Advanced Software Protection:
Integration, Research and Exploitation

D3.06
Remote Attestation and Server Mobile Code Report

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 / D3.04 / 1.0
WP and tasks contributing:	WP3 / Tasks 3.1, 3.2, 3.3
Due date:	Apr 2016 – M30
Actual submission date:	3 June 2016
Responsible Organization:	POLITO
Editor:	Cataldo Basile
Dissemination Level:	Public
Revision:	1.0

Abstract:
This deliverable documents the tool support and the research undertaken in WP3 at M30. The document starts describing the new version of ASPIRE client-server communication logic, and then progresses about different online code protections are documented, namely: remote attestation, reaction mechanisms, anti-cloning, and applications of code mobility to other protections. It also documents the prototypes delivered with D3.05.

Keywords:
Remote attestation, reaction mechanisms, anti-cloning, ACCL/ASCL, code mobility

Editor

Cataldo Basile (POLITO)

Contributors (ordered according to beneficiary numbers)

Bert Abrath, Bjorn De Sutter (UGent)

Cataldo Basile, Alessio Viticchie' (POLITO)

Alessandro Cabutto, Paolo Falcarin (UEL)

Andreas Weber (SFNT)

Jerome d'Annoville, Christian Cudonnec, Philippe Jutel, Paul Hariyanto (GTO)

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

Executive Summary

This deliverable reports the status at M30 on the topic of online protection techniques in WP3, with focus on Remote Attestation and Code Mobility. However, we also report progresses in the other WP3 task. Moreover, we report the prototypes delivered with the deliverable D3.05. This deliverable, together with the deliverable D5.08 allows the ASPIRE project to reach the milestone MS15.

First, Task 3.1 (Client-Server Code and Data Splitting) is reported. UEL, which developed and maintains the ASPIRE Client-side Communication Logic (ACCL) and the ASPIRE Server-side Communication Logic (ASCL), has improved the bidirectional communication based on Websocket protocol and updated the related network APIs. ASCL/ACCL are now used by all the online techniques developed in WP3.

Second, Task 3.2 (Remote attestation) is reported. POLITO reports on the updates on the Static Remote Attestation framework architecture, which is now integrated in the ASPIRE Compiler Tool Chain (ACTC) with a joint effort from POLITO and UGent. Several static attestators have been developed and described, based on the use of different algorithms for computing attestation data (random walk), hashing attestation data, accessing attestation data information (memory management), and parsing the nonces sent by the server with attestation requests. Moreover, a new feature has been developed, which allows the attestation of selected code areas when the application is launched. POLITO also reports the new version of the static RA annotations. Finally, NAGRA reports the final version of the Anti-Cloning mechanisms developed by NAGRA, including its database structure and API. Anti-Cloning is now integrated into the ACTC.

GTO reports the reaction mechanisms, which ensure that applications that fail to prove to the trusted entity their integrity are rendered unusable. Reactions are triggered by ad hoc server side components based on a reaction policy. A reaction policy are used to analyse the verdicts of current and past attestations and decide the proper reaction. Reaction are classified in a scale of nine values, from no reaction to immediate corruption of the application, with intermediate values forcing a more graceful degradation. Finally, the ASPIRE DB, already introduced in D3.04, has been extended to support reactions with new tables that describe the reaction policies and the reaction statuses.

Third, Task 3.3 (Renewability) is reported. UGent and NAGRA report the design and implementation of the WBC mobility, which required the support for making the large data structures used by WBC mobile. UGent and SFNT report the design and implementation of the SoftVM mobility, which required modifications to the binder in order to support mobile bytecode. Finally, POLITO and UGent report the investigations on alternative approaches to make static remote attestators mobile. Thus, it is proposed a set of changes to the static remote attestation and code mobility to renew static remote attestators, by sending part of the attestation code after making it mobile, and to renew the Attestation Data Structures (ADS). The planning of the next months include the release of last renewability components, the support for renewability in space of WBC and SoftVM, which have been already made mobile, the integration of renewability in Diablo, and application of renewability to the NAGRA and SFNT use cases.

The deliverable D3.05 includes eight prototypes, which have been delivered in the period M24-M30. The prototypes are Client/server Code Splitting, Code Mobility, Static Remote Attestation, Reaction, ASCL/ACCL, Anti-cloning, Mobile WBC, and Mobile SoftVM.

Table of Contents

List of Figures

List of Tables

Section 1 Introduction

Section Author:

Cataldo Basile (POLITO)

The goal of this deliverable (see GA Annex II DoW part A) is to document the updates and the tool support for the online protection techniques delivered in the ASPIRE's Work Package 3 at M30 for the Remote Attestation and Code Mobility techniques. This deliverable, together with the deliverable D5.08 allows the ASPIRE project to reach the milestone MS15.

However, we decided to use this document to report the entire WP3 progress. The following techniques are developed in WP3: Code Mobility, Client/Server Code Splitting, Remote Attestation, Anti-Cloning and Renewability. The updates presented here are in part normal progress due to the continuous improvement of protections and continuous integration in the ASPIRE Compiler Tool Chain (ACTC). However, they have been further stimulated by the preparation of the tiger team experiments.

The remainder of this deliverable reports the updates implemented to the ASPIRE Client Server Architecture, with more details on the ASPIRE Server-side Communication Logic implemented with Web Sockets (ASCL-WS) and the redefined APIs. Moreover, it presents the other online protection techniques that are now integrated into the ASPIRE Compiler Tool Chain, such as Remote attestation (detection, verification, and reactions), and Anti-cloning, while all the diversity and renewability techniques will be integrated, as expected, by the end of the project at M36. However, we report in this document updates on the effort to support mobility for the White Box crypto, for the SFNT SoftVM and of the Static Remote Attestation client-side components.

We also report here both Code mobility and Client Server Code Splitting did not require an explicit report in this deliverable, as these techniques were already fully integrated and tested at M24 and did only require very minor updates and bug fixes that do not deserve to be mentioned here.

Moreover, this deliverable also documents the deliverable D3.05 (which is of type prototype). The deliverable D3.05 includes eight prototypes, which have been delivered in the period M24-M30. The prototypes are Client/server Code Splitting, Code Mobility, Static Remote Attestation, Reaction, ASCL/ACCL, Anti-cloning, Mobile WBC, and Mobile SoftVM.

This deliverable is structured as follows. 0 introduces the new version of the ASPIRE ASCL-WS. 0 reports the updates on Remote Attestation, while Section 4 focuses on the Reaction components. 4.5.1 reports the integration into the ACTC of the Anti-cloning technique. Section 6 presents White Box Crypto mobility, SoftVM mobility, and the design of mobile and renewable static remote attestators. Finally, Section 7 lists the D3.05 prototypes and their current status.

Section 2 The ASPIRE Client-Server Architecture

Section authors:

Paolo Falcarin, Alessandro Cabutto (UEL)

This section reports on the updates on the ASPIRE Client-Server Architecture and includes the final Web Socket protocol design, which was initially drafted in the first version of D1.04 document (reference architecture) and then finalized in D1.04 v 2.0 (M24).

2.1 The ASPIRE Client/Server Communication Logic (ACCL/ASCL)

The ASPIRE Client/Server Communication Logic has been finalized following the design presented in D1.04 v2.0. Its support for client-initiated and server-initiated communication is now integrated with all the online protection techniques. The ACCL/ASCL libraries in fact powers Code Mobility (UEL), Remote Attestation (POLITO), Client-Server Code Splitting (FBK), Anti-Cloning (NAGRA) and the Reaction Manager (GTO) client-server communication features.

A previous implementation of the ACCL Web Socket Protocol was incorporated in the M24 ACTC release. In the M30 release, we improved that implementation by refining the code and its ACTC integration, and by making it ready to use out of the box.

2.1.1 ASPIRE Application ID and modifications to the ACTC integration

The ACCL functions automatically embed the ASPIRE Application ID in each request to the server but since the AID is randomly generated by the ACTC during its execution we changed the build process sequence in order to make such identifier available at compile time. When the communication logic is required by a protected application, the ACTC compiles the ASPIRE Application ID into the ACCL source code and links the whole ACCL into the final binary. Previously (in the M24 ACTC release) the ACCL library was provided as single, pre-compiled object file to be linked into the client application by the ACTC.

2.1.2 Finalization of the ACCL/ASCL Implementation

The ASCL/ASCL final implementation fully reflects the design presented in D1.04 v2.0, with some minor changes with respect to its last report in D3.04.

The Simple Protocol implementation did not require any remarkable update since that last report. On the server side, the ASCL component is provided as a single object file to be linked into protection techniques' backend services.

The Web Sockets API is simple and easy to use by technique owners. We provided a use example on the Framework SVN in the development branch. Official documentation is available on the ASPIRE project's wiki at `https://ASPIRE-fp7.eu/wiki/accl-ascl-deployment`.

In its final implementation, the API exposes 5 functions to the user:

- `asclWebSocketsInit`: initializes the channel accepting connections from clients
- `asclWebSocketsShutdown`: closes the channel, terminating server side operations
- `asclWebSocketsSend`: sends a message to a client expecting no response (non blocking behaviour)
- `asclWebSocketsExchange`: sends a message to a client expecting a response (blocking behaviour)

- `asclWebSocketsDispatcherMessage`: is a call-back function to be implemented by techniques owners that is used to dispatch messages. It is invoked by the ASCL library when one of the following events occur:
    - OPEN: a client connected to the server
    - CLOSE: a client disconnected from the server
    - SEND: a client sent a message expecting no response
    - EXCHANGE: a client sent a message expecting a response

For ease of integration the named pipes based message dispatching method presented in D3.04 Section 2.1.1 has been abandoned. The `asclWebSocketsDispatcherMessage` call-back function replaced it, which reduced the effort required to implement the API.

While the `asclWebSocketsDispatcherMessage` function implementation is mandatory, techniques owners can decide to handle just some of the available messages (i.e., the ones required by their protection architecture).

When a client connects to the server, a technique identifier and the ASPIRE Application ID are provided such that the proper server-side logic can be activated and possibly initialised.

On the client side the ACCL library provides a similar counterpart API consisting of 4 functions:

- `acclWebSocketInit`: connects to the server establishing a bidirectional channel
- `acclWebSocketShutdown`: terminates an existing connection
- `acclWebSocketSend`: sends a message to the server expecting no response
- `acclWebSocketExchange`: sends a message to the server expecting a response

The library including the final Web Sockets implementation consists of ~1.3k lines of ANSI C code.

2.1.3 ACCL components protection

During M30 a tiger team composed by hackers from NAGRA carried out an experiment consisting of a series of attacks on their own use case (see D6.01 v2.1 section 2). The use case has been protected with both state of the art off-line and on-line protection techniques. Since the ACCL library is automatically linked into the client by the ACTC when at least one on-line protection is applied to a target application it has to be protected as well. To achieve this goal the following protection scheme has been applied to every ACCL functions:

- Code Obfuscation
    - Opaque predicates insertion with application chance of 20%
    - Branch functions insertion with application chance of 25%
    - Control flow flattening with application chance of 25%
- Call Stack Checks

The successful application of these protections proved that this portion of client-side code is protectable.

2.2 Plan

The ASPIRE Client/Server Communication Logic is now finalized. It satisfies all on-line techniques owners' actual needs. Apart from possible support to partners and limited bug fixing operations no more effort is required on ASPIRE Client/Server Communication Logic maintenance.

Section 3 Remote Attestation

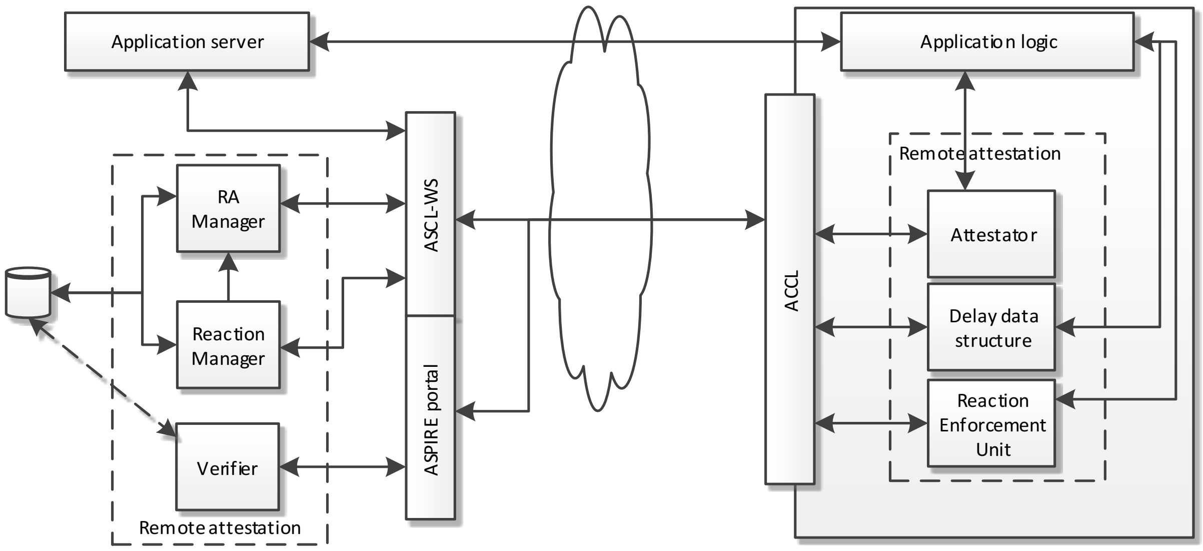
Section Authors:

Cataldo Basile, Alessio Viticchié (POLITO)

This section covers the work performed in Task T3.2. It presents the updates to the static remote attestation, originated by the need for protecting the use case applications for the tiger team experiments. Together with the details reported in the sections below, effort has been spent to support the integration of the last version of the ASCL that now better supports multiple attestators and verifiers, and to achieve a better integration into the ACTC. Finally, we sketch the planning and the expected deadlines for the next months.

