do not implement payload encryption. we also determined that applying encryption to the channel is only an engineering task that can be easily performed by companies when integrating ASPIRE’s protection techniques into their existing frameworks. While choosing our external HTTP stack this concern influenced our choice, in fact the full stack supports TLS/SSL natively.

If necessary, the additional code needed to support this encryption can be protected with the ASPIRE tool chain, just like any other application code and other protection code. It then suffices to annotate the source code of the encryption routines. Of course, the impact of applying these techniques on performance then needs to be considered, but that needs to be done as part of the performance-protection trade-off the whole client-side app.

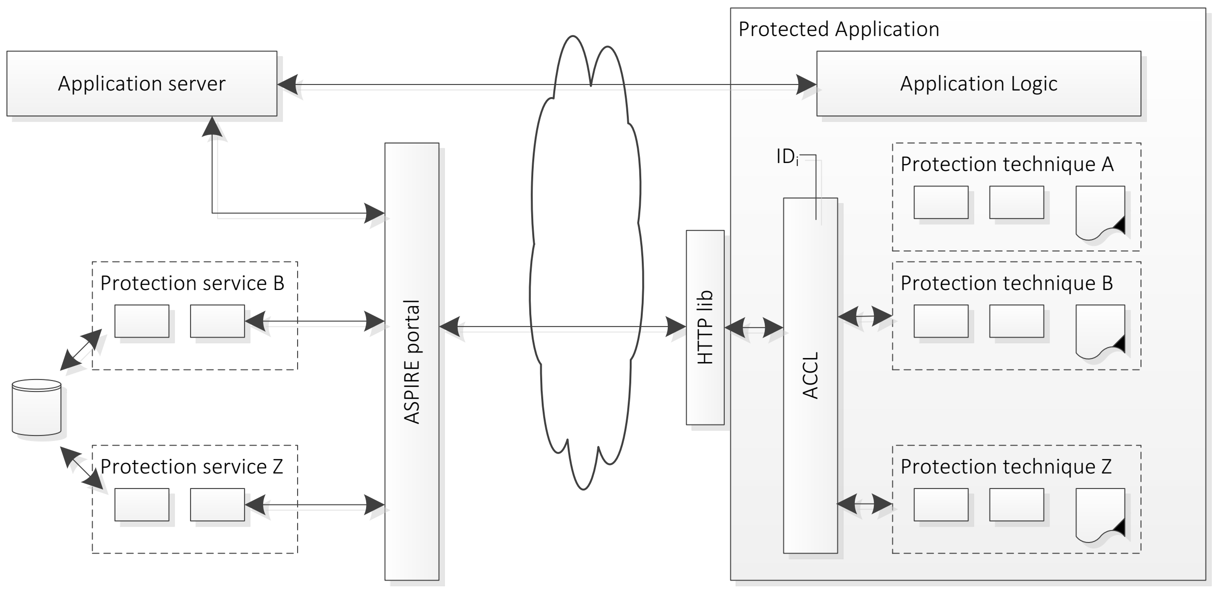

Figure 3 – The ASPIRE architecture with an external HTTP stack

2.2 Simple Request Protocol

The Simple Request Protocol captures a very simple protocol where the ACCL sends a single payload to the ASPIRE portal and optionally waits for an answer. In other words, the ACCL sends a single query to the ASPIRE portal, similar to sending a simple HTTP request.

Most of the network-based software protection techniques that are developed in the ASPIRE project use this protocol for the communication between their client-side components and the server-side support. It is the favoured protocol because it is the most natural and simple one, and does not impose a significant overhead on either the client or the server.

In this protocol, the protected application initiates the communication on a per-event base. Whenever a client-side component invokes the ACCL for sending a payload, the ACCL will interface with the HTTP stack to setup the communication and send the payload. This is depicted in the high-level sequence diagram in Figure 4, in which we make abstraction from the protocol used between the ACCL and the portal by depicting that the package is encrypted with a key *k*.

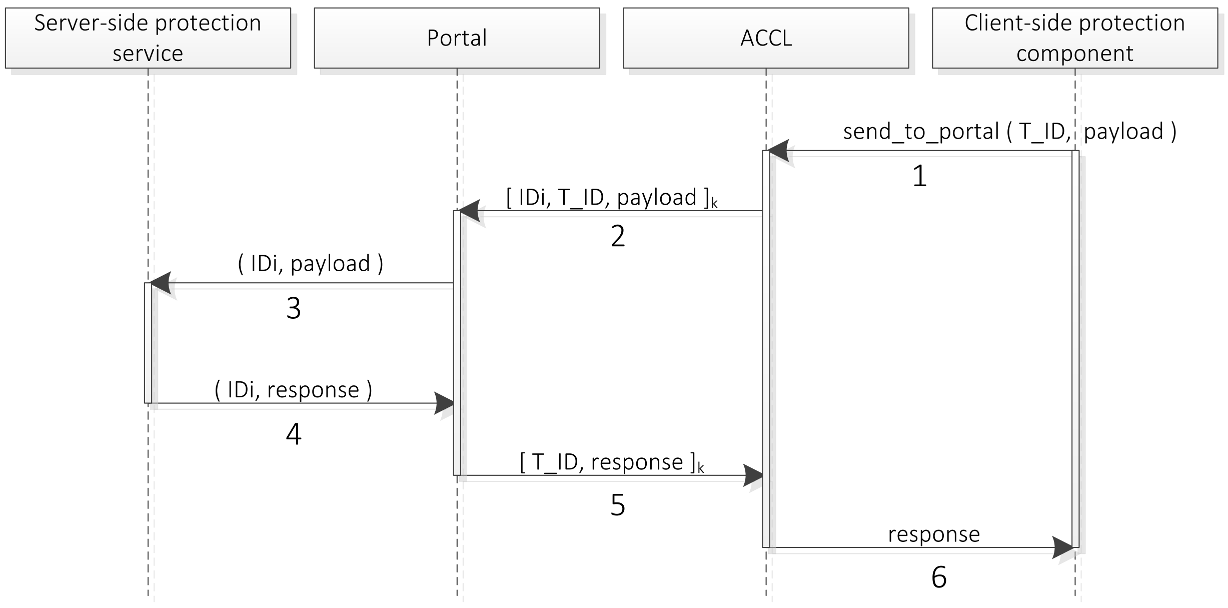

Figure 4 – The Simple Request Protocol

Seq#	**Operation description**
1	The ACCL is invoked by some client-side protection component.

Details: The client-side protection technique component calls the `send_to_portal()` function of the ACCL, with as argument

- The identifier of the technique, as listed in Section 7, and
- The payload that the component wishes to send to the technique server-side support.

2	The ACCL packages the received payload and transfers it to the ASPIRE Portal.

Details: The ACCL packages together the technique identifier and the payload with the application identifier. It protects the package in a proper way, for example by encrypting it with a session key, and sends the content to the ASPIRE portal.

3	The ASPIRE portal redirects the received package to the appropriate back-end service.

Details: The ASPIRE Portal, decrypts the received package and extracts the technique identifier then it sends the obtained (decrypted) package to the appropriate server-side support.

4	The server-side component of the relevant protection technique processes the package.

Details: The technique-specific server-side support receives the package, comprising the application identifier and the payload, and processes this. When appropriate, it sends back a response together with the application identifier to the ASPIRE portal

5	A response is sent back from the ASPIRE portal.

Details: The ASPIRE portal sends the response back to the appropriate application, in encrypted form.

6	The ACCL returns the return payload.

Details: The ACCL receives the encrypted package, decrypts it, and sends the payload (without the identifier) back in response to the call that has been made in Step 1.

2.3 WebSocket Protocol

For some software protection techniques tat we aim to develop in the ASPIRE project, the Simple Request Protocol may not suffice. This is for example the case with the remote attestation technique, described in Section 4.1.1.2.1. which is less secure if the client has to start the attestation process. To enable such techniques, it does not suffice that the client takes the initiative to send requests to the server: The server also needs to be able to invoke certain actions at the client.

In a standard setting, it is for several reasons not practically feasible for a server to invoke an action of the client application. The application may not have the privileges to listen onto network interfaces of the execution platform; the execution platform may not be reachable directly by the server because it might be behind a firewall; or it may be that the network configuration of the execution platform (which may be a mobile device) changes during the execution of the protected application.

To overcome these issues, we decided to use WebSocket technology. WebSocket is a protocol that has been standardized in 2011 and provides full-duplex communication channels over TCP [WS]. WebSockets are designed to be implemented in web servers. it is fully supported by NGINX since version 1.3, and there exist standard open sourced libraries for client-side support. We chose to rely on libwebsockets due to its lightweight footprint, robustness and its pure C implementation.

This technology is actually used within the ASPIRE project by the Remote Attestation protection technique. At application launch time, or upon a specific instance during the execution of the ASPIRE protected application, a WebSocket-based channel between the ASPIRE protected application and the ASPIRE portal is initiated. This channel is then used by the protection back end to invoke client-side functions. Additionally, we decided to use this technology to improve certain ASPIRE protection techniques (e.g. Client-Server Code Splitting) reducing the impact of communication overhead introduced by the Simple Request Protocol. WebSocket provides scalable low latency communication between peers so it can be helpful in scenarios where performances matter.

The WebSocket protocol is different from the HTTP protocol, but the WebSocket handshake is compatible with HTTP, using the HTTP Upgrade facility to upgrade the connection from HTTP to WebSocket. This allows WebSocket applications to more easily fit into existing infrastructures. For example, WebSocket applications can use the standard HTTP ports 80 and 443, thus allowing the use of existing firewall rules.

At client side a separate thread is needed to listen to the WebSocket channel and thus the ASPIRE-protected application inevitably becomes multi-threaded (if it wasn't already).

A new server side component called ASCL (ASPIRE Server Communication Logic) is introduced in order to manage WebSocket logic at server side.

Note that WebSockets may expose the applications to a number of security issues (both at the server- and client-side) and they are affected by a number of vulnerabilities. Namely, WebSockets are vulnerable to DOS attacks, require additional mechanisms for authenticating clients trying to send data and to implement authorization policies, require additional protections to mitigate attacks (e.g., sensitive data exposure, injection, malformed input data, Cross site WebSocket Hijacking), and they are vulnerable to tunnelling attacks.

However, we decided to use them for a set of reasons:

- As already mentioned in Section 2.1.2, we explicitly stated in the DoW that we exclude the creation and securization of the communication channels from the project scope.
- Similarly, protecting the server is not the primary focus of the ASPIRE project.
- Web sockets are certainly imperfect from a security standpoint. However, given the above limitations to the project scope, the ASPIRE partners agreed that they are the best solution to implement a server-to-client asynchronous communication, which allows us to focus on software protection rather than network-level channel implementations and computer security issues.

Thus our purpose here is to warn the reader that the use of Web Sockets may expose protected applications to security issues. Thus any party interested in exploiting the ASPIRE protections using WebSocket server (for remote attestation or client-server code splitting) should consider that administrators and application developers need to take care of the security issues or develop ad hoc communication channels.

2.3.1 Protocol initialization

Figure 5 visualizes the WebSocket protocol initialization to be initiated by a client app. It consist of 4 steps, that are detailed below.

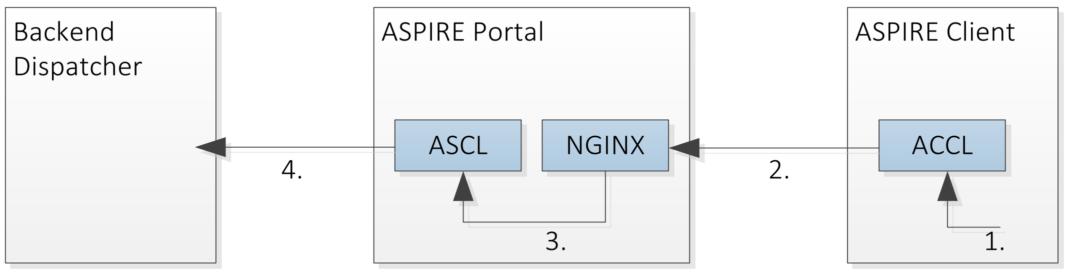

Figure 5 – WebSocket Protocol Initialization

Seq#	Operation description
1	The ACCL WebSocket is initialized by some client-side protection component.

Details: The client-side protection technique component invokes the `acclWebSocketInit()` function of the ACCL, with as argument

- The identifier of the technique, as listed in Section 7, and
- The callback to be invoked when data arrives from server.

2	The ACCL sets up a new connection to the Portal

Details: The communication logic assigns an instance identifier (handle) to the connection and sets up a new connection to the Portal passing through the technique identifier and the application identifier encrypted with a key *k*.

3	The ASPIRE Portal initializes an creates a new entry in the ASCL module

Details: After decrypting the request, a new entry composed by technique identifier and application is memorized into the ASCL module in order to manage data coming from the new client. An instance ID for the connection is combined with the entry.

4	The protection backend is informed about the new client

Details: Information about the new connection is provided to a resident service called Dispatcher for the given technique. From now on payloads coming from this channel will be forwarded by the ASCL to the dispatcher using a named pipe and identified by the instance ID generated at step 3.

2.3.2 Client initiated communication

1	The client-side component of the relevant protection technique processes a message to be sent to the server

Details: The client application prepares a payload to be sent to the server and calls the acclWebSocketSend() function passing the handle obtained during initialization and the payload. The ACCL packages together the technique identifier and the payload with the application identifier, encrypt the package and sends it to the ASPIRE portal.

2	Payload delivery

Details: The ACCL sends the payload to the Portal. The ASPIRE Portal decrypts the received package and then, based on the technique identifier, sends the obtained package to the appropriate server-side support. To minimize the latency in communication the package is transferred directly to a resident service (protection backend dispatcher) via a named pipe.

2.3.3 Server initiated communication

1	The server-side component of the relevant protection technique processes a message to be sent to the client

Details: The technique-specific server-side produces a package, comprising the application identifier, the payload and the connection instance identifier.

2	Payload delivery

Details: When appropriate, it sends the payload to the ASPIRE Portal via a named pipe so that the payload can be encrypted and delivered to the ACCL using the existing connection.

2.3.4 Scalability and Performances

The use of this technology raises a scalability issue: in an industrial scenario a huge amount of clients could request WebSockets based protection services at the same time. Possibly long-running connections between clients and the server will be kept active over time loading the ASPIRE Portal and making it a potential bottleneck. NGINX can act as a reverse proxy and load balancer for WebSocket applications improving the scalability of the solution; NGINX supports WebSocket by allowing a tunnel to be set up between a client and a back-end server.

According to WebSocket Performance test run by NGINX developers (https://www.nginx.com/blog/nginx-websockets-performance/) it requires less than 1Gb of memory and less than 1 core of CPU capacity to support 50.000 concurrent connections. Moreover, when loaded up with very busy connections, memory usage is stable and increase more slowly than payload size.

Therefore the overall architecture does not need an update: in a complex real world scenario the solution can scale up by adding the required computational power to the ASPIRE Portal server. The required amount of RAM and CPU can be deterministically sized depending on the number of expected active clients.

2.4 ACCL API

We present an API for the ACCL. This is a C API, which is used by the client-side components of the online protection techniques. It is an internal API that will not visible any more in the protected application as the API is obfuscated during the final steps of the ASPIRE tool flow operation.

The following functions need to be supported:

- `acclExchange( T_ID, payload)`
- `acclSend( T_ID, payload)`
- `acclWebSocketInit (T_ID, callback)`
- `acclWebSocketSend (handle, payload)`
- `acclWebSocketExchange (handle, payload)`
- `acclWebSocketShutdown (handle)`
-

In production, we will probably need some additional functions, such as `getServerState()`. This is not currently necessary but it might be defined later if needed by some techniques.

2.4.1 acclExchange

Description

Send a request to the ASPIRE portal, and wait for a return value.

Definition

```
int acclEchange (
     const    int         T_ID,
     const    int         payloadBufferSize,
     const    char*       pPayloadBuffer,
     unsigned int         returnBufferSize,
                char**    pReturnBuffer
);
```

Parameters

`T_ID`	[in] The identifier of the technique, according to the table presented in Section 6. This identifier will be used by the ASPIRE portal for redirecting the request to the appropriate security service.
`payloadBufferSize`	[in] Size of the payload char buffer in bytes.
`pPayloadBuffer`	[in] A pointer to a payload char buffer.
`returnBufferSize`	[out] Size of the return char buffer in bytes.
`pReturnBuffer`	[out] A pointer to a return char buffer.

Return

`int`	The return status of the operation	
	ACCL_SUCCESS = 0	if the operation was successful
	`else`	if the operation failed. Specific error codes as positive integers are defined by the API so that components at client-side can use for more fine-grained reaction.

2.4.2 acclSend

Description

Send a request to the ASPIRE portal, and return control as fast as possible. I.e., do not wait for any result from the Portal.

Definition

```
int acclSend (
     const    int        T_ID,
     const    int        payloadBufferSize,
              char*      pPayloadBuffer
);
```

Parameters

`T_ID`	[in] The identifier of the technique, according to the table presented in Section 6. This identifier will be used by the ASPIRE portal for redirecting the request to the appropriate security service
`payloadBufferSize`	[in] Size of the payload char buffer in bytes.
`pPayloadBuffer`	[in] A pointer to a payload char buffer.

Return

`int`	The return status of the operation	
	0	if the operation was successful
	`else`	if the operation failed.

2.4.3 acclWebSocketInit

Description

Initialize all the internal structures needed to operate on the WebSocket channel. This function must be called before sending or receiving any data though the channel. A dedicated thread is spawn in order to manage incoming data.

The function can be called only once per technique.

Definition

```
int acclWebSocketInit (
     const    int        T_ID,
              void*      (* callback) (void*, size_t)
);
```

Parameters

`T_ID`	[in] The identifier of the technique, according to the table presented in Section 6. This identifier will be used by the ASPIRE portal for redirecting the request to the appropriate security service
`(* callback) (void*, size_t)`	

[in] Callback function to be invoked when data arrives from the Portal. The callback must accept a pointer to the buffer containing data and the buffer size as arguments.

Return

`int`	The return status of the operation	
	-1	if the operation failed
	`else`	handle (numeric identifier) of the channel. This value must be used as reference for next WebSocket functions calls.

2.4.4 *acclWebSocketExchange*

Description

Send a websocket message to the ASPIRE portal, and return control to the application when a response is received.

Definition

```
int acclWebSocketSend (
        const    int         handle,
        const    int         payloadBufferSize,
                 char*       pPayloadBuffer,
        unsigned int         returnBufferSize,
                 char**      pReturnBuffer
);
```

Parameters

`handle`	[in] The handle obtained at WebSocket initialization.
`payloadBufferSize`	[in] Size of the payload char buffer in bytes.
`pPayloadBuffer`	[in] A pointer to a payload char buffer.
`returnBufferSize`	[out] Size of the return char buffer in bytes.
`pReturnBuffer`	[out] A pointer to a return char buffer.

Return

`int`	The return status of the operation	
	ACCL_SUCCESS	if the operation was successful
	`else`	if the operation failed.

2.4.5 *acclWebSocketSend*

Description

Send a websocket message to the ASPIRE portal, and return control to the application immediately.

Definition

```
    int acclWebSocketSend (
        const   int         handle,
        const   int         payloadBufferSize,
                char*       pPayloadBuffer
    );
```

Parameters

`handle`	[in] The handle obtained at WebSocket initialization.
`payloadBufferSize`	[in] Size of the payload char buffer in bytes.
`pPayloadBuffer`	[in] A pointer to a payload char buffer.

Return

`int`	The return status of the operation	
	ACCL_SUCCESS	if the operation was successful
	`else`	if the operation failed.

2.4.6 *acclWebSocketShutdown*

Description

Terminates the specified WebSocket connection. The communication with the server is closed and the thread associated with the channel ends.

This function should be called when the communication with the server is no longer needed, e.g. when the application quits.

Definition

```
    int acclWebSocketShutdown (
        const   int         handle
    );
```

Parameters

`handle`	[in] The handle obtained at WebSocket initialization.

Return

`int`	The return status of the operation	
	ACCL_SUCCESS	if the operation was successful
	`else`	if the specified handle is not valid or the connection was already shut down

Section 3 Anti-reverse engineering techniques

3.1 Client-side code splitting

Section Author:

Andreas Weber (SFNT)

3.1.1 Introduction

Client-side code splitting is the subject of Task T2.3. In this section, we present the components that are introduced to the ASPIRE protected application to support this protection technique. The actual splitting was reported in WD2.03 and D2.03 (M12) and its basic binary-level tool support was implemented for D2.02 (M12), in-time for integration into the ASPIRE tool chain in T5.1. More advanced binary-level tool support was implemented for D2.08 (M24).

Client-side code splitting raises the bar for program analysis and tampering by statically removing code portions from the native app or library and by instead executing semantically equivalent bytecode sequences in a security-oriented virtual machine (VM) locally embedded in the native app or library.

The technique takes an unprotected binary (executable or shared object) as input and translates suitable parts into functional equivalent bytecode, links the bytecode and an appropriate bytecode interpreter (i.e., the VM) into the binary and replaces the original instructions with glue code that, at run-time, executes the embedded bytecode inside the in-process VM.

The partitioning of the application code into bytecode and native code should to some extent be steered by the ADSS, which has to find a balance between the obfuscation goal (sensitive code runs inside the VM) and the performance overhead caused by security-oriented bytecode interpretation (which lacks, e.g., just-in-time compilation and fast dispatch mechanisms) and by the necessary serialization and deserialization of the physical processor state before and after each VM invocation.

3.1.2 System requirements and assumptions

- The architecture for client-side code splitting described here supports multiple ASPIRE-protected components, be it a (dynamically or statically) linked executable or a dynamically linked library: Each component links their own VM.
- Protecting multi-threaded applications is considered, including when the threads originate from unprotected code such as a Java VM that executes a Java application that invokes a native ASPIRE-protected code library.
- The successful application of this technique on Linux-based systems requires that the application is written in C and built using a Diablo-aware compiler/linker.

3.1.3 Client-side components

The following components are added to the application to implement the protection technique.

3.1.3.1 The embedded Virtual Machine

The VM consists of a collection of procedures that together implement the functionality of a custom bytecode interpreter. This code is linked into the application binary by the ASPIRE tool chain. Furthermore, its code is dispersed throughout the application code by means of Diablo's code layout randomization support. As such, this VM component is not a single, easily identifiable code region.

During its execution, the application invokes from time to time the VM and passes it the relevant program state and the address of the bytecode to interpret as a replacement of some original, native code that was removed from the application to hide it from inspection and tampering. The VM then fetches the bytecode and, starting from the passed program state, interprets the bytecode. This includes the computation of the address at which the execution of native code continues after the interpretation has finished.

The ASPIRE tool chain will customize the VM, i.e., its instruction set and/or implementation, to some extent, so that an attacker cannot simply reuse results such as a bytecode disassembler from previous analysis without modification. SafeNet will implement the diversification techniques as background during the project’s third year and will deliver them as an updated version of their cross translator.

3.1.3.2 Bytecode to be interpreted

For each code fragment that is removed from the application, a corresponding bytecode image is provided instead. All bytecode images are provided in object files that can be linked into the application binary by the ASPIRE tool chain linker. Again, Diablo's layout randomization capabilities are used to disperse the bytecode images throughout the app's own data and code.

3.1.3.3 Mobile Bytecode

Instead of embedding the bytecode produced to be run into the SoftVM this can be delivered and installed at run time using the Code Mobility technique (Section 3.4). Please see the Composability Section 5 for further details about this implementation.

3.1.3.4 VM Invocation Stubs

Each bytecode image is accompanied by a distinct native code stub. This stub is responsible for passing the relevant program state to the VM according to the interface accepted by this particular VM, for passing control to the VM, for translating the updated state computed by the VM back to the native app, and for passing control back to the native app at the correct address. More concretely, the stub captures the contents of the physical processor registers and then calls the VM with the captured register values and the address of the corresponding bytecode image. When the VM finished the execution of the bytecode, the stub writes the updated values back into the physical processor registers and passes control back to the application. The necessary continuation address is provided by the just interpreted bytecode image.

Inside the application, the original instructions are replaced with a jump to the corresponding native code stub.

Once the stubs are linked into the application, and the jumps have been inserted, Diablo will optimize and obfuscate the stubs in its surrounding code. The result will again be that the stubs are not easily recognizable code fragments.

3.1.4 Run-time behaviour of client-side code splitting

Figure 6 presents the basic sequence diagram, depicting the run-time behaviour of this protection technique.

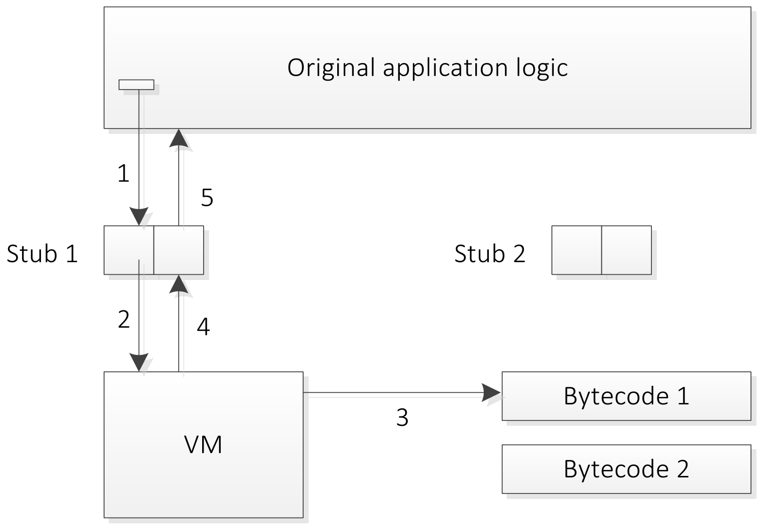

Figure 6 – Client-side code splitting run-time behaviour

A detailed description of each step depicted in Figure 6 is presented below.

Seq#	Operation description
1	The original application transfers control to the stub.

