SEVENTH FRAMEWORK PROGRAMME

Advanced Software Protection:
Integration, Research and Exploitation

D3.01

Preliminary Online Protections 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 / D3.01 / 1.01
WP and tasks contributing:	WP3 / Tasks 3.1, 3.2
Due date:	Oct 2014 – M12
Actual submission date:	25 November 2014
Responsible Organization:	UEL
Editor:	Paolo Falcarin
Dissemination Level:	Public
Revision:	1.01

Abstract:

This deliverable reports about the project activities on online protections, developed within Task 3.1 (Client Server Code and Data splitting) and Task 3.2 (Remote Attestation). We present the current status of four online protection techniques in ASPIRE: code mobility, client-server code splitting, implicit remote attestation and anti-cloning.

Keywords:

Code mobility, remote attestation, client-server code splitting, anti-cloning, online protections

Editor

Paolo Falcarin (UEL)

Contributors (ordered according to beneficiary numbers)

Bart Coppens, Bjorn De Sutter (UGent)

Cataldo Basile (POLITO)

Brecht Wyseur (NAGRA)

Mariano Ceccato, Andrea Avancini (FBK)

Alessandro Cabutto (UEL)

Andreas Weber (SFNT)

The ASPIRE Consortium consists of:

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

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

Disclaimer

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

The information in this document is provided as is and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

Document Revision History

Version 1.0 31 Oct 2014	Original deliverable submitted to the EC.
Version 1.01 24 Nov 2014	Overlooked 'XXX' placeholders for references to other deliverables are replaced by actual references on page 9/30. No other changes.

Executive Summary

This document presents the first results obtained for the ASPIRE online protections that aim to improve the resilience of traditional offline protections, such as information hiding, obfuscation, anti-tampering.

In the ASPIRE vision, online protection techniques can prevent (static) code analysis by means of two protection techniques: *client/server code splitting*, and *code mobility*. *Client/server code splitting* removes code from the deployed software program, and lets it run a trusted server. *Code mobility* allows installing sensitive parts of a program at run time by downloading binary code blocks from a trusted server.

Other online protections aim at detecting the attack as soon as it is enacted. These include *remote attestation* and *anti-cloning. Implicit remote attestation* (IRA) (implicit, because of limited use of local attestators in the client code) collects relevant run-time data on the client program execution to be sent to a remote verifier for tamper detection. Anti-cloning aims at preventing the attacker from run multiple copies of the same program, by 'locking' the software with a remote server.

The four online protections introduced in this document are still in design or prototyping stage and plans of future developments are also discussed.

The preliminary architecture of the code mobility protection assumes that only whole procedures can become mobile, and that downloaded mobile code blocks are allocated dynamically, at addresses determined at run time. Control transfers into and out of mobile code blocks will be rewritten in a Diablo tool to overcome the issue that even relative addresses are not known until code has been downloaded from the server. For that, we designed a Binder and Downloader component. The invocation of the Binder is designed around the necessary data structures, including look-up tables, to avoid unnecessary overhead once mobile code is downloaded. We also designed an extension to the offline SoftVM-based protection to allow bytecode to become mobile. The next steps regarding code mobility are the implementation of the first prototype tool support.

For client-server code splitting, we will rely on the barrier slicing technique. A source-level tool flow has been designed to implement barrier slicing. The requirements with respect to control flow and data flow analyses have been identified, together with the required source-to-source code transformations. Those include the removal of sensitive variable definitions and uses, and transformations of the barrier variable operations. At the current point in time, the tool prototype is working but it is not yet integrated into the ASPIRE Compiler Tool Chain.

As the work on IRA and anti-cloning techniques started very recently, i.e., in M10, only preliminary design aspects have been researched so far.

The important research issues to solve to make IRA usable in practice is identifying a set of properties/peculiarities the remote IRA Verifier can observe to establish the client-side application integrity. Currently, we have identified two approaches (decision on usability still pending): CFG-based IRA and remotely watched invariants monitoring that we will research. With respect to IRA, we refined the remote attestation reference architecture of deliverable D1.04. The detailed design and implementation of this technique is planned at a later phase in the project.