3.1 Remote attestation reference architecture

To make this deliverable more readable, we report here the reference architecture of the remote attestation technique. It has not been changed since the last version but it is useful to have it here for quick references for both 0 and Section 4. However, more in-depth explanation of the components can be found in the deliverable D3.02 and D3.04, the latter containing the most up-to-date version and the description of the characteristics of the architecture in presence of multiple clients with multiple attestators.



The only component that presents an update is the ASCL-WS, which has been adapted to support the last version of the ASCL described in 0.

3.2 Static Remote Attestation

3.2.1 Annotations

There are two types of annotations to add to support the static remote attestation:

- *Definition of an attestator*, by means of annotations using as first protection parameter `static_ra`;
- *Definition of the areas to attest*, by means of annotations using as first protection parameter `static_ra_region`.

3.2.1.1 Attestator declaration (static_ra parameter)

First, it is needed to declare the use of one or more attestators by including (at any place in the application source code) remote attestation annotations having as first protection parameter `static_ra`. These annotations characterize the attestator to insert by defining every attestator configurable feature. The definition of the attestator also (implicitly) characterizes the other related RA infrastructure components (i.e., the extractor and the verifier).

In turn, the `static_ra` protections parameter accepts six parameters as follows:

1. `RW` parameter, it specifies which random walk must be performed to extract attestation data. Admissible values for this parameter are:
   - `RW_NORMAL`, the attestator will perform the exponentiation based random walk (as described in D3.02);
   - `RW_GOLDBACH`, the attestator will perform the random walk that uses the Goldbach hypothesis (as described in D3.04).
2. `HF` (hash function) parameter, it specifies which hash function must be used to generate attestation data digest. Allowed value for this parameter are:
   - `HF_BLAKE2`, it sets the hash function to Blake2;
   - `HF_MD5`, it sets the hash function to MD5;
   - `HF_SHA1`, it sets the hash function to SHA1;
   - `HF_SHA256`, it sets the hash function to SHA256;
   - `HF_RIPEMD160`, it sets the hash function to RIPEMD160.
3. `NI` (nonce interpretation) parameter, it specifies how nonces are interpreted in order to extract parameters for the random walk. Allowed values for this parameter are:
   - `NI_1,` with this value the parameters are
     - `area_label=((uint16_t) nonce[nonce_size-4])%total_monitored_areas` where `total_monitored_areas` is assumed to be equal to 2^n.
     - `buffer_size`=attested_area_size (that is, buffer_size=total extracted .
     - -bytes)
     - `actual_buffer_size=`largest prime number less than or equal to the attested area size
     - `generator=((uint32_t)nonce[0]) % actual_buffer_size`
     - `initial_offset=(uint32_t) nonce[4] % (buffer_size - actual_buffer_size)`
   - `NI_2,` with this value the parameters are
     - `area_label=((uint16_t) nonce[nonce_size-4])%total_monitored_areas` where `total_monitored_areas` is assumed to be equal to 2^n.
     - `buffer_size`=attested_area_size (buffer_size=total extracted bytes)
     - `actual_buffer_size= attested_area_size`
     - `generator`=the largest prime number less than the attested area size
     - `initial_offset=(uint32_t) nonce[4] % (buffer_size - actual_buffer_size)`
   - `NI_3`
     - `area_label=((uint16_t) nonce[nonce_size-4])%total_monitored_areas` where `total_monitored_areas` is NOT assumed to be equal to 2^n.
     - `buffer_size=attested_area_size` (buffer_size=total extracted bytes)
     - `actual_buffer_size= attested_area_size`
     - `generator` = the largest prime number less than the attested area size
     - `initial_offset=0`

- NI_4
  - `area_label=((uint16_t) nonce[nonce_size-4])%total_monitored_areas`
  - where `total_monitored_areas` is assumed to be equal to 2^n.
  - `buffer_size`=attested_area_size (buffer_size=total extracted bytes)
- `actual_buffer_size`=largest prime number less than or equal to the attested area size
- `generator`=the largest prime number less than the attested area size
- `initial_offset=0`

4. `NG` (nonce generation) parameter, which specifies how the nonces are generated. Only one implementation for nonce generation has been implemented so far. The existing implementation generates random nonces without further customization. Thus, only one value is allowed for this parameter, namely `NG_1`. As future work, it is possible to implement other nonce generation functions that encode in the nonces the information required to understand the area to attest and random walk parameters, or to generate the parameters randomly and build a nonce accordingly. They are no major updates that considerably affect the performance or security of static RA, they just create more variability in the attestators.
5. `MA` (memory area) parameter, which specifies the memory areas management API implementation that the attestator must use to access the data in the Attestation Data Structure. Only one implementation for this API has been implemented so far. Hence, only one value is allowed for this parameter, namely `MA_1`. As future work, it is possible to implement other memory area functions that map memory areas depending on the low-level layout.
6. `DS` (data structure) parameter, which specifies the RA data management API to use. For instance, this API is used to parse an attestation request and prepare an attestation response, to read and write all request and response components, to read and write RA prepared data and hashed data. Only one implementation for this API has been implemented so far, so only one value is allowed for this parameter, namely `DS_1`. As future work, it is possible to implement another data structure or to adapt to other request/response protocols.

The overall RA architecture has been designed to be modular and to work independently from the actual implementation of its components. It means that if it is needed to tailor the RA system for particular kind of hardware or low-level software architectures, it is possible to generate ad-hoc RA components that fit the system features.

The `static_ra` protection parameter also requires that every added attestator be assigned to a label. The label is the unique identifier of the attestators. The label is specified by using the `label` protection parameter, which has the following format:

```
label(name)
```

where `name` is a string of characters specified without any quote.

When the `static_ra` is used as protection parameter, one additional protection parameter can be passed through the annotation, the `frequency` protection parameter. The `frequency` protection parameter has the following format:

```
frequency(seconds)
```

where `seconds` is an integer that specifies the number of seconds between two subsequent attestation requests to be sent to the defined attestator.

An example of the described annotation is reported hereafter:

```
_Pragma("ASPIRE begin protection(remote_attestation,
static_ra(RW_NORMAL, HF_SHA256, NI_1, NG_1, MA_1, DS_1),
label(first_attestator), label(first_attestator),
frequency(100))")_Pragma("ASPIRE end")
```

The annotation below requests the inclusion of an attestator, named `first_attestator`. The server must be configured to send to this attestator one request every 100 seconds on average (`frequency(100)`). This attestator uses SHA256 as hash function (`HF_SHA256`), performs exponentiation based random walk (`RW_NORMAL`) and interprets received nonces according to nonce interpretation version 1 (`NI_1`).

As an alternative way to declare attestators, the `static_ra` protection parameter has been overloaded to accept just a parameter in the form:

```
    static_ra(id)
```

where `id` is an integer that represents a combination of the six parameters described before. The possible values and related meanings are reported in Table 1.

Table 1. Attestator IDs

	Parameters					
id	**RW**	**HF**	**NI**	**NG**	**MA**	**DS**
1	RW_NORMAL	HF_BLAKE2	NI_1	NG_1	MA_1	DS_1
2	RW_NORMAL	HF_BLAKE2	NI_2	NG_1	MA_1	DS_1
3	RW_NORMAL	HF_BLAKE2	NI_3	NG_1	MA_1	DS_1
4	RW_NORMAL	HF_BLAKE2	NI_4	NG_1	MA_1	DS_1
5	RW_NORMAL	HF_MD5	NI_1	NG_1	MA_1	DS_1
6	RW_NORMAL	HF_MD5	NI_2	NG_1	MA_1	DS_1
7	RW_NORMAL	HF_MD5	NI_3	NG_1	MA_1	DS_1
8	RW_NORMAL	HF_MD5	NI_4	NG_1	MA_1	DS_1
9	RW_NORMAL	HF_SHA1	NI_1	NG_1	MA_1	DS_1
10	RW_NORMAL	HF_SHA1	NI_2	NG_1	MA_1	DS_1
11	RW_NORMAL	HF_SHA1	NI_3	NG_1	MA_1	DS_1
12	RW_NORMAL	HF_SHA1	NI_4	NG_1	MA_1	DS_1
13	RW_NORMAL	HF_SHA256	NI_1	NG_1	MA_1	DS_1
14	RW_NORMAL	HF_SHA256	NI_2	NG_1	MA_1	DS_1
15	RW_NORMAL	HF_SHA256	NI_3	NG_1	MA_1	DS_1
16	RW_NORMAL	HF_SHA256	NI_4	NG_1	MA_1	DS_1
17	RW_NORMAL	HF_RIPEMD160	NI_1	NG_1	MA_1	DS_1
18	RW_NORMAL	HF_RIPEMD160	NI_2	NG_1	MA_1	DS_1
19	RW_NORMAL	HF_RIPEMD160	NI_3	NG_1	MA_1	DS_1
20	RW_NORMAL	HF_RIPEMD160	NI_4	NG_1	MA_1	DS_1
21	RW_GOLDBACH	HF_BLAKE2	NI_1	NG_1	MA_1	DS_1
22	RW_GOLDBACH	HF_BLAKE2	NI_2	NG_1	MA_1	DS_1
23	RW_GOLDBACH	HF_BLAKE2	NI_3	NG_1	MA_1	DS_1
24	RW_GOLDBACH	HF_BLAKE2	NI_4	NG_1	MA_1	DS_1
25	RW_GOLDBACH	HF_MD5	NI_1	NG_1	MA_1	DS_1
26	RW_GOLDBACH	HF_MD5	NI_2	NG_1	MA_1	DS_1
27	RW_GOLDBACH	HF_MD5	NI_3	NG_1	MA_1	DS_1
28	RW_GOLDBACH	HF_MD5	NI_4	NG_1	MA_1	DS_1
29	RW_GOLDBACH	HF_SHA1	NI_1	NG_1	MA_1	DS_1
30	RW_GOLDBACH	HF_SHA1	NI_2	NG_1	MA_1	DS_1
31	RW_GOLDBACH	HF_SHA1	NI_3	NG_1	MA_1	DS_1
32	RW_GOLDBACH	HF_SHA1	NI_4	NG_1	MA_1	DS_1
33	RW_GOLDBACH	HF_SHA256	NI_1	NG_1	MA_1	DS_1

34	RW_GOLDBACH	HF_SHA256	NI_2	NG_1	MA_1	DS_1
35	RW_GOLDBACH	HF_SHA256	NI_3	NG_1	MA_1	DS_1
36	RW_GOLDBACH	HF_SHA256	NI_4	NG_1	MA_1	DS_1
37	RW_GOLDBACH	HF_RIPEMD160	NI_1	NG_1	MA_1	DS_1
38	RW_GOLDBACH	HF_RIPEMD160	NI_2	NG_1	MA_1	DS_1
39	RW_GOLDBACH	HF_RIPEMD160	NI_3	NG_1	MA_1	DS_1
40	RW_GOLDBACH	HF_RIPEMD160	NI_4	NG_1	MA_1	DS_1

3.2.1.2 Declaration of the areas to attest (`static_ra_region` parameter)

An area will be attested if it is enclosed within a remote attestation annotation that specifies, as first protection parameter, the `static_ra_region` parameter. The annotation must specify the attestator (among the defined ones) that will monitor the code region that is being protected. The attestator reference is specified by the attestator protection parameter that has the following format:

```
attestator(label)
```

where `label` is a non-quoted string of characters. If the specified `label` is not defined among the defined attestators the application of the static RA protection fails.

It is possible to specify that the attested area must be attested as soon as the application starts by using the `attest_at_startup` parameter. This protection parameter accepts a Boolean parameter that specifies whether the region must be attested or not. The parameter has the format:

```
attest_at_startup(bool)
```

where `bool` is either the `true` or `false` non-quoted string.

The `attest_at_startup` parameter is optional and, if omitted or set to false, the region is not attested at start up.