Details: Currently this is implemented as an unconditional jump into the first part of the stub 1 code. Conceptually but not yet implemented this jump could be removed by Diablo by means of branch forwarding, so, that the stub is inlined in the application code.

2	The stub sets up state for VM and transfers control.

Details: The stub collects the contents of the physical ARM processor registers and calls the VM, passing the address of the corresponding bytecode (VM-image) as argument.

When different stubs have different entry points into the VM, those entry points can be inlined in the stubs as well.

3	The VM fetches the Bytecode and interprets it.

Details: In case the bytecode is stored in encrypted form, the VM will need to decrypt it during this process.

4	After interpretation is finished, control is transferred to second part of the stub.

Details: The bytecode comprises code to calculate the address where the native execution should continue. This address and the updated register values are returned to the stub.

5	The stub cleans up and transfers control back to the application.

Details: The stub updates the physical ARM registers with the values the VM returned and jumps to the continuation address, transferring control back to the application.

3.2 Anti-debugging

Section Authors:

Bart Coppens (UGent), Stijn Volckaert, Bjorn De Sutter (UGent)

3.2.1 Introduction

The anti-debugging technique is part of Task T2.5 on Anti-Tampering. This section specifies the debugger component used for the anti-debugging requirement REQ-NFS-012 of D1.03.

The initial work on anti-debugging is reported in deliverables WD2.08 (M18), D2.08 (M24), with initial tool support in time for D2.07 (M24). This initial support will be extended in the following months, to be delivered in D2.09 (M30) and will be reported in D2.10 (M30).

The anti-debugging technique that will be developed in ASPIRE will be based on inserting a debugger component into the ASPIRE-protected application. This will allow us to reach two goals:

- **Anti-debugging**: First, the debugger component will serve as a debugger of the protected application, thus preventing an attacker from attaching his own debugger. The debugger component is tightly integrated into the protected application to prevent the attacker from easily disabling or removing the debugger component. This is achieved by migrating and rewriting parts of the protected application such that they are executed in the debugger's execution context instead of their original application context.
- **Obfuscation:** Secondly, because the transfer of control between the application context and the debugger context can be obfuscated, the migration of code from one context to the other allows us to obfuscate the application code.

The partitioning of the application code into the debugger and application execution contexts should to some extent be steered by the ADSS, which has to find a balance between the anti-debugging goal (which requires the debugger component to be launched before sensitive code is executed), the obfuscation goal (which requires the debugger component to be invoked during the execution of sensitive code), and performance overhead.

3.2.2 System requirements and assumptions

- The architecture for anti-debugging described here only works when the application contains only one ASPIRE-protected component to which the anti-debugging techniques has been applied, be it a (dynamically or statically) linked executable or a dynamically linked library. It is possible to extend the architecture to support applications comprising multiple ASPIRE-protected libraries on which the anti-debugging technique is applied, but to that end the described architecture and execution flow needs to be revised and extended.
- Protecting multi-threaded applications is considered possible, including when the threads originate from unprotected code such as a Java VM that executes a Java application that invokes a native ASPIRE-protected code library.
- While an extension towards multi-process applications is possible, in which the protected application process forks off a new application process that is also protected with the anti-debugging protection, that extension is not considered in the currently described architecture and execution flow.
- The successful application of this technique on Linux-based systems requires that:
    - The application can fork itself. This is the case for current Android versions, and is unlikely to change in future versions.
    - The forked off process can attach itself as a debugger to its parent process with *ptrace*. It is possible that in some future Android versions, not all applications will

have the permissions to do so. However, our current estimate based on studying online information sources and experimental validation is that is possible on (unrooted) Android 4.0, which the tool chain is required to target as per REQ-ASR-010 of D1.03, as well as on Android 4.4 and Android 5.

- The overhead of the protection technique can be mitigated if the target Linux-based platform supports reading /proc/pid/maps and /proc/pid/mem.

3.2.3 Client-side components

A debugger component is inserted into the application, and the original application code is partitioned in code to be executed in the debuggee's execution context, and code to be executed in the debugger's execution context.

3.2.3.1 Debugger component

Dynamically, the debugger component of the application will be a separate process, from hereon called *debugger process*. This debugger process

- is launched by the *application process* to be protected;
- runs concurrently with that application process;
- is attached to that application process as a debugger.

Instead of executing all code fragments in the application process, the application process will from times to times pass control to the debugger process by performing actions that are intercepted by the debugger process, and wait for control to return. The debugger process will then execute a code fragment that replaces the fragment to be executed in the application process, after which it will pass control back to the application process by letting it resume its execution.

Statically, the debugger component, from hereon called debugger code, consists of the code that controls the execution of the debugger process. This code takes care of the proper initialization where needed such as launching the debugger process by cloning (forking) the application process during its initialization, and initializing it, attaching to the application process, etc. The debugger code also aids in transferring control and data between the application process and the debugger process.

This debugger code is embedded in the application binary, i.e., in an executable program or in a dynamically linked library. The debugger code is inserted by Diablo and is based on code that is independent of the original application. The protected application or library initialization code is modified by Diablo to launch the debugger process and its initialization.

3.2.3.2 Code for Application/Debugger Contexts

The original program code is partitioned into code to be executed in two different execution contexts, corresponding to the application process and the debugger process. There is no *a priori* limitation on which code is executed in which context. However, guidelines will be made available that will focus on choosing a partitioning that reduces the performance overhead while offering the necessary obfuscation and/or anti-debugging strength.

Whenever a program fragment is migrated from the application context to the debugger context, Diablo replaces the code fragment with the necessary code to transfer control to the debugger, e.g., by replacing it with code that raises an exception of which the debugger component can identify the origin. Furthermore, the migrated code is rewritten to allow it to execute correctly in the debugger process, e.g., by replacing memory access instructions, which were originally executed in the application's memory space, by memory accesses through the ptrace API. In the scope of the project, only single basic blocks will be migrated.

3.2.4 Anti-debugging run-time behaviour

Figure 7 comprises the sequence diagram of the anti-debugging protection technique.

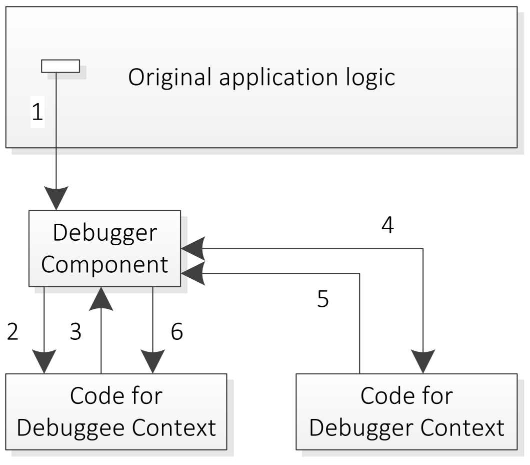
Original application logic
1
Debugger Component
4
5
2
3
6
Code for Debuggee Context
Code for Debugger Context

Figure 7 – Anti-debugging workflow diagram

A detailed description of each step of the workflow is described in this section.

Seq#	Operation description
1	The debugger initialization code is started from the library's or application's initialization code.

Precondition: If the protection is applied to shared libraries, only a single dynamically linked library may be protected with anti-debugging.

Overhead: There is a one-time cost per program execution, when the application starts (or the library is loaded).

Details: The initialization code forks the running application process. The forking thread in the application process halts until receiving a resume signal. The forked off process attaches itself as a debugger to its parent process, i.e., the application process, and sends the resume signal.

2	The debugger initialization process transfers control back to the application process, which continues execution where it previously halted.

3	The application reaches a code fragment that was migrated to the debugger context and transfers control to the debugger, which fetches the register context from the application process.

Overhead: Significant: throwing an exception, context switch.

Details: The debugger is invoked by the application by the latter throwing an exception, for example by dereferencing an invalid pointer or dividing by 0. The pointer and zero value can be dynamically computed with opaque computations to thwart static analyses. The available information at that time should suffice to let the debugger decide which fragment to execute.

4	The debugger transfers control to the corresponding, rewritten version of the migrated fragment. This code operates on the fetched register context and

	whenever it needs to access the application process state, it invokes utility functions in the debugger code.

5	Upon exit of the migrated code fragment in the debugger context, control is transferred back to the debugger code.

Details: Unless the debugger is being debugged by the debuggee (which is a possible extension of this technique), it cannot throw an exception to transfer control: the code has to call into the debugger to explicitly transfer control.

6	The debugger writes back the updated register context to the application process, and transfers control to the correct location in that process.

3.2.5 Impact

- Since there is no server-side component, this technique introduces no server-side overhead.
- There is a small overhead to initialize the debugger.
- For each switch between execution contexts, there is a fixed, significant overhead.

Depending on the instruction mix of the code executing in debugger context, and in particular on the number of memory accesses, the overhead on that code can be significant.

3.3 Client-server code splitting

Section Authors:

Andrea Avancini (FBK), Mariano Ceccato (FBK)

3.3.1 Introduction

The client-server code splitting technique in ASPIRE is part of Task T3.1. It is based on a set of source-to-source code transformations to modify the original application into the new ASPIRE-protected one.

The goal of client-server code splitting is to remove sensitive, attackable parts from the original client program and to move them on a trusted server. Let the *sensitive* variables of the original unprotected client be those variables that can be tampered by an attacker to interfere with the normal behavior of the application. The identification of these sensitive portions of code is performed by relying on a technique called *barrier slicing*. A barrier slicing algorithm returns barrier slices of code, similar to the concept of (backward) slices, as output. Let the slicing *criterion* be a set of program variables and a set of program statements. A backward slice is a subpart of the original program that is equivalent (assuming termination) to the original program with respect to the variables in the criterion, observed in the statements of the criterion. Practically, the slice for a given criterion includes all the statements that directly or indirectly (i.e., transitively) hold data or control dependencies on the variables in the criterion.

The notion of backward slice can be extended to the barrier slice. A barrier slice is a slice where some statements are considered "barriers", such that they block the backward propagation of control and data dependencies. Practically, variables in the criterion are those sensitive variables that are intended to be protected and thus should be moved on the server-side component, while variables in the barriers are those not security critical. They define a non-sensitive portion of program that does not need to be moved into the server. This kind of slice is computed by stopping the backward propagation of dependencies of a regular backward slice whenever one of the barrier statements is reached.

When the sensitive code is correctly identified, a new (protected) client is automatically generated, where the sensitive code is *sliced* away and only the subset of those variables that can be considered non-sensitive remains in the client. Then, any reference to sensitive variables is removed from the client. The new server component contains the slices, with the original references to sensitive variables preserved.

The new protected client, without sensitive variables, and the new server component execute synchronously and exchange data as needed by the distributed computation. Since the client still needs values of sensitive variables to run properly, a communication is established between client and server. The novel client-side component facilitates the message exchange between the modified client code and the ACCL (see Section 2) that will eventually take care of client-server network communication.

3.3.2 System requirements and assumptions

The portion of the application to protect must be single-threaded. The original application can be multi-threaded, but client/server code splitting can be applied only on single threads, i.e., client/server code splitting cannot be inter-thread.

3.3.3 Architecture Overview

The reference architecture for client/server code splitting, in case of an offline application is depicted in Figure 8. In the case of an on-line application, the architecture is exactly the same, with the only difference being the presence of the original server. However, nothing changes for the original client/server communication protocol and behaviour.

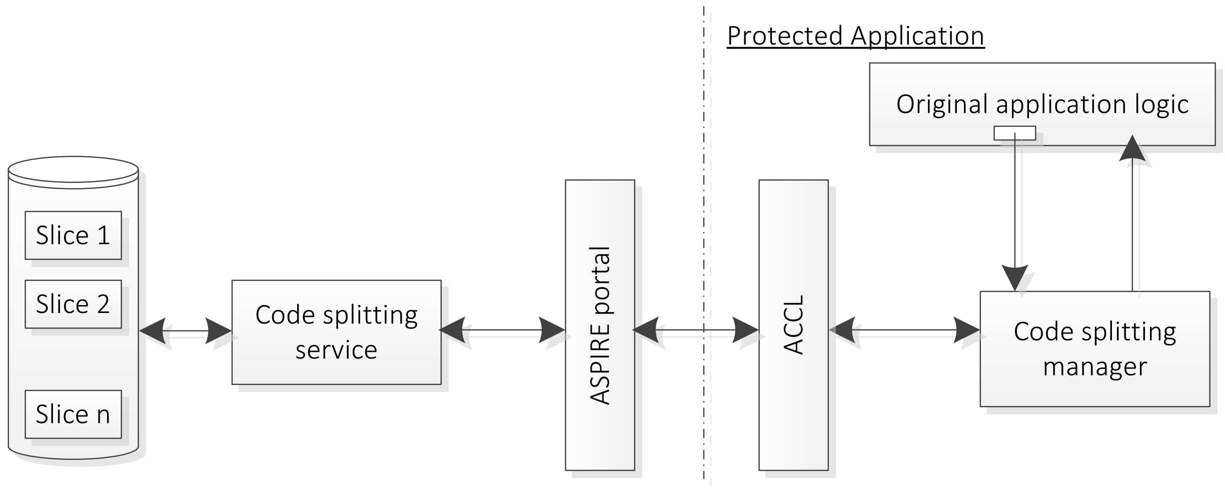

Figure 8 – Reference Architecture for client-server code splitting

3.3.4 Client-side components

3.3.4.1 Code splitting manager

The client/server code splitting technique introduces a new client-side component that manages the communication between the protected client and the server-side support, such that both sides remain synchronised. We denote this new client-side component as the *Code Splitting Manager*. It acts as a proxy and interacts with the ASPIRE communication logic to send messages to the server side, and to receive responses accordingly.

The code splitting manager exposes an API that is used by the protection technique. An overview of the API functions is presented in Table 1. At various places in the protected client application, calls to the API functions are introduced by the tool flow in such a way that the now split client application operates as intended. Note that this API is not visible in the protected application, as the (internal) boundaries will are obfuscated.

Table 1 – Code Splitting Manager API

Function Signature	Parameters	Payload	Description
int sync(int LABEL)	int LABEL	Message type (SYNC), LABEL, size of the message	Synchronizes with the server by sending the current execution point reached (indicated by parameter LABEL). Bootstrap message is a special sync message with different message type.
int send(int LABEL, int varLABEL)	int LABEL, int varLABEL	Message type (SEND), LABEL, variable value, size of the message	Sends a required value to server. LABEL works as for sync, while varLABEL marks the variable value to be sent.
int ask(int LABEL, int varLABEL)	int LABEL, int varLABEL	Message type (ASK), LABEL, label of the required variable, size of the message	Sends request for value. Waits until server responds.
int waitForValue(int LABEL, int varLABEL)	int LABEL, int varLABEL	None	Checks if required value of variable varLABEL from synchronization point LABEL is available.
int exit()	None	None	Builds and sends the exit message to notify the server to close the connection.

Synchronization points implemented by calls to the `sync` function are used to keep client and server executions aligned. These calls replace any definition of sensitive variables that was present in the original code. Calls to the function sync are non-blocking, in fact the client communicates the server which point of the execution is reached and then continues its execution.

Whenever a value of any of the sensitive variables is required by the client, calls to function *ask* are used. Calls of this type are blocking, in fact the client sends a request for a value of a sensitive variable that is needed for progressing in the computation, and waits for the answer before resuming its execution. The function `ask` replaces the uses of sensitive variables in the original code.

Immutable statements, like user inputs, are those statements that cannot be moved to the server, since they represent an active and required task in the original application. This means that values from immutable statements need to be sent to the server in order to perform the correct computation of the sliced code at the server side. Calls to the function *send* are used by the client to deliver the requested values to the server, and also as synchronization points like in case of sync function. Calls to function send are non-blocking, values are sent to the server without waiting for any confirmation of reception.

Barrier variable values are sent to the server by using the same function *send* described earlier.

While there is an `exit()` function, note that there is no `init()` function. The initialisation will be invoked by the server when the first message from the client application is received.

3.3.5 Server-side components

3.3.5.1 Slice manager

This server-side component handles connections and messages from and to the client. It is also responsible to launch the correct sliced code whenever a new client connects.

The backend dispatcher parses the payload received from the ASPIRE portal, and then invokes an internal server-side function. Table 2 presents an overview of the API that is supported.

Table 2 – Server-side internal code splitting API

Function Signature	Parameters	Payload	Description
void process()	None	None	Handles incoming messages and connections
int sendValue(int LABEL, int varLABEL)	int LABEL, int varLABEL	Message type (SEND-VALUE), LABEL, value of the required variable, size of the message	Sends a required value to client. LABEL identifies a previous request from client, while varLABEL marks the variable value to be sent (when needed)
void * loadSlice()	None	None	Executes the requested slices.
int checkSync(int LABEL)	int LABEL	None	Checks if current synchronization point (identified by LABEL) is reached by client.
int waitForValue(int LABEL, int varLABEL)	int LABEL, int varLABEL	None	Checks if required value of variable varLABEL from synchronization point LABEL is available.

3.3.6 Messages

The local code slice manager and the backend dispatcher at the server side exchange messages structured like in Figure 9. It comprises the following data fields:

- *Message Type*, which represents the type (i.e., synchronization, value delivery, request for values) of the message itself and it is encoded as a 32 bit integer.

- *Message Label:* a label that identifies the point in the code that originated the current message. Messages originated by different parts of the application have different labels, while messages produced within loops by the same origin carry the same label.
- *Variable Label* identifies a variable for which a value request has been originated by either the client or the server.
- *Message Size* represents the total size of the message.
- *Payload* contains variable values when requested.

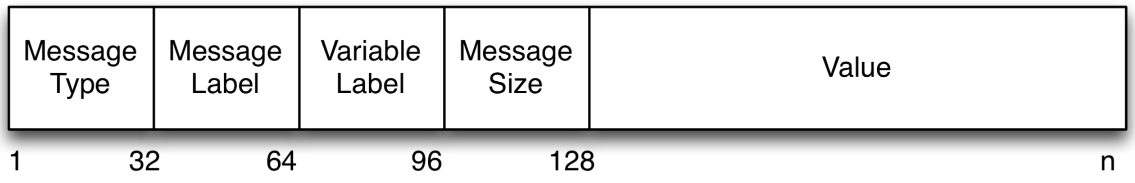

Figure 9 – Structure of a message

3.3.7 Client/server code splitting splitting sequence diagram

Figure 10 comprises the sequence diagram of the protection technique, followed by a detailed description of each step depicted. The figure depicts a prototypical execution of the protected application, where `client:Client` represents the client, while `backendDispatcher:Server` represents the slice manager that handles connections and messages, and `slicedCode:Server` is the sliced code at the server side.

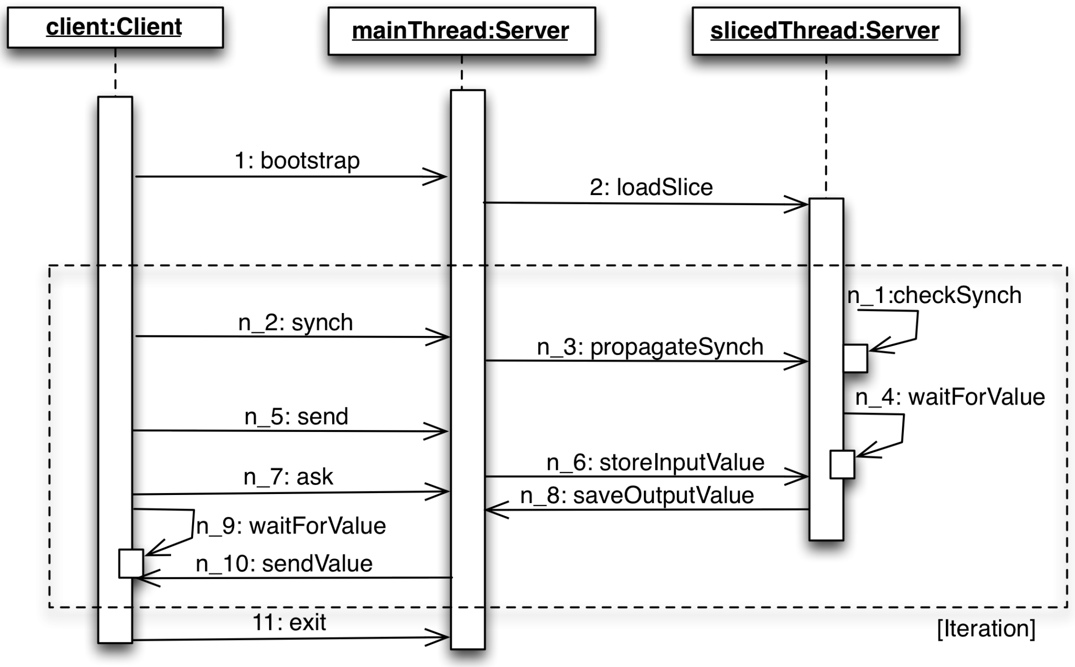

Figure 10 – Sequence Diagram for Code Splitting

Seq#	**Operation description**
1	The protected client starts and sends a bootstrap message to the server.

Details: The client (labelled `client:Client` in Figure 10 starts its execution and sends a bootstrap message (bootstrap) to the server's dispatcher (`backendDispatcher`)

Pre-condition: The server is up and able to handle connections. The client has not sent other bootstrap messages.

Post-condition: The server is ready to start execution of the sliced code.

Data passing: Client message to server contains specific message label for bootstrap.

2	The dispatcher at the server side loads the requested slice by invoking the corresponding process.

Details: Upon receiving the bootstrap notification, the server invokes a new process (`slicedCode:Server`) that is responsible for executing the sliced code.

Pre-condition: The server has received a bootstrap message from a client.

Post-condition: Requested slice is running. Both client and server are running the same piece of code synchronously.

The following operations, operation 11 excluded, can be executed multiple times in an iterative process

n_1	The execution of the slice code waits for synchronization messages from client.

Details: The process that handles the sliced code reaches a synchronization point and suspends its execution; it waits for a message from the client to communicate the same synchronization point has been reached also on the client-side.

Pre-condition: Sliced code is running.

Post-condition: Execution of the sliced code is suspended.

n_2	The client sends a synchronization message to server.

Details: The client, whenever a synchronization point is reached, sends a message to the server to signal the current status of the execution

Pre-condition: The client reaches a synchronization point while executing its copy of code without sensitive variables.

Post-condition: The server is ready to propagate synchronization information to sliced code.

Data passing: Client message to server contains specific message label for synchronization.

n_3	The server propagates the synchronization acknowledgement to the sliced code.

Details: The serve propagates synchronization status coming from client to sliced code to resume execution until the next synchronization point.

Pre-condition: The server has received a synchronization message from client; the sliced code is waiting for notification.

Post-condition: The sliced code resumes its execution; the two executions (on client-side and on server-side) are now aligned.

n_4	The sliced code is waiting for values coming from server.

Details: An input value or a value of a barrier variable is required by the sliced code to continue its execution. Since these values do not come with the barrier slice, the sliced code on server needs to be fed by the client with proper communication messages. When a value is needed, the sliced code stops its execution and waits that value to be available.

Pre-condition: The sliced code requires input or barrier values to proceed.

Post-condition: The sliced code stops to execute and waits for communication from the client.

n_5	The client sends a value to the server.

Details: The client sends a new message that contains the value needed by the server as payload.

Pre-condition: The client reaches a synchronization point while executing its copy of code without sensitive variables. The synchronization point requires the client to send values to server. The sliced code is waiting for values from client.

Post-condition: The server has received the required values and it is ready to notify the sliced code.

Data passing: Required values, specific message label for synchronization.

n_6	The server stores the required values to resume sliced code execution

Details: Upon reception of message from client, the server extracts and propagates the value to sliced code; the execution of the slice can resume.

Pre-condition: The server has received a message coming from the client; sliced code is waiting for values.

Post-condition: The execution of the slice code is resumed; executions on client-side and server-side are aligned.

n_7	The client sends a message to the server, requesting values of sensitive variables.

Details: Whenever a protected value is required, the client prepares a message that is delivered to the server.

Pre-condition: The client needs a value that is computed on the server-side.

Post-condition: The client is ready to suspend its execution.

Data passing: Specific message label for value request.

n_8	Sensitive values computed by the sliced code are stored and ready for delivery.

Details: Protected values are computed by sliced code and then stored to be accessible by the server. After receiving a request message, the server checks the availability of a fresh value for the variable requested: if the value is ready, the server started packing it; if the value is not ready; the server waits until the sliced code emits a fresh value and then proceeds as in the previous case.

Pre-condition: The client is waiting a fresh value for a sensitive variable; the server waits for this value from sliced code.

Post-condition: The value is ready and the server is about to pack and send it.

n_9	The client is waiting for message from the server.

Details: The client has stopped its execution since new sensitive values are required to continue. After having sent a request to the server, the client is waiting for answer.

Pre-condition: The client has sent a request to server.

Post-condition: The client is waiting for answer.

n_10	The server sends the requested value to the client.

Details: When ready, the requested value is sent to the server.

Pre-condition: The client is waiting for the value.

Post-condition: The client resumes its execution.

Data passing: Required values, specific message label for value request.

11	The client sends a message to server to notify its exit.

Details: The sliced code terminates to execute autonomously, when all the synchronization points are passed and no other instructions remain to execute. The client, whenever the computation reaches its conclusion, sends a closing message to the server and exits, while the server closes all the connections and also exits.

Pre-condition: The client has concluded its execution.

Post-condition: The client stops; the server exits if sliced code has terminated and no other operations are running.

Data passing: Specific message label for exiting.

3.3.8 Impact

- The client-server code splitting technique introduces a new server-side component, and adds communication between the protected application and the server-side support. This introduces additional complexity and latency.
- Sliced code runs in an Android emulator that must be available at server-side.
- For each client connection, the server component needs to launch a a new Android process in the emulator to serve such client. In case this protection is applied to N distinct places in the same client, and they belong to N distinct threads, the server needs to activate up to N different Android processes per connected client (one for each independent slice).
- The protected application needs to pause and resume execution when values need to be sent between the client and server. This introduces additional latency, and may cause issues when the server is not responsive.
- Client/server connectivity is required to run client code; offline execution is not supported.