For anti-cloning, this deliverable describes the overall architecture and the related code annotations. The detailed design and implementation of this technique is planned at a later phase in the project.

Contents

Section 1 Introduction **1**
1.1 Document Structure 1
1.2 Related Work 1
Section 2 Code Mobility **4**
2.1 Introduction 4
2.1.1 Basic components 4
2.1.1.1 Code Mobility Server Component *4*
2.1.1.2 Downloader Component *5*
2.1.1.3 Binder Component *5*
2.2 Design refinement 5
2.2.1 Assumptions 5
2.2.2 Non-Mobile Client Application Code 5
2.2.3 Mobile Code 8
2.3 Mobile bytecode 8
2.3.1 Required design changes 9
2.4 Planning 10
Section 3 Client/Server Code Splitting **11**
3.1 Problem Definition 11
3.2 The Protection 12
3.2.1 Background 12
3.3 Design 14
3.3.1 Client/Server 14
3.3.2 Structure of the Tool 15
3.3.2.1 Annotations for Client/Serve Code Splitting *16*
3.3.2.2 SLP01 – CodeSurfer Analyzer *17*
3.3.2.3 SLP02, SLP03 – Client Generation and Server Code Generation *18*
3.3.3 Current Implementation 20
3.4 Plan 21
Section 4 Implicit Remote Attestation **22**
4.1 Design 24
4.2 Plan 25
Section 5 Anti-Cloning **26**
5.1 Introduction 26

5.2 Design 26
5.2.1 Architecture 26
5.2.2 Annotations 27
5.3 Plan 27
Section 6 List of Abbreviations 28
Bibliography 29

List of Figures

Figure 1 - Code Mobility Reference Architecture 4
Figure 2 - Call to procedure *f* in the original control flow 6
Figure 3 - Calling function *f* (passing through Code Mobility) 6
Figure 4 - Mobile function code layout 7
Figure 5 - Calling function *f* passing through already downloaded mobile code 8
Figure 6 - Bytecode dumping process 9
Figure 7 - Modified stub 10
Figure 8 - Running example of C code before and after an attack 11
Figure 9 - C code snippet and the SDG with the barrier slice (nodes in red). 13
Figure 10 - Original application to protect with ASPIRE 14
Figure 11 - New client/server application after splitting 14
Figure 12 - Architecture for client-server code splitting 15
Figure 13 - Tool flow for client/server code splitting 16
Figure 14 - Pseudo-code of the barrier slicing algorithm 17
Figure 15 - Annotated running example 18
Figure 16 - Code containing definitions of sensitive variable dd2 19
Figure 17 - Protected code for the example of Figure 16 19
Figure 18 - Code containing uses of sensitive variables dd1 and dd2 19
Figure 19 - Protected code for the example in Figure 18 19
Figure 20 - Protected code for the example of Figure 18 (server side) 20
Figure 21 - Annotated code for barrier variables 20
Figure 22 - Protected code for the example of Figure 21 20
Figure 23 - Remote Attestation, reference architecture. 22
Figure 24 - The architecture of the IRA Protection Tool. 24

Section 1 Introduction

Section Author:

Paolo Falcarin (UEL)

The analysis of binary code is a common step of Man-At-The-End attacks to identify code sections crucial to implement attacks, such as identifying private key hidden in the code, identifying sensitive algorithms or tamper with the code to disable protections (e.g. license checks or DRM) embedded in binary code or use the software in an unauthorized manner.

The main objective of WP3 is to realize a paradigm shift in software protection by designing and developing network-based protections to enforce software protection.

The goal of Task 3.1 in the ASPIRE project is to provide effective techniques to prevent code analysis and tampering by removing code from the deployed software program, and let it run a trusted server or installing it at run-time by downloading binary code blocks from a trusted server.