An example of code region protection using a static remote attestation annotation is reported below:

```
_Pragma("ASPIRE begin protection(remote_attestation,
static_ra_region, attestator(first_attestator),
attest_at_startup(true)))

/* code to attest */

Pragma("ASPIRE end")
```

This annotation defines an area to be protected by the static remote attestator named `first_attestator`, the corresponding area is mandatory attested when the client connects to the server (`attest_at_startup(true)`).

It is worth noting that after first attestation, the areas marked with `attest_at_startup` are randomly selected among the areas to attest, with the same probability of all the other `attest_at_startup(false)` (or all the annotation where this field is omitted).

The areas that need to be attested at start-up require a proper preparation. Indeed, since the areas to attest are determined by random nonces, it is necessary a preliminary offline phase to select the nonces that will force the attestation of these areas. This operation is performed by the Extractor that has been modified for this purpose.

Practically, when performing the preparation of attestation data, the Extractor associates the randomly generated nonces with the areas to attest and computes (by emulating the extraction on the binaries) the attestation data. Everything is stored on the ra_prepared_data table in the ASPIRE DB. When computing the Attestation Data Structure, Diablo also outputs all the IDs of the areas that need to be attested at start up. The IDs of areas to attest are processed by the static RA deployment tool, which is integrated in the ACTC, and inserted in the DB. The RA Manager, when a new client connects, checks in the DB for the presence of areas to attest at start up, reads all their IDs, then it queries the DB to obtain nonces that will force the attestation of those areas.

3.2.2 ASPIRE Database tables for static RA

The ASPIRE DB includes the following tables to manage remote attestation:

- **ra_prepared_data**, used to store the association between nonces and pre-computed attestation data, already presented in details in the deliverable D3.04. In this table, a column has been added to store the id of the memory area to which each record is associated. The new column is described below.

Column name	MySQL type	C type	Description
`memory_area`	smallint(5)	`Uint16_t`	Memory are id assigned in the ADS

- **ra_request**, used to store the all the information related to attestation requests (time, response, verification results), already presented in details in the deliverable D3.04, no changes since then.
- **ra_reaction**, which reports the overall status of clients as established by the server-side reaction logic. The table is reported below.

Column name	MySQL type	C type	Description
`id`	Bigint(20)	`uint64_t`	Record id and primary key
`application_id`	bigint(20)	`uint64_t`	Foreign key that refers to application record in ra_application table
`reaction_status_id`	bigint(20)	`uint64_t`	Foreign key that refers to ra_reaction_status the associates status in the ra_reaction_statuses table

- **ra_reaction_status**, which reports the possible values for the overall application status. The table is reported below.

Column name	MySQL type	C type	Description
`id`	bigint(20)	`uint64_t`	Record id and primary key
`name`	varchar(32)	`char[32]`	Enumerative name of the status
`description`	varchar(50)	`char[50]`	Optional textual description of the status value

- **ra_attest_at_startup_area**, which stores the label of the memory areas that must be attested as soon applications start. The table is reported below.

Column name	MySQL type	C type	Description
`id`	bigint(20)	`uint64_t`	Record id and primary key
`attestator_id`	bigint(20)	`uint64_t`	Foreign key that refers to the attestator record
`memory_area`	smallint(5)	`uint64_t`	The label of the memory area to attest at startup

- **ra_request**, stores the remote attestation transactions statuses, already presented in details in the deliverable D3.04. Since then, a column has been added in order to specify if the tracked attestation has been performed normally or at application startup. The request added column is described here.

Column name	MySQL type	C type	Description
`is_startup`	tinyint(1)	`bool`	True if the attestation request has been sent at startup, false otherwise.

3.3 Reactive attestation

Attestation determines if a client has been modified, however, without proper reaction, it is pointless. Client-side reactions will be presented in Section 4 together with the infrastructure to decide and enforce reactions. Reactive attestation is reported here, as it will not be managed by means of the standard reaction architecture.

We have designed a method, alternative to the reaction mechanisms that will be presented in Section 4, to react in case of compromised applications based on the cooperation between remote attestation and client-server code splitting. The idea is to make an application dependent on the server in order to have at our disposal the easiest of the reactions: stopping to serve compromised applications.

The application of this technique is based on this workflow:

- *Obtain profile information*, traces are collected in order to obtain data useful to assess performance overhead in case of particular slices being removed.
- *Decide what to split*, based on the profile information and on the annotated assets, a decision process determines the slices that are to be moved to the server.
- *Annotate*, add annotations in the application source code to mark the slices to be moved on the server, based on the decision process and on the performance constraints.
- *Execute the ACTC*, which will execute first the client-server code splitting then the remote attestation component to attest modifications and react to compromised applications.

Notice that the use of the client-server code splitting as a reaction mechanism is different from when it is used as an independent technique. When client-server code splitting is used as a protection, it starts from the annotated assets, it determines the actual slice to be moved to the server from performance and security considerations. That is, there is not a lot of freedom in the decision of the slices to move as the assets and other important related parts must be necessarily moved. This also poses challenges from the performance point of view. If the parts to move are used very frequently and if they are computationally demanding, the risk is that the performance degradation due to network and server overhead may render this protection unusable.

Currently, reactive attestation can be used in practice as both remote attestation and client-server code splitting are integrated into the ACTC. However, this kind of reaction is not yet covered by the ADSS and the decision process that determines the areas to move to the server to achieve server dependency is not integrated in the ACTC tool flow.

Reactive attestation is currently described in a paper that will soon be submitted at the SPRO 2016 Workshop and will be documented in more details in the deliverable D3.09.

3.4 Plan

The effort for static remote attestation can be considered completed and finalized, even if the protection of the GTO use case will require some minor effort.

Dynamic attestation is in the debugging phase, we expect to complete the integration into the ACTC by M33. It will be reported in D3.09.

Implicit dynamic attestation will be completed and optionally integrated in the ACTC by the end of the project. It will be reported in D3.09.

Reactive attestation will be released at M33. It will be reported in D3.09.

Section 4 Reaction

Section Authors:

Christian Cudonnec, Philippe Jutel, Paul Hariyanto (GTO)

The Reaction mechanism has been introduced in the D1.04-Reference Architecture document – sections 4.1.2 and 4.8 – and described with more details in D3.04-Intermediate Online Protections Report, in Section 5.2.

4.1 Reaction architecture

The actual reaction mechanism that degrades the application is located in the *Reaction Enforcement Unit.* This *Reaction Enforcement Unit* might be invoked either locally or remotely. Offline protections can trigger locally the *Reaction Enforcement Unit* by setting the adequate data in *Delay Data Structures*. Online protections can set fields in the Database on the ASPIRE server to report that tampering has been detected based on the protections' criteria. The *Reaction Manager* is regularly querying the database and tracks those fields to take decisions. According to reaction policies, the *Reaction Manager* can send notifications to the device. The *Reaction Waiting Unit* is listening for notifications sent by the *Reaction Manager* and sets adequate data in *Delay Data Structures.* This settings of the *Delay Data Structure* triggers the *Reaction Enforcement Unit*. Figure 1 shows all the components involved in the Reaction mechanism.

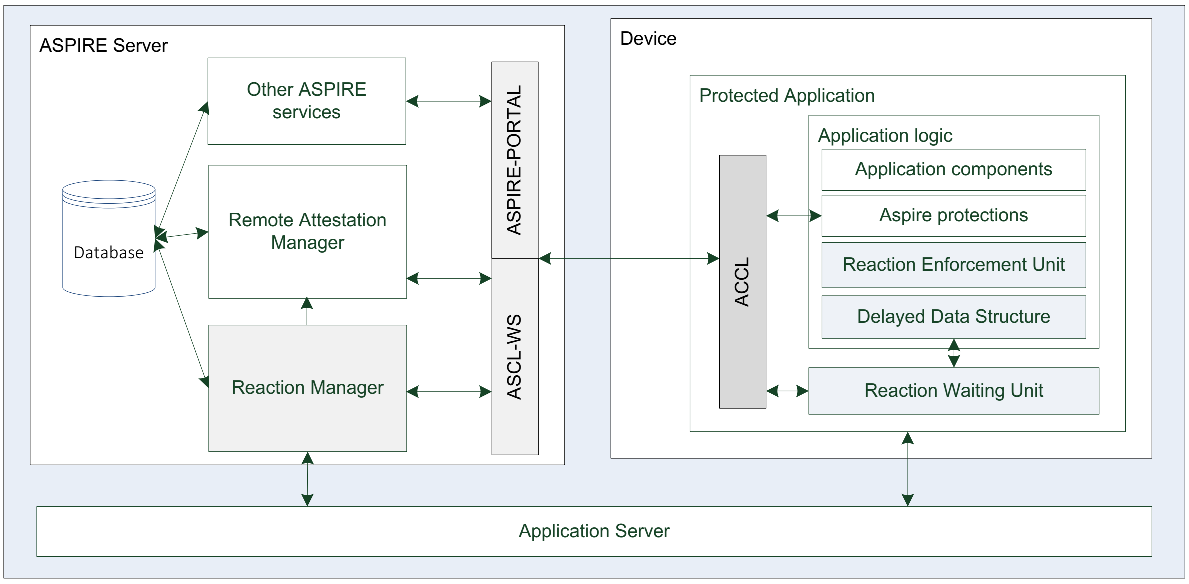


Figure 1 – Reaction Architecture

4.2 Reaction Annotations

The reaction mechanism is invoked by means of annotations set in the source code of the application and reaction code is inserted based on the location of these reaction annotations.

Another option would be to automatically deduce from the Control Flow Graph where to insert the reaction code and would relieve the application developer from setting these extra annotations considering he must already set many ASPIRE annotations to protect its code. This option has been rejected mainly because of the implementation cost. To insert the code at the best place in the application, there is a need to run the application in a monitoring mode before any insertion of code in order to detect and register the insertion points of the reaction code in the application. This monitoring mode would require too much engineering effort and a simpler option has been taken by using annotations. However, the automated approach is interesting from the exploitation point of view.

The reaction mechanism needs several annotations. Some updates have been done compared to the specification given in the annex of D5.02 and the updated description is in the D5.08 Online Protection Framework document.

4.3 Device Reaction mechanism

The implementation of the Reaction mechanism in the application is based on two main components: the Reaction Waiting Unit (RWU) and the Reaction Enforcement Unit (REU) as shown in Figure 2.

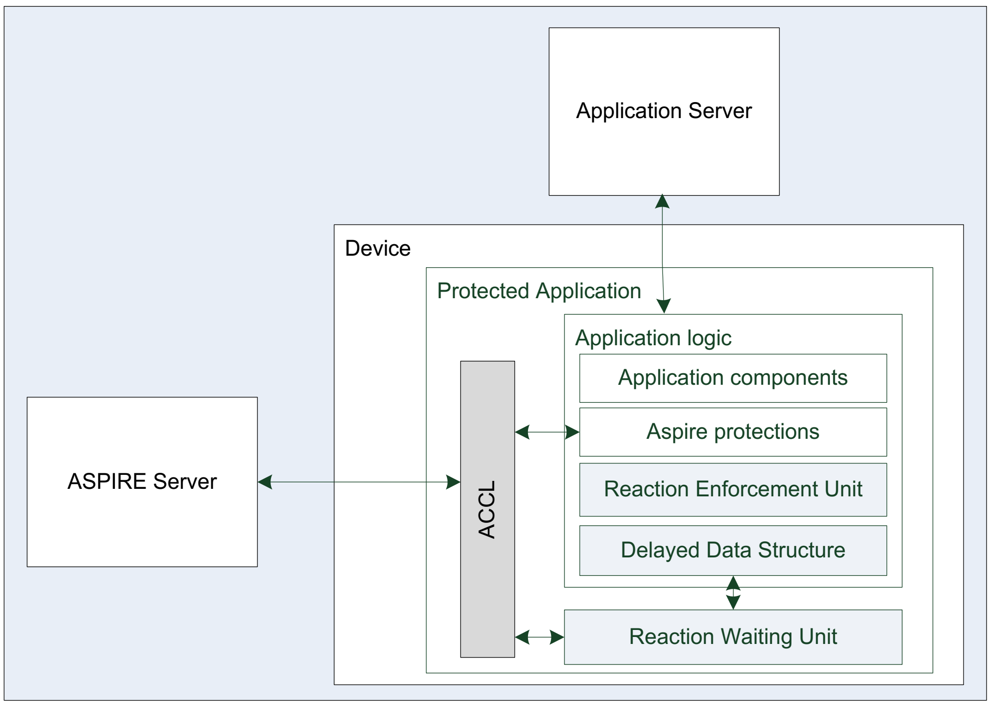


Figure 2 – Reaction components in the application

As explained in the Reference Architecture, the RWU collects the notifications coming in from the Reaction Manager located on the server. These notifications trigger reaction actions by setting adequate data in the Delay Data Structures (DDS). The REU modifies the application according to the tampering severity level set by the RMU in the DDS. These modifications are extra code that can slightly alter the application to degrade performance, break it over time, or provoke an immediate exception when the maximum severity level is set.