3.3.9 Limitations

- Support to C structs is limited. Whenever a field of a struct is annotated as sensitive variable, the computation of the barrier slice propagates the dependencies from the annotated field to the whole struct, which is consequently moved on the server.
  Sliced code runs in an Android emulator, that must be available at server-side. Moreover, a slice can be considered a distinct Android process to be executed when a client connects. This can pose pose concerns in terms of scalability of the approach.

3.4 Code Mobility

Section Authors:

Paolo Falcarin, Alessandro Cabutto (UEL)

3.4.1 Introduction

The Code Mobility technique that is being developed in the ASPIRE project, along with other online network-based techniques, aims to overcome the drawbacks of local protection techniques, by using a trusted server placed on the network, which is in charge of providing static code blocks dynamically delivered to the untrusted client. In this approach, a client application (or library) is stored on the user device as an incomplete executable that does not contain all the application's code. A Downloader component and a Binder component are introduced on the client-side by this technique: they are able, respectively, to fetch binary code blocks from a trusted server at run time, and to patch these into the running process' memory, allocating the dynamically delivered code in the application's heap memory. On the server-side a Code Mobility Server component responsible of blocks delivery is introduced.

This approach aims for mitigating reverse engineering: instead of preventing analysis of code by making the code complex, we make sure that the code is not available for analysis on the client device as long as possible, and deliver the necessary code only when it actually needs to be executed on the client device.

The Code Mobility technique can be seen as a dynamic binary obfuscation approach based on the deployment of an incomplete application whose code arrives from a trusted network entity as a flow of mobile code blocks, which are arranged in memory at run-time with a configurable memory layout.

Code Mobility (T3.1) is one of the ASPIRE methodologies to perform code splitting along with other techniques like client-server code splitting (barrier slicing, T3.1) and VM-based client-side code-splitting (T2.3). More in general, code mobility might be seen as the key technology of WP3 as it is the framework on which other online protection techniques might rely. For example the code attestators in remote attestation (T3.2) can be sent through the code mobility framework, and renewability (T3.3) will extend code mobility by allowing renewable code blocks.

The initial work on this subject has been reported in deliverable D3.01 (Preliminary Online protections report - M12), current status is described in deliverable D3.04 (Intermediate Online Protections report – M18) and future work will be reported later on in deliverables D3.06 (Remote Attestation and Server report - M30), and D3.08 (Renewability report – M33).

Mobile Blocks granularity is actually at a function level and the amount of functions to be made mobile can be defined in the ASPIRE tool-chain JSON annotations input file by specifying their names, even using wildcard character '*'. The amount of functions made mobile can be tuned to achieve an acceptable trade-off between overhead (bandwidth consumption, execution delay) and protection level. Mobile blocks are downloaded by the protected application when needed using the ACCL API.

In the protected application each and every call to mobile code is wrapped by an invocation to the Binder component (they are actually replaced by an indirect jump) and, during execution time, the mobile code is downloaded and installed into the heap where an appropriate amount of memory is allocated by the Download component. This process is better explained later on in Section 3.4 and fully treated in deliverable D3.02 Section 3.3.

Further extensions to this approach might be considered to make dynamic analysis more complex. For instance, downloaded code blocks, can be erased from memory after use making harder possible dump attacks.

The network communication between the Code Mobility server and the client can only be initiated unidirectionally from client to server; meaning that the client asks for a new code block and the server answers by sending it. These connections are short-lived and the server is not actually required to be keeping status about clients.

We will explore the possibility of a bidirectional communication with the exchange of control information when code mobility will be extended to integrate other online techniques such as remote attestation and renewability.

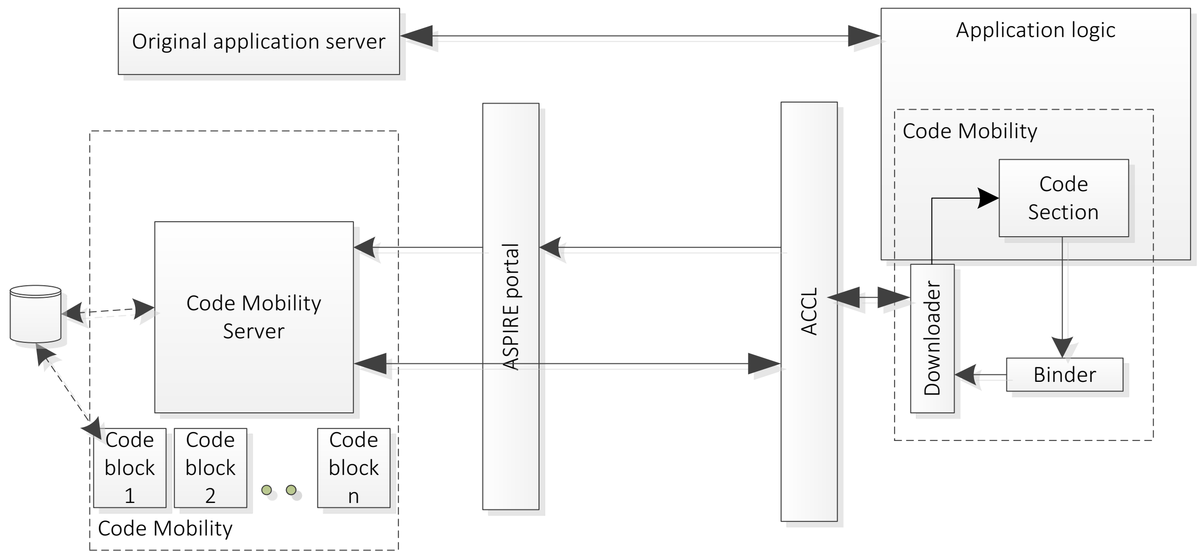

Figure 11 – Code Mobility High-Level Architecture

3.4.2 *System requirements and assumptions*

Code chunks are delivered by a remote service, and thus stable network access is strictly required at that point. In the current design, a continuous network connection will be required. Finally, when all the Mobile Code blocks are delivered, a connection is not required anymore.

The execution of mobile code blocks from the heap requires the application to call the mprotect() syscall in order to change the protection of that specific memory area to execute only. The assumptions here are that mprotect() can be called over the heap and mobile code blocks are stored in dedicated page aligned memory areas. The latter requirement is imposed by mprotect() which can only be used with full memory pages and, of course, by the need of applying the PROT_EXEC permission to the code block; moreover allocating dedicated memory pages for each mobile code block prevents possible concurrent write-access to the same block.

3.4.3 *Client-side components*

3.4.3.1 Downloader

The Downloader component invokes the ACCL API in order to obtain a specific mobile code block from the Code Mobility Server. It is in charge of allocating memory for the incoming code block and to provide a pointer to the buffer containing it.

3.4.3.2 Binder

When control is to be transferred in the client application to a mobile code fragment, the Binder relays on a set of addresses that act like a redirection table to determine whether the actual Mobile Code Block has to be downloaded or not.

The Binder component needs a custom, statically allocated table that stores target addresses of jumps into mobile blocks, i.e., the entry points of the mobile code blocks. Initially the table is filled with the address of the Binder, so that upon the first jump into a mobile code block, the Binder is actually invoked. After loading the block (via the Downloader component), the Binder will then overwrite the target address of the block's entry points with the addresses in the downloaded block. Then each subsequent jump into that block will directly go to the block rather than invoking the Binder again and again.

As previously discussed we will consider the option of erasing downloaded code block after use (or after a certain number of uses) restoring the initial indirection through the Binder.

3.4.4 Server-side components

3.4.4.1 Code Mobility Server

The Code Mobility Server is responsible for sending the binary code blocks to the clients when they are requested. It does not keep track of existing sessions with clients.

3.4.5 Code Mobility run-time behaviour

Figure 12 depicts the mobile code workflow diagram, and is followed by a detailed description of the referenced steps.

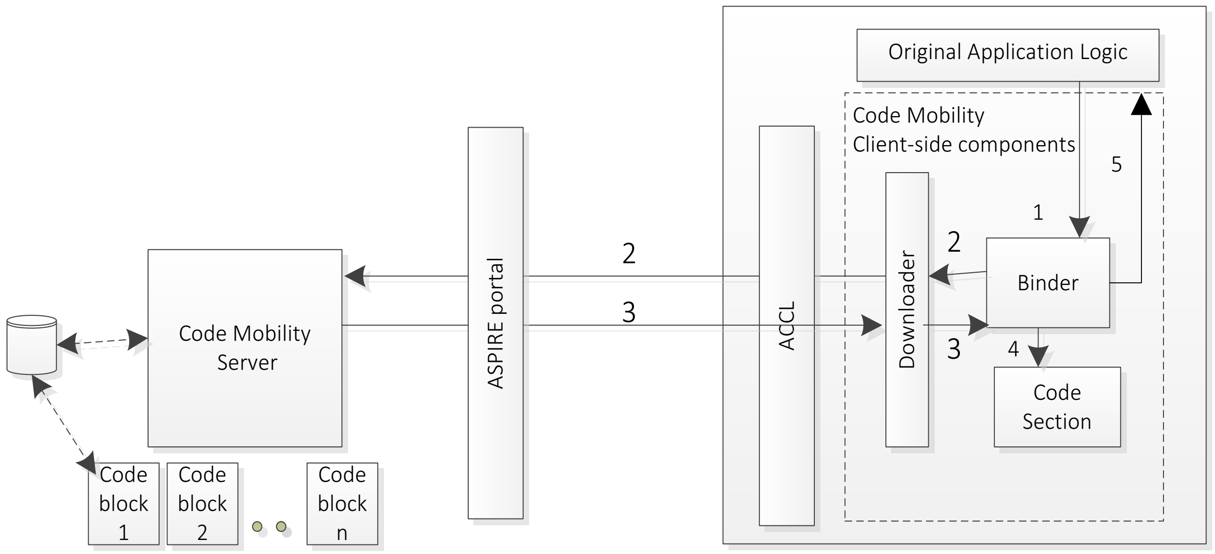

Figure 12 – Code Mobility workflow diagram

Seq#	Operation description
1	Binder invocation

Details: In the protected application all jumps into mobile code have been replaced by indirect jumps that take their target addresses from a statically allocated table that initially contains the Binder address for each entry.

2	Downloader invocation

Details: The Binder invokes the Downloader passing it an identifier to the mobile code block that should be downloaded. The Downloader establishes a connection to the Code Mobility server, through the communication logic and the ASPIRE portal, by sending sending the appropriate Technique ID and Application ID.

3	Code block delivery

Details: The Code Mobility Server Component serves the requested mobile code block to the client application.

4	Code patching

Details: The downloaded code block is allocated in the heap in a memory area that is made executable.

The Binder will then replace the addresses in the statically allocated table with the addresses of the just patched entry points. Then each subsequent jump into that block will directly go to the block rather than invoking the Binder again for the same call.

5	Return to original application logic

Details: Finally the Binder transfers control back to the original application, by continuing execution at the target address of the jump that was diverted to the Binder.

3.4.6 Impact

The technique comes with additional server load and significant performance impact.

The client-side performance impact mainly comes from the download latency and less significantly from patching process. This could be tuned by configuring the download process to transfer several code blocks into single packages and/or or pre-ship such packages. The pre-shipping could be fine-tuned using heuristics or special-purpose deterministic techniques that predict which code blocks the application may need for its execution. This, as well as the discarding of code blocks once they have been executed, introduces a trade-off between download latency and code hiding.

Additionally, the code blocks may be compressed prior to sending to reduce the bandwidth consumption.

A detailed report about overhead introduced by Code Mobility and performances analysis can be found in D3.04 Section 2.

3.4.7 Error management

Network access is assumed. If the server does not respond within a predefined timeout when the application asks for a specific code block it will be shut down gracefully.

The Downloader will check the format of received info before providing it to the Binder.

3.4.8 Composability

This technique is in general orthogonal and independent from most of the other offline and online techniques developed in ASPIRE. Mobile code is downloaded through a secure channel and passed to the Binder but at this stage it could be tampered with;. therefore, it could be paired with other anti-tampering techniques.

As far as we know, it might conflict with any other protection trying to access the code segment at run time, such as code guards, as these could try to calculate the hash of a code section not yet downloaded. Remote attestation could also read the code segment and conflict with code mobility, unless the attestators will be implemented into the Code Mobility framework as a special code block to be downloaded, run and then discarded from memory.

These interactions are discussed in detail later on in this document in Section 5.

3.5 White-box cryptography

Section Author:

Brecht Wyseur (NAGRA)

3.5.1 Introduction

Standard cryptographic implementations such as those available in open source libraries such as OpenSSL and LibTomCrypt, are vulnerable to key extraction attacks. During their execution, they store information related to the cryptographic key in memory; memory dump analysis can then easily lead to the recovery of the cryptographic key. White-box cryptography aims to prevent key recovery attacks. It does so by replacing the original implementation with a special-purpose implementation.

We distinguish two different types of white-box implementations:

- Fixed-key white-box implementations: these are implementations that hard-code the cryptographic key into the code. Changing the key requires the entire implementation to be replaced.
- Dynamic-key white-box implementations: these are implementations that can be instantiated with a key. Obviously, the cryptographic key itself cannot be presented to such implementation; a protected/obfuscated form needs to be presented. This requires an additional building block that is able to protect such key. This ProtectKey building block can reside at the client-side (but then needs to be well hidden) or it can be used at server-side.

Task 2.2 comprises the R&D track on white-box cryptography. This comprises both the research for new techniques (both theoretical approaches as practical approaches) as the implementation thereof. This includes the implementation of a white-box tool (WBT). The WBT is a framework that is capable of generating fixed-key and dynamic key white-box implementations and any supporting functions (such as a ProtectKey function) that might be relevant. The details of these activities have been disclosed in deliverable D2.04 and D2.08. The fixed-key implementations that have been developed in Year 2 of the ASPIRE project are considered a trade-off between performance and security. We consider them only to have a limited time validity and thus they will need to be renewed in due time. This is subject of the research in Task 3.2 that will be executed in Year 3 of the ASPIRE project. We elaborate on this in Section 3.5.7.

In Year 2 of the ASPIRE project, time-limited WBC techniques will be developed. This is an approach where a trade-off between fixed-key and dynamic-key white-box implementations is established. Fixed-key implementations have the advantage of being more secure and faster than dynamic key ones, but they can only instantiate a single key in their code. With time-limited implementations, we envision to develop faster fixed-key white-box implementations, but compensate the security loss with renewability: updating these implementations regularly. The support for this will be developed in Task 3.3 and reported in deliverable D3.04. Since the design of these time-limited white-box implementations have not yet started, we do not elaborate on this in the current reference architecture. This will be reported in the revision version of the reference architecture in M24.

3.5.2 Client-side components

3.5.2.1 White-box crypto library

The code that represents the white-box implementation itself is a library that is statically linked into the protected application or library. It comprises an API that can be used to invoke the cryptographic functions.

For fixed-key implementations, the implementation is invoked with as argument a pointer to a plaintext and ciphertext buffer. In the case of a dynamic key white-box implementation, a pointer to a protected key buffer is provided additionally. The API of these calls is presented in Deliverable D2.04 as the WBGC API.

3.5.2.2 Encoding/decoding/protectKey function

The client-side statically linked white-box library may optionally comprise additional supporting functions.

- ProtectKey – a function that transforms a cryptographic key into a protected/obfuscated key that is compatible with the white-box implementation itself.
- EncodeInput – a function that can encode the input to the white-box implementation (plaintext/ciphertext for an encryption/decryption function).
- DecodeOutput – a function that can decode the output from the white-box implementation (ciphertext/plaintext for an encryption/decryption function).

The EncodeInput and/or DecodeOutput functions are sometimes required, because white-box implementations might comprise additional functions to protect their input and output. These encodings aim to mitigate attacks on the first or last rounds of the cryptographic implementations. We refer to deliverable D2.01 and D2.08 for more technical details and motivation on this matter.

Any of these three functions that is present at the client side must be well protected. Reverse engineering these components may make their purpose of existence, which is to improve the security of the white-box implementation, obsolete.

3.5.3 Server-side components

3.5.3.1 WBS

The encoding/decoding/protectKey functions that have been presented as client-side components can also be used as server-side components in the ASPIRE protection server. These will be implemented into a White-Box Library Server-side (WBLS), which is part of the White-Box Server (WBS). Additionally, the WBS also comprises the logic to query the ASPIRE backend DB.

3.5.3.2 ASPIRE Database

White-box implementations are generated based on a set of parameters amongst which a cryptographic seed. Not only can the syntactical representation modify for different seeds, but also the functional behaviour, such as the semantic definition of the encodings. This offers a natural way of introducing diversity.

To manage the fact that different instances of white-box implementations may co-exist in the field, the ASPIRE database will be used.

3.5.4 Offline white-box crypto workflow

We present the offline workflow, where an encryption function is implemented. In case of a decryption function, all logic stays the same, with plaintext and ciphertext swapped in the text, and “decryption” instead of “encryption”.

The functions EncodeInput/DecodeOutput/ProtectKey are optional components. In the protected application, these components should never be present as individual components, but rather integrated into other components. For that reason, there has not been defined an API for these functions. Therefore, in Figure 13, we depicted these components in gray, indicating that they are only implicitly available.

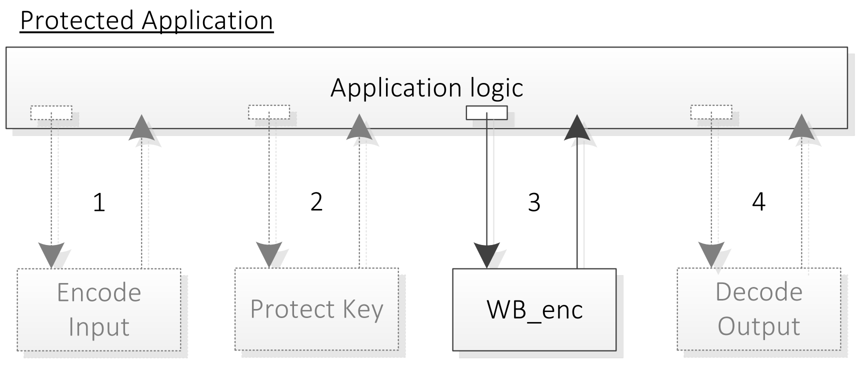

Figure 13 – Client-side white-box workflow diagram

Seq#	Operation description
1	The application logic operates with the EncodeInput function (optional).

Details: This function transforms the plaintext into an encoded plaintext, compatible with the white-box encryption function that is integrated into the application.

2	The application logic operates with the ProtectKey function (optional).

Details: This function transforms the cryptographic key into a protected key, compatible with the white-box encryption function that is integrated into the application.

Such a function is not available when it concerns the case of a fixed-key white-box implementation, or when the ProtectKey function is used at server-side instead.

3	The application logic calls the encryption function.

Details: The encryption function is called via the white-box API that has been defined in Deliverable D2.04, and corresponds to the `wbgcClientEncrypt` function defined in Section 3.5. As arguments, pointers to the plaintext/ciphertext buffers are provided; in case of a dynamic-key white-box implementation, a pointer to the buffer containing the protected key is additionally provided.

4	The application logic calls the DecodeOutput function (optional).

Details: This function transforms the encoded ciphertext into the original ciphertext, compatible with the white-box encryption function that is integrated into the application.

3.5.5 Online white-box crypto workflow

We describe the case of a dynamic-key white-box implementation that is integrated into the protected application, and where the application server aims to deliver a protected key. For the delivery of the protected key, the application server can use the same transport as it used for the original (unprotected) key – assuming that the transport can handle the protected key (which might be larger than the original key).

Figure 14 depicts the workflow of this use-case.

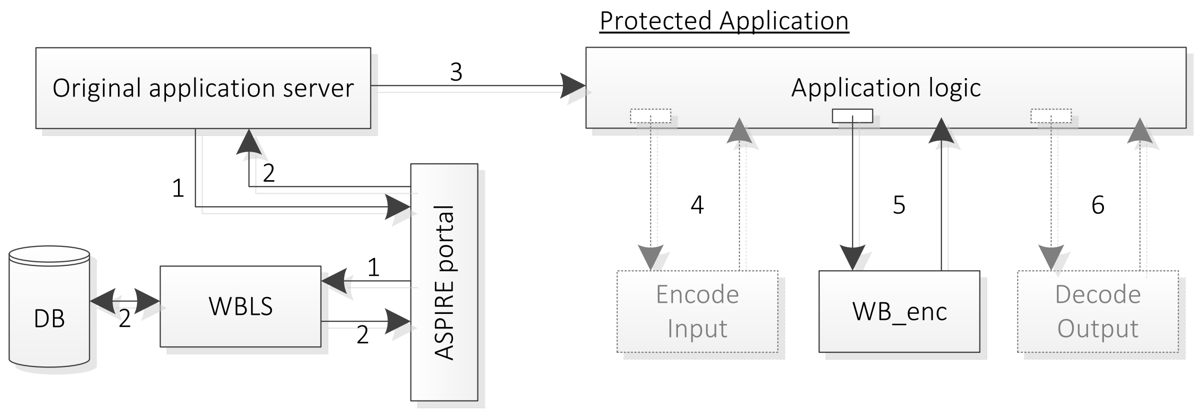

Figure 14 – Online dynamic-key workflow diagram

Seq#	Operation description
1	The original application server connects to the ASPIRE portal, presenting the key that needs to be obfuscated together with the application identifier.

Details: The original application server has a cryptographic key that it intends to send to the application. With the client-side cryptographic processing now replaced by a white-box implementation, a protected key needs to be sent to the client application rather than a 'plain' key.

The application server will therefore present the plain key to the ASPIRE portal, along with the client application identifier, requesting the WBS to protect the key as such that it is compatible with the white-box implementation that is integrated into the client application.

2	WBS returns the protected key.

Details: The WBS receives the key and the application ID. It will query the ASPIRE database for the information it needs to be able to compute the protected key. After this computation, it returns the protected key.

Note that the protected key may be larger (in size) than the plain key.

3	The application server sends the protected key to the protected application.

Details: The application server sends the protected key to the client application. It can use the original transport for doing so, under the assumption that it can deal with the (potential) larger size of the protected key.

4	The application logic operates with the EncodeInput function (optional).

Details: The application aims to encrypt a plaintext with using the protected key and the white-box implementation. In the case where this white-box implementation is protected with some input encodings, the application logic will need to use the EncodeInput function to encode the plaintext.

Note that this operation should be implicit: the encoding function should be hidden into another operation such that its definition cannot be reverse engineered.

5	The application logic calls the encryption function.

Details: The application logic calls the white-box encryption function, in compliance with the white-box API. It will present as argument pointers to the (encoded) plaintext, the protected key and a buffer where the result needs to be stored.

6	The application logic calls the DecodeOutput function (optional).

Details: In the result buffer, the ciphertext will be stored. In the case where the white-box implementation comprises an output encoding, the buffer will comprise an encoded ciphertext. In this case, the application logic will need to use the DecodeOutput function to decode the ciphertext.

Note that this operation should be implicit: the decoding function should be hidden into another operation such that its definition cannot be reverse engineered.

3.5.6 Impact

White-box cryptography induces significant impact on size and performance. A white-box implementation of a cryptographic function is considerably larger and slower than the original (non-white-box) cryptographic implementation.

Protected keys that are used to instantiate dynamic white-box implementations are in general also larger than the original keys.

3.5.7 Renewable White-Box Cryptography

In this section, we describe the workflow of how the time-limited white-box implementations can be renewed at client application run-time. This leverages on the Code Mobility techniques that have been described in Section 3.4.

The approach is as follows: before the usage of the white-box routine, the downloader of the Code Mobility technique will be triggered (via a function call that has been inserted into the application) and will request to the white-box server-side backend service an update of the white-box routine. This update will comprise the data that corresponds to new tables for the time-limited white-box implementation. This approach is depicted Figure 15, and each of the steps is detailed in the step-by-step overview below.

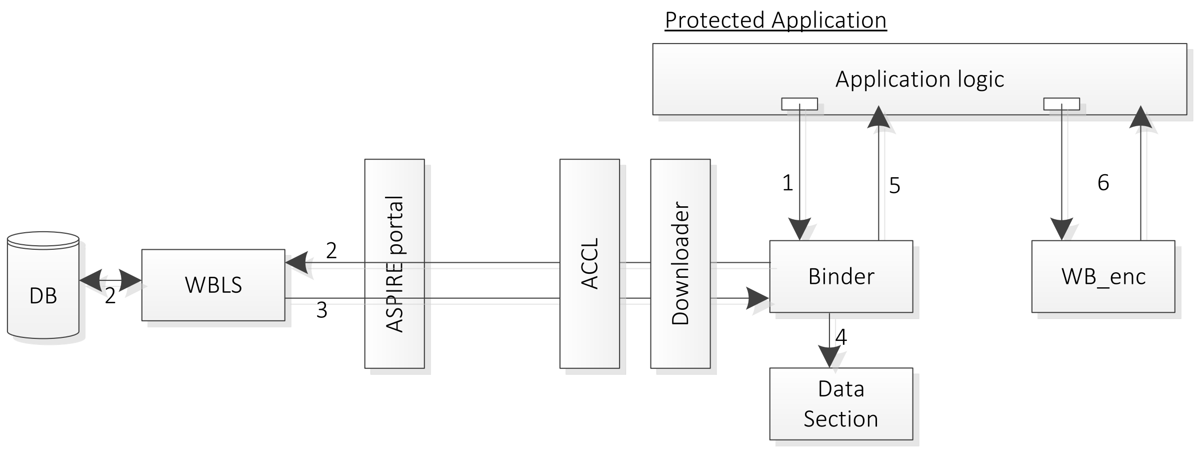

Figure 15 - Renewable WBC workflow

Seq#	Operation description
1	The application invokes the Binder

Details: The original application invokes the Binder to instruct it to fetch the new information from the server that needs to be used to update the white-box implementation.