The goal of Task 3.2 in the ASPIRE project is to develop effective techniques to prevent and detect tampering by collecting relevant run-time data on program execution, to be analysed server-side to detect malicious behaviour.

This deliverable presents the initial outcome of these two tasks to implement solutions for code mobility and client-server code splitting (Task 3.1, started at M7), as well as for remote attestation and anti-cloning (Task 3.2, started at M10).

This document is the first deliverable of WP3 about online protections. The presented working prototypes will be further improved and extended during the development of the project. Moreover, they will be integrated in the ASPIRE Compiler Tool Chain and integrated with the other protection strategies that will be delivered by ASPIRE.

Deliverable D3.02 (M18) will provide preliminary working prototypes and support of online protections, and in Deliverable D3.03 (M24) will provide Client-server code splitting and code mobility support and the status of online protections will be described in D3.04 (M24), the Intermediate Online Protections report.

Later (M30), D3.05 and D3.06 will provide information about the other techniques, while D3.07 and D3.08 (M33) will describe the renewability features added to the online protections as a result of Task 3.3 (to be started at M19).

Finally D3.09 (M36) will describe ASPIRE online protections, their integration with the toolchain and how they will be applied to the case studies.

1.1 Document Structure

Preliminary architectures and designs of working prototypes for code mobility (Section 2) and client-server code splitting (Section 3) are presented, while more recently started work on (implicit) remote attestation (Section 4) and anti-cloning (Section 5) are briefly introduced at architecture and design level.

1.2 Related Work

One of the main goals of software protection is to prevent code from being observed and analysed, and then eventually illicitly modified and tampered with.

To protect against code analysis, developers usually try to make reverse-engineering harder, by applying different obfuscating transformations [Col09]; attackers can use binary code inspection tools like IDA Pro [IDA] and binary instrumentation tools [Mad05] to extract run-time information such as execution traces and memory dumps.

To protect against illicit modifications, anti-tampering approaches are utilized to detect when code has been tampered with and to react by stopping or delaying program execution. Tamper-resistant software typically uses built-in integrity checks to detect code tampering by guarding the code being executed [Cha02] or by checking that the flow of control through the program confirms to the expected flow [Che02].

However, when these techniques are used to protect standalone applications, their security is limited, because the expected checksum values and the reaction mechanism is hard-coded in the application itself, where it can be analysed and altered.

Online protections techniques aim at extending state-of-the-art static protection techniques (described in more detail in the project deliverable of WP2) with network-based protections.

To protect against code analysis, the online protection client-server code splitting removes sensitive code from the binary code and executes it on a trusted server, while the code mobility framework delivers such binary code to the client at run-time.

To protect against illicit modifications, online protections such as implicit remote attestation and anti-cloning are going to be developed in the project.

Remote attestation is static when the identity and authenticity proof are computed on static properties, such as program binaries or a priori knowledge of the allocated memory pages, while it is dynamic if the properties used to check the authenticity depend on the program's execution.

Remote attestation can be based on client-side computations, e.g., by checking the execution traces [Mon99] or by verifying assertions [Cec07]. The Tisa system [Raj08] checks a set of properties on execution traces specified in terms of linear temporal logic expressions. Dymo [Gil11] tracks the run-time integrity by means of cryptographic hashes over executable regions in the process' address space, also detecting library additions. ReDAS [Kil09] automatically extracts and checks a set of monitoring properties, the invariants, from the application; a modification of dynamic objects would change the related invariants thus providing evidence of integrity violation.

Previous works proposed to dynamically replace the code fragments that generate the attestations [Fal06, Sca08]. With the Integrity Verification Proxy [Sch12b], a client-side service mediates connections with the server and locally implements some functionalities of the remote verifier, thus eliminating the need for continuous remote attestation.

Static remote attestation is considerably weaker than dynamic attestation, as the former cannot protect from cloning attacks based on the simultaneous execution of a correct version of the program along with a tampered one [Few08,vOo05].