4.3.1 Reaction Waiting Unit

As explained in Section 5.2.2.1 of the deliverable D3.04 – Intermediate Online Protections Report, the purpose of the RWU is to listen for reactions notifications sent by the Reaction Manager located on the server side and to set data in Delay Data Structures.

The RWU cannot run in the main thread of the application because it would miss notifications, thus a specific thread has to be launched. RWU is launched from the application based on an initialization reaction notification. This notification is replaced by a call to the following function:

```
void reactionUnitInitialization (void);
```

This call launches the RWU thread and an initial Web Socket request is sent to the server to establish the connection. Then RWU listens for notifications sent by the Reaction Manager and processes those notifications when received. The RWU has been designed in a relatively simple way to enforce multiple reactions depending on a tampering severity level.

The DDS value is set with a call to the following function:

```
void reactionUnitSyncNotification (
      int nTechniqueID,              // The technique ID
      bool fHasBeenTampered,         // The tampering status flag
      int nResponseLevel             // The tampering severity level
      );
```

The technique ID is the identifier of the protection technique verifier sending the notification. When called by RWU, the Reaction Manager sets this field. The tampering status flag indicates if a tampering action has been detected. Depending on this flag, the RMU could start, stop, or restart the reactions. The Tampering Severity Level value ranges from 0 to 8; the semantic is explained in Section 4.3.3 Reaction Enforcement Unit.

As already mentioned in the D3.04 document, the RWU is the weakest component of the current implementation of the reaction mechanism regarding the security and its capacity to resist to attacks. An attacker who understands the design and is able to stop the listening thread by any means would break the reaction. Solutions are either to merge the RWU with some application service to make it difficult to stop without breaking the application or by using positive reaction as described in Section 4.1.2.3 of the reference architecture.

4.3.2 Delayed Data Structure

The Delay Data Structures (DDS) have been reduced to a plain structure set by a synchronous interface. Some more sophisticated data structures such as described in the ASPIRE Reference Architecture document section 4.7.1.2 may be implemented if the project resources allow so.

The `reactionUnitSyncNotification` introduced in Section 4.3.1 called by the RWU can also be called by any offline verifier to set the DDS. The Control Flow Tagging verifier will use this function to trigger the reaction when required.

4.3.3 Reaction Enforcement Unit

The REU component is made of pieces of code inserted in the source code at the location specified by an annotation. This component implements actions in response to an altered execution of the program. These actions depend on the tampering severity level notified to the REU by the protection technique verifiers. The current implementation is described in **Error! Reference source not found.**.

Other implementation options are however possible to respond gradually to tampering attempts

Table 1 – Reaction enforcement unit actions

Tampering Severity Level	Comment
0	The REU will drop the reaction notification; it is not to be considered. Notifications with this severity level are extra notifications sent by the

	Reaction Manager to fool the attacker.
1 to 4	The reaction enforcement unit slows down the application only. Based on the time base given as a parameter of the annotation described in Section 4.2.3, the REU slows down the application. The delay is gradually increased on the tampering severity level. Therefore, the application becomes less and less responsive.
5 to 7	One or more annotated variables are altered after the delay mentioned above.
8	This is the highest level. A critical attack has been detected and it is required to stop the application. A signal is raised simulating a memory corruption (Segmentation fault).

The *Software Time Bomb* reaction mechanism mentioned in the DoW and in various ASPIRE documents is a reaction action with level 5 to 7.

One of these actions is to alter the content of specified application variables. These variables are marked using a dedicated attribute annotation. ACTC generates a call to the `alterVariable` function based on this annotation. This function enables to register the applications variables that might be altered by the REU.

```
void alterVariable (
      void* pVar,               // The variable address
      int nVarSize              // The size of the variable
      char* szCodeID            // The code ID of the enforcement unit
      );
```

The *szCodeID* is the identifier of the piece of code of the enforcement unit that alters the variable.

Another annotation indicates where reactions can be triggered. This annotation is replaced by a call to the `applyReactionEnforcement` function.

```
void applyReactionEnforcement (
      char* szCodeID      // The fixedcode ID of the enforcement unit
      long lTimeBase      // The time base in ms
);
```

The time base parameter is the one given as a parameter of the annotation. This is the base of the computation of the delay to slow down the application provided the Reaction mechanism has been activated or not according to the tampering severity level updated by the RWU.

4.4 Online Reaction mechanism

4.4.1 Architecture

The Reaction Manager (RM) follows the design presented in the Reference Architecture. **Error! Reference source not found.** details the two queues managed by the RM.

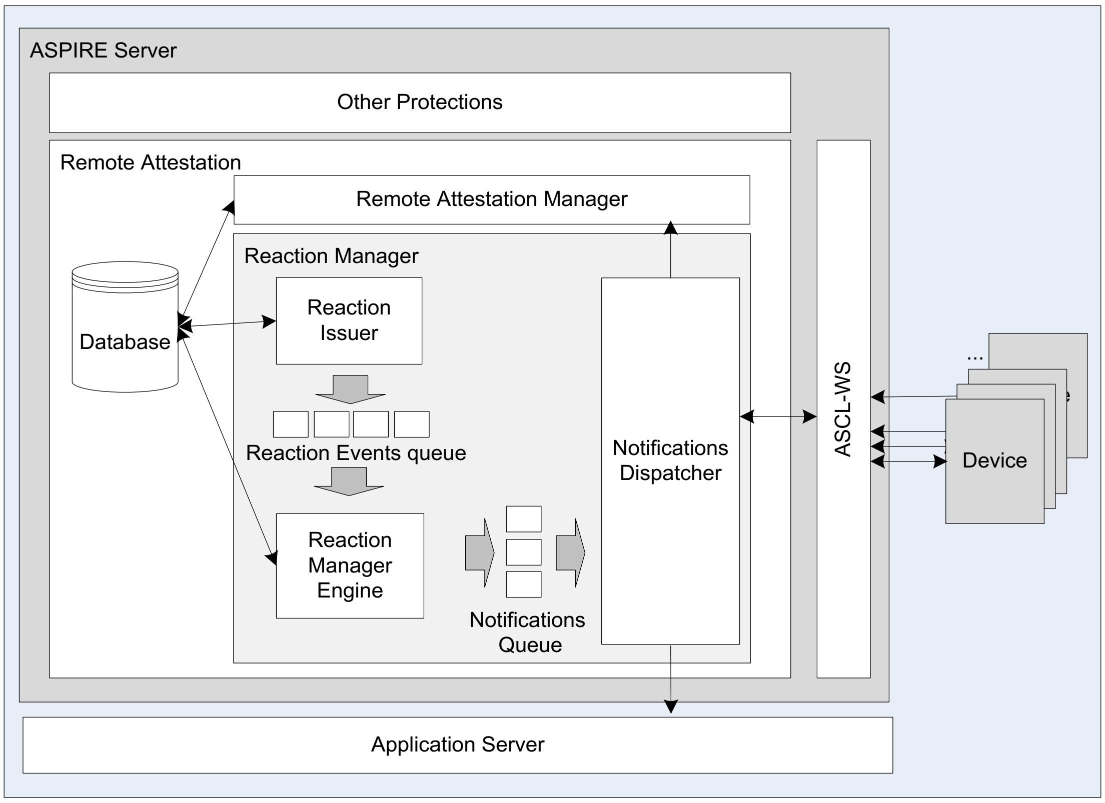


Figure 3 – Online Reaction Manager Architecture

The role of the *Reaction Issuer* is to query the database and to test the fields set by the Remote Attestation. Because there might be a large number of records to check in the database, the Reaction Manager runs in its own thread.

The *Reaction Manager Engine* applies the reaction logic expressed through the Reaction Policies. The split between the Reaction Issuer and the *Reaction Manager Engine* is to maintain acceptable performance but these two internal components are the actual implementation of the *Reaction Manager*. The Reaction is just there to optimize the *Reaction Manager Engine*.

The *Notification Dispatcher* off-loads the *Reaction Manager Engine* onto the Dispatcher that acts as a satellite service that routes notifications to either the Remote Attestation Manager, the Application Server or the device. So far, notifications are sent to the device only. It is not planned to extend the component to send notifications to the application server during the project lifetime. However, it is a possible option that can be easily supported by the Notification Dispatcher. Notifications will be sent to the Remote Attestation Manager when the corresponding API will be fully validated.

4.4.2 Reaction Manager Logic

The Reaction Manager logic is implemented in the Reaction Management Engine based on rules expressed in the Reaction Policies. Reaction Policies are configuration files given as input to the Reaction Management Engine (see Sections **Error! Reference source not found.** and **Error! Reference source not found.**). A policy specifies what should be done with all possible attestation states recorded in the database by the Remote Attestation. A Reaction Policy has to be given for all applications protected by the Remote attestation protection.

Attestation results recorded in the database are actual results that are sensitive to network messaging congestion or to devices that might not be accessible anymore because of loose 3G coverage. The Reaction Manager Engine cannot take decisions based on a single attestation status only because there is a high probability that it does not reflect the actual remote attestation status. The Reaction Manager Engine must preferably consider the history of the attestations. Then, in addition to the latest attestation status, other previous attestations shall be analyzed to confirm or mitigate the status of the latest attestation. The approach taken is to consider the latest attestation status received and another status that reflects all previous attestation statuses not yet analyzed by the Reaction Manager Engine. This status that aggregates many other attestation statuses could be the result of a statistical operation on all statuses or by assigning a weight to each status to promote the most recent one by giving lower weights to older statuses. Another option is to arbitrarily limit the depth of the list of the attestations considered. The current implementation uses only the last two attestations recorded in the DB to simplify the processing and the implementation.

The RM is driven by the connection events returned by the ASCL library. Thanks to the callbacks provided by the library the Reaction Manager Engine maintains a list of connected devices. This list is regularly parsed by the Reaction Issuer that enables the check of the relevant attestations in the ASPIRE DB. Attestations that have been visited in the database by the Reaction Issuer are marked in the rm_status field. This field is checked by the Reaction Manager Engine when taking a verdict decision.

The Reaction Policies indicates the verdict to be taken by the Reaction Manager Engine and what severity level must be put in the reaction notification sent to the Reaction Enforcement Unit.

4.4.3 Reaction Policies

The reaction mechanism 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 who bought the application, etc. More details on this component are presented in Section 4.1.2.1
- The REU described in Section 4.3.1.**Error! Reference source not found.**

The Reaction Enforcement Unit takes decision based on the Tampering Severity Level. This severity scale is application agnostic and shall not depend on the type of Attestation. The appropriate Tampering Severity Level supported by the REU as defined in Section 4.3.3 is specified in the policy for the various Attestations answers that might be recorded in the database. The Reaction Policy specifies how the Reaction Manager Engine decides if a notification reaction shall be created and sent to the RWU.

The status of the attestation set by the RA Verifier in the ra_request table of the database may content the following values

- 0, 'PENDING', 'Pending request'
- 1, 'SUCCESS', 'Right response received in time'
- 2, 'FAILED', 'Wrong response received in time'
- 3, 'EXPIRED_SUCCESS', 'Right response received out of time'
- 4, 'EXPIRED_FAILED', 'Wrong response received out of time'
- 5, 'EXPIRED_NONE', 'No response received'

The Reaction Manager Engine logic is described in **Error! Reference source not found.**. In case a reaction notification is created, the severity is taken from the policy. This table can be customized for an application and this specification is done in a configuration file called the Application Reaction Policy.

In the table, the notation (Previous -1) Attestation means the second last attestation.

Table 1 – Reaction policy expressed in form of a decision table

Current Attestation status	Previous Attestation status	Verdict	Comment
PENDING	PENDING	No action	Some network issues expected. The RM can detect on its side if connection with device is lost, in this case the attestation responses relative to this device in the database are not even checked anymore. If the connection with the device is still available with the RM, the RM can challenge the RA by reducing the delay between two attestation requests.
SUCCESS	PENDING SUCCESS, EXPIRED_SUCCESS, EXPIRED_NONE	No action	Nothing to do, skip to next attestation.
SUCCESS	FAILED, EXPIRED_FAILED	No action	Strange behavior, nothing to do if (Previous -1) Attestation status is SUCCESS.
Either EXPIRED_SUCCESS EXPIRED_FAILED, EXPIRED_NONE	Either EXPIRED_SUCCESS EXPIRED_FAILED, EXPIRED_NONE	Send notification	Severity taken from the policy.
FAILED	FAILED, EXPIRED_FAILED, EXPIRED_NONE	Send notification	Severity taken from the policy.
FAILED	SUCCESS, EXPIRED_SUCCESS	No action or Notification	Depends on Previous -1 status.
EXPIRED_SUCCESS	EXPIRED_SUCCESS	No action	
EXPIRED_FAILED	EXPIRED_FAILED	Send notification	Severity taken from the policy.
EXPIRED_NONE	EXPIRED_NONE	No action or send notification	If no disconnection event is received, then send notification.