2	The Binder invokes the Downloader which will query the WBLS server backend

Details: The Binder invokes the Downloader passing it an identifier to the mobile block that should be downloaded. The Downloader establishes a connection to the ASCL, using the identifier related to the renewable white-box.

3	The WBLS generates the new information

Details: The WBLS receives the request for generating new white-box tables, and will send these back to the Downloader who will pass this information on to the Binder.

4	The Binder allocates the retrieved table definitions.

Details: The Binder allocates memory on the heap at a randomized location.

5	The Binder returns control to the original application logic

6	The White-Box Implementation itself is invoked

Details: The white-box implementation itself is now invoked. This is the same workflow as the offline white-box workflow. The only difference is that the white-box implementation itself will now use the updated tables rather than the original ones.

3.6 Multi-threaded cryptography

Section Author:

Jerome d'Annoville (Gemalto)

3.6.1 Introduction

The multi-threaded cryptography technique is an obfuscation technique that is included in Task 2.4 of the DoW as the domain-specific obfuscation technique. Originally, it was envisioned that this would be an offline binary obfuscation technique, but during the design phase, it has been decided that this technique can be more easily deployed at source level. Additionally, up to some extent this can now also be considered an online technique because it requires a server to generate keys that is used by the crypto processing. The multi-threaded cryptography protection is delivered in D2.07 (M24) and reported in deliverable D2.08 (M24).

The use-case addressed by this protection is related to symmetric cryptography where an application needs to share a master key with an Application Server in order to send a cryptogram to this server.

Key derivation is a cryptography function that enables to generate a derived key from a master key. The master key is a common secret that is shared by both sides. A non-secret data can be used to get the derived key from the master key. The advantage is that there is no need to provision the client application with a specific secret. The master key is a common secret for all deployed applications and is kept within the application. This master key is only used to produce the derived key and this derived key is to cipher the data to protect. The master key is stored somewhere in the application and can be found by an attacker that can either use it or better export it on an attacker server.

The aim of this technique is threefold:

- Provide a way to keep the master key in a secure place. It is not exposed in the client application in clear.
- Hide the derivation key generation processing.
- Prevent the attacker to reuse the derivation key.

Instead of being exposed in clear within the application the master key in the client application is ciphered by a crypto server public key. The master key is passed to a crypto server, still encrypted. The key derivation is performed on this crypto server side which retrieves the master key thanks to its crypto server private key. Then the crypto server sends several derivation keys to the client application where only one is the valid key and remaining ones are dummy keys. A seed is returned together with the keys that enables to retrieve the valid key.

The encryption crypto processing is done without exposing the valid secret key in the client application because the plain text is ciphered with several keys in parallel and all generated cryptograms are sent to the Application Server. Each ciphering processing is done in its own thread and at each round cryptograms and round keys are exchanged between threads. Neither the client application nor an attacker can locate which key is the valid derived key and as a consequence what will be the valid ciphertext.

The multithreading crypto technique does not protect against a replacement of the plain text by other data prepared by an attacker.

The implementation is done for the AES encryption for the purpose of the project.

This protection technique can be used to encrypt data to be sent to a recipient. It has no value when a data needs to be decrypted like in the DRM use-case because the attacker can track the use of the plaintext in the logic of the application. It can be interesting if the decryption process can be isolated with another protection technique.

3.6.2 System requirements and assumptions

The crypto server must be able to perform crypto functions such as RSA deciphering function, key derivation and pseudo random number generation.

The technique needs to use the ASPIRE protocol described in Section 2, because it needs a strong authenticated channel between the server and the application. In the remainder of this section, it is assume that application authentication is taken care of.

3.6.3 Client-side components

The client-side components, as depicted on the right of Figure 16 are

- A crypto library that performs all crypto operations and thread obfuscation.
- A communication component to perform communication between the crypto-server and the crypto library.

The Crypto library performs the cryptographic operations and thread obfuscation. It performs the computation of the various ciphertexts, and it performs the thread obfuscation as well during this computation process.

As defined in the reference architecture the communication component is the interface between the crypto server and the crypto library. The communication protocol is implemented in this component.

3.6.4 Server-side components

The server side components are shown on Figure 16 below. The following features are provided:

- Authentication token validation.
- Derivation key generation of a master key and a seed
- Provisioning of multiple random dummy keys to perform thread obfuscation in the client application.

No direct communication is required between the Application server and the crypto server. No application data needs to be maintained on the crypto server.

3.6.5 Multithreaded crypto workflow diagram

Figure 16 presents the multi-threaded crypto processing workflow diagram, followed by an overview of each of the referenced steps.

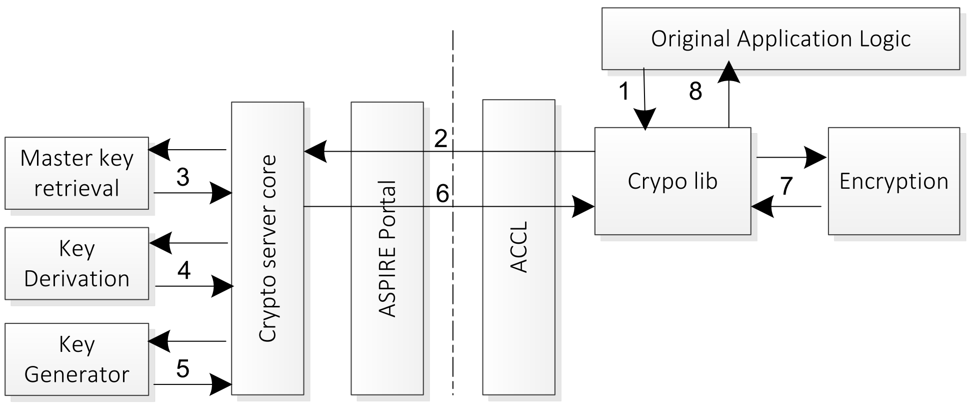

Figure 16 – Multi-threaded Crypto Encryption Processing

Seq#	Operation description

1	AES crypto library invocation.

Details: The application calls the crypto library with the encrypted master key, a fingerprint, and the plaintext as argument.

2	Crypto server invocation

Details: The crypto lib sends the master key to crypto server. This is a synchronous call and the crypto library waits for the answer.

3	Crypto server: Master key retrieval

Details: The Master key is retrieved: the encrypted Master key received by the server can be decrypted thanks to the Crypto server private key.

4	Crypto server: key derivation step.

Details: A derived key is generated with a PBKDF2 function. It is a standardized Password-Based Key Derivation Function. Input arguments are the Master key and the fingerprint data.

5	Crypto server: dummy keys generations.

Details: The random Number Generator will produce multiple random dummy keys in order to be able to obfuscate the crypto process in the application. Step 4 and 5 can be called in parallel. These processes are independent.

6	Crypto server: The answer to the request is sent back to the client application.

Details: The answer is prepared. A random number generates a seed that gives the position of the valid key thanks to a pseudorandom number generator. Then all keys and the seed are returned to the client.

7	Encryption step.

Details: The crypto lib produces a set of cipher texts. During this process the same seed is used to indicate how to permute data at each rounds. The seed will enable to retrieve the position of the valid result.

8	Results are returned to the application.

Details: The application is then able to send the set of cipher text and the seed to recipient according to the application logic. The recipient has the Master key and is able to derive the same key as the crypto server thanks to the fingerprint. It retrieves the valid ciphered text to decrypt thanks to the seed.

Section 4 Anti-tampering

4.1 Overall Anti-Tampering Architecture

Section Author:

Bjorn De Sutter (UGent)

To check the integrity of a program's execution, many techniques have been proposed in literature. Some prevent attackers from tampering, others try to detect ongoing tampering and respond appropriately. In this section, we focus on such detection and response techniques, all of which monitor certain static program features or dynamic program behaviour. The monitoring happens in so-called *attestation routines*, that perform computations on the observed features or behaviour, and that produce an *attestation report* as a result of those computations. A *verifier routine* then checks the validity of the attestation report, and returns an *attestation verdict*. This is essentially a Boolean value marking whether the verification succeeded or failed, with failure indicating that (likely) tampering was detected.

If the verification was successful, the application is allowed to continue executing. In case the verification fails, a response is invoked. The possible responses span a wide range, and are to some extent application-specific. Possible responses include halting the program, corrupting the program state, graceful degradation, etc. For intrinsically distributed applications, disconnecting from the server or temporarily limiting access is another viable solution. Furthermore, responses can depend on multiple verdicts, and on the time frame in which successes and failures occur.

To obtain strong protection through anti-tampering techniques based on tamper detection, it is important to delay the response following a tampering detection, such that the attacker cannot easily identify (and circumvent) the cause of the response he will obviously observe. It is hence necessary to implement a delay mechanism as part of anti-tampering techniques.

While in some cases there might be good reasons to deploy specific combinations of (i) attestation and verifier routines with specific forms of (ii) delay mechanisms and (iii) tamper response, these three aspects are mostly orthogonal. Moreover, the choice to implement the verifier locally in the client application, or remotely on a server is also mostly orthogonal to the other aspects.

For that reason, we propose an overall anti-tampering architecture that composes complete solutions from four types of components:

1. **Attestator components:** The purpose of this type of component is to collect either static or dynamic characteristics of the application and to prepare attestation reports for verifiers. Examples of Attestators are code guards or CFG tagging.
2. **Verifier components**: Based on attestation reports these components control that the application conforms to its characteristics and does not deviate from expected behaviour. These components are closely linked to the Attestator components in terms of functionality, but may be executed locally, partially remotely, or completely remotely.
3. **Delay components**: These components are the means by which verdicts made by verifiers are stored for later activation of response components. Delay components consist of data structures and of their update and query APIs that can stealthily encoding that tampering has been detected and that can be queried to extract that information.
4. **Reaction components**: These components implement the actual responses in cases of tampering detection. These components are the reaction part that trigger adequate

actions to change the behaviour of the application. These actions may vary according the policy configured for the application.

Conceptually, these components cooperate as depicted in the workflow diagram depicted in Figure 17, and further detailed in the steps following the figure.

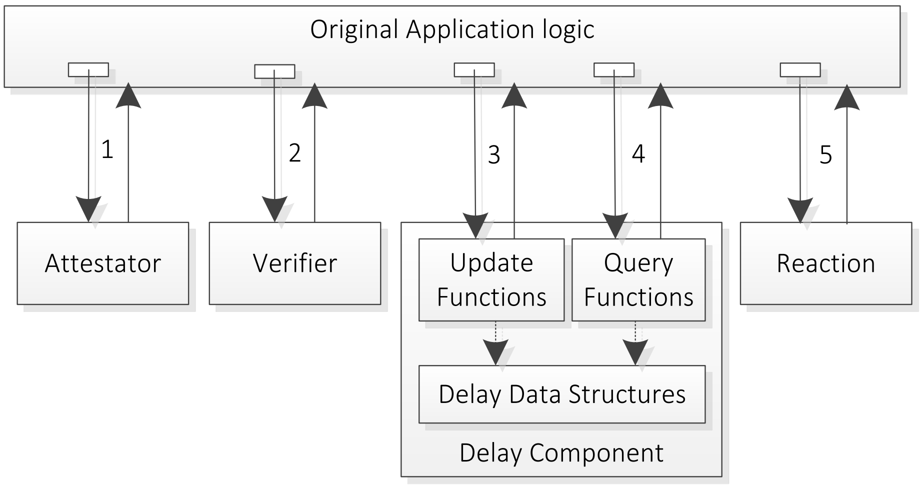

Figure 17 – Anti-tamper components

Seq#	Operation description
1	The attestator routine is invoked.

Details: An attestator routine is invoked that returns the attestation report in the form of some data. It should be noted that the attestator can be a single monolithic routine, but it can also consist of a collection of smaller routines that are invoked one after the other to iteratively compute an attestation report. In the latter case, the routines can actually also be in-lined into the program and hence be indistinguishable from the application code.

2	A verifier is invoked.

Details: In step 2, which will typically be executed immediately after step 1, the attestation report is verified. This can occur locally, but it the verification can also be offloaded (completely or partially) onto a secure server. By offloading the verification to a server beyond the reach of an attacker, the attacker cannot learn how to fabricate correct responses by studying the verification routine. In practice, the attestator and verifier can also be combined into one routine. Alternatively, their invocation can be pulled apart to some degree, in order to hide the dependency between them. However, there always has to remain a guarantee that whenever the attestator routine is invoked, so is the verifier routine, and vice versa.

3	Update the tamper detection status.

Details: In step 3, which typically will follow immediately after step 2, the result of verification (the verdict) is used to encode the tamper detection status of the application. Based on the verdict, an update function is invoked that alters the delay data structures to encode that some form of tampering was or was not detected.

It is important to note here that the update functions can also be invoked from random places in the original program, as long as those random invocations do not alter the information regarding detected tampering encoded in the data structures. This is important because such random updates will give the delay data structures the appearance of being integral data structures of the original application, such that their true functionality is not easily identified or analysed by attackers.

4	Query of delay structures.

Details: Later in the execution of the application, in step 4, at times and places not necessarily linked to those where step 3 was executed, the delay data structures are queried through access functions. Based on the retrieved information, indicating which forms of tampering have been detected or not, normal program execution is continued, or step 5 is executed.

5	Invocation of response mechanism.

Details: A tamper response is invoked. Clearly, when step 4 and 5 would be executed immediately after step 3, which our architecture does not exclude, the response is immediate rather than delayed. So our architecture covers both versions.

Thus, to summarize, anti-tampering comes in 2 phases: tamper detection, and tamper response. The detection comprises of Attestator, Verifier and update components (or a subset thereof); response comprises query functions and some reaction logic. In case of direct response (in contrast to delayed response), the verifier can directly invoke the reaction logic.

Note that in the remote mechanisms additional logic can be available to coordinate the execution of attestations and verifications.

In subsequent sections, we describe some concrete forms of the components that we will develop in the ASPIRE project. Furthermore, we discuss the range of functionality and interfaces that can be covered by the delay and response components to provide a wide range of delay and response tactics. Finally, we describe alternatives for implementing the verification step locally or remotely, and alternatives to control the invocation of the different mechanisms. In case a remote server handles that control, this composition is known as remote attestation.

Finally, compared to the previous version of this document, the synchronous mode of remote attestation is no longer presented. Indeed, after the availability of the ASCL-WS, we have preferred the asynchronous mode, as it is more secure and closer to the theoretical case, as it does not allow unsolicited attestations.

4.1.1 Tamper detection

Section Author:

Cataldo Basile (POLITO), Bart Coppens (UGent), Jerome D'Annoville (GTO), Alessio Viticchié (POLITO)

The ASPIRE anti-tampering mechanisms can be divided in two categories:

1. **Offline code guards.** Inserted invocations of hashing functions that hash part of the program memory are (almost) immediately followed by the verifying routine that locally compares the computed hash value with a value pre-computed by the ASPIRE tool chain.
2. **Remote techniques**. We implemented completely remote anti-tampering mechanisms, i.e., remote attestation. In this phase the ASPIRE protection server will decide which code regions or application properties to attest and pass that information, along with, e.g., nonces to ensure protection from reply attacks.

This section presents how the basic blocks described above can be used to implement the two ASPIRE code guard solutions.

The architecture and the workflow of these three solutions will be presented in this section individually.

4.1.1.1 Completely offline combinations: offline code guards

In case of an offline code guard, the following client-side components are required:

- A hashing function for the Attestation component, as described in Section 4.2.2.1.
- A hash verification function for the Verifier component, as described in Section 4.2.2.2.
- A (possibly delayed) tamper response, such as described in Section 4.7.

Figure 18 comprises the use-case diagram of completely offline code guards with immediate response, followed by a table comprising the details of each step. For delayed responses, additional steps like steps 3 and 4 in Figure 17 need to be added.

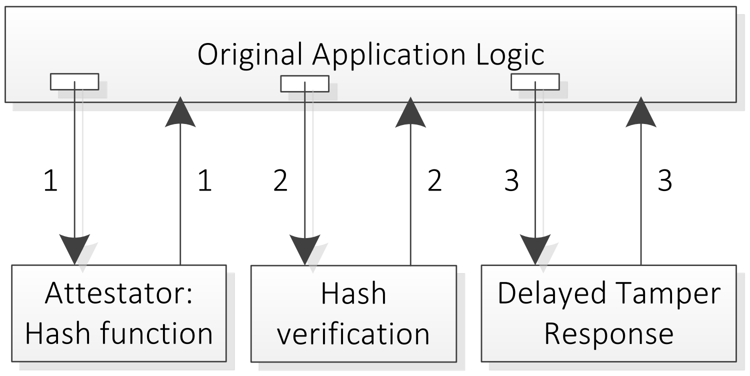

Figure 18 – Code guards workflow diagram

Seq#	Operation description
1	Attestator: Hash function computes the hash of a code region.

Details: One of the diversified hash functions spread throughout the program is executed. The hash function reads from a region of memory and computes a hash of this region. A hash bookkeeping function stores the result of this hash (or a value that depends on it) in the program's data section.

Data passing: The resulting hash value is stored to be verified in Step 2.

2	The computed hash is verified.

Details: The diversified hash verification code corresponding to the diversified hash function from Step 1 is executed.

The verification needs to be called after the corresponding hash has been computed.

Data passing: The verification returns a Boolean value signifying the verification status.

3	The tamper response is activated.

Details: As described before, the delayed tamper response is invoked depending on the result of the Verification step. The update functionality of the tamper response component is invoked, whose reaction logic can then decide when and how to respond to a verification failure.

4.1.1.2 Remote techniques

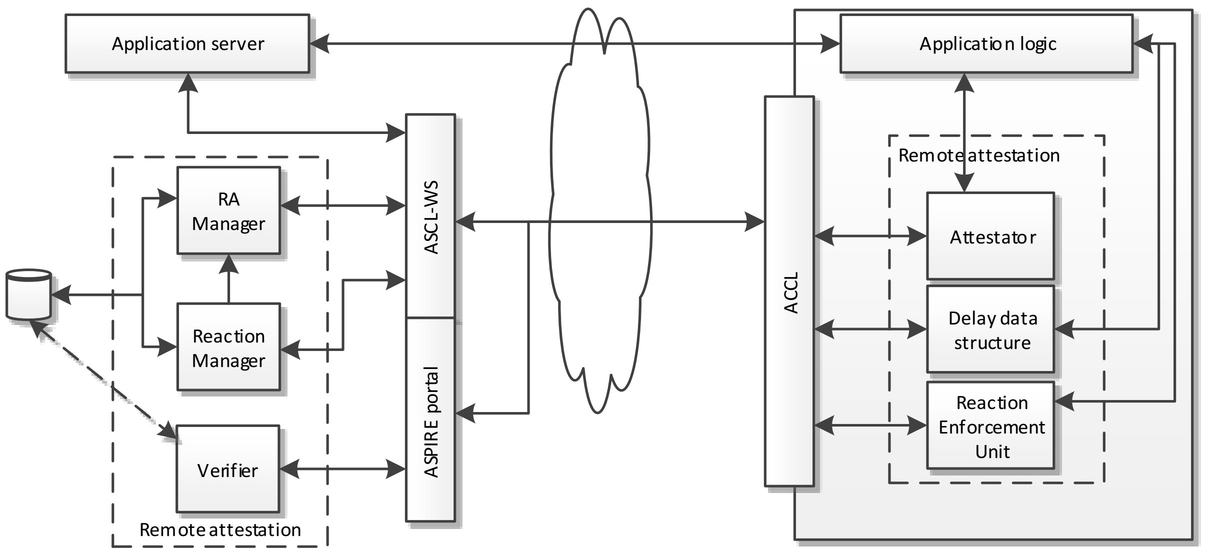

Figure 19 – Remote Attestation Architecture (from D3.04)

In case of remote attestation, the following components as depicted in Figure 19 are needed:

- (Client-side) an Attestator, as in Section 4.1.
- (Client-side) a Reaction Enforcement Unit, like in Section 4.1.2.
- (Client-side) a Delayed tamper response component, like in Section 4.1.2.
- (Client-side) a communication logic, such as the ACCL as described in Section 2.
- (Server-side) the ASCL, as described in Section 2.
- (Server-side) a Reaction Manager, to request attestations to the client Attestator like in Section 4.1.2.1;
-
- (Server-side) a Verifier, able to check values provided by the attestator, like in Section 4.1.1.2.2.1;
- (Server-side) a Reaction Manager connected to a state DB (optionally available to the Verifier).

The server side part of remote techniques must manage several clients. Therefore server-side components are more complex.

4.1.1.2.1 Remote attestation workflow diagram

This workflow, as depicted in Figure 20, describes the operations to perform a remote attestation.

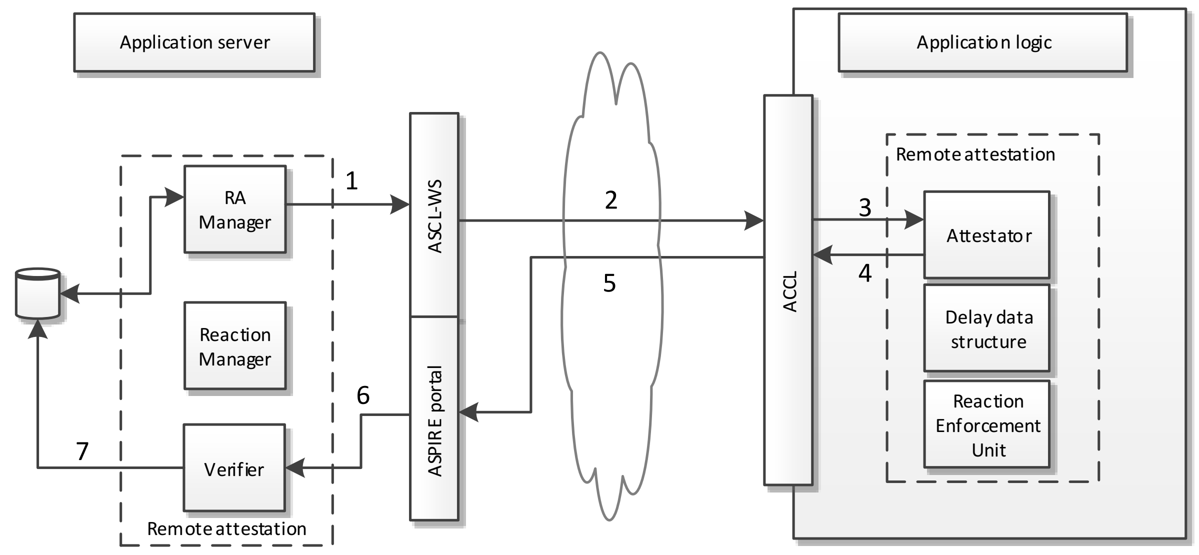

Figure 20 – Remote attestation workflow.

Seq#	Operation description
1,2,3	The RA Manager prepares and sends the attestation request to the Attestator

Details: The RA Manager decides that the client needs to be attested and sends an attestation request message to the ASPIRE portal. This decision is triggered by a timeout defined, at first, in the ASPIRE database which can be altered at run-time according to the needs.

The content of the attestation request depends on the technique the Attestator implements. It will certainly contain a nonce, which has the objective to provide anti-replay protection.

The attestation request is then passed to the ASCL-WS (step 1), which forwards the message to the ACCL (step 2) exploiting the WebSocket channel, which forwards the attestation request to the Attestator (step 3).

4,5,6	The Attestator performs the attestation routine and sends back the attestation report to the Verifier.

Details: The Attestator prepares the attestation report according to the directives and using nonce and other data from the request. The Attestator then passes the attestation report to the ACCL (step 4), which forwards the message to the ASPIRE portal (step 5) over the standard HTTP channel, which forwards the message to the Verifier (step 6).

7	The Verifier reports the verdict in the database.

Details: The Verifier checks the report, decrees about the received attestation response and writes the result in the ASPIRE database.

The result of the attestation is then available for any server side component which has to infer about the integrity of the client application (e.g. the Reaction Manager).

4.1.1.2.2 Server-side component: RA Manager

The RA Manager is the server side remote attestation component that is in charge for sending the attestation requests to clients. It is supposed that the Remote Attestation Manager is a bit more complex than in case of simple remote code guards, as it must be able to determine when the right moment has arrived to force an application instance to execute the remote attestation.

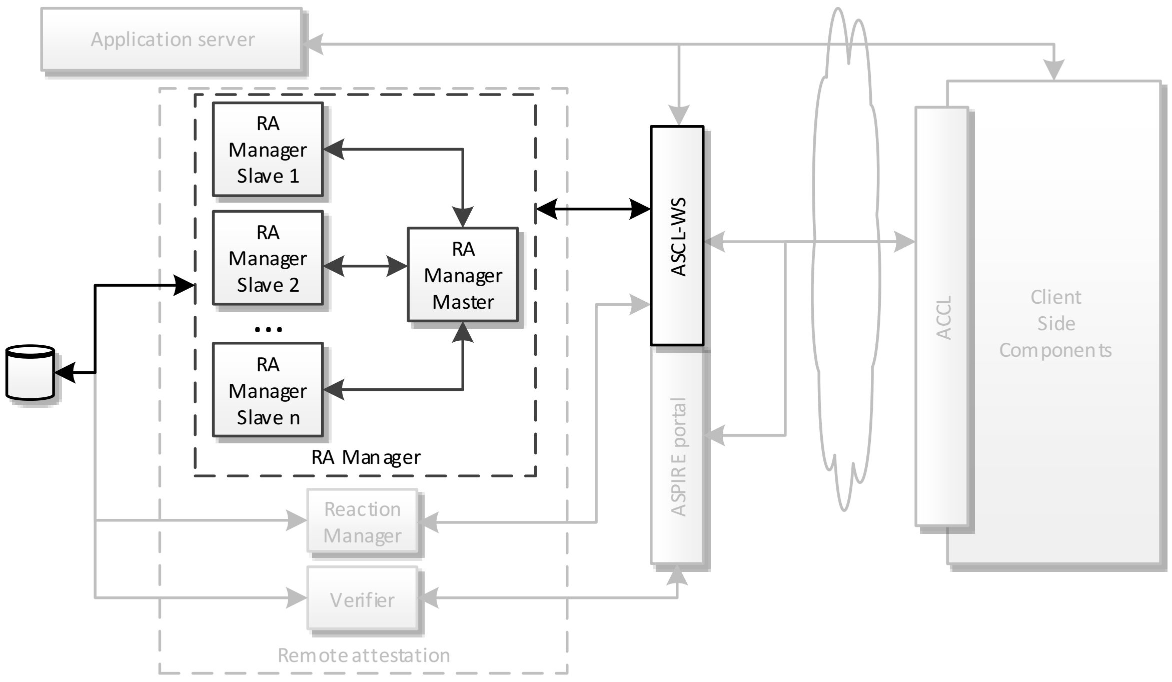

Figure 21 – Architecture of the RA Manager (from D3.04).