However, the dynamic techniques in literature usually check dynamic properties against pre-computed checksums: they do not monitor the "current" execution of the program because they do not detect which part of the program is being executed. That makes them weak.

In the ASPIRE protection consisting of implicit remote attestation, a remote server will monitor the execution of a client by keeping track, through frequent communication, of dynamic properties of the client's internal state. Authenticity verdicts will then be based on that state instead of the attestation proofs easily computed by local code guards.

Several online protections use dynamic code replacement to periodically replace the copy of the program running on the untrusted machine with the goal of limiting the amount of time that the attacker has to reverse engineer the application. The replacement may be implemented for the functional part of the program, and/or for the protection techniques used

to protect it [Jak01]. Collberg et al [Col08,Col12] and Falcarin et al [Fal11] propose the continuous replacement of Java and binary code respectively, in which the remote trusted entity frequently sends a set of new code fragments to the untrusted machine.

Some research prototypes implemented dynamic replacement of protection code using code mobility features offered by dynamic aspect-oriented platforms [Fal06] or by ad-hoc JVM extensions [Sca08], and the time interval for replacement can vary in order to avoid that attackers exploit the replacement rate.

Section 2 Code Mobility

Section authors:

Paolo Falcarin, Alessandro Cabutto (UEL), Bjorn De Sutter, Bart Coppens (UGent), Andreas Weber (SFNT)

2.1 Introduction

Code Mobility is an online technique that aims to overcome the drawbacks of local protection techniques by introducing a client – server architecture. Its reference architecture is fully described in deliverable D1.04 Section 3.4 and is summarized in Figure 1.

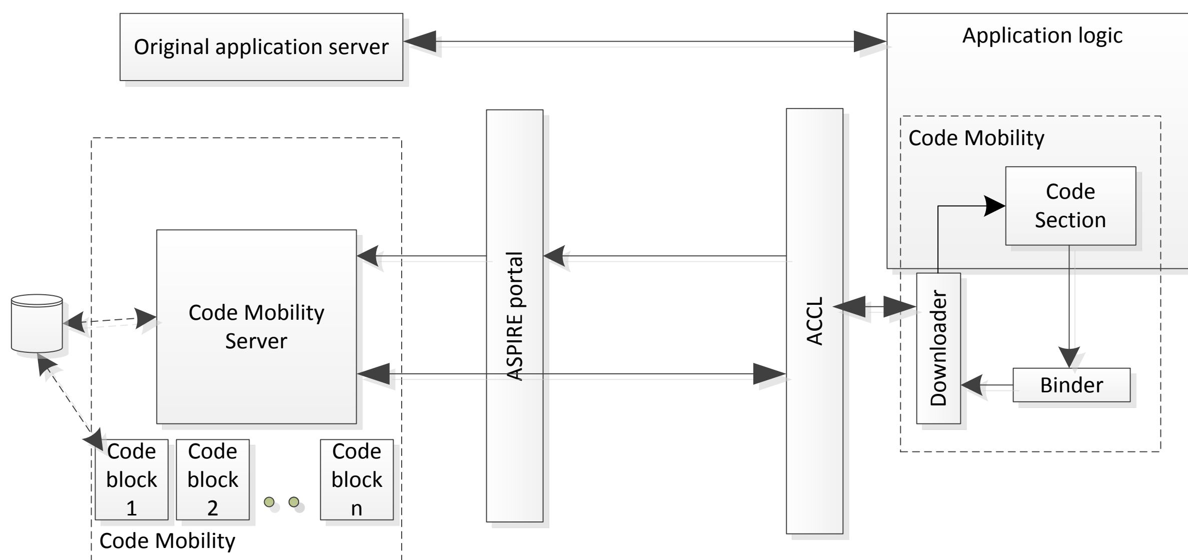

Figure 1 - Code Mobility Reference Architecture

In Figure 1 (on the right) two components are highlighted in the Application Logic: the Downloader and Binder. The Binder and Downloader components will be compiled from separate source code files, and then linked into the application. This is very similar to the way the SoftVM code is linked into an application, as presented in deliverable D5.01 Section 9.4.