4.4.4 Policy description

Reaction Policies are passed to the Engine as configuration files. As an example, the policy presented in **Error! Reference source not found.** is shown below. There is one configuration file per application and the application identifier is in the name of the configuration file.

```
//The status returned by the RA are:
//PENDING, SUCCESS, EXPIRED_SUCCESS, EXPIRED_NONE, EXPIRED_FAILED, FAILED
//We will rely on those status to build the policies
//the PENDING status means the request is not be processed by the
//Reaction Manager; It is an internal status of the RA, meaning that the
//attestation request has no response yet
//
//The policy relies not only on the last status of an attestation response,
//but can also be influenced by older status of attestation responses
//This is expressed by the property
//          "status#n = most_recent_status.past_status" described below.
//
//According to the current status and past status, a policy will be
//described for each attestator.
//This policy may vary according to the Security mechanism used, or the
//piece of code of the application which needs to be protected. But in any
//case, the policy needs to be defined by the developer of the application,
//otherwise, a default behavior will be ued by the Reaction Manager.

//The notion of status_group is used to define some common behavior for
//status belonging to the same group

status_group1 = SUCCESS, EXPIRED_SUCCESS, EXPIRED_NONE
status_group2 = FAILED, EXPIRED_FAILED
status_group3 = FAILED, EXPIRED_FAILED, EXPIRED_NONE
status_group4 = SUCCESS, EXPIRED_SUCCESS
status_group5 = SUCCESS, EXPIRED_SUCCESS, FAILED, EXPIRED_FAILED

status1 = SUCCESS.status_group1
status2 = SUCCESS.status_group2
status3 = FAILED.status_group3
status4 = FAILED.status_group4
status5 = EXPIRED_SUCCESS.status_group1
status6 = EXPIRED_SUCCESS.status_group2
status7 = EXPIRED_FAILED.status_group3
status8 = EXPIRED_FAILED.status_group4
status9 = EXPIRED_NONE.EXPIRED_NONE
status10 = EXPIRED_NONE.status_group5

// Reactions
// May be adressed to the RAM:
//The base delay in milliseconds to use as Attestation Polling
//reaction#n.RAM.delay = 300000
// May be adressed to the CLIENT:
//severity: 0 to 8
//0 no tampering ,
//8 highest level => reaction = crash of application
//reaction#n.CLIENT.severity = 0
// The base delay in milliseconds to use before applying the action

//reaction#n.CLIENT.BaseDelay = 10000
```

```
reaction1.CLIENT.severity = 0
reaction2.CLIENT.severity = 8
reaction3.CLIENT.severity = 4
reaction4.RAM.delay = 300000
reaction4.CLIENT.severity = 0
reaction5.RAM.delay = 300000
reaction5.CLIENT.severity = 2
reaction6.RAM.delay = 300000
reaction6.CLIENT.severity = 4
reaction7.RAM.delay = 150000

attestator_1.status1 = reaction1 //default value for status 1 in reaction
manager if policy no defined here will be no reaction
attestator_1.status2 = reaction1 //default value for status 2 in reaction
manager if policy no defined here will be no reaction
attestator_1.status3 = reaction2 //default value for status 3 in reaction
manager if policy no defined here will be the most severe response level  8
on client
attestator_1.status4 = reaction3 //default value for status 4 in reaction
manager if policy no defined here will be severity 4 sent to CLIENT
attestator_1.status5 = reaction4 //default value for status 5 in reaction
manager if policy no defined here will be  no reaction but requires a new
attestation to client in 30 seconds
attestator_1.status6 = reaction5 //default value for status 6 in reaction
manager if policy no defined here will be  severity 2 and requires a new
attestation to client in 30 seconds
attestator_1.status7 = reaction2 //default value for status 7 in reaction
manager if policy no defined here will be the most severe response level  8
on client
attestator_1.status8 = reaction6 //default value for status 8 in reaction
manager if policy no defined here will be  severity 4 and requires a new
attestation to client in 30 seconds
attestator_1.status9 = reaction1 //default value for status 9 in reaction
manager if policy no defined here will be  wait until the network
connection is restored. Not even sure we can send a reaction to the client
attestator_1.status10 = reaction4 //default value for status 10 in reaction
manager if policy no defined here will be  no reaction but challenge the
RAM requiring a new attestation to client in 30 seconds

//attestator 2
//attestator 3
//......etc......
//attestator_10
```

4.4.5 ASPIRE Database

Some updates have been done to the ASPIRE DB to support the RM. An additional field is required in an existing table and two tables already introduced in Section 3 have been designed for the purpose of the RM. The ra_request table needs one more field compared to the description given in Section 3.2.2 to avoid repetition, only this extra field is described below.

- **ra_request**:

Table 2 – Update to the ra_request ASPIRE DB table.

Column name	**MySQL type**	**C type**	**Description**
`rm_status`	varchar(8)	`uint8_t`	State of the request as seen by the Reaction Manager. Values can be (pending, expired, in_progress,

			completed).

The *pending* value is the default value that means the ra_request has not been processed yet by the RM. The *expired* value means that the RM considers the ra_request as exceeding the maximum time limit and should be dropped. The *in_progress* state means the RM is currently processing the ra_request; The *completed* value means that no further action is required on the ra_request

Two tables are specific to the RM: ra_reaction and ra_reaction_status; the descriptions are given in the following tables.

- **ra_reaction**

Refer to Section 3.2.2.

- **ra_reaction_status**

One extra severity field is required compared to the initial description, it is described in the table below.

Table 3 – ra_reaction_status ASPIRE DB table.

Column name	**MySQL type**	**C type**	**Description**
`severity`	varchar(8)	`uint8_t`	Severity of the reaction; it ranges from 0 to 8

The RM retrieves the attestator number and the Application ID from the ra_request in order to be able to retrieve the corresponding Reaction Policy. As previously mentioned the ra_reaction and the ra_reaction_status are managed by the RM.

Each ra_reaction is identified in the DB by its id is associated to one application_id; the application_id is the unique identifier that permits to retrieve both the application and the device.

tion that is called when a notification reaction is received from the Reaction Manager.

```
void applyReactionEnforcement (
      char* szCodeID    // The fixedcode ID of the enforcement unit
      long lTimeBase    // The time base in ms
);
```

The time base parameter is the one given as a parameter of the annotation. This is the base of the computation of the delay to slow down the application.

4.5 Plan

4.5.1 Device Reaction mechanism

The RWU and the REU have been committed on the svn server in development/reaction_unit. The corresponding ACTC task has been committed in development/ACTC/GTO directory.

The reaction ACTC task has to be validated with other protections. This work will be done in M31.

According to the remaining budget another iteration of the REU will be implemented before the end of the project to provide a more stealthy component.

4.5.2 Online Reaction Manager

The Reaction Manager Engine and the Notification Dispatcher are implemented and have been unitary tested. They are available on the SVN server in the development/reaction_manager directory. Components have been committed in source form with a makefile.

A script creates Reaction events and puts them in the Reaction Events queue. The Reaction Issuer will be implemented in M31 and the full Reaction Manager will be integrated with the RA early M32 before the GTO Tiger Team Experiment. Integration here means mostly global testing since the only interface of the RM and the RA are the database.

Section 5 Anti-cloning

Section Author:

Patrick Hachemane (NAGRA)

5.1 Introduction

The anti-cloning mechanism consists of forcing the client to regularly connect to the server in order to provide its device ID and an incremented counter. If two clients share the same device ID, the server will not receive the counter incrementally, which allows detecting that the software has been cloned. If some valuable item used by the application, typically a license, is linked to the device so that it can be used only with the correct device ID, this ensures that this item cannot be shared by several devices.

Two different protections can be used: a status send (silent report) or a decision request (pro-active report). We refer to Section 4.5 of ASPIRE deliverable D1.04 for a full description of the mechanism.

5.2 Overview

Figure 4 depicts the anti-cloning workflow diagram, followed by a detailed overview of the referenced steps.

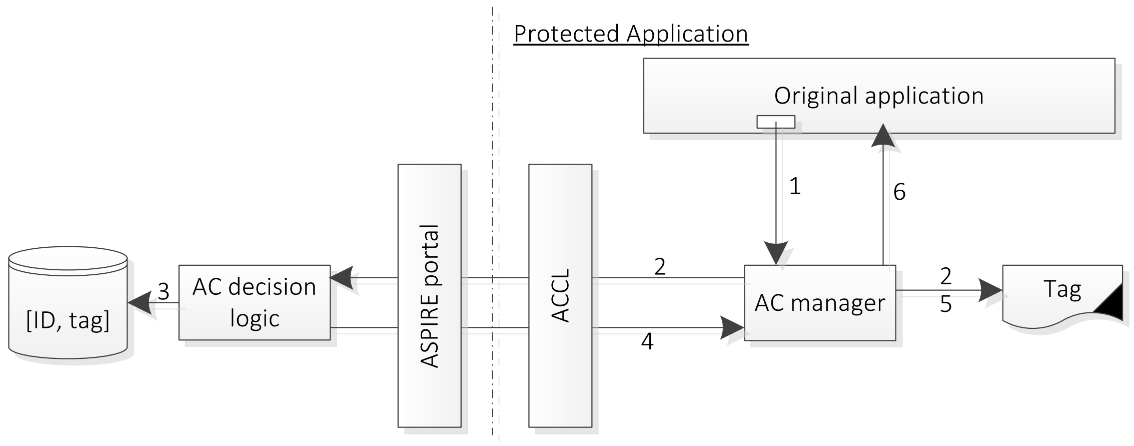


Figure 4 – Anti-cloning workflow diagram

During the execution of some specific instructions, annotated by the anti-cloning protection, the original application (1) requests the anti-cloning (AC) manager to activate the mechanism. The AC manager (2) gets the current value of the tag and sends it to the ASPIRE portal, using the anti-cloning client library (ACCL). The value is retrieved by the ASPIRE portal and transmitted to the anti-cloning backend (AC decision logic). This backend compares it to the expected value, possibly informs the remote attestation tool about an incorrect value, and stores the value in its database. In case of proactive report, the portal (4) sends back the status to the AC manager. Finally, the AC manager (5) updates the tag value to use during next activation.

5.3 Implementation

5.3.1 Annotation

In order to trigger anti-cloning mechanism, two different annotations have been defined. To insert a silent report into the code, the following annotation must be used:

```
__attribute__(ASPIRE("protection(anti_cloning, status)"))
```

To insert a pro-active report, the following annotation must be used:

```
__attribute__(ASPIRE("protection(anti_cloning, decision(response))"))
```

Note that the annotation is not related to any variable or code fragment. In the latter case, the response is stored in the variable `response`, that must be defined (outside of the annotation) as `int`. This variable gets the anti-cloning status as evaluated on server side; possible values are:

- 0 if the counter always has been sent correctly to the server;
- 255 if the counter value is not correct during this report;
- N (0 < N < 255) if the counter value is correct, but has been sent incorrectly at least N times in the past.

5.3.2 Connection from client side

If the anti-cloning mechanism is active, the client has to connect to the server. The ACCL is used; a specific file `anti_cloning.c` has been developed to handle the connection and exchange the data with the server.

The payload sent from the client to the server is composed of the ASPIRE application ID, the device ID and the counter value. As these values must remain persistent on device side, they must be stored in a local file. This file is named `anti_cloning.bin` and stores these two fields.

As simplification in the implementation, the value used as device ID is the property `ro.build.display.id` as found in the file `/system/build.prop`. This value is read from the file on first installation of the application, i.e. if the file `anti_cloning.bin` is absent.

Note that the ASPIRE application ID is not sufficient to distinguish between two cloned devices: it is by definition the same for all devices. Therefore the anti-cloning mechanism must use a specific device identifier to distinguish two different physical devices running the same application. As a reminder, the objective of anti-cloning mechanism is to ensure that two different devices cannot share the same valuable item, e.g. the same access license.

In summary, the file `anti_cloning.c` is in charge of reading/updating the file `anti_cloning.bin`, building the transmission payload and sending it to the server using the ACCL.

5.3.3 Compilation

During code compilation, the annotation is replaced by a call to the corresponding anti-cloning function; this replacement has been integrated to ACTC. More accurately, the ACTC performs following operations:

- it adds at the beginning of the file containing the annotation the declarations of the used anti-cloning functions;
- it replaces the annotation by a call to the corresponding anti-cloning function;
- it adds to the build the files `anti_cloning.c` and `accl.c`;
- it adds to the build the library l`ibcurl.a` and its dependencies.

5.3.4 Server backend

In order to receive anti-cloning requests, the anti-cloning *backend* has been developed. This backend is written in Python and processes anti-cloning requests, i.e., having TID 70 or 75.

In order to determine if a device is compromised, the backend manages its own device database. It is stored in an XML file called `anti_cloning.xml`; we refer to Section 5.3.5 for details about this file. On each request, the backend checks if the received counter value matches the expected one, then updates the database and (optionally) sends back the response with the device status.

In case of incorrect counter value, the backend also notifies the reaction logic by reporting failed attestation in the ASPIRE DB. Reaction logic will decide the proper ways to react, as presented in Section 4. This ensures the integration of the anti-cloning mechanism to the reaction logic. This reaction has been currently tested by means of a dedicated script called `mark_application_as_compromised.sh`.