Therefore, the RA Manager uses one RA Manager Master, which is able to manage connection and disconnection of clients, and several RA Manager Slaves, which actually generate and send attestation requests to the clients they are assigned to by the RA Manager Master. These components are presented in details in D3.04.
When the client application is reachable by the server (known because the ASCL-WS records the connect operations of the clients), the RA Manager (independently from the client) decides that a client must prove its authenticity, based on a timeout and other information from the Application server, like the fact that the application is connected, and the manager contacts the client. In both cases, the time between two consecutive requests for attestation must be not predictable by the client, e.g., it can be randomly chosen within a range around a fixed average value.

Additionally, to optimize performance, the RA Manager may have the intelligence to force the use of a complex and time consuming attestation when the client is at risk, while for clients that have always shown a good behaviour, it can ask a fast attestation method.

4.1.1.2.2.1 Server-side component: Verifier

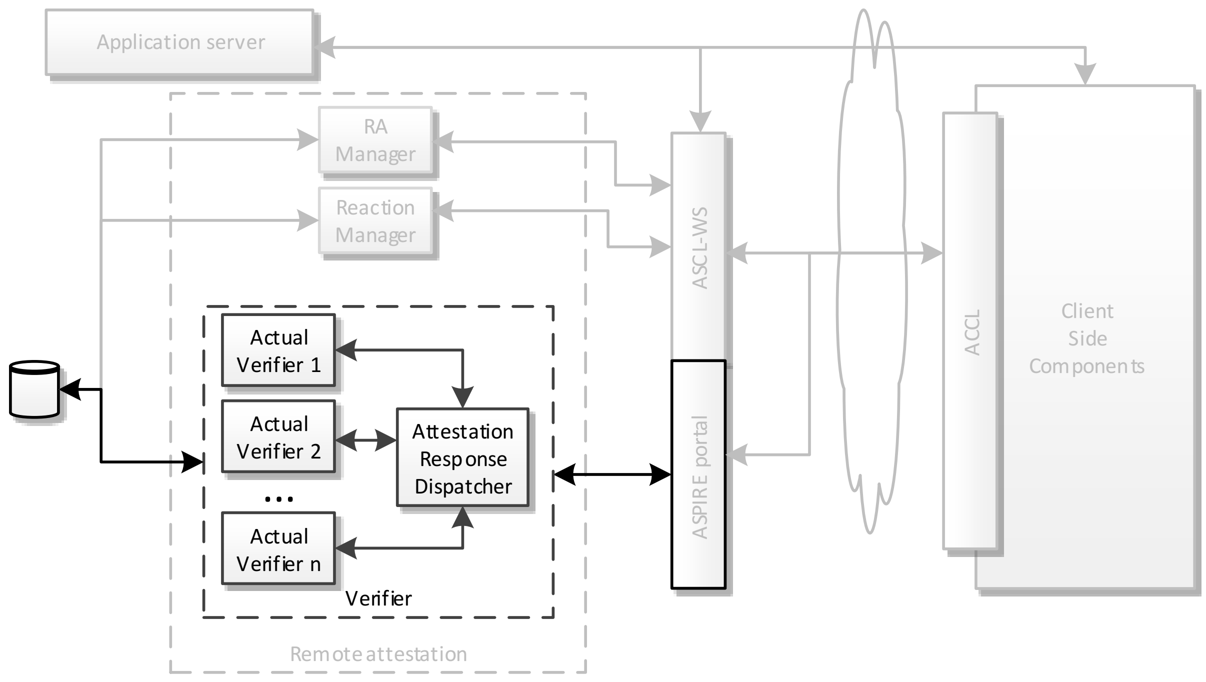

Figure 22 – Architecture of the Verifier (from D3.04).

Figure 3 shows the architecture of the Verifier, which is composed of an Attestation Response Dispatcher and several Actual Verifiers. Actual Verifiers are needed because we support more than one client, and each client may be protected with different RA techniques. More precisely, one client may be protected with zero or more attestation techniques, and several clients may use the same attestation technique. Therefore, the Attestation Response Dispatcher forwards attestation responses to the proper Actual Verifier.

4.1.2 Delay components and tamper response

The tamper response is performed by a Reaction Logic, which is in charge of enforcing the decision of the verifier. In the end, the Reaction Logic must render tampered applications unusable, while still guaranteeing the correct behaviour of original un-tampered applications. As explained before, the Reaction Logic communicates with the Verifier, which is in charge of determining if an application has been tampered with or not, by means of a *covert channel* in the form of the delay data structures.

The reaction Logic must perform two separate tasks, which are associated to two distinct components:

- The Reaction Manager is the component that selects the correct reaction mechanisms against the tampered applications, i.e., the punishment for tampered applications. This decision can be made by correlating different data, e.g., the severity of the tampering, the frequency of verification failures as detected by the verifier, history data about the customer which bought the application, etc. More details on this component are presented in Section 4.1.2.1
- The Reaction Enforcement Unit comprises the actual code deployed to execute the tamper response prescribed by the Reaction Manager.

4.1.2.1 The Reaction Manager

The Reaction Manager (RM) is the central component of the Reaction Logic. Its role is to decide if a reaction action must be taken for an application running in a device. The RM can run on the server side or it could also be deployed on the client side.

For offline protections that needs all the reaction logic to be located on the client side the recommendation is to implement it directly within the protection. A RM module packaged separately and shared by several protections could be spot by the attacker that might block it. Indeed, it may be difficult to transparently maintain a large history of negative and positive events. The RM could be kept relatively simple, e.g., the RM may enforce one reaction or a small set of different reactions, ordered by severity and triggered by a Tampering Severity Code.

The RM on the server side is described hereafter.

The architecture would enable sophisticated mechanisms such as an inference engine running on the ASPIRE DB where facts are the Attestation reports and the status of the connection with the Application server. For the purpose of the project the RM will run simple rules made of queries on the history of the Verification reports and the verdicts wrote by the Verifier in the database. The RM do not deduce new knowledge that would enrich the ASPIRE DB like in an expert system.

From the rules decision and based on policies set for the various applications the RM Engine will create adequate notifications. The RM may send reaction notifications to the RA Manager, to the Application server and to the Reaction Waiting Unit located in the application. The Reaction Waiting Unit will activate the Reaction Enforcement Unit through the mean of the Delayed Data Structures

The Delayed Data Structures are a covert channel between the RM and the Reaction Enforcement Unit. So in that case, Delay Data Structures will encode one of the different types of reactions the Reaction Enforcement Unit is able to enforce (or no reaction). Therefore in this case, the RM will use the Delay Data Structures to communicate the decision to enforce.

4.1.2.2 Reaction Enforcement Unit

As anticipated before, the Reaction Enforcement Unit is the component that actually contains the code that enforces the reaction prescribed by the Reaction Manager.

The reactions (and the types of Reaction Enforcement Units) can be classified in two types:

Immediate response

Immediate responses render applications unusable right after the Reaction Enforcement Unit notices that the Reaction Manager has decided to punish the application. Possible Reaction mechanisms that implement an immediate reaction include:

- Halting: The application code is modified in such a way that it is not executable;
- Disconnection: Block the Application Logic of a tampered application to interact with the Application Server. It may be also limited in time (e.g., for 24h or one week). This only applies to intrinsically distributed applications – those applications that require an interaction with the server for their core functionality (e.g., an multi-player online game).

Delayed response

Delayed responses render the application unusable at some time after the Reaction Enforcement Unit notice that the Reaction Manager has decided to punish the application. That is, the Reaction Enforcement Unit starts a process of application degradation that can last minute, but possible also for hours or days. Possible Reaction mechanisms that implement a delayed reaction include:

- Performance degradation, which consists of corrupting selected parts of the program's internal state as such that the program does not fail (e.g., by entering into an unstable state) but shows performance degradation [Tan06]. Practically, it consists in inserting reaction enforcement code that changes something in the program when the detection routines check a fail a save failure information in the delayed structure. The reaction must not halt the program immediately; it is the cumulative effect of several failures that must make the program unusable. For instance, some solutions proposed to place conditions in the loops that take more time to be satisfied. In other cases, graphical routines have reaction enforcement units that increase distortion of depicted images/draws. These techniques have been applied to games, where even a small delay or imprecision in movement's reaction makes the game unusable.

- Time bombs, as described in Section 4.8.

It is worth noting that, in case of remote guards or remote attestation code guards on an intrinsically distributed application, there is no need to implement at client-side a Reaction Enforcement Unit, as the Application server can simply stop serving application disconnect applications that have been identified as tampered.

4.1.2.2.1 Reaction Enforcement unit workflow diagram

This workflow, as depicted in Figure 23, describes how a remote RM is able to inform a local Reaction Enforcement Unit about its decisions.

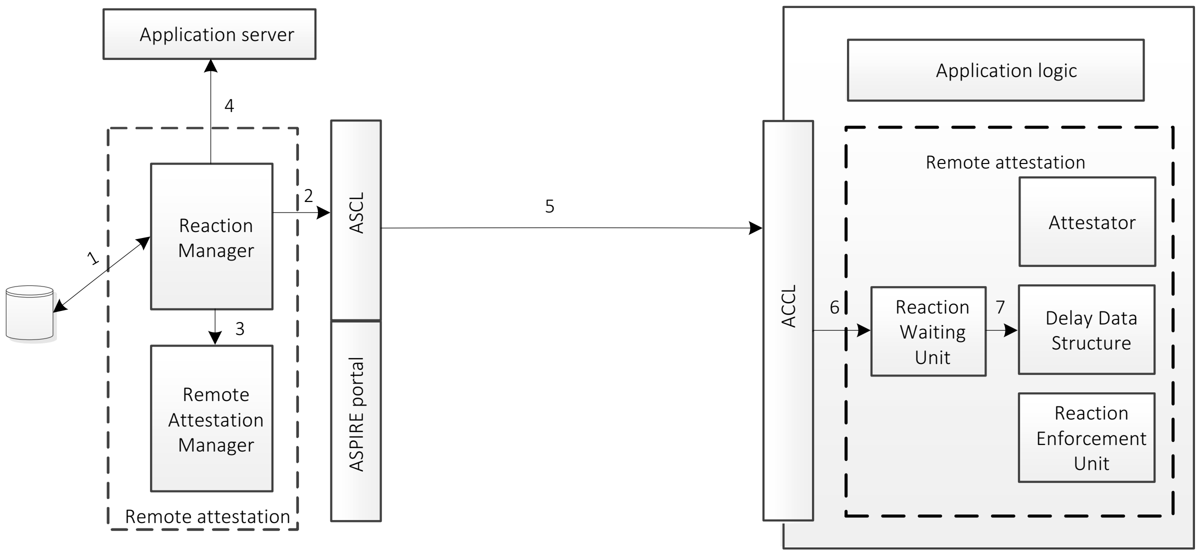

Figure 23 – Reaction enforcement workflow diagram

Seq#	Operation description
1	The RM queries the ASPIRE database.

Details: The RM queries the database for application currently connected. Based on the current and previous verdicts the RM triggers a reaction if required. Based on the policy of the application the RM may or may not notify the Reaction Waiting Unit on the client side (Step 2), the Remote Attestation Manager (Step3) and the Application server (Step 4)

2	The RM notifies the Reaction client side

Details: In case the RM triggers a reaction action then a notification is sent to the Reaction Waiting Unit on the client side through the ASCL.

3	The RM notifies the Remote Attestation Manager

Details: This step is optional. In case the RM triggers a reaction action then it notifies the Remote Attestation Manager to enable possible adjustments in the Attestation request management. This notification is sent based on the policy of the application.

4	The RM notifies the Application Server.

Details: This step is optional. In case the RM triggers a reaction action then it notifies the Application Server to enable possible adjustments in its service policy for the device where a reaction action has been triggered. This notification is sent based on the policy of the application.

5, 6	The notification is forwarded to the Reaction client side

Details: The notification is forwarded to the Reaction Waiting Unit on the client side by the communication layers on the server (ASCL) and the client (ACCL).

7	Delay data structures are updated.

Details: The Reaction Waiting Unit updates the Delay data structures to trigger asynchronously the Reaction Enforcement Unit.

4.1.2.3 Positive reactions

At a first glance, one may expect that the Reaction Logic will always render the application unusable after one or more verification failures (or other sophisticated algorithms as explained before). We name this behaviour or the Reaction Logic reaction to negative events, or simply *negative reaction*.

However, we also envision an alternative, dual approach that we name reaction to positive events, or simply *positive reaction*. Intuitively, positive reaction works differently: The application is initially assumed as corrupted and a delayed reaction is started, but every time the client proves to the Verifier that it is a legitimate client, the delayed reaction is re-initialized. For example, every time a positive verdict is reached, the countdown of time bombs is restarted or the effects of the degradation are annihilated. If the verification fails only occasionally, the degradation or the countdown speed can be left at the same pace. If a certain amount of successive verifications failures has been detected, the Reaction Manager may decide to accelerate the pace or reset the counter, thus causing an immediate negative reaction. It is evident that positive reaction only works with Reaction Enforcement Units that use techniques that implement a delayed reaction such as time bombs, graceful degradation.

Another possible exploitation of positive reaction is to oblige applications that are functionality-wise able to survive without interacting with the application server for a long time, to interact with the security server more frequently. Applications are forced to provide an attestation within a given time, since they need to receive inputs from the security server to stop the code degradation or reset the time bombs. Of course, forcing applications to interact with the security server is needed in case of remote attestation or remote code guards.

Another possible application of positive reaction is arises when applications are not intrinsically distributed, for instance, if the Application Logic does not require to interact with a server to continue its operation as discussed above. Another case may appear in case of applications protected with renewability, if an attacker is able to collect all the blocks that form the target application (even if they are not available on the client at the same time) it would be theoretically able to disconnect the application from the server). Positive reaction is also needed for applications that are designed for occasionally connected scenarios. In practice, in all these cases the disconnection from the services is not feasible as punishment a Reaction Enforcement Unit that uses positive reaction should be preferred.

A Reaction Enforcement Unit that implements positive response may include the following methods:

- `void restartReactionLogic(reactionparams)`
- `void triggerReactionLogic(reactionparams)`
- `void fastenReaction(reactionparams)`
- `void slackReaction(reactionparams)`

The variable `reactionparams` is a data structure that will be used to pass additional information to the Reaction logic, whose type is dependent on the precise technique implemented.

The Software Time Bombs reaction mechanism that is proposed in the project could be stopped or blocked but without absolute certainty that a degradation of the application has not actually started. Then no positive reaction can be implemented with this reaction mechanism. Still, the theoretical approach of positive reaction described above is still valid.

4.2 Tamper detection technique 1: Code guards

Section Authors:

Bjorn De Sutter, Bart Coppens, Stijn Volckaert (UGent)

The code guards technique is part of Task T2.5 on Anti-Tampering. This section describes the code guards, which will be used to satisfy the code integrity requirement REQ-NFS-010 of D1.03. Furthermore, the response on detected tampering by this technique should have a delayed tamper response as per REQ-NFS-011.

The initial work on code guards will be reported in deliverables WD2.08 (M18), D2.08 (M24), with initial tool support in time for D2.07 (M24). This initial support will be extended in the following months, to be delivered in D2.09 (M30) and will be reported about in D2.10 (M30).

The ASPIRE tool chain will insert code guards in the protected binary. When executed, each guard verifies the integrity of a part of the protected binary by hashing a region in the code image in the application's address space in memory.

There are two ASPIRE code guards mechanisms: **offline code guards**, and **online code guards.**

4.2.1 System requirements and assumptions

For offline code guards and online code guards we identified the following system requirements and assumptions:

- By themselves, code guard computations can be identified easily by an attacker using dynamic techniques, and eventually be worked around [Van05]. However, when combined with other protection techniques, such as remote attestation, the protection offered by code guards increases dramatically.
- It is easy to provide support for code guards that check other code guards, as long as there is no cyclical dependency between their hash values. Such dependencies can be avoided by excluding hash-value dependent code or data from the checked regions, such as when the hash values are verified on a server instead of in the client application itself.
- Obviously, all expected hash values (to be embedded in the program itself or to be used on the server side) should be computed on the final binary, i.e., after all other protections have been applied.
- Only the code of the protected library itself is guarded. External code that is invoking protected code can only be protected by means of call stack checks (see Section 4.3).

For remote attestation with code guards we identified the following additional system requirements and assumptions:

- Data authentication for attestator and verifier messages is mandatory to avoid simple black box attacks. An attacker may tamper with server-messages to trigger the attestator to produce a set of valid attestation reports. To avoid this, server-authentication verification is needed. Client-side authentication is also needed, for example to allow the server-side verifier to univocally identify attestation report originators.
- An attestation report may contain privacy-sensitive data, i.e., data that univocally identifies the client, or simply valid attestation reports. Therefore confidentiality is advisable.
- Network access is needed, albeit not necessarily continuously. Note that this requirement is valid also for remote code guards.

4.2.2 *Client-side components*

4.2.2.1 Attestator routines: Hashing functions

Code guards are known to get their strength from protecting each other. This requires multiple diversified guards spread throughout the code, including multiple diversified hashing functions. We foresee to inject many of them into the protected binary or library, with each instance guarding a hard-coded region.

At each program point where a code guard needs to be invoked, a call will be inserted to the linked-in hash function, as well as code that sets up the necessary arguments, like the start and end addresses of the code region to be guarded.

When used with online verification or remote attestation, the hashing functions will rely on a nonce (to avoid replay attacks).

Furthermore, when used with an online verifier, the computed attestation will consist of the hash, plus an identifier of the specific guard that was invoked, such that the online verifier knows which guard (i.e., hash algorithm and parameter combination) was invoked.

4.2.2.2 Offline verifier routines: hash check functions

For offline code guards, each hash function instance will come with a corresponding *hash check* function that performs the verification of the computed hash. This function takes as a parameter the return value of the hash function, and returns a verdict.

This verdict can be a Boolean value, but it can also be a set of parameter values to pass upon invocation of an update function of the delay data structures. In the former case, the invocation of the verifier will be followed by a conditional (i.e., on the Boolean return value of the verifier) invocation of an update function with fixed parameters to record the verdict in the delay data structures. In the latter case, an unconditional invocation with the provided parameters is executed after the verifier routine. Such unconditional code is typically much less easily identified by an attacker as code passing verdicts. It is, in other words, much more stealthy.

This hash check function will be compiled and diversified from the same library as the hash functions.

4.2.3 *Server-side components for online code guards*

The client-side online components require interaction with server components. We foresee three server components for the online code guards: a hash randomization component, a hash verification component, and the ASPIRE database to keep track of state.

4.2.3.1 Hash randomization

In the online scenario, the server selects the client's hash function and/or hashing key. The client has initiated a connection with this component through the ASPIRE portal. The client sends to the server an ID identifying the application, and an ID identifying the specific code guard communicating with the server. The hash randomization component of the server chooses the hashing function, hashing key and hashed code regions for a client's code guard, sends this information to the client. Furthermore, in the code guard database this component stores which nonce, hashing function, and code region should now be used by this client application's code guard.

4.2.3.2 Hash verification

The client's code guard sends its computed hash back to the server, together with information identifying the code guards. The server accesses the code guard database for information on this code guard, and whether or not the hash sent by the client matches the correct hash, based on the selected hashing function, hashing key and code regions. The server sends the verification result back to the client, and possibly informs the original application server about the failed verification.

4.2.3.3 ASPIRE database

The server needs to keep track of which hash function, hash key and code region is associated with each code guard. This information is stored in the ASPIRE database.

4.2.4 Code guards offline techniques workflow diagram

Figure 24 comprises the use-case diagram of offline code guards, followed by a table comprising the details of each step. We only detail the Attestator and Verification components of the code guards, the combination of code guards with a Tamper response is further detailed in the Section 4.1.

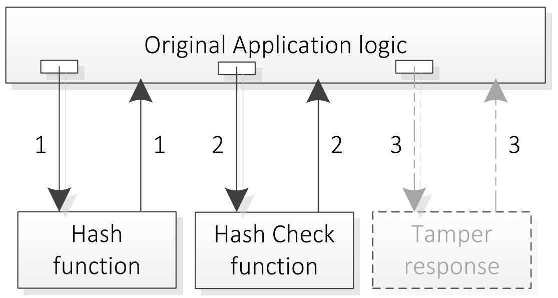

Figure 24 – Offline code guards workflow diagram

Seq#	Operation description
1	Attestator: Hash function computes the hash of a code region.

Details: One of the diversified hash functions spread throughout the program is executed. The hash function reads from a region of memory, and computes a hash of this region. A hash bookkeeping function stores the result of this hash (or a value that depends on it) in the program's data section.

Data passing: The resulting hash value is stored to be verified in a later step.

2	The computed hash is verified.

Details: The diversified hash verification code corresponding to the diversified hash function from Step 1 is executed.

The verification needs to be called after the corresponding hash has been computed. This step can either be merged with the hash computation for immediate hash verification, or it can be executed later during the program's execution for delayed hash verification.

Data passing: The verification returns a Boolean value signifying the verification status.

4.3 Tamper detection technique 2: Call Stack Checks

Section Authors:

4.3.1 Introduction

The call stack checks technique is an anti-callback technique that is part of Task T2.5 on Anti-Tampering. It implements part of requirement REQ-NFS-013 of D1.03 that code injected by an attacker cannot call back into protected code. Furthermore, the response on detected tampering by this technique can have a delayed tamper response as per REQ-NFS-011.

Implementing call stack checks as originally described in D1.04 v1.0 proved to be harder than anticipated. Therefore, we had not concrete initial work to report in M18 yet. The initial work on call stack checks is reported in deliverables WD2.08 (M18), D2.08 (M24), with initial tool support in time for D2.07 (M24). This initial support will be extended in the following months, to be delivered in D2.09 (M30) and will be reported about in D2.10 (M30).

Call stack checks provide anti-tampering for protected binaries by regularly checking that certain features of the call stack are consistent with non-tampered execution. For the ASPIRE project, we will implement simple return address checks that verify that the return addresses of functions originate in allowed code regions. The allowed code region consists of the entire executable segment of the protected library or application.

4.3.2 System requirements and assumptions

- Call stack checks prevent a program under attack to execute with invalid code addresses on the stack. Attackers can avoid the occurrence of invalid code addresses, however, by allocating their malicious code at valid addresses, i.e., by overwriting original application code at valid addresses. To prevent such attacks, code guards need to be applied (See Section 4.2).
- Call stack checks will only verify *internal* functions, i.e., functions that provably can only be called from other functions of the protected program. Functions that are exported, or that can be called as a call-back function from external code will not be protected.
- Call stack checks will only verify one single caller of a function, rather than the entire call stack.
- If a protected function can only be called from a single call site, the call stack checks will verify that, on function entry, the return address corresponds to this single call site. In all other cases, call stack checks will verify only that the return address originates in the protected library.
- While these call stack checks could in principle be extended to interact with mobile code, we will not implement this in the scope of the ASPIRE project.

4.3.3 Client-side components

The call stack checks themselves are small code fragments that are in-lined in diverse locations in that application code.

Figure 25 presents the call checks workflow, followed by a detailed description of each of the steps below the figure.

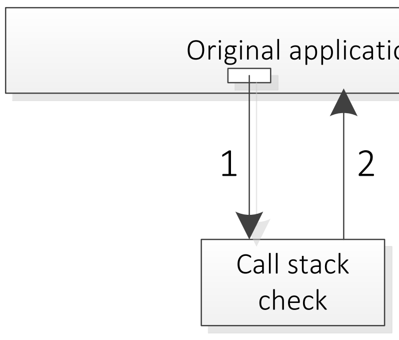

Figure 25 – Call stack check workflow diagram

Seq#	Operation description
1	**Call stack check is triggered and executed.**

Details: Code that has been protected with call stack checks has had small check stubs inserted. These check stubs perform the call checks themselves.

Dependencies: The usefulness of this step depends on the integrity of the code located on the addresses in the call stack. Thus, code guards need to be used in combination with this technique.

2	Control flow is redirected.

Details:

When the check succeeds, regular program execution continues. When it fails, for example because a return address is found on the stack that is not allowed there, a tamper response is triggered. This response can be delayed by means of the functionality provided through delay data structures and a range of response mechanisms. However, in case the check detects that sensitive code is invoked in unauthorized ways, the response should not be delayed. Instead, the execution of the code should be interrupted immediately to prevent the execution of computations or communications that leak sensitive data.