In summary, the anti-cloning can either react by itself, using its own database and the pro-active report, or rely on the ASPIRE DB to take appropriate reaction depending on the reported status. The first alternative has been mainly developed for testing purposes; in ASPIRE context, the second alternative should be used, so that the reaction is coordinated with other ASPIRE protections.

5.3.5 Database

In order to store the state and the counter of each application running on a known device, a "database" is needed on server side; it is stored in the file `anti_cloning.xml`. The file is updated on each request. The figure below depicts the grammar used for this file.

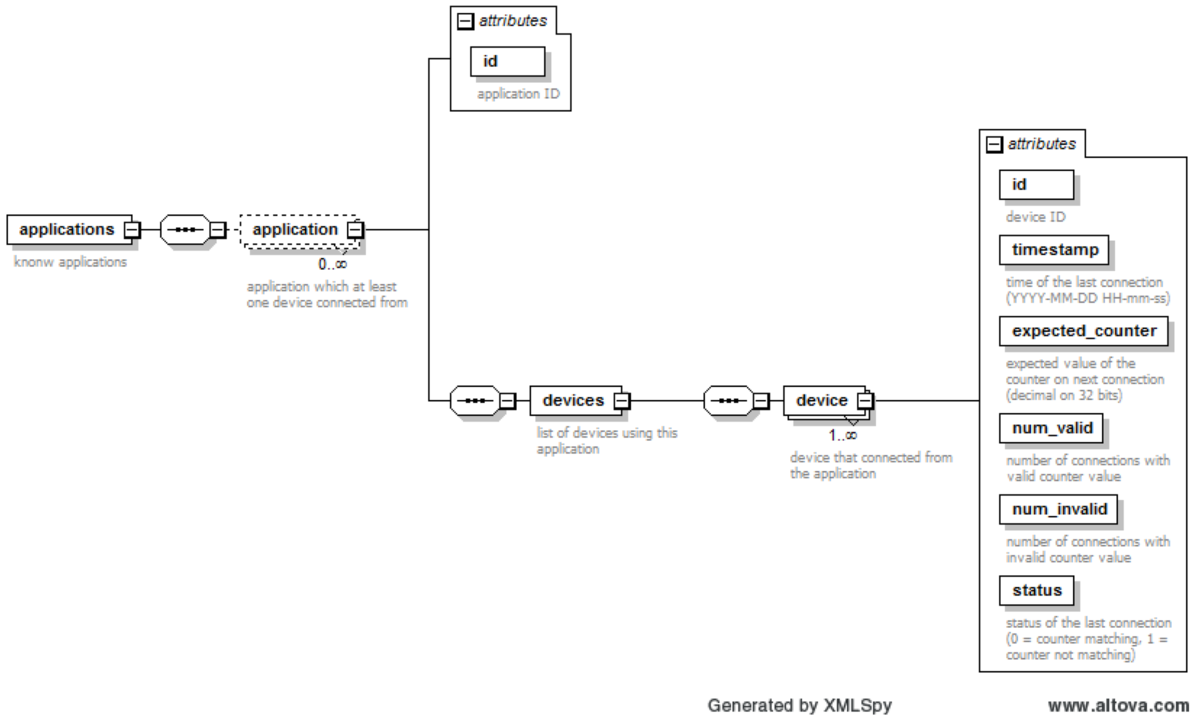

attributes
id
application ID
applications
knonw applications
application
0..∞
application which at least one device connected from
devices
list of devices using this application
device
1..∞
device that connected from the application
attributes
id
device ID
timestamp
time of the last connection (YYYY-MM-DD HH-mm-ss)
expected_counter
expected value of the counter on next connection (decimal on 32 bits)
num_valid
number of connections with valid counter value
num_invalid
number of connections with invalid counter value
status
status of the last connection (0 = counter matching, 1 = counter not matching)
Generated by XMLSpy
www.altova.com

Figure 5 – syntax of database file

The node `application` is associated to the following attribute:

- `id`, which stores the application ID, as string.

The node `device` is associated to the following attributes:

- `id`, which stores the device ID, as a string;
- `timestamp`, which stores the timestamp of the last connection, formatted as YYYY-MM-DD HH-mm-ss;
- `expected_counter`, which stores the expected value of the counter on next connection, as decimal value on 32 bits;
- `num_valid`, which stores the number of connections with valid counter value;
- `num_invalid`, which stores the number of connections with invalid counter value;
- `status`, which equals 1 if the last connection has been done with valid counter value, 0 otherwise.

5.4 Simplifications

During the implementation, some simplifications have been done. These simplifications eased the development of the feature with a limited impact on the mechanism validity. These simplifications are related to:

- Device ID
- File used on client side
- First counter value

- Next counter value
- Transmitted payload
- Remote attestator

5.4.1 Device ID

It is difficult to extract the Android device ID from native code. Most solutions imply using JNI to access the corresponding Java field. To validate the mechanism, a "sufficiently" unique value was needed, therefore it has been chosen to use a device property. If two devices get the same property, they would be marked as compromised.

In other words, it is mandatory to use devices having a different value for the property `ro.build.display.id`.

Moreover, for confidentiality and privacy reasons, it would be even better to use the hash of several device identifiers rather than a single value.

5.4.2 File used on the client side

On the client side, the current value of the counter (along with the device ID) is stored in a binary file. This file is not encrypted, nor authenticated. In a real mechanism, authentication should be used; encryption could be useful.

5.4.3 First counter value

The first value used for the counter has been set to a fixed value. This value could be used to easily let the anti-cloning mechanism think that an old device is a new one. In a real system, the value should be hidden and modified depending on some external conditions.

5.4.4 Next counter value

The next value used for the counter is equal to the previous value, incremented by 1. This should be replaced by a more complex algorithm (hash, encryption, …) in order to prevent the predictability of the valid values. The next value could also be sent by the server if the response mechanism is active.

5.4.5 Transmitted payload

The payload is transmitted in the clear, without authentication. In a real system, the transmission layer (ACCL / ASCL) should protect the payload against possible attacks. A better way would be to include the mechanism into the application protocol used to exchange any data between the client and the server, in particular when delivering a valuable item (license).

5.4.6 Remote attestator

As the remote attestator does not know the notion of *device*, if a device is detected as compromised by the anti-cloning mechanism, the calling application is marked as compromised rather than the device. This is due to the fact that the remote attestator is application-based, while the anti-cloning mechanism is device-based.

5.5 Validation

In order to validate the anti-cloning mechanism, a code sample has been developed, released and archived along with the source code. Moreover, an anti-cloning protection has been added to the NAGRA use case and was successfully checked.

On the reference board, the extra application size used by the anti-cloning mechanism is about 3 KB. For each anti-cloning transmission, the payload exchanged with the server (without header and signalization) is less than 100 bytes.

Section 6 Code mobility and renewability

This section presents the design and implementation of extended versions of existing White Box Crypto (WBC), SoftVM, and Static Remote Attestation protections that exploit code mobility support (T3.1), a Task T3.3 topic. Code mobility and renewability in time have several objectives, including limiting (static and dynamic) inspection possibilities of client-side code, and reducing the likelihood and mitigating the effects of successful attacks. Renewing protections has the effect that they are changed before they are defeated and/or before they are understood. For instance, in case of mobile WBC, a new version of WBC code and data are sent to the client before the key is extracted (that in the NAGRA scenario allows to access the media content). On the other hand, for static remote attestation, a change of attestator allows to detect changes that could have been hidden to a previous version.

The purpose of provisioning code at run time serves to make available to the client some sensitive parts of the application (either code or data structures) only when they need to be executed. The results presented here can be considered a first step towards renewability in time and the preparation of the next activities that will be performed in Task T3.3. whose planning is presented in Section 6.1.

6.1 Renewability status and planning

Section Author:

Alessandro Cabutto, Paolo Falcarin (UEL)

We present an update of the renewability plan presented in the deliverable D3.04 until M30:

M26: creation of DB for storing different code blocks. Done by UEL. This objective required a minor change to the Mobility Server repository organization. Possibly this will be adapted again during next development phases.

M27: Extension of the Code Mobility tool to transfer data blocks. Done by UGent, as documented in this deliverable in Section 6.2.2.

M28: first implementation of the Renewability Manager. Not completed yet. Probably the Renewability Manager will be delivered in M31. This really has no impact on the other tasks, thus it will not negatively affect other activities in this planning.

M29: testing of the approach with WBC and SoftVM. Data mobility is working for both WBC and Code VM as documented in this deliverable in Section 6.2 and 6.3. No renewability is applied so far and will be probably supported in M31 or M32.

M30: design of renewable RA and complete integration of RA in the ACTC for deliverables D3.05 and D3.06. Done, as documented in this deliverable in 0 and 6.4.

These are the next deadlines:

M31: release of the Renewability Manager.

M32: renewability in space of with WBC and SoftVM.

M33: Integration with Diablo for diversity and Renewability support in ACTC

M35: Implementation of mobile remote attestators.

M36: Renewability on use cases (Nagra or SFNT).

6.2 White Box Crypto Mobility

Section Author:

Bjorn De Sutter, Bert Abrath (UGent)

6.2.1 Requirements Analysis

For supporting white box crypto (WBC) mobility, two basic mobility features need to be supported: mobile code and mobile data. This follows from the fact that WBC primitives can be implemented using two components: large, complex code fragments with irregular control flow, and large read-only data tables. To hide both components from static analysis tools, both components need to be made mobile.

The WBC functionality developed by NAGRA in the ASPIRE project relies on relatively simple code, that accesses large read-only data tables in a set of nested loops. These are defined as multi-dimensional arrays that are local to the C functions that implement the encryption and decryption primitives. The following pseudo-code captures the relevant aspects.

```
void encrypt(char* in, char * out, int size){
   char LUT1[ ][ ] = { ... } // large number of initialization values;
   for ( ... i ... )
      for ( ... j ... )
          ... = ... LUT [i][j] ...
}
```

With standard compilation flags, the initial values of LUT1 end up in the single .rdata (read-only data) section of the object file generated by the compiler, together with the other read-only data generated for the whole source code file of which the encryption routine is a part. However, When the code is compiled with the `-fdata-sections` flag – which all modern compilers support – the initial values end up in their own .rdata.LUT1 read-only data section.

Furthermore, as LUT1 is a function-local array, only the body of the function in which it is defined contains so-called *address producers* of that variable. These address producers are instruction sequences that compute the address of the variable and put it in a register to serve as base address for all the memory accesses to the array in the loops.

This results in a very interesting feature: no (relocatable) (computations of) addresses of the .rdata.LUT1 section or of similar sections relevant to making the data involved in WBC mobile, will be found outside the bodies of the functions implementing the WBC primitives themselves.

From this feature, it follows that whenever such a function is marked in its entirety as mobile code by source code annotations, and then extracted from the program to become mobile during the ACTC's binary rewriting processes, all of the relevant address producers of the large data arrays will automatically be extracted as part of that process.

The functionality to make entire functions mobile is already available, as was reported in multiple previous WP3 deliverables. This functionality is implemented in the ACTC's binary code rewriting phase, on top of the link-time rewriting tool Diablo. Furthermore, Diablo's Augmented Whole-Program Control Flow Graph (AWPCFG) [DeS07] already models the relation between code fragments, address producers, and data sections in a way that is ideally suited to identify the data sections that become accessible from within certain code fragments and not from somewhere else. We exploit this in the developed support for making the WBC arrays mobile.

6.2.2 *Mobile WBC design*

As indicated, to make the WBC code and the involved tables mobile, we rely on the existing code mobility support, and Diablo's AWPCFG.

6.2.2.1 Developer support

The code mobility source code annotation is extended to let an ACTC user indicate whether or not the read-only data exclusively accessed from a mobile code region needs to become mobile together with that code region.

6.2.2.2 Tool support

In BLP04, which implements code mobility in the ACTC, the code mobility step is extended as follows:

- As was already the case in the code mobility protection, code regions marked to become mobile are extracted from the main program CFG (which is in fact the AWPCFG, a fact that need to be mentioned so far in the project deliverables) and put in separate CFGs.
- A reachability analysis is performed on the AWPCFG and the separated CFGs to determine which read-only data sections are reachable only through a separate CFG.
- Any such data section is migrated from the main AWPCFG to its corresponding, separate CFG, such that the data will become part of the same mobile block as the code in that separate CFG.
- The necessary code rewriting is performed to make all necessary code offset-independent (see previous WP3 reports), as was already the case in the code mobility protection. Very few adaptations are necessary to rewrite the involved address producers, as the offsets between the code and the data in a single mobile block is known at link time.
- For each mobile block, Diablo produces a binary blob that contains both the code and the data.

This design flow not only supports mobile WBC data in the ASPIRE WBC implementations, but in fact any read-only data accessed exclusively in any mobile code region. The only requirement is that the source code is compiled with the `-fdata-sections` flags. This is not a strange requirement: like many existing programs, the ASPIRE use cases are already compiled with that flag anyway, because it often enables link-time size savings in the binary that would otherwise be wasted.