4.4 Tamper detection technique 3: Static Remote Attestation

Section Authors:

Cataldo Basile, Alessio Viticchié (POLITO)

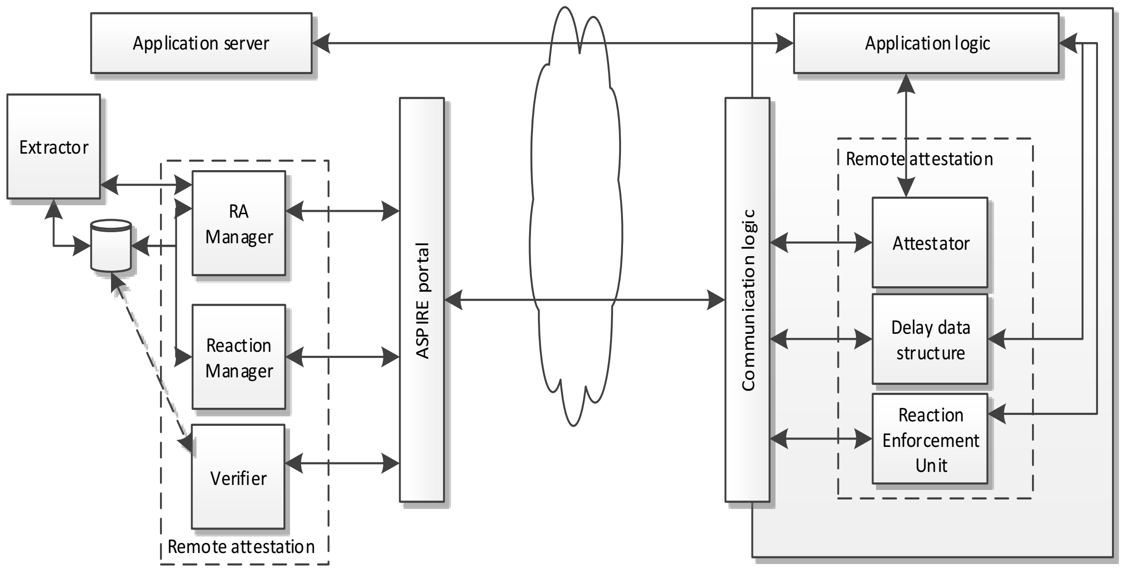

Figure 26 – Static remote attestation reference architecture

Static remote attestation is an instantiation of remote anti-tampering technique.

The purpose of static remote attestation is to attest selected code areas of the application in memory that need to be protected for integrity. One of the main design constraints is that static remote attestation techniques are not vulnerable to replay attacks (i.e., reusing previously generated attestation data that can be resent by a man-in-the-middle attacker). Therefore, attestations are computed using a random nonce that is sent by the RA Manager. Moreover, the value of the nonce drives the attestation process: the area to attest and how areas' data are processed

Compared to the general architecture, in the static remote attestation in Figure 26 an additional component is present, the Extractor, whose purpose is to pre-compute attestation data, as will be described later in Section 5.2.2.4.

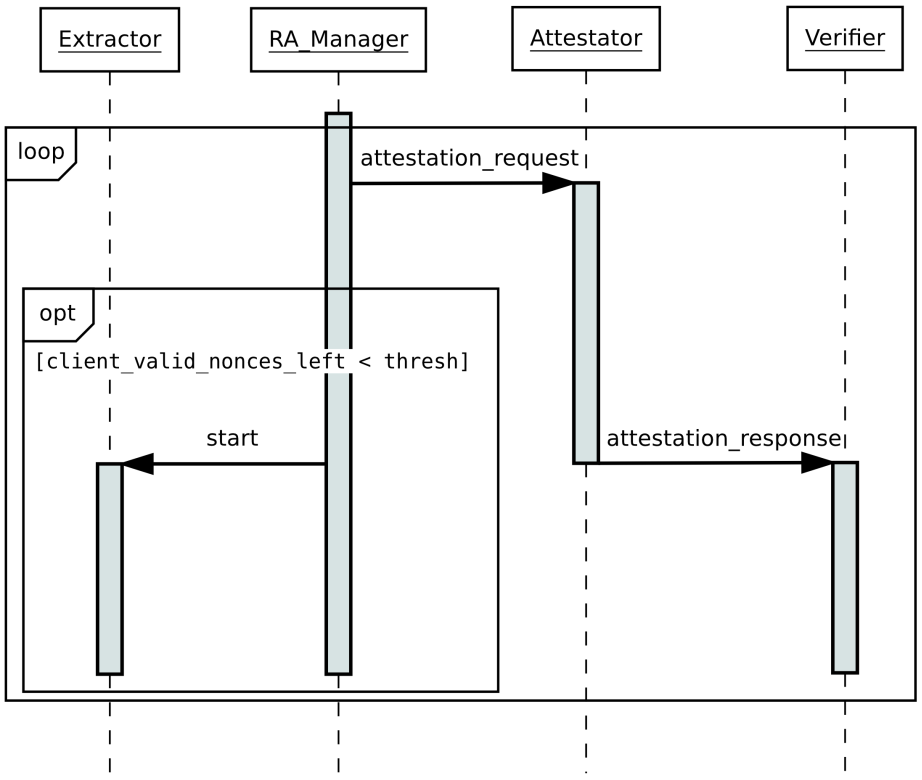

Figure 27 – Static remote attestation workflow.

Figure 27 presents the up-to-date workflow for remote attestation:

- When the RA Manager retrieves a new fresh nonce to prepare an attestation request for a client, it also verifies how many remaining nonces are available for a given user
- The server-side RA Manager sends an attestation request to the client Attestator using the ASCL. The format of an attestation request is described in Section 4.4.1.4.1.
- The client-side Attestator computes an attestation response and sends it to the server-side verifier. The format of an attestation response is described in Section 4.4.1.4.2.

The M24 remote attestation prototype implements exactly this protocol.

4.4.1.1 Attestator

This client-side component is in charge for processing attestation requests received by the RA Manager. It produces the attestation response, with an ad hoc attestation data generation algorithm, and the attestation response, by hashing the attestation data and other client identification information. Afterwards, the Attestation sends the Verifier the attestation Reply by means of the ACCL.

The Attestator is executed when an attestation request is received by the client communication logic and directed to the Attestator. Due to the characteristics of the current implementation of the ACCL based on web sockets, the Attestator is executed in a separate thread.

There are several variants of the attestator (and corresponding Actual Verifier) depending of four components:

- Random walk algorithm, which defines how the memory area to protect is processed to obtain the attestation data;
- Nonce interpretation, which defines how the nonce are split and processed to derive the parameters used by the random walk algorithm;
- Representation of the memory areas to protect, which encodes the memory block that form all the memory areas to protect;
  Hash functions, which are used to digest the attestation data.

4.4.1.2 Static RA Actual Verifier

The Actual Verifier is the server-side component that checks the correctness of attestation responses. To perform this verification it compares the attestation response with the result of the same computation performed on the same attestation data obtained from an untampered version of the application to protect. There is no direct communication between this Actual Verifier and the RA Manager. When the Verifier receives a new attestation response, it collects all the needed information about the request that solicited the response in the ASPIRE DB.

The attestation data may be generated after a response is received or pre-computed and stored on the ASPIRE DB.

The Verifier logs on the DB the result of the attestation according to the following codes:

- PENDING, if no answer has been received yet, the timeout has not expired;
- SUCCESS, if the request has been received on time and the Verifier has verified it as correct;
- FAILED, if the requests has been received on time and the Verifier has verified it as incorrect;
- EXPIRED_SUCCESS, if the request has been received late and the Verifier has verified it as correct;
- EXPIRED_FAILED, if the request has been received late and the Verifier has verified it as incorrect;
- EXPIRED_NONE, if no answer has been received and the timeout has expired.

4.4.1.3 Extractor

The Extractor is the server-side component that pre-computes and stores in the ASPIRE DB the attestation data associated to the nonces.

Practically, it randomly generates a set of nonces. Starting from an untampered version of the application to protect, it then computes the attestation data. To permit the reuse of the same nonces with different clients, the extractor does not compute attestation responses, only attestation data. As presented in Section 4.4.1.1, attestation responses are obtained by concatenating client-specific to the attestation data before computing the hash, thus attestation data can be shared among clients.

The presence of an Extractor allows the reduction of the verifications' time and load at run-time, even if the same computations need to be performed off-line. However, from our experience, the Extractor is very efficient, hundreds of nonces and related attestation data can be produced in seconds on an off-the-shelf laptops. The nonce generation can be easily parallelized.

The Extractors is a helper component that is neither exposed as an ASPIRE service nor directly reachable by clients, it is only invoked by the RA Manager when the number of remaining nonces is less than a predefined threshold.

4.4.1.4 Static RA Messages

This section presents the current attestation request and response message format. These format is being rearranged to support, in Y3, the use in the same clients more variants attestators in the same client to attest the same memory areas.

4.4.1.4.1 Attestation request

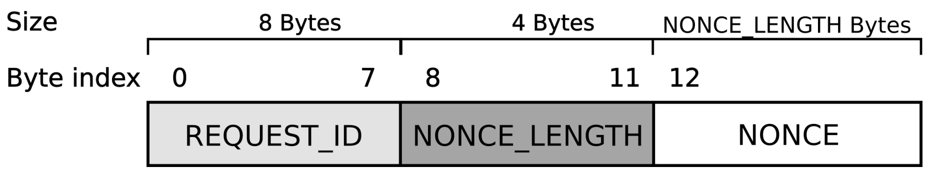

Figure 28 – Attestation request format

The attestation request uses the following format (see Figure 28):

- REQUEST_ID, 8 bytes used (together with the client ID) to univocally identify the attestation request (in the ASPIRE DB)
- NONCE_LENGTH, 4 bytes used to indicate the length in bytes of the nonce, in our case NONCE_LENGTH = 32
- NONCE, the actual nonce.

Currently, the

4.4.1.4.2 Attestation response

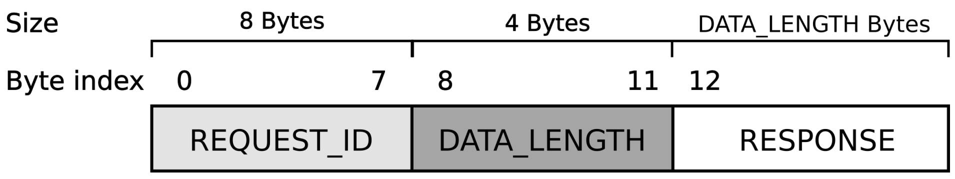

Figure 29 – Attestation response format

The attestation response uses the following format (see Figure 29):

- REQUEST_ID, 8 bytes used to univocally identify the attestation request to which this response pertains;
- DATA_LENGTH, 4 bytes used to indicate the length in bytes of the nonce. This length depends on the output of the hash algorithm used;
- RESPONSE, the actual response.

4.5 Tamper detection technique 4: CFG Tagging

Section Author:

Jerome D'Annoville (GTO)

4.5.1 Introduction

CFG tagging is an anti-tampering technique that aims to detect if the execution flow graph is modified in a way that is not expected by application developers. This protection originates from code coverage tools used in software testing where it enables to measure the amount of code that is activated by a test suite. A common implementation of such tools is to insert probes in the code to measure the edges of the CFG that are actually activated by the tests.

The same idea is reused with the CFG tagging protection where some counters are inserted in the code and each time a supervised basic block is activated the corresponding counter is incremented. The counter management code, call the Gates, are the Attestator part of the tagging protection. The Gates are distributed in the CFG of the application according to the annotations set in the source code. The Verifier part of this protection technique checks if counters values are consistent. Based on rules set in annotations the Verifier gets counter values and checks that rules are respected. If it is not the case, it means that an incorrect behaviour is detected in the execution flow of the program and the Verifier triggers the reaction mechanism by setting adequate indicator in the Delay Data Structures forcing the program to activate an appropriate response.

This protection technique requires the application developer to specify which paths in the execution flow have to be supervised and to set coherence rules that will be checked in the Verifier. It is also theoretically possible to perform an exhaustive protection of the execution flow, but this would probably lead to heavier control computations, which first would make them easier to spot by an attacker and second would bloat the code with useless controls.

The Verifier code can be inserted in the application code or be located remotely and combined with the online remote attestation protection technique. In the latter case counter values are sent to the server in a payload. For the purpose of the project only the offline release will be implemented. See the plan in the next paragraph about the online CFG support.

This CFG tagging technique has been initially described in the WP3 part of the DoW as a remote attestation technique. However, further analysis has revealed that the tagging technique can be decoupled from the response step and that the verifier component can be either local or remote. Then the offline part with the Attestator –the Gates- and offline Verifier will be available in D2.09 (M30) and described in D210 (M30). In the meantime the Reaction logic will be supported on the server and combined with the Remote Attestation protection. Then in Q2 2016 the CFG online will be implemented with the Verifier on the server. This feature will be available in D3.07 (M33) and reported in D3.08 (M33).

4.5.2 System requirements and assumptions

The purpose of the CFG tagging protection is to provide a way to check that identified parts of the application have been executed as expected. Code transformations can be applied after Gates and Verifiers have been inserted, which will reinforce the tagging protection by hiding more deeply the tagging code, thus preventing the attacker to spot the protection code through pattern matching.

Tagging is also complementary with code integrity checking techniques brought by code guards. A Gate would take advantage of the static protection by guards, as any code modification attempt to change the Verifier part could be detected by the guards.

4.5.3 Client-side components

The CFG Tagging technique is focused on the application execution flow by tracking the dynamic execution path with the following components:

- Attestators: these components are a set of probes, called the Gates. A Gate is the place where a counter is incremented.
- Verifiers: in accordance with the information provided by the various counters, the Verifiers are detecting changes in the expected execution flow by checking the consistency of the counter values. Verifier code can be either local or remote and in this later case another component is required to interface with the server. This component is a Verifier connector.
- Verifier connector: This component sends data remotely in case the Verifier is on the server side.

4.5.3.1 Counter

A counter is a data location in memory to save an integer value. It should simply be correctly hidden, in order to make its detection more difficult for attackers. Indeed, the counter location is critical information with this protection technique. Rather than trying to add protections on the counter, the option has been taken to keep it as lightweight as possible.

4.5.3.2 Attestators

The application is sprinkled with Gates, which are small pieces of code inserted in a basic block. The density of Gates is a parameter that is left to the user's choice and will be driven by annotations. Once a Gate is reached, the related Counter is updated to reflect this node of the CFG has been activated.

Each Gate needs to access two data elements in memory: a factor and the Counter. It is important to hide these addresses computation and not to hardcode them directly in the Gate. Thus, those values are split in two: one part is hardcoded in the Gate, the other is in memory and accessed indirectly by offset.

4.5.3.3 Local Verifier

A local Verifier retrieves the Counter values required to check a rule, it checks the rule, and in case it fails it sets up the data structures in the delay component that will be used by the reaction code.

4.5.4 Verifier connector

The role of this component is to get counter values, prepare a payload and send it to the remote verifier. A Verifier identifier is also passed in the payload to enable the Verifier to retrieve the corresponding rule and to make the matching with the counters passed in the payload.

4.5.5 Server-side components

The advantage of using a remote Verifier is that no expected counter values appear in the client application code as immediate values. This makes it more difficult for an attacker to guess what should be restored in case the code has been tampered with. A drawback is that calls to the server are easy to retrieve and arguments passed in the payload can perhaps be analysed by the attacker. The main issue using a remote Verifier is that the reaction client part waiting for reaction notification from the Reaction Manager cannot hidden within the application is can be blocked by the attacker.

4.5.5.1 Remote Verifier

The rule is checked the same way it is done in the local Verifier. The main difference is that data have are saved and retrieved to/from the ASPIRE database. The reaction can differ because the application server can be notified in case a rule is not checked.

4.5.5.2 ASPIRE database

The ASPIRE database contains the rules and the Counters passed for a rule.

4.5.6 CFG tagging offline technique run-time behaviour

Figure 30 depicts the offline CFG tagging workflow diagram, followed by a detailed overview of the referenced steps.

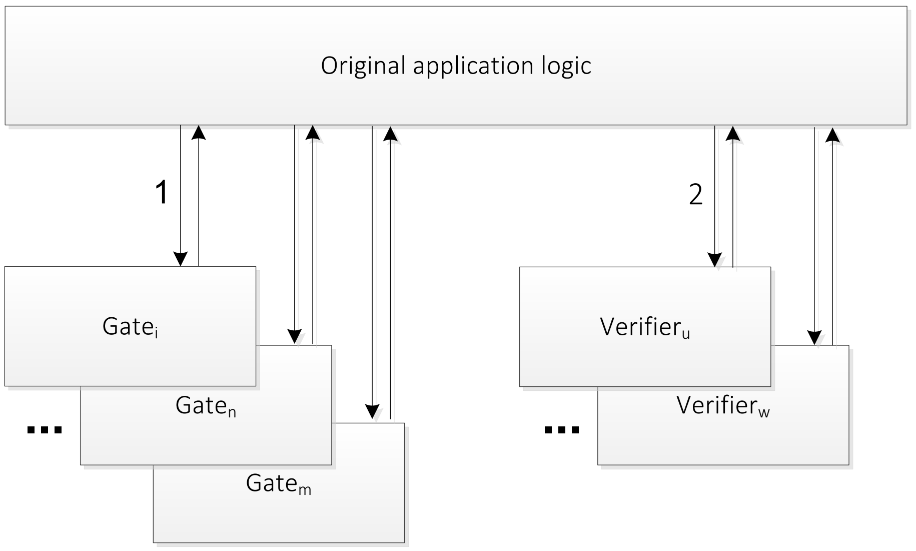

Figure 30 – Offline CFG Tagging workflow diagram

Seq#	Operation description
1	**The Gate increments the Counter's value.**

Details: The execution flow reaches a Gate. The Gate increments the Counter value.

2	The Verifier checks for deviation.

Details: The execution flow reaches a Verifier. It retrieves the required Counters' values and performs several computations with it. The details of these computations are determined by the rules set by annotations in the source code. According to result of these computations the adequate field in the Delay Data Structures is set.

4.5.7 CFG tagging online technique run-time behaviour

Figure 31**Error! Reference source not found.** depicts the online CFG Tagging workflow diagram, followed by a description of the referenced steps.

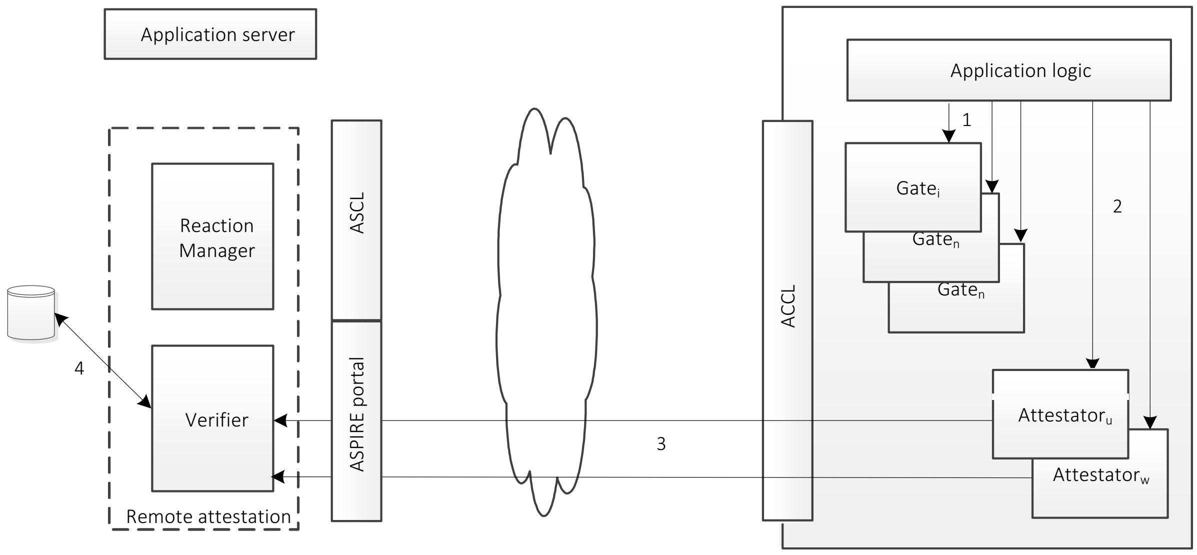

Figure 31 – Online CFG Tagging workflow diagram

Seq#	Operation description
1	**The Gate increments the Counter's value.**

Details: When the execution flow reaches a Gate then the Gate increments the Counter value.

2	The Verifier Connector prepares a payload.

Details: The execution flow reaches a Verifier Connector. It retrieves the required Counters' values and prepares the payload.

3	The Verifier Connector sends a payload to the Remote Verifier.

Details: The payload is sent to the ASPIRE portal that forwards it to the Verifier..

4	Verdict is saved in the Database.

Details: The Verifier check if the counter values conform to the rule. The verdict is stored in the database. The reaction logic will be triggered independently from those verdicts.

4.6 Tamper detection technique 5: Anti-cloning

Section Author:

Brecht Wyseur (NAGRA)

4.6.1 Introduction

An inherent problem with software applications running on open platforms is that its code and data can always be copied. An attacker can always replicate the binary representation of an application to run it on another machine.

One solution to mitigate such attack is by binding the application and its execution to the platform. This could be achieved using special-purpose hardware that ensures unique execution of applications. This however is not feasible on open platforms as we envision in the ASPIRE project. In our attack model (as described in deliverable D1.02), an adversary can intercept and emulate any communication between the application and the hardware.

In Task 3.2 of the ASPIRE project, we will develop a solution to this problem, which we denote as the anti-cloning technique. This is a technique that allows for remotely managing unique client identification, and can detect when two versions of the exact same application (clones) are running. This is challenging to detect, in particular when the two instances are running each in a different time frame. Two instances that are running at the same time could be detected through behavioural analysis of the network connectivity; two instances running sequentially can be hard to distinguish, as they are identical.

The underlying idea of the technique is to make applications *evolve* uniquely, and enforce this using the network connection. While two applications may be identical at the moment when one is cloned from the other, as soon as one connects to the ASPIRE anti-cloning security service, they will differentiate. This evolution can be enforced prior to the delivery of any valuable service.

Attackers that aim to mitigate this technique will need to clone the application again each time they wish to request the service, as their instance will be desynchronised once the other instance has connected. This is an attack that is hard to scale, and meets the objectives of the ASPIRE project, where we aim to discourage attackers, in this case by forcing them to make continuous efforts.

In this section, we describe a preliminary proposal of such anti-cloning mechanism. As T3.2 starts at M10, only initial ideas are available so far. Details on this construction will be reported in Deliverable D3.01 (M18).

4.6.2 System requirements and assumptions

The Anti-Cloning mechanism requires an *occasionally* connected system.

4.6.3 Client-side components

4.6.3.1 Tag

A special-purpose value needs to be introduced in the application. We denote this value as ‘Tag’ value. It embodies the *evolution* of the application instance. The variable will be frequently updated during the execution of the program.

The tag value should be persistent between different executions of the application and should thus be stored in non-volatile memory, preferably in a way that makes it non-trivial to identify and copy the variable.

4.6.3.2 AC manager

The client-side anti-cloning functionality is introduced by the AC manager, which is a library that needs to be statically linked with the application. The AC manager can be invoked with a call to its API to launch the mechanism, will manage the responses from the server, and update the tag value when needed.

4.6.4 Server-side components

4.6.4.1 AC decision logic

The AC manager interacts with a server component, which we denote as the Anti-Cloning decision logic. This component can verify the correctness of tag values of application instances by comparing the received tag value with the expected tag value that is stored in the ASPIRE backend DB.

This component is also able to signal to the AC manager that the tag value needs to be updated. This AC decision logic is defined in a policy which can be configured. The configuration of this policy is out of scope of the ASPIRE project.

4.6.4.2 ASPIRE database

A list of expected tag values corresponding to the client identifiers needs to be stored in the ASPIRE database.

4.6.5 Anti-cloning workflow diagram

Figure 32 depicts the anti-cloning workflow diagram, followed by a detailed overview of the referenced steps.

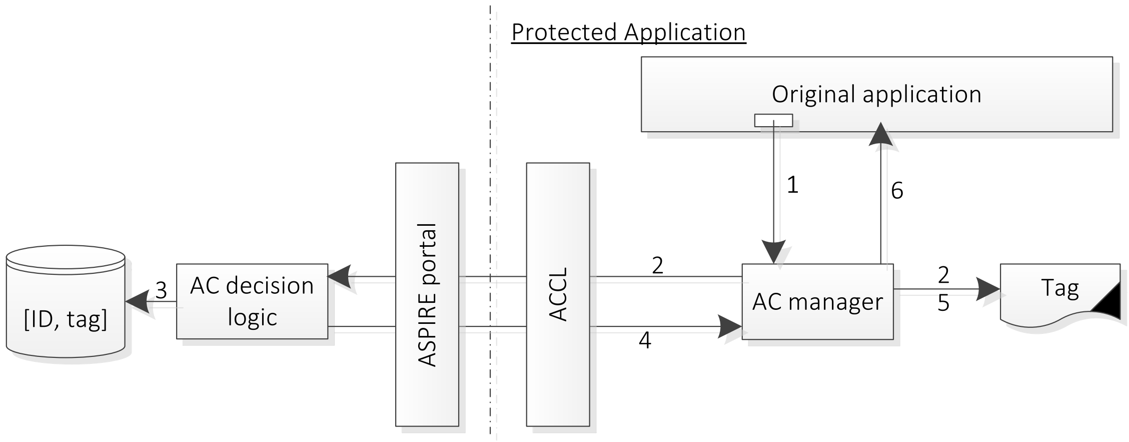

Figure 32 – Anti-cloning workflow diagram

Seq#	Operation description
1	AC manager is invoked by the client application.

Details: The AC manager is implemented as a native library, and exposes a C function that can be invoked.

2	Tag value is sent to the server-side support logic.

Details: The AC manager fetches the tag value, and sends this value to the AC decision logic.

3	The AC decision logic queries the ASPIRE database, and decides what its response will be.

Details: The response depends on

- The value of the (ID, tag) couple that is received,
- The policy that is implemented by the AC decision logic
- The Tag value that is stored in the ASPIRE database, associated to the application ID.
- The current state of the application (e.g., the application may have been blacklisted)

There are three possible responses:

1. The AC decision logic decides that the (ID, tag) couple that is received is OK, and no further updates are needed. It will respond "OK".
2. The AC decision logic decides that the tag value needs to be updated at client side. In its response, it will send payload to the application that it needs to use to generate a new tag value. It will then respond "UPDATE, Nonce"
3. The AC decision logic decides that the (ID, tag) couple is unacceptable, and will blacklist the application associated to the ID. It will respond "NOT OK".

4	The AC manager receives the response from the server, and takes subsequent action.

Details: The action that the AC manager takes is defined from the response received.