In the run-time tools (the mobile block Downloader and the Binder, see previous WP3 deliverables) no changes are required. Mobile blocks are downloaded and allocated as a whole, and the Binder only needs to bind external code to the entry point of the code fragment in the mobile block (as was already supported) because there is the guarantee that the static part of the binary contains no reference to the data part of the mobile block at all.

6.2.3 *Implementation*

The necessary analyses and transformations were implemented in Diablo, and released on 3 Feb 2016. Relatively little effort was needed for this, given Diablo's existing code and data reachability analysis and its flexibility in specifying relocatable computations to be injected into rewritten code. Most effort was in fact spent on inserting extra checks for pre-conditions and on code refactoring to allow reuse of existing functionality without code duplication.

The preceding requirements analysis and design was a joint effort between UGent and NAGRA. The implementation in Diablo was done by UGent.

6.2.4 *Evaluation*

To test the implementation, unit tests were used, as well as the NAGRA use case. This testing was successful.

6.2.5 Future work

In Task 3.3, this line for RTD will be continued to provide WBC renewability, which will first be reported in D3.07-08 in M33. In summary, the plan is as follows:

- The ACTC will be extended to generate the necessary scripts to compile WBC primitives on an ASPIRE server with the same compiler settings as the original WBC primitives in a client, but for new keys and/or random seeds.
- New scripts will be developed to extract code and data from the output of that server-side compilation and to replace code and data chunks in the mobile code blocks that were extracted from the client-side application by the ACTC.
- The mobility server will then be extended to let it serve the server-generated mobile code blocks instead of the client-extracted ones.

In this way, we will deliver the necessary server mobile code in support of renewable, time-limited WBC.

6.3 SoftVM Bytecode Mobility

Section Author:

Bjorn De Sutter, Bert Abrath (UGent), Andreas Weber (SFNT)

6.3.1 Requirements Analysis

UGent collaborated with SFNT for analysing and designing the SoftVM mobility protection. In this case, the goal is to enable mobile bytecode. For static bytecode, the following steps are already in place:

- Extract native code blocks from the client-side executable.
- X-translate them to bytecode.
- Integrate the bytecode blocks into the client-side executable together with a SoftVM to interpret the bytecode and the necessary stubs to invoke the SoftVM.
- Fix up the integrated chunks to encode the necessary addresses as they occur in the finalized, protected binary.

This existing process was documented extensively in various WP2 deliverables.

To make bytecode mobile, one option would have been to use a similar approach as was used for WBC Mobility. In this case, the stubs that invoke the SoftVM contain an address producer to the relevant, read-only bytecode section, so it is doable to extract the bytecode section together with the stub.

This approach was determined to be problematic for a number of reasons, however:

- First integrating externally generated code (in this case X-translated bytecode) only to extract it again is overkill and could impose unnecessary limitations.
- In source code, the client-server code splitting annotations typically mark much bigger regions than those of the individual bytecode fragments. A user might very well want to make parts of those bigger regions mobile, but not all of them. Likewise, he might want to make only the stubs mobile (as those are in a code format an attacker might understand during static analysis), but the bytecode itself not (to save bandwidth, and as those are in a custom format not known to the attacker anyway). So it makes sense to keep the annotations of mobile code and SoftVM code separate yet composable, rather than relying on an extension of the existing code mobility annotation to make bytecode mobile as well.

6.3.2 Mobile Bytecode Design

For these reasons, an alternative approach was designed.

6.3.2.1 Developer support

The SoftVM source code annotation is extended to let an ACTC user indicate whether or not the X-translated bytecode is to be mobile.

6.3.2.2 Tool support

When native code is extracted (in step BLP01 in the ACTC) by Diablo and passed to the X-translator (BLP02), Diablo now informs the X-translator about the static resp. mobile character of the bytecode. In case static bytecode is requested, the X-translator generates the same stub and bytecode to be integrated in the client binary as it did before, in the form of an assembly file to be assembled and linked into the client-side app before integration in BLP03.

For mobile bytecode, however, the X-translator now generates an external mobile bytecode block that is not statically integrated into the client-side binary at all. Instead, it is generated in the same directory where the mobile code blocks extracted in BLP04 reside, ready to be stored on the code mobility server. The X-translator still generates a stub to be integrated into the client-side app, but this stub will be able to determine it is handling mobile bytecode based on its arguments.

Whereas native mobile code is downloaded from the server by invoking the Binder via an indirect control flow transfer right before the execution of such a native code block, in the case of mobile bytecode the adapted stub invokes a custom SoftVM binder, with the following API:

```
binder_softvm(application_uuid, mobileId, &vmImageAddr, &vmImageSize );
```

The first two parameters are input parameters, used to specify which block to download, the second two parameters are output parameters that specify the size and the location in memory to which the bytecode was downloaded.

This SoftVM Binder, although we designed it specifically for making the bytecode mobile, operates completely independently of whether it is downloading bytecode to be interpreted or any other form of read-only data. As such, UGent was able to develop the SoftVM Binder, relying on the Code Mobility Downloader developed by UEL, completely independently of the X-translator adaptations and the SoftVM adaptations developed by SFNT. In other words, the functionality of the SoftVM Binder is orthogonal to the design and implementation of the X-translator, the bytecode format, etc. The X-translator-generated stub is responsible for invoking the SoftVM Binder in the appropriate way.

This design keeps the benefits of the separation of concerns that was achieved with the existing tool support for the client-side code splitting (i.e., SoftVM) protection in the ASPIRE project.

As the format and size of the bytecode, as well as the internal structure of the mobile bytecode blocks are independent of the operation of the mobility Downloader and the SoftVM Binder, this design also prepares for temporal bytecode diversification as a possible extension after the project. In such an extension, both new, diversified bytecode and new corresponding SoftVM internals would be generated on the server side and delivered at run time to the client using mobile code and mobile data support.

6.3.3 Implementation

UGent developed the minor adaptations to the binary processing steps of the ACTC in its Diablo tools, and implemented the SoftVM Binder.

To support mobile bytecode SFNT slightly extended its existing X-Translator/SoftVM solution.

The first extension affected X-Translator's JSON parser as Diablo flags mobile chunks with a `mobile_id`, which is a 32-bit integer. Therefore, the parser had to be extended so it understands the `mobile_id` and makes it available for the following code generation pipeline.

The second extension affected X-Translator's dynamic library interface, which gained an additional API `bin2vm_setMobileCodeOutputDir`. This function enables Diablo to specify the directory where the X-Translator will output the files containing the bytecode of the mobile chunks.

The third extension affected the X-Translator's code generation pipeline. X-Translator's first phase was extended, so that it behaves differently for mobile chunks. If a mobile chunk is encountered, it does not create the usual placeholder bytecode image but instead outputs a special "referral" image consisting of exactly two 32-bit words. The first word is a header and is present in both regular and "referral" images. This header indicates the type of the image using the most significant bit. When the bit is set, the image is a reference to mobile bytecode and the second word contains the `mobile_id` of the associated mobile bytecode. Otherwise, the image contains the actual bytecode whose size is specified by the header's remaining 31 bits.

In addition, the X-Translator's second phase was also slightly adapted for mobile chunks: Instead of returning the final bytecode images to Diablo, the X-Translator outputs the actual bytecode of each mobile chunk into a separate file. These files are created in the previously specified output directory and are named `mobile_dump_` followed by the `mobile_id` as an eight characters wide hex number, e.g. `mobile_dump_00001234`.

The last extension affected the SoftVM stub code. This code was changed, so that it first evaluates the most significant bit of the passed image and in case the bit is set, it calls `binder_softvm` with the `mobile_id` obtained from the second word to retrieve the actual bytecode before calling the SoftVM. Otherwise, it directly calls the SoftVM with the passed bytecode.

6.3.4 Evaluation

To verify the correct behaviour of the XTranslator and the stub code, SFNT wrote a simple dummy implementation of the `binder_softvm` function that does not retrieve the mobile bytecode over the network but instead just expects the presence of the `mobile_dump_<mobile_id>` files in the current directory and then loads the file content into a memory buffer. With this setup, the correct functioning of mobile bytecode was verified for both the original stack based SoftVM as well as for the newer LLVM-based SoftVM before sending the new XTranslator/SoftVM release to UGent for the ACTC integration.

UGent evaluated the correct functioning of the SoftVM Binder and the overall approach on unit tests and on the SFNT use case. The evaluation was successful.

6.4 Attestator mobility and renewability

Section Author:

Cataldo Basile, Alessio Viticchié (POLITO), Bert Abrath (UGent)

Remote attestation uses Attestators to collect information that must prove to the remote server the integrity of the application to protect. An attacker may thus want to perform static analysis of the Attestator's code in order to understand its functioning and defeat it or to implement methods to attack the integrity of the application assets that are not noticeable by the remote attestation protection. Making Attestator's code mobile (that is, the Attestator code is received the first time that it is needed), has the advantage that no purely static analysis of the Attestator's code can be performed. Moreover, the aim is to support renewability of Attestators. Indeed, supporting renewability of Attestators can also help in limiting the consequences of successful attacks, as methods to defeat one Attestator type do not (necessarily) work for another one and modifications that are hidden for one Attestator type are not (necessarily) non-noticed by another one.

Moreover, by watching the parts of the application that are monitored by the Attestator, an attacker can gain insights on the most sensitive parts of the application. By allowing the application to receive information on the areas to attest only when needed and by renewing the areas to attest periodically, we can achieve a better protection.

The next sections present the effort to design mobile and renewable Static Remote Attestation, by investigating the changes needed to Static Remote Attestation and to the Code Mobility developed in WP3. The design reported here involves the research performed in T3.2 for Static Remote Attestation and in T3.3 for supporting code mobility and possibly renewability.

6.4.1 Basic facts concerning protection with static attestation

We report here some basic information concerning the ACTC support of static remote attestation. A more in-depth description is available in D5.08.

The annotations are parsed to determine how to use the remote attestation:

- static_ra is processed to determine the attestators to use;
- static_ra_region annotations determine the code areas to attest.

All the code of the selected attestators is customized and compiled by the ACTC and linked in as a unique object file. The object links with other external libraries (e.g., the hash function). Static remote attestation relies on an additional object file (racommon.o) that includes all the features shared among all the attestators, which have been factorized for ease of code management. The Diablo-based binary rewriting step BLP04 of the ACTC reads the static_ra_region annotations and generates the Attestation Data Structure (ADS), which it inserts into the binaries. The ADS describes the data needed to reconstruct the areas to attest (after layout randomization and obfuscation), that is, every area is represented as a sequence of code blocks (see D3.02).

At run time, when the application starts, the attestator is launched in a separate thread. The attestator calls the ASCL-WS code needed to create a persistent connection with the server (web socket in our implementation).

6.4.2 Requirements Analysis and design

Supporting mobile static remote attestators requires the support of two basic mobility features: mobile code and mobile data. Two possible uses are:

- *Renew the attestator*, that is, either the attestators to use are decided at run-time and sent to the client-side application or the (some or all the) attestators initially provided with the application is substituted with a new one. This feature requires the support for mobile code (the attestator) and mobile data (the ADS). Sending a new ADS is only needed if the new attestator supports a different memory area management function (required with the `MA` annotation parameter, as explained in 0). Currently, we only have implemented one memory area management function, thus ADS substitution would not be necessary.
- *Change the areas to attest,* this feature requires the support for mobile data (the ADS).

After our preliminary analysis, we determined that attestators' code can be made mobile following the design already documented in the deliverable D3.04 Section 5. Only, attention must be paid to preserve the links to the external libraries. Moreover, since more than one attestator may have already been inserted in the original application, to make single attestators mobile requires the ability to substitute (and invalidate) individually each attestator. Currently, all the functions related to one attestator are made unique during the insertion (see Section 3.2), therefore, since Diablo is able to identify functions based solely on their name, this requirement is easy to address.

Attestators may differ in size. Therefore, attention must be put to the area to allocate for the sent attestators. This is a common case and has been already addressed by code mobility features. In general, the location of a mobile block is not known beforehand, nor does the downloader now the size of the block before it starts downloading it. The downloader requests a block with a certain ID from the server, and then receives a block of a certain size (N bytes). It then dynamically allocates (malloc) an area of N bytes, copies over the mobile block, and returns this to the binder. Given this characteristic, the attestators may have any size, and at run time will be assigned an address that can vary with the different executions.

To avoid known attacks (like the OWASP Broken Authentication and Session Management) the Web Socket mechanism, used by the ASCL-WS, has been designed to make hard to share already established persistent connections. In practice, Web Socket persistent connections are stored as context objects. These objects cannot be easily shared or passed to other applications or other threads in the same application). Currently, every attestator is executed in a separate thread and it individually establishes a persistent connection with the server through the ASCL-WS. As a consequence, new attestators sent by the server could have problems in using already established persistent connection with the server if executed as new threads. Solving this issue could require consistent effort.

Server-side requirements

The regeneration of the ADS should be done by a tool based on Diablo (similar to BLP04) from the binaries if new areas need to be attested or if the memory area management function is changed.

The procedures to dynamically change the association between Application ID and attestator and verifiers (an API) in the ASPIRE DB must be updated in order to support the change of the attestator.

The procedures to invalidate the nonces associated to clients must be provided, as prepared attestation data may change, both for the normal areas and for the ones attested at startup.

6.4.3 Mobile attestator design and support for renewability

Given the requirements concerning static remote attestation and code mobility, the most promising approach is to support mobile attestators and renewability is to make only part of the attestators' code mobile.

The first advantage is that ASCL-WS initialization code needs not to be executed every time that a new attestator is sent. Furthermore, the library is linked in once thus not affected by dynamic linking of mobile code (unless racommon.o library it not made mobile, but in that case, the extraction of mobile code must preserve linking among mobile blocks).

The parts of attestators to be made mobile and renewable are the hash function and the random walk routine. Currently, there is no need to make the racommon.o mobile or renewable (as it includes the parts common to all the attestators and they are not sensitive, an attacker cannot gain any advantage on compromising RA if he understand the racommon.o code). The exact part of the source code of the attestator to be made mobile is thus annotated with code mobility annotations. The same approach should be followed to mark the parts that must be made mobile.

There are three possible approaches to renew attestators.

- **Received once**. The attestator is first sent when an attestation request needs to be served. Then the attestator remains available during the whole period when the application is up and running. This approach requires no changes to current code mobility and remote attestation protections. It is only needed the server-side logic to select the next attestator to send. This is the simple mobility case.
- **Occasionally renewed**. The attestator is first sent when an attestation request needs to be served. However, after a pre-defined number of execution the attestators becomes eligible to be substituted. This approach requires minor changes at client side code mobility procedure (a counter of the executions is maintained before requiring to renew a block) or at server side (the request for a new mobile code is sent to the server however, in some cases the server can answer to reuse the piece of code previously sent. As renewing attestators require removing the previous attestator (whose code might still be running), this case will likely require source-level changes in the attestator code as well. For example, the removing of the old attestator and requesting of the new one might happen on request of the (non-mobile) attestator logic present in the application. The attestator's code must be designed so that it will know whether or not some thread is still executing on the mobile block that was downloaded.
- **Renewed every time**. The attestator is sent every time an attestation request needs to be served. The attestator code is reached then a new code block is requested. This approach requires limited effort on the code mobility side (it possibly works with current implementation with no modifications). However, it requires a consistent modification at server side logic of the remote attestation. Indeed, attestation data are prepared offline to save computational effort during the verification phases. If the attestator is renewed at every attestation, either the sequence of attestators that will be sent is known in advance (this also requires re-engineering of the code mobility protection) or no pre-computation is used at all. This issue would affect in particular the 'attest-at-startup' feature. In this case, attestators code may need changes to be aware of the presence of threads.

Different approaches are available to make the ADS mobile.

- **Mobile ADS**. With this approach, ADS raw data is collected during the preparation made by the RA tool (the ACTC tool that applies remote attestation as presented in Section 3.2) to be sent by the server. The prepared ADS is downloaded when needed. It requires the implementation of synchronization mechanisms to inform the attestator when the download of the ADS has been completed, in order to properly launch the ADS parsing activity and attestation computation. ADS must be computed based on the attestators, that is, the ADS must be compatible with the memory area management functions of the attestator. This synchronization could also be implemented as a call to the code mobility binder API to download in a synchronous way the ADS. Alternatively, by annotating a part of the attestator to be made mobile, the right mobile block should be downloaded automatically and made available the first time it is required. In general, a substantial quantity of engineering effort is required.

- **Mobile ADS and management code**. With this approach, as in the previous one, ADS raw data is collected during the preparation made by the RA tool. The prepared ADS is downloaded when needed. However, together with the ADS, the mobile code conveys the functions to access and correctly parse the memory area information. That is, the ADS is sent together with the memory area management code. The attestator thus accesses the API provided by the new code block. As in the previous case, this approach requires the implementation of synchronization mechanisms to inform the attestator when the download of the ADS has been completed, in order to properly launch the ADS parsing activity and attestation computation, as explained in the previous case. This synchronization could also be implemented as a call to the code mobility binder API to download in a synchronous way the ADS. Moreover, it requires the pre-computation of ADS, which needs to be made mobile as data block together with the memory area management code. Furthermore, this approach requires a different management of the static RA code, as the memory area management code must be marked as mobile together with the ADS. It requires non-trivial work to extend Diablo to support this approach and derive proper blocks, however, it requires minor modifications to the RA tool and to the code mobility features.
- **Individual mobile area descriptions**. With this approach, the ADS is computed and never sent to the client, it does not even exist as a data structure on the client. A proper number of attestation area descriptions from the computed ADS are sent together with the attestators' code (works both for occasionally renewed and renewed every time approaches). This approach requires substantial engineering effort to synchronize the next attestation requests to be sent to the client with the anticipated description of the areas to attest. With proper redesign of the attestators, this approach will very likely work with the existing code mobility protection without further extensions.
- **Dynamically populated pre-allocated ADS**. With this approach, the ADS space is pre-allocated in the client-side binaries but it is empty (or nearly empty). This approach simplifies the management of the attestator links, as the ADS position is known in advance. However, the mechanisms to dynamically update records in the pre-allocated ADS needs to be implemented by the code mobility protection (it is actually not supported). Therefore, this approach, which looks promising and effective, requires too much engineering.

For what concerns making the ADS mobile, support developed for WBC could be extended to this purpose, however, none of the previously presented approaches seems to require limited engineering and implementation effort.

Another design aspect that needs to be addressed concerns the server-side estimation of delays when receiving attestation responses. Currently, it is expected to receive a response in a given time, which is estimated roughly as a few seconds (it is not needed to have very precise time and platform information, as it is not a temporal-based attestation). When recording the result of an attestation verification, it is needed to subtract the delays introduced by the code mobility functions when estimating if an attestation response has been received in time.

6.4.4 Plan

Since making the renewable mobile does not introduce significant research issues, only engineering issues, we will support mobile attestators as specified in the DoW (i.e. attestators that can be downloaded once, relying on the code mobility framework), but renewable attestators (downloadable many times at run-time) will not likely be implemented during the project. Making the attestators renewable is thus an exploitation activity that can be performed outside the project by industrial partners interested in more effective remote attestation procedures and a better integration with code mobility and renewability.

Section 7 Prototypes released with D3.05

We report here the list of prototypes of online protections in WP3 released with D3.05.

7.1 Client/server code splitting

Owner : FBK

Technique ID: 10

Last stable version: 1.0.1

Development Version: 2.0.1 (both inter and intra-procedural variants)

Last delivery: 12/04/16

Testing: Tested with the SFNT use case (Diamante), license toy example

ACTC integration status: completed, ACTC step SLP06, documented in D5.08 Section 3.3

Annotations: stable, documented in D5.06

Logging: fully supported

Availability: /development and /testing for version 2.0.1, /stable for version 1.0.1

7.2 Code Mobility

Owner : UEL

Technique ID: 20

Last stable version: 08Mar16

Development version: 28Apr16

Last delivery: 28/4/16

Testing: The protection technique has been successfully tested both on the ASPIRE VM and on the ARM development board.

ACTC integration status: finalized, ACTC step BLP04, documented in D5.08 Section 4.5

Annotations: stable, documented in WD5.02.

Logging: fully supported. Code Mobility Server logs are collected in /opt/online_backends/code_mobility/mobility_server.log. Code Mobility client-side logging support relies on the ACCL logging functionality (accl.log file in working path)

Availability: Source code, support scripts, object files are provided for both development and testing branches on the SVN.

7.3 Static Remote Attestation

Owner: POLITO

Technique ID: 80-89

Last stable version: 1.0.0

Last delivery: 21/4/16

Testing: The protection technique has been successfully tested both on the ASPIRE VM and on the ARM development board. It has been tested on open sources applications (hello world, bzip2), on the NAGRA use case, and on the SFNT use case.

ACTC integration status: finalized, ACTC step SLP07, documented in D5.08 Section 3.5

Annotations: stable, documented in this deliverable in Section 3.2.1.

Logging: fully supported.

Availability: Source code, support scripts, object files are provided for in the /testing and /testing-log and /development branches on the SVN.

7.4 ACCL

Owner : UEL

ACCL (Technique ID: N.A.)

Version: 09Mar2016

Last delivery: 09 Mar 2016

Last stable version: 09Mar2016

Last development version: 29Apr2016

Implementation status: Completed

Testing: The component has been successfully tested both on the ASPIRE VM and on the ARM development board.

ACTC integration status: Completed

ACTC step name: COMPILE_ACCL

Annotations: There is no explicit annotation referring to the ACCL component; it is compiled and linked into the target application by the ACTC when at least one on-line protection is applied.

Logging support: client-side logging support creates a log file called accl.log into the working directory

Availability: Source code, support scripts, object files are provided for both development and testing branches on the SVN

7.5 ASCL

Owner : UEL

Technique ID: N.A.

Version: 09Mar2016

Last delivery: 09Mar2016

Last stable version: 09Mar2016

Last development version: 29Apr2016

Implementation status: Completed

Testing: The component has been successfully tested both on the ASPIRE VM and on the ARM development board.

ACTC integration status: Completed

ACTC step name: SERVER

Annotations: There is no explicit annotation referring to the ACCL component; it is compiled and linked into the target application by the ACTC when at least one on-line protection is applied.

Logging support: client-side logging support creates a log file called accl.log into the working directory

Availability: Source code, support scripts, object files are provided for both development and testing branches on the SVN

7.6 Reaction

Owner: GTO

Technique ID: N/A

Last stable version: Still in development environment only, no committed in the stable environment yet

Development version: 17/05/2016

Last delivery: 17/5/2016

Testing: The technique has been internally tested by GTO on the unitary tests.

Annotations: stable, documented in D5.01 and WD5.02, Section B.13

Logging: Not supported.

Availability: All source files of the Reaction Manager have been committed on the SVN server in /development/reaction_manager. The directory contains include files, C/C++ files, the makefiles to build the Reaction Manager and the properties file that contains the policies.

The Reaction Unit has been committed under source form in development/reaction_unit directory. It contains Python scripts, the Reaction Unit source code and unitary tests. A branch in development/ACTC contains the Python script that calls the Reaction Unit.

7.7 Anti-Cloning

Owner: NAGRA

Technique ID: 70-75

Last stable version: 1.0.0

Development version: none

Last delivery: 4/3/16

Testing: The techniques have been successfully tested on Aspire VM with the toy sample available in /development/anti-cloning/test. They have been deployed on the Nagra use case, precisely in the DRM plugin, and successfully activated on the ARM development board.

ACTC integration status: finalized, ACTC step SLP09, documented in D5.08 Section 3.5

Annotations: stable, documented in D5.01 and WD5.02, Section B.13

Logging: fully supported.

Availability: all source elements (C source code to add to the application, bash script to parse and replace annotations, Python script used as ACSL backend) are available in /development/anti-cloning on SVN.

7.8 Mobile WBC

Owner : NAGRA

Technique ID: N.A.

Last stable version: 08Mar16

Development version: 28Apr16

Last delivery: 28/4/16

Testing: The protection technique has been successfully tested both on the ASPIRE VM and on the ARM development board.

ACTC integration status: N.A.

Annotations: There are no explicit annotations for mobile WBC. This technique is achieved through code mobility annotations.

Logging: Same as code mobility.

Availability: Same as code mobility.

7.9 Mobile SoftVM Bytecode

Owner : SFNT/UGent

Technique ID: N.A.

Last stable version: N.A.

Development version: 18Apr16

Last delivery: 18/4/16

Testing: The protection technique has been successfully tested both on the ASPIRE VM and on the ARM development board.

ACTC integration status: Completed

Annotations: Stable. Documented in this deliverable, Section 6.3.2.1.

Logging: Fully supported.

Availability: Object files are provided on the SVN.

Section 8 List of Abbreviations

AC	Anti-Cloning
ACCL	ASPIRE Client Communication Logic
ACTC	ASPIRE Compiler Tool Chain
ASCL	ASPIRE Server Communication Logic
ASCL-WS	ASPIRE Server Communication Logic, Web Socket based implementation
ADS	Attestation Data Structure
ADSS	ASPIRE Decision Support System
AID	Application Identifier
API	Application Programming Interface
ASPIRE	Advanced Software Protection: Integration, Research and Exploitation
AWPCFG	Diablo's Augmented Whole-Program Control Flow Graph
BLPxx	Binary-level software processing step nr. xx
CFG	Control Flow Graph
DB	Data Base
DDS	Delay Data Structure
DoW	Description of Work
JSON	JavaScript Object Notation
RA	Remote Attestation
REU	Reaction Enforcement Unit
RM	Reaction Manager
RWU	Reaction Waiting Unit
VM	Virtual Machine
WBC	White Box Crypto
WP	Work Package