If an update response is received from the server ("UPDATE, Nonce"), the AC manager will compute a new tag value, using that Nonce, and proceed to step 5.

In the case of "OK", the AC manager will return control to the original application (step 6).

In the case of "NOT OK", the AC manager can invoke the delay component to update its data structures to trigger a delayed tamper response (See Section 3.6), or can directly trigger a tamper response.

5	The tag value is updated

Details: The AC manager will update the tag value with the new computed tag.

Subsequently, the sequence flow will proceed with step 2 again: fetching the (new) tag value and send it to the server, waiting for its response.

6	The AC manager returns control to the application.

4.6.6 Server status report request

The application server can request the ASPIRE portal if a specific application ID is blacklisted or not, as depicted in Figure 33.

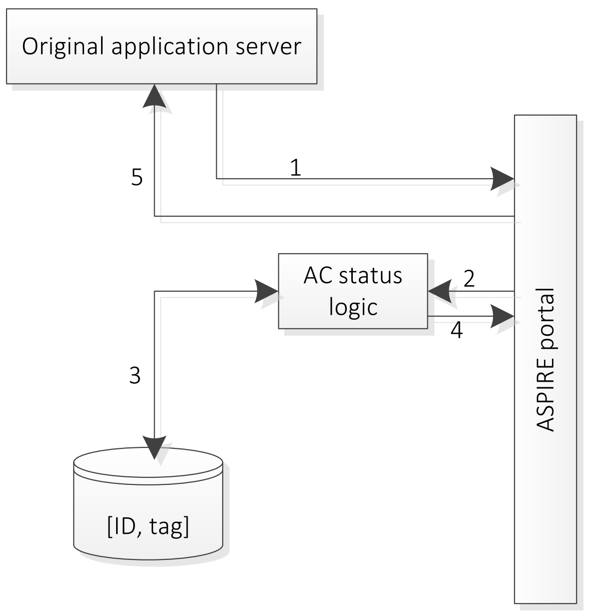

Figure 33 – Request for trustworthiness status report

Seq#	Operation description
1, 2	The server initiates the request.

Details: The server connects to the ASPIRE portal, requesting for a status update on the application instance identified with ID. The ASPIRE portal forwards the request to the AC status logic.

3,4,5	The AC status logic returns the status report.

Details: The AC decision logic fetches the information related to the application instance from the ASPIRE database, and derives a status report from it. It sends the report as return message to the application server that requested the report.

4.6.7 Error management

There might be an issue in case of de-synchronisation between the tag value at client-side and the tag value that is stored in the ASPIRE database. This can occur when the AC decision logic updated the ASPIRE database value, but the AC manager failed to update the tag value.

Fall-back mechanisms to resolve such issues in case of de-synchronisation can be designed, but we consider them to be out of scope of the ASPIRE project. They are not of core relevance to the protection technique but rather a pure engineering task. This relates to the implementation of a policy and management of blacklisted clients.

4.7 Delayed Tamper Response: Delay Data Structures

Section Authors:

Bjorn De Sutter, Bart Coppens (UGent)

Sections 3.2 to 3.6 focused on techniques with which verdicts about tampering could be made. Now we shift the focus to reacting on those verdicts, and to delaying that reaction. As explained in Section 3.1, the overall strategy and architecture is to record the verdict in so-called delay data structures by invoking update functions on them, and to react upon the recorded verdicts in (unrelated) locations later during the execution of the program on the basis of queries to those data structures. In this section, we focus on the delay data structures and their possible APIs, and on the possible reactions.

4.7.1 Delay data structures and their API

4.7.1.1 Data structures

In support of delayed tamper responses, a set of statically and/or dynamically allocated data structures is injected into the program, together with a number of update functions and query functions that manipulate and query the data structures. The core concept is that under normal operation of the program, the data structures maintain one or more invariant known at compile time. During the execution of the program, update functions on the data structures can be invoked that maintain the invariants. However, whenever a negative verdict occurs, an update function will be invoked that breaks one or more invariants. Thus, the data structures keep track of possibly multiple forms of tampering having been detected or not, and they do so in an obfuscated manner. By querying the data structure, the encoded properties can be retrieved, and delayed responses can be implemented without directly linking their trigger to the failed code guard.

To avoid easy detection by attackers, the data structures used to encode information about the status of protections should not be fixed or special-purpose. Instead multiple variations should be available that span a wide range of commonly used data structures. Overhead and protection level can then be considered when choosing particular implementations, and variations can be chosen that are in line with the data structures already present in the original program. For example, if a program already involves many linked lists or hash tables, encoding verdicts in the data contained in such container structures will be much stealthier than using a number encoding based on prime numbers that occur nowhere else in the original program.

Furthermore, the interface to those data structures should not be fixed. It cannot be fixed because a wide range of data structures has to be supported, and it should not be fixed because it should not be easy to identify its use in a program, e.g., through the pattern matching attacks described in D1.02.

Some potential data structures that might be useful, ranging from very simple ones to very complex ones are the following:

- Numbers encoding information, in which case bit combinations or mathematical properties of numbers encode the status.
- Containers where the number of elements with certain keys or values store can store the status, as can the presence of aliasing iterators operating on the data structures.
- Graphs, in which presence of certain aspects encodes the status, such as nodes with multiple incoming edges, or loops in the graphs, ...
- Automata in which the states encode the information.

The number of available options is only limited by human imagination.

Depending on the overall form of the anti-tampering technique (e.g., offline techniques only vs. remote attestation) and on the types of response that need to be activated, different forms and amounts of information need to be encoded: In the simplest case, only the presence of tampering needs to be encoded. In more complex schema, it might be encoded which technique resulted in a negative verdict, how many times that has happened, how many times that has happened since the last positive verdict or since the last verdict executed under the control of a remote server, etc. For example, the information stored could be a combination of the following Boolean values:

- `t_bool has_been_tampered`:[1] This directly stores whether any tampering took place.
- `t_bool CFG_tagging_violation`: This stores whether one specific form of tampering was detected.

4.7.1.2 Update and Query Functions

Based on a negative verdict, the protected program will invoke an update function to record the verdict in the data structures. A wide range of interfaces can be used for this purpose, such as

- `set_tampering()` This update function is to be invoked conditionally, i.e., only when the verdict was negative. This is relatively easy to spot by an attacker.
- `set_CFG_tagging_result(t_bool verdict)` This update function can be invoked unconditionally, and is given the Boolean verdict as an argument.
- `insert(int key, void * value)` The verdict and the status update to be recorded is encoded in some property (known at compile time, but not easily determined by an attacker) of the key value, the values of where the pointer points to, and/or the pointer value itself.

While the former interface is easier to handle in a protection tool flow, the latter interface is much stealthier; in particular if no conditional statements are executed inside the update function for which the verdict needs to be extracted explicitly from the passed arguments. Human imagination again seems to be the limiting factor, as well as the needed compiler support, and the fact that the verifier routines (local or remote) need to be able to generate the appropriate arguments for the update functions. The more complex the interface and the encoding of the verdict in the passed arguments, the more tightly coupled the verifier routine and the delay data structures become. This more complex interface can be automated to a large degree, by allowing users to define the complex data structures, and the effect that different functions and their arguments have on these data structures. The tool chain can then automatically inject the correct function calls with the correct arguments.

Similarly to the update functions, the query functions can span a wide range of interfaces, such as

- `t_bool check_tampering()`
- `void check_tampering(int * result)` (result in memory)
- `void * check_tampering()` (pointer is result)

With each of the above interfaces, the result can be turned into a Boolean predicate (i.e., a conditional branch) that is used to invoke a tamper response or to continue normal execution. Alternatively, and much stealthier, the query operations can result in data, like pointers, on which unconditional actions are performed that, in case no tampering was detected, leave the

[1] We should note that while we use meaningful function and variable names in this section for the sake of clarity, obviously no function or variable names will be present in the generated code.

normal program execution unaltered, and in the other case trigger the actual response, e.g., by corrupting the memory. For example, with the latter query function, the returned pointer value might be passed to the `memset()` function. In case no tampering was detected, the pointer points to an injected array that may be overwritten without affecting the rest of the execution, while in case tampering was detected, it points to some random location in the middle of the program's true data, which is therefore corrupted by the `memset` operation.

Again it is clear that a simpler interface supports a looser coupling between the query functions and the actual tamper responses, while a more complex one will allow a stealthier coupling.

4.8 Reaction: Software Time Bombs

Section Author:

Jerome d'Annoville (Gemalto)

4.8.1 Introduction

Each software protection needs to be protected itself to prevent attackers to make the protection ineffective. Several strategies exist to achieve this, like integrating a protection mechanism in the protection itself. This technique is used, e.g., by viruses that are encoded as bytecode to a custom embedded VM to make their detection more difficult. Another strategy is to hide the protection as much as possible making it difficult to spot for an attacker. This latter strategy is used for Software Time Bombs (STB).

STB is a technique developed for the purpose of providing a reaction mechanism for other protection techniques. It is a partial defence mechanism, since it does not provide any *detection* functionality but only a *reaction* functionality. It is accessible through an API enabling other protection techniques to alter the behaviour of the application in case of attack detections.

STB provides protections with a *delayed* damaging action on code execution. When started, STB eventually stops the execution of the program . Through the API, methods are provided to control the behaviour of this mechanism. Besides enabling the activation of STB, the defence action delay offers an advantage of stealth to the protection. Indeed, it makes it more difficult for an attacker to find that STB is the cause of the program stoppage if the related reactive action is noticeable some amount of time only *after* its triggering.

STB is one of the protection of Task 3.2 of the DoW, as a reaction mechanism of the Remote Attestation protection method based on CFG tagging. During the design and brainstorming about different techniques, however, it became clear that this technique could be proposed as a service that anti-tampering techniques can use as reaction mechanism..

The experimentation on this technique are reported in D3.04 (M24). Final support will be provided in D2.09 (M30) with associated report in D2.10 (M30).

4.8.2 System requirements and assumptions

Once started, STB performs very few actions until a certain point. Once this point is reached, the real defence mechanism is triggered. From then on nothing can be done anymore to make the program working correctly. The behaviour of STB is controlled through its API:

- `startReaction():` Start the reaction timer. Other API methods (`restartReaction` put aside, obviously) have no effect on the STB behaviour if this method is not called.
- `restartReaction():` Restart the reaction timer as if it was just started for the first time. The speed of reaction although remains the same as before the `restartReaction` invocation.
- `stopReaction():` Stops the reaction. If the reaction is restarted, it restarts from the beginning.
- `fastenReaction():` Ups the reactions speed.
- `slackReaction():` Slows the reaction speed down.

Two usage scenarios supported with such an API include both *Positive Reaction* or *Negative Reaction*, as discussed in Section 4.1.2.3. Negative Reaction is the direct triggering of the reaction mechanism on detection of a threat. STB could be useful for such a usage in the case of unsure detections. For example, if a detection mechanism notices a threat but is not sure of it, it can start the STB mechanism and stop it if the threat is later proved unfounded (*false positive*).

Positive Reaction works the other way around. The STB has its reaction started from the beginning of the execution. Security checks are performed on regular basis and if they succeed, the reaction is restarted. If they fail, obviously the reaction is not restarted and the execution eventually fails. In this usage scenario, functions `startReaction` and `stopReaction` are not required. For the purpose of the project only Negative Reaction is proposed for online protection techniques and it is not recommended to use STB as a positive reaction mechanism by the offline protection techniques because there is no control on the time where the degradation has reached the non-return point.

Again, STB is not intended to work by itself. It is both a Delay and Response component that acts on behalf of protection techniques and which action is controlled by protection techniques.

Client-side components

The STB Reaction Enforcement Unit is based on two components:

- Accumulator: This is a simple counter that is incremented. Its only purpose is to be accessible by Incrementors. There can be several Accumulators but the number is limited.
- Incrementors: These are small software components whose only purpose is to modify their Accumulator value. Incrementors are part of STB. They are not part of Verifier of protection components. A set of Incrementors are related to a single Accumulator.

4.8.3.1 Accumulator

An Accumulator is a global variable in memory. It is accessible to Incrementors that increase the Accumulator value each time one of its Incrementor is called. When the maximum value of the counter is exceeded, a punishment is triggered on execution.

The accumulator is a passive component. Its presence in the model is only for the purpose clarity, but it is not a real dynamic or active component.

3 functions in the API directly modify the Accumulator’s value.

- `startReaction():` Activates the accumulator..
- `stopReaction():` Deactivates the Accumulator..
- `restartReaction():` Sets the reaction time to its initial value..

4.8.3.2 Incrementors

Incrementors are small pieces of code positioned at places in the CFG that have a high probability to be executed like a common node of two sub-graphs. Their only role is to increase the Accumulator’s value, in order to make it reach its maximum value. All Incrementors perform the same action on their Accumulator: Each one makes Accumulator’s value grow at the same speed. The Incrementor's behaviour is not conditioned: they are always executed whatever is the verdict of the protection Verifiers.

Each Incrementor must be able to access its Accumulator through its address. Moreover, in order to ensure that each Incrementor performs the same transformation on its Accumulator, all Incrementors access the same growth parameter. This growth factor is saved in memory and its address is given to all Incrementors of an Accumulator.

Two functions in STB API perform modifications on Incrementors.

- `fastenReaction():` Makes Incrementors action twice as fast..
- `slackReaction():` In a symmetric manner, this cuts the Incrementors action’s speed by half.

4.8.4 *Software Time Bombs run-time behaviour*

Two mechanisms are described hereafter. The first one depicts a static invocation of STB. The reaction actions are performed regardless of the STB API. This is typically the way online protections will activate the degradation of the application through the Delay Data Structures. The second one depicts the dynamic behaviour of STB that is triggered by offline protection techniques.

4.8.4.1 Passive operations workflow

Passive operations are the executions of Incrementors throughout the application. This is depicted in Figure 34 and each corresponding step of the Figure is subsequently detailed.

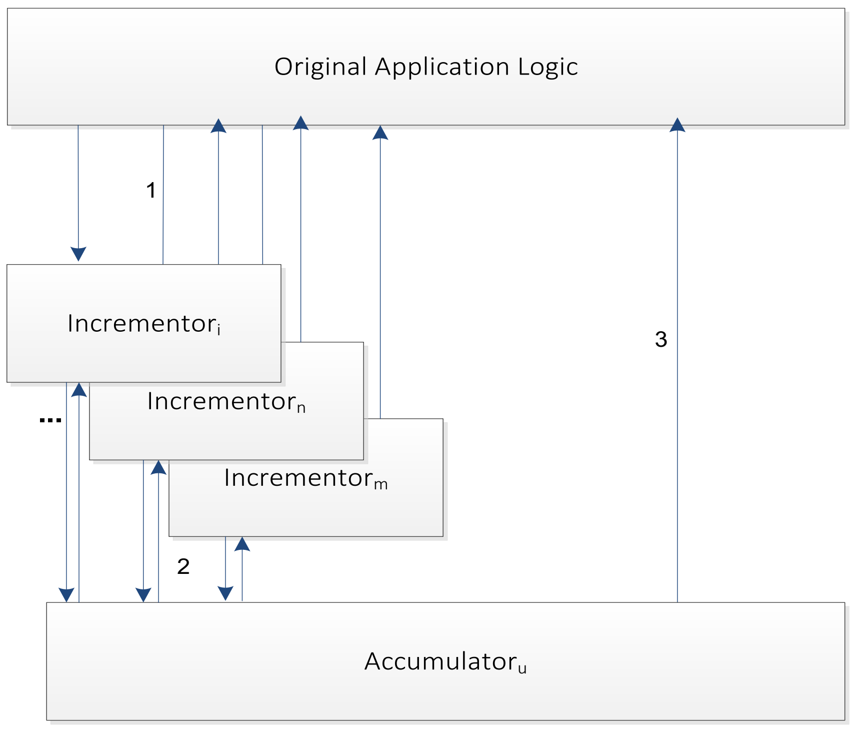

Figure 34 – STB workflow diagram for Passive Operations

Seq#	Operation description
1	**Incrementor is activated.**

Details: Execution passes through Incrementor #i. It activates the growth factor if the Delay Data Structures related field is set.

2	Accumulator value is updated.

Details: Accumulator value is multiplied by growth factor: `accumulator *= growth_factor`.

The Accumulator is an integer variable. If allocated on four bytes then with a growth_factor equal to 2 there will be an overflow after 32 updates of the Accumulator. This is the expected behaviour of the Accumulator that will spread memory corruption once it has been activated and updated until a threshold where the overflow occurs. Once there is an overflow it exceeds the allocated Accumulator variable to corrupt the previous allocated variable.

Speed and effect of the overflow propagation will vary according to the numbers of executed Incrementors and the reaction speed

3	Punishment is triggered.

Details: This operation is conditioned by the reaching of its maximal Accumulator value. When the maximal value is reached, punishment gets triggered. This happens on one of the many operations #2 that will be executed, but it cannot be predicted when exactly it occurs, as that depends on the last intervening reset of the Accumulator.

4.8.4.2 Active operations: Workflow diagram

Figure 35 depicts the way STB is used by offline protection techniques. Verifiers may call the STB to manage the Delay and the Response component.

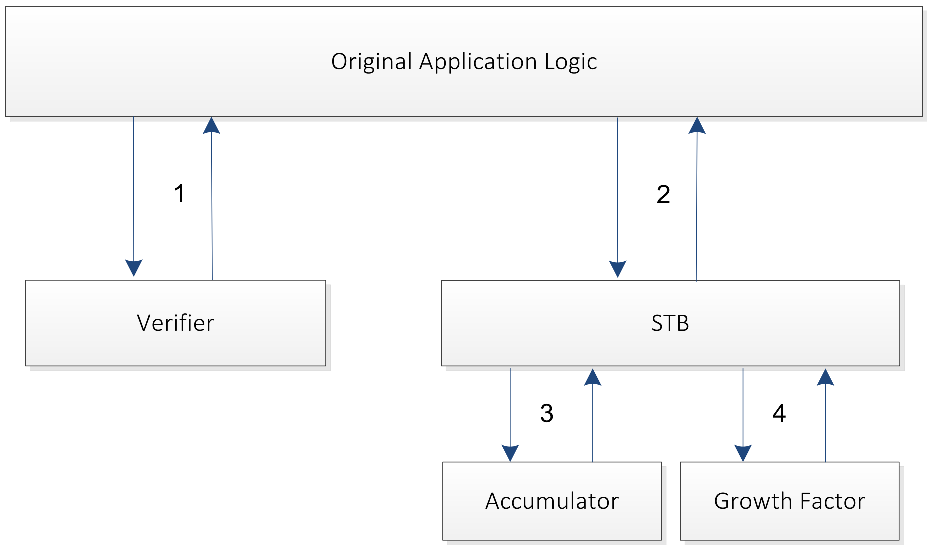

Figure 35 – STB workflow diagram for Active Operations

4.8.4.3 Active operations: Workflow details

The first operation described below is not part of STB. It is there to shown how STB is used by a anti-tampering protection technique to control the delay component of the STB.

Seq#	Operation description
1	**A Verifier is invoked.**

Details: This operation is not really part of STB. The attestation report is verified.

2	STB API is called to update the tamper detection status.

Details: Based on the verdict, any function in STB API can be called to activated and controlled the response mechanism.

Data passing:

API functions arguments have to be passed at this moment.

- `startReaction()` takes no argument.
- `stopReaction()` takes no argument.

- `restartReaction()` takes one argument: `int reaction_advancement`. It specifies which level of advancement the reaction must restart from.
- `fastenReaction()` takes one argument: `int added_speed`. It specifies how much the reaction speed must be increased.
- `slackReaction()` takes one argument: `int removed_speed`. It specifies how much the reaction speed must be reduced.

3	Accumulator’s value is modified.

Details: The function called in API masks modifications on Accumulator’s value.

4	Growth factor’s value is modified.

Details: The function called in API masks modifications on Growth Factor’s value.

Section 5 Composability

The composability of multiple protection, on the same code fragment, is of course an important aspect of ASPIRE, given the underlying principle of five lines of defence that strengthen one another.

So this section discusses to what extent the ASPIRE integrated tool flow supports compositions of individual protections, to what extent custom support is foreseen to let multiple protections work together, and where the consortium has identified synergies between protections: which protections protect each other, and which protections, when combined, provide more than 1+1 protection of the original application assets to be protected.

5.1 Composability of different protections in the ACTC

Section Authors: Bart Coppens, Bjorn De Sutter (UGent)

As a first-order approximation of how the implementation of different ASPIRE protections are composable, we investigated whether or not techniques can be applied to the same code block or not. This is an approximation of all possible compositions, because the investigation only considers the deployment of protections in the order they are actually deployed in the ASPIRE tool chain, rather than all possible theoretic orderings.

We carefully analysed each combination of protection techniques during two dedicated conference calls. Even though the conclusion from this analysis is that most combinations of protections will pose no problems, some issues were identified; these will be discussed in more detail in the following subsections.

Most of these issues are limitations of the current implementation, and are not necessarily an absolute prohibition of composability. They simply reflect that the project has limited resources, and that the often significant additional engineering that would be required to overcome the limitations has given a low priority.

By contrast, one composability issue was identified as too important, and was hence addressed immediately. With the existing infrastructure for extracting annotations from the source code and using those annotations to steer the protections, it was not possible to specify that binary protection techniques should be applied to binary code that was automatically linked-in as part of some protection techniques. This concerns, for example, the code that implements the embedded interpreter, the attestator routines, the anti-debugging component, the code mobility downloader and binder, etc. As this severely limited the useful compositions, we modified the ACTC to allow users to specify additional annotation fact files, which can contain annotations (i.e., specifications) for the binary protections that should be applied to code that was pre-compiled instead of being compiled during the invocation of the ACTC on the application to be protected.

5.1.1 *Code mobility combined with binary obfuscations*

Mobile code blocks are currently implemented as on a per-function basis. Binary obfuscations (including factoring) split up functions into smaller functions. This means that when obfuscations are applied, this results in multiple, smaller functions, which in turn results in more mobile code blocks being necessary. While this is not an issue per se, composing both techniques can affect performance negatively.

5.1.2 *Code mobility combined with the SoftVM*

Making the VM and the VM invocations mobile will pose no problem. However, making the byte code itself mobile requires data mobility in addition to code mobility. This will require some additional engineering to support.

5.1.3 Code mobility combined with anti-debugging

When control flow leaves a code block that has been protected by anti-debugging, the anti-debugging component needs to know to which address control flow should return. However, it might require quite some engineering to inform the anti-debugger component that this address is actually mobile code, and to correctly invoke the downloader/binder if required.

5.1.4 Code mobility combined with WBC

As with the combination of code mobility and the SoftVM, making the WBC tables mobile requires data mobility in addition to code mobility, and will require additional effort to complete.

5.1.5 Code mobility combined with binary attestation techniques

Mobile code is no longer present in the static part of the protected application. Rather, it is downloaded, and on each execution this code will be located in potentially different memory regions. Furthermore, not all mobile code will be present at any given point during the exaction of the protected application. This poses problems when combined with techniques that try to attest the integrity of the binary code:

1. Code guards would need to be informed that code is currently (not) downloaded, and code guards should be provided with a dynamic mapping between the attested code regions, and where these are currently loaded in memory. Furthermore, downloaded blocks cannot be diversified if their hashes are stored in the static binary.
2. Remote attestation similarly would need to keep track of which code blocks are downloaded, and where they are loaded in memory. It is definitely possible for the RA server to keep track of diversified copies of each binary, however, which makes this combination a potential candidate for actual implementation.
3. Call stack checks currently verify only if the address of the calling function resides in the protected application. Mobile code will blocks will be located in the heap, rather than in the protected application. Thus, in the current implementation, call stack checks and mobile code cannot be combined at all.

5.1.6 Code guards and remote attestation combined with the SoftVM and WBC

Currently, constant data such as the VM byte code is not yet attested. However, we foresee that this can be solved with relatively little effort, and should pose no real problem of composability.

5.1.7 CFG Tagging combined with attestation techniques, SoftVM, anti-debugging and mobile code

All of these techniques contain code fragments that are injected in the binary, such as a downloader component, a binder, the anti-debugger component, etc. It is not possible to determine up-front in which order and with what frequencies all these components will be executed. As such, they are unsuited to be combined with CFG tagging.

5.1.8 Remote attestation combined with anti-debugging

Care should be taken that the anti-debugger component can correctly deal with multiple threads, as it could be called from the asynchronous RA thread.

5.1.9 Call stack checks combined with binary obfuscations

We should ensure that the factoring transformation that is part of the set of binary obfuscations does not factor the entry block of a function that is protected with call stack checks.

5.1.10 Invariant Monitoring combined with all Diablo-implemented techniques

Invariant monitoring will require the use of debugging information to read information about the locations of variables on the stack. Currently, Diablo has no support whatsoever to track debugging information through its transformations. Thus, invariants monitoring cannot be combined at present with techniques in which Diablo has to transform the code, which currently are: binary code obfuscations, SoftVM, and anti-debugging, and code mobility.

5.1.11 Invariant Monitoring combined with Client-Server code splitting

Code fragments that have been moved to a server, can of course no longer be monitored for invariants on the client.

5.1.12 Invariant Monitoring combined with data obfuscations and WBC

Data obfuscations will transform values throughout the execution of a function. Thus, what is an invariant value in the unprotected application can be transformed by data obfuscation into different values that no longer have the same invariant. Similarly, the White-Box Crypto code might remove a fixed key, and thus also remove an invariant.

5.1.13 Multi-threaded crypto combined with client-server code splitting

It is possible that moving the multi-threaded code to the server might pose some minor issues.

5.2 Custom support for specific protection compositions

Section Author: Bjorn De Sutter (UGent

One of the basic concepts of the ASPIRE project are its five lines of defence, which are envisioned to strengthen each other. A careful investigation of a number of techniques and their composability has uncovered a number of challenges. In this section, we discuss these challenges, and the extensions of the individual techniques we propose to overcome the challenges.

5.2.1 Composability challenges

5.2.1.1 Native code mobility vs. white-box crypto data and bytecode mobility

Automatically making native code blocks from a client-side application mobile is relatively easy, even if they are targeted through indirect control flow transfers (i.e., using code pointers): All indirect transfers can be diverted to direct transfers, so-called trampolines. Making statically allocated data mobile is harder, however, because aliasing prevents compiler tools from performing precise enough data flow analysis to determine when and where all possible data pointers are used in the program, and from rewriting a program to redirect all accesses through such pointers (with acceptable overhead).

There are at least two important cases, however, in which protections can be made stronger by making data blocks mobile as well.

The first is **white-box cryptography**, of which the implementation depends on (very) large chunks of code and (very) large look-up tables in which the cryptographic keys are embedded. In order to support renewability of those tables (and of the keys embedded in them), they need to become mobile.

Secondly, in order to support renewability by means of bytecode generated for the bytecode interpreter embedded in an application protected with the **client-side code splitting**, bytecode fragments, which are data from a technical perspective, need to become mobile as well.

Clearly, the existing design of the code mobility protection will need to be revised and extended to support mobile data.

5.2.1.2 Code mobility vs. remote attestation

Mobile code is downloaded into the client application at run time and placed at randomized addresses in the client's memory space, as explained in Section 3.4. Static remote attestation, however, relies on executing code guards in the client on known code, i.e., code at known locations. This prevents that the basic static remote attestation techniques and offline code guards protect the integrity of mobile code. As this typically concerns security-sensitive code (otherwise it would not be made mobile), this is a serious issue.

Moreover, the binding performed for mobile code needs to have its integrity checked as well. This binding is performed by means of mutable data (tables of code pointers) which are vulnerable to tampering, so purely static code guards do not suffice, and special care is needed instead.

Finally, the static remote attestation as foreseen in Section 4 does not yet check the integrity of immutable data (i.e., data in read-only sections). Also for that, specialized extensions are needed. Those extensions need to cover mobile byte code was well.

Clearly, remote attestation will need to be customized to check the integrity of mobile code, of mobile data (incl. mobile bytecode), and of the client-side components implementing code mobility.

5.2.1.3 Code mobility vs. client-side code splitting

As already stated, downloaded mobile code is placed at random memory locations. This conflicts with some implementation aspects of the client-side code splitting approach. To obfuscate the flow of control into and out of bytecode fragments, the native continuation points of the bytecode (i.e., the points in native code where execution continues after a bytecode fragment has been interpreted) are hardcoded (in a position-independent manner) in the bytecode itself, in a custom manner not easily interpretable by attackers. Such hardcoded addresses are of course not possible for mobile code.

This limitation is problematic, because both mobile code and client-side splitting are supposed to be applied on security-sensitive code fragments, and because it is hence likely useful to make them more composable.

We hence need to design a method to make the two techniques composable. At the very least, we need to make sure that any implementations of code mobility and of client-side code splitting resolve all conflicts correctly, albeit possibly by separating the fragments on which the techniques are applied.

5.2.2 Solutions

5.2.2.1 Mobile data blocks (incl. mobile bytecode)

In addition to mobile code blocks, we will let our Code Mobility protection (as described in Section 3.4) support mobile data blocks. A couple of restrictions will apply, however:

- **Each mobile code block has to correspond to exactly one object file data section**, i.e., to one statically allocated variable. All statically allocated arrays in C can be allocated into separate sections by invoking the compiler (such as LLVM or GCC) with the -fdata-sections flag. So this restriction will in general pose no problems. We checked that it definitely does not pose any problem for bytecode chunks generated as part of the client-side code splitting and for the lookup-tables of the WBC functions.
- **Those object file sections should only be referenced from within the code section**, not from within other data sections. For bytecode chunks, this holds by construction, as the address of the bytecode chunk is produced only in the native code chunk that invokes the embedded interpreter to executed the bytecode chunk. For the WBC lookup tables generated in the ASPIRE compiler tool chain, this property also holds.
- Mobile code blocks will only be supported for position-independent code. This requirement is met in all ASPIRE use cases, is by definition met in all dynamically linked libraries, and is also met in all so-called position-independent executable (PIE). Given that recent version of LLVM and GCC can generate PIE binaries at a very low overhead, this limitation imposes no significant burden.

The run time behavior of an application with mobile data blocks will be very similar to the case of mobile code blocks as documented in Section 3.4.5. Every reference to the object sections that have become mobile (i.e., every position-independent production of an address in those object sections) will be replaced by

- a lookup in a statically allocated table maintained by the Binder to check for the presence of the downloaded block;
- a check whether or not the loaded address is a valid address;
- if so, simply continue executing;
- if not, instead invoke the Binder (and the Downloader) to download the block and to fill in the table, and continue executing with the new address determined by the Binder.

5.2.2.2 Custom remote attestation

The static remote attestation framework in ASPIRE already includes a large set of different code guard functions (i.e., attestation functions and verifier functions). The current ones can only scan code at fixed offsets in the binary/library code segment.

We will in addition develop data guards that can also scan read-only data sections to check their integrity.

Moreover, we will develop custom attestation and verification routines that interact with the bookkeeping tables of the Code Mobility Binder to check the integrity of those tables and of mobile code and data blocks. In year 3 of the project, we will study how the Binder can introduce redundant data that allows for stronger integrity checks, and we will study what is the best ways to perform the integrity checks themselves. If necessary, we will also let the RA server and the Code Mobility server communicate directly such that the server obtains additional information about the state of the Binder, to allow for stronger integrity checks.

These custom routines and the necessary support in the Binder, as well as the potential support on the server side will of course tie the remote attestation implementation in ASPIRE to the implementation of remote attestation. This can be considered a violation of the plugin-based design that was envisioned for the ASPIRE Compiler Tool Chain. This is inevitable, however, to obtain that 1 + 1 > 2 in terms of protection, which is also a major goal of ASPIRE. So in this case, we prioritize the protection strength over the flexibility of the tool support.

5.2.2.3 Composing client-side splitting with code mobility

To make the client-sidecode splitting compose with code mobility, we will not extend any of the reference architecture components. Instead, we impose the restriction that bytecode fragments can only feature continuation points in non-mobile code blocks.

To support cases where a bytecode fragment still is followed directly by a mobile code fragment, as occurs when an outer, larger code fragment in the application to be protected is marked to made mobile, and a nested, smaller fragment in it is marked to become (mobile) bytecode, we will introduce trampolines in the code. The original outer fragment will be split into multiple smaller ones, each of which has a single entry point corresponding to one of the continuation points of the bytecode fragment. Transfers from one of the split fragments to another will then be diverted via an injected trampoline, and that trampoline will remain static. As such, all continuation points of bytecode fragments extracted from mobile code fragments will still be located in the static code sections of the binary/library.

5.3 Server-generated bytecode

Section Author: Andreas Weber (SFNT)

SafeNet will investigate the feasibility of an alternative code generation route for their SoftVM that starts from C source instead of ARM binary code. A route starting from source code would enable the ASPIRE security server to easily generate and package additional code modules that can be sent to the client and run inside the process of the ASPIRE protected component. This could be used to strengthen other protections such as remote attestation by providing the server with the capability to dynamically generate and deploy customized data collection and response modules.

SafeNet will also investigate the interworking between the ASPIRE protected component and the separately compiled bytecode. The result should be a design that describes how the ASPIRE protected component can invoke a separately compiled bytecode module and how the bytecode can access resources (code/data) of its surrounding process.

5.4 Synergies of protections

Section Author: Cataldo Basile (POLITO)

The most important aspect of using multiple lines of defence is to achieve better levels of protection. We often indicated this case with the formula 1+1>2. In the ASPIRE project, we have developed techniques that used in combination present synergies that improve the overall level of protection.

As a first level of approximation, one protection is suitable to protect some security properties of application assets if it makes it more difficult (or costly) for an attacker to violate these security properties compared to the vanilla application.

Two or more protections join forces to improve the overall security if the effort (or cost) needed to achieve the attacker goals is more than the effort (or cost) to remove the two protections in isolation. This scenario happens every time a technique renders the attacks more difficult to violate the security properties that the other projection aims at preserving. In other words, the risk mitigation obtained with the two techniques in place is much better than the individual risk mitigations.

By analysing data collected from the protection owners (see deliverable D5.07 for more details on the questionnaires), we have identified several cases where synergies manifest:

- When a technique blocks or renders useless tools and techniques that are not (or only partly) blocked or addressed by the other protection techniques;
- When a technique blocks or renders useless tools and techniques that are used to remove or circumvent the other protection techniques;
- When a technique blocks specific classes of attacks (e.g., dynamic analysis or static analysis), that are used to detect and defeat some techniques;
- When a technique blocks specific classes of attacks that are not (or only partly) covered by the other protection techniques;
- When a technique detects and reacts to complementary attacks against the assets protected by another protection technique;
- When a technique detects and reacts to attacks against another protection.

Given the previous considerations, for the synergy analysis it is not important to assume that the protections are deployed on the same asset. For instance, anti-debugging renders dynamic analysis at application level harder, not only on the piece of software that will be actually protected.

The same analysis allowed us to abstract when synergies are expected in a more practical way:

- Some techniques insert parts that may be detected with some analysis
- Some techniques have peculiar behaviours that can be identified with some analysis
- Some techniques rely on parts, added to the application, that if modified, do not work as expected (or a rendered useless)

From this synergy analysis we have also modelled, for the sake of more effective ADSS, that in several cases some protection techniques can be deployed to strengthen some of the techniques. That is, protections can be used not only to protect application assets. For instance, remote attestation can be deployed to check whether renewable white box cryptography code, even if no assets require to protect integrity.

Table 3 summarizes the synergies among ASPIRE protection techniques. The Scope column explains if the technique must be deployed on the same asset (e.g., same variable of piece of code), if their impact is positive if they are deployed on the same application, and if the protection techniques are helpful to protect the code added by the other protection.

Table 3 – Summary of the synergies between ASPIRE protections.

Protection	Synergy with	Scope	Motivation
Anti-Cloning	None	---	Since this technique stores on the client data that are valid only between two invocations of the application, it cannot be strengthened by other techniques nor it strengthen other protections.
Anti-Debugging	Code guards	On the protection code	An attacker could try to analyze and circumvent anti-debugging by modifying and/or instrumenting the anti-debugger component. Code guards protect from these attacks.
Anti-Debugging	Remote attestation + Reaction	On the protection code	An attacker could try to analyze and circumvent anti-debugging by modifying and/or instrumenting the anti-debugger component. Code guards protect from these attacks.
Client-Server code splitting	Data obfuscation	On the protection code	Barrier variables are sensitive part of the applications, even if they are not assets (i.e., they are not annotated by users). Data obfuscation can help to preserve them by making them more difficult to understand.
Binary Obfuscation	Anti-Debugging	global	Anti-Debugging protects against dynamic attacks that can be used to de-obfuscate protected code.
Call Checks	Anti-Debugging	global	Anti-Debugging prevents certain components of the anti-debugger from being executed outside of the anti-debugger component by an attacker.
CFG Tagging	Code guards	On the protection code	Integrity protections work against an attacker trying to bypass or activate artificially a gate set by the CFG tagging.
CFG Tagging	Remote attestation + Reaction	On the protection code	Integrity protections work against an attacker trying to bypass or activate artificially a gate set by the CFG tagging.

CFG Tagging	Binary Obfuscation	On the protection code	Binary Obfuscation makes more difficult to detect where tags are inserted and used.
Code Guards	Remote Attestation+ Reaction	On the protection code	Remote attestation can detect react code written to skip guards or defeat reactions and react according to a user-defined policy.
Code Guards	SoftVM	Globally	SoftVM can increase the confidentiality of the guards as it protects against dynamic attacks and tampering.
Code Guards	Anti-debugging	Globally	Anti-Debugging protects against dynamic attacks, which use debuggers, mounted to detect guards attestators and verifiers.
Code Mobility	Binary obfuscation	On the protection code	Binary obfuscation techniques that protect against static analysis can help to increase the resilience of this protection .
Code Mobility	CFG tagging	On the protection code	Integrity protections work against an attacker trying to modify the mobile code that is, by definition, sensitive code.
Code Mobility	Multi-Threaded Crypto	On the protection code	Integrity protections work against an attacker trying to modify the mobile code that is, by definition, sensitive code.
Code Mobility	Remote Attestation	On the protection code	Integrity protections work against an attacker trying to modify the mobile code that is, by definition, sensitive code.
Data Obfuscation	Anti-debugging	Globally	Anti-Debugging protects against dynamic attacks that may help to discover data modifications operated by Data Obfuscation protections.
Data Obfuscation	Code guards	Globally	Circumventing data obfuscation techniques requires modifications of the application binaries. This can be detected by integrity protection techniques.
Data Obfuscation	SoftVM	Globally	SoftVM can increase the confidentiality of the transformations operated by Data Obfuscation (xor, merge_var, rnc technique).
Data Obfuscation	Remote attestation + Reaction	Globally	Circumventing data obfuscation techniques requires modifications of the application binaries. This scenario can be detected by integrity protection techniques.

Data Obfuscation	Client-Server code splitting	Globally	Client-Server code splitting increases the effectiveness of overall Data protection by moving on the server selected variables (data_to_proc technique).
Data Obfuscation	Binary Obfuscation	On the protection code	Binary Obfuscation can increase the confidentiality of the transformations operated by Data Obfuscation (xor, merge_var, rnc technique).
Multi-Threaded Crypto	Code Mobility	Globally	Code mobility helps against static analysis that allows attackers to identify multi-threaded crypto code
Non Renewable White-Box Crypto	Binary Obfuscation	On the protection code	WBC code is automatically generated thus it has a structure that can be recognised. Binary Obfuscation helps in hiding this structure.
Non Renewable White-Box Crypto	Code Guards	On the protection code	Integrity protection can detect and react to modifications of the WBC code aimed at extracting the key or against reuse if WBC in applications different from the protected one.
Renewable White-Box Crypto	Binary Obfuscation	On the protection code	WBC code is automatically generated thus it has a structure that can be recognised. Binary Obfuscation helps in hiding this structure.
Renewable White-Box Crypto	Code Guards	On the protection code	Integrity protection can detect and react to modifications of the WBC code aimed at extracting the key or against reuse if WBC in applications different from the protected one.
Renewable White-Box Crypto	Remote Attestation	On the protection code	Remote attestation can help strengthening this technique if used to check if the renewed WBC tables have been updated correctly
SoftVM	Binary Obfuscation	On the protection code	Binary obfuscation helps in making more difficult for an attacker to detect the VM code Code layout randomization mixes application code with VM code and hence helps hiding the VM.
Software Time Bombs	Code guards	On the protection code	Code guards are helpful against an attacker trying to bypass the software time bombs code.
Software Time Bombs	Binary Obfuscation	On the protection code	Techniques to improve the confidentiality of the STB code help in making harder to defeat this kind of reaction.

Software Time Bombs	Code Mobility	On the protection code	Dynamically sending the reaction code help preventing attacks that completely defeat them.
Static RA	Code guards	On the protection code	Integrity protection can detect and react to modifications of the WBC code aimed at extracting the key or against reuse if WBC in applications different from the protected one.
Static RA	Anti-Debugging	Globally	Anti-Debugging protects against dynamic attacks that can be used to detect RA code.

Section 6 Renewability techniques

Section Author:

Paolo Falcarin, Alessandro Cabutto (UEL)

This section describes the high level design of the reference architecture related to renewability techniques, which aims at making renewable in time and in space as many as possible protection techniques.

As described in the previous section, the composability of different offline and online techniques, when feasible, come with a price in terms of additional engineering effort or run-time performance overhead.

Software diversity applied to renewability consists in producing and delivering semantically equivalent or semantically different versions of the same application in order to obtain various benefits in terms of protection:

- A patch able to successfully circumvent a certain protection technique on a given application instance cannot be applied with no effort on a diversified instance of that application.
- A patch that used to work on an application instance at a specific time will not work later.

The introduction of software diversity (renewability in space) and renewability in time, adds other two variables (space and time) to the above-mentioned composability constraints.

Renewability techniques will come with many challenges: the generation of diversified code, its transport to the client side, and then its integration and invocation in a manner that does not break the run-time behaviour of the protected application and does not introduce any new attack vectors.

Moreover, the main concern will be the components that will manage the diversified code blocks both at server as at client side, how they will interact and what impact they will have on other components.

Renewability will be based on the Code Mobility technique that has been developed in Task 3.1 and described in deliverables D3.02, D3.03, and D3.04. This technique can already download code blocks on-demand and install them on the client at run-time, but it is currently limited to code blocks that have been extracted from the original binary application. It would need to be adapted to work also with diversified code blocks and to transfer data blocks. Figure 36 shows the extended Reference Architecture for Renewability.

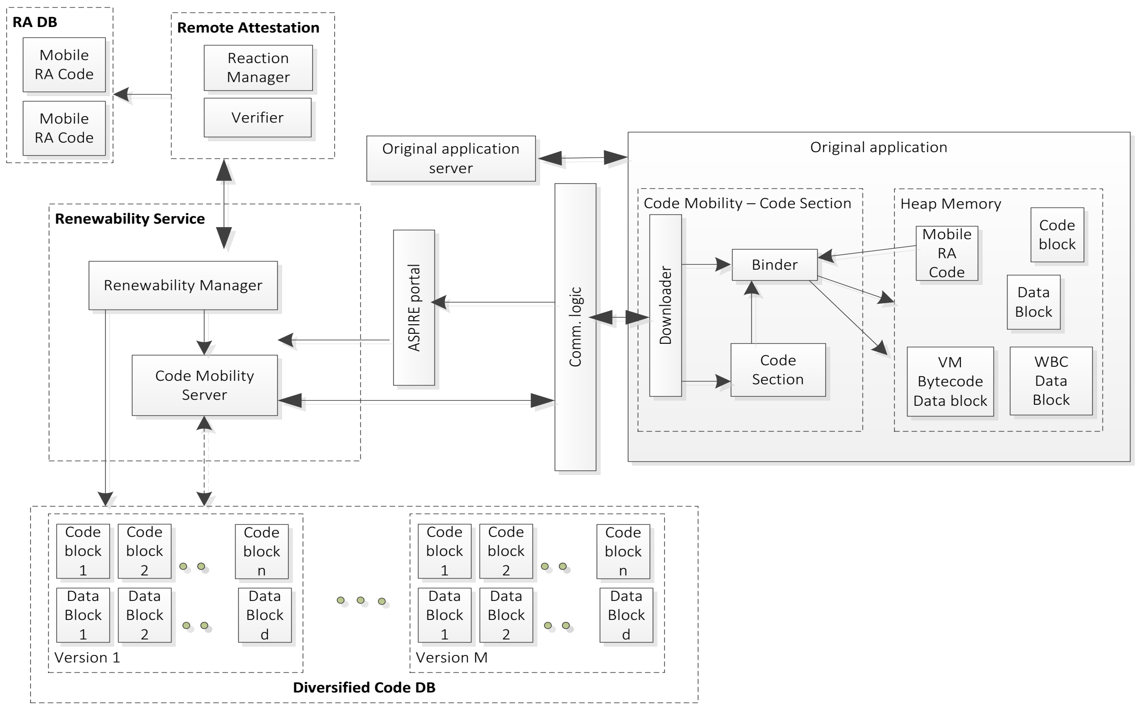

Figure 36 – Extended Reference Architecture for Renewability

The Code Mobility prototype discussed in Section 3.4 was designed to manage one version of the client application for one particular version of the client application.

In order to implement renewability, the Code Mobility Server will be extended to be able to initiate transactions using the ACCL WebSocket Protocol in order to start some tasks on the client side (e.g. code blocks deletion This capability is useful when the server decides that a certain code block has to be delivered again from the server before next use. The subsequently transferred block can then be a diversified version of the initial one taken from the Diversified Code DB shown in Figure 36.

Renewability of the client application is also achieved by means of the Data Mobility extension described in Section 5.2.2.1 that allows SoftVM bytecode renewability (see Section 5.3) and WBC data table run-time replacing.

The Renewability Manager component is now introduced to manage the different lifecycles of the various client applications and to orchestrate the renewability schedule of each mobile block depending on different and configurable renewability strategies:

1. The **Basic renewability strategy** will split the original code in N code blocks and then diversify each code block in M semantically-equivalent versions stored in the Diversified Code DB. The Renewability Manager will set a timeout for each code block and when it will expire the Code Mobility Server will send an erase request for that code block to the client. When the code block will be requested again to the server one of the M diversified versions will be selected for delivery. In this case once the initial split in code blocks is defined it will be maintained during the application lifecycle, as diversity is then applied to the next version of each single block after it is expired; however multiple clients will have the same initial version and the renewable code blocks will be randomly chosen among the corresponding blocks in the M versions in the database.
2. The **Mobile Diversity strategy** will create M diversified binary versions and then split them in N code blocks: this is equivalent to apply diversity in order to have different versions running on different clients and each version having its own code mobility protection applied afterwards; it is important to notice that in this case different clients can have an initial diversified version with a different binary structure, thus the code

mobility split in blocks will be different in such versions as the original binary structure will be different;

3. The **Renewable Remote Attestation Strategy** will require the design of a information exchange protocol between the Renewability Manager and the Remote Attestation Service in order to create renewable remote attestators (code guards) deployed with the Code Mobility framework;
4. The **Renewable Actuator Strategy** will be designed to work with the Remote Attestation Service in order to create renewable actuators to implement reactions after tampering has been detected by an attestator: a possible implementation of this strategy would require making mobile current reaction components such as time-bombs, or delayed tamper-response code;
5. The **Renewable Data Strategy** will leverage on the extended Code Mobility framework (that will be able make data blocks mobile), to force an update of specific data whenever a timeout expires (e.g. WBC data tables).
6. The **Mobile Data Strategy** will leverage on the extended Code Mobility framework (Able to make data blocks mobile), to download data at run-time once, without any further renewability. For example this will be the case for the VM bytecode that can be made mobile (i.e. downloadable from a server) but not renewable as the bytecode is created at compile-time together with the corresponding VM implementation. This strategy can be combined with the basic code diversity of the VM implementation, but there cannot be diversified bytecode for the same VM.

In conclusion, the renewability framework goals will be slightly less ambitious than the ones in the DoW, due to the technical constraints, and the composability issues highlighted in Section 5.

Section 7 List of technique identifiers

In this section, we define the list of technique identifiers. These identifiers will be used by the ASPIRE portal to redirect messages that it received from the protected application to the correct protection service. In other words, we provide a list of server-side support components with which the ASPIRE portal can communicate.

The list is provided in Table 4, and comprises a column with the identifier (T_ID) that needs to be used by the client-side components when invoking the ACCL (see Section 2.4), the technique, and the server-side support component.

Table 4 – List of technique identifiers

T_ID	Technique	Service
10	Client-server code splitting (Section 2.3)	Code splitting service (2.3.5.1)
20	Mobile code (Section 2.4)	Code mobility service (2.4.4.1)
21	Mobile data	
30	White-box cryptography (Section 2.5)	WBS (2.5.3.1)
40	Multi-threaded cryptography (Section 2.6)	Crypto server code (2.6.4)
41	Diversified crypto (conditional)	
50	Code guards (Section 3.2)	Hash randomization (3.2.3.1)
55		Hash verification (3.2.3.2)
60	CFG tagging (Section 3.3)	Remote verifier (3.3.5.1)
70	Anti-cloning (Sect 3.5)	AC decision logic (3.5.4.1)
75		AC status logic – only for application service providers.
80	Remote attestation (Section 3.8.3)	Reaction manager (3.6.2.1)
To be defined		The verifier to be used depends on the attestators or the code guards deployed. Identifier that needs to be defined needs to be specific to the verifier.

Section 8 List of Abbreviations

Abbreviation	Meaning
3G	Third Generation of mobile communications technology
ACCL	ASPIRE Client-side Communication Logic
ADSS	ASPIRE Decision Support System
API	Application Program Interface
ASPIRE	Advanced Software Protection: Integration, Research and Exploitation
CFG	Control Flow Graph
DB	Database
DoW	Description of Work
HTTP	HyperText Transfer Protocol
MATE	Man-at-the-end
MITM	Man-in-the-middle
Mx	Month x. A reference to a specific time in the ASPIRE project. It refers to the x'th month since November 2013.
PBKDF2	A standardized Password-Based Key Derivation Function (see http://csrc.nist.gov/publications/nistpubs/800-108/sp800-108.pdf)
RA	Remote Attestation
STB	Software Time Bomb
VM	Virtual Machine
WB	White-Box
WBGC	White-Box Generated Code
WBLC	White-Box Library Client-side
WBLS	White-Box Library Server-side
WBS	White-Box Server

Bibliography

[Curl] Curl, http://curl.haxx.se

[Nginx] Nginx, http://nginx.org

[RFC_WS] Internet Engineering Task Force (IETF), "The WebSocket Protocol", RFC 6455, December 2011, http://tools.ietf.org/html/rfc6455.

[Ses04] A. Seshadri, A. Perrig, L. van Doorn, and P.K. Khosla. SWATT: SoftWare-based ATTestation for Embedded Devices. In Proceedings IEEE Symposium on Security and Privacy, 2004, pp. 272–282.

[Sha05] M. Shaneck, K. Mahadevan, V. Kher, and Y. Kim, "Remote software-based attestation for wireless sensors," in In ESAS. Springer, 2005, pp. 27–41.

[Tan06] Delayed and Controlled Failures in Tamper-Resistant Software. Gang Tan, Yuqun Chen, and Mariusz H. Jakubowski. Information Hiding, volume 4437 of Lecture Notes in Computer Science, page 216-231. Springer, (2006).

[Van05] Van Oorschot, Paul C., Anil Somayaji, and Glenn Wurster. "Hardware-assisted circumvention of self-hashing software tamper resistance." In IEEE Transactions on Dependable and Secure Computing (TDSC),, Volume 2, Issue 2, pages 82-92, 2005.

[WS] The WebSocket Protocol, IETF RFC 6455, http://tools.ietf.org/html/rfc6455, December 2011