Besides the insertion of the Downloaded and Binder, the application code itself also needs to be transformed. Wherever code in the original application directly transfers control to some mobile code fragment, such transfers need to be redirected via the Binder (and possibly the downloader) to ensure that the mobile code is downloaded and bound before it is actually executed. Moreover, because the mobile code fragments are stored in locations determined at run time rather than at link time, many occurrences of PC-relative address computations need to be adapted as well. All of this rewriting will be performed in a binary-level rewriting step in the ASPIRE Compiler Tool Chain (ACTC), similar to the SoftVM integration step BLP03 presented in D5.01 Section 9.4.

2.1.1 Basic components

2.1.1.1 Code Mobility Server Component

The Code Mobility Server is a server-side component able to serve code blocks (kept in its local storage) on demand to remote clients. It runs on a trusted node so that it is assumable that it cannot be tampered with.

2.1.1.2 Downloader Component

The *Downloader* is a client-side component able to fetch binary code blocks from the trusted server as required by the application control flow at run time. This component relies on the ASPIRE Communication Logic.

2.1.1.3 Binder Component

The client-side Binder component is in charge of invoke the Downloader when required. The Binder is invoked by the application when the control flow reaches a mobile code block. If that block has not been downloaded from the server yet, the Binder asks the Downloader to retrieve the requested missing code block. The Downloader queries the Code Mobility server in order to obtain such a block and finally the Code Mobility Server sends the proper block back to the Downloader. At a conceptual level, this process was already presented in deliverable D1.04 Section 3.4.3.2.

After the fetch process the Binder places the block in memory and makes sure that the just downloaded block will not be downloaded again, reducing the overhead effort introduced by the protection technique.

Eventually the Binder redirects the control to the entry point of the downloaded code, where the application can continue normally.

2.2 Design refinement

Since the initial reference architecture for Code Mobility was designed (as documented in deliverable D1.04), this design has been refined. The refined design presented below is the result of a collaborative analysis and feasibility study building on UEL's background in Code Mobility (an existing Windows x86 mobile code prototype was partially ported to Linux as a proof of concept) and on UGent's ARM architecture and binary transformation expertise.

2.2.1 Assumptions

Since the Code mobility Reference architecture was first presented in D1.04, a number of design options have been revised. The overall architecture has not changed, but the placement and forms of mobile code blocks has been redesigned.

First of all, mobile code blocks coming from the Mobility Server will not be placed in a statically known location in the binary, but will instead be placed in dynamically allocated memory. So the location of what we called Mobile Code Space in deliverable D1.04 will not be fixed. This implies that the mobile code needs to be position-independent code (PIC) that can be relocated dynamically, and independently from the non-mobile code part of the binary or library. Thus, indirections need to be inserted in the transformed code to deal with these variable code locations, both in the static, non-mobile parts of the client application and in the mobile code. Fortunately, only local code transformations are required for this: instructions will be replaced with small code snippets that can deal with the a priori unknown addresses at which the code has been loaded.

Secondly, for sake of simplicity, we will initially only consider entire procedures to become mobile code. This offers the advantage that the mobile code blocks have one entry point only, i.e., the entry point of the procedure. This significantly simplifies the implementation of the Binder and its bookkeeping data structures. In the remainder of this document, we can therefore use the term mobile procedures. In a later phase, it is possible that we also transform code regions that do not necessarily coincide with procedure boundaries.

2.2.2 Non-Mobile Client Application Code

In the original client application, procedure calls to mobile procedures need to be transformed such that

1. upon the first execution of a call to a mobile procedure, the Binder and Downloader components are properly invoked in order to obtain the code from the server;
2. upon subsequent calls to the same mobile procedure, the control is immediately transferred to the already downloaded mobile code. By avoiding going through the Binder again, the performance overhead of mobile code can be limited.

This principle is illustrated in Figure 2, Figure 3, and Figure 5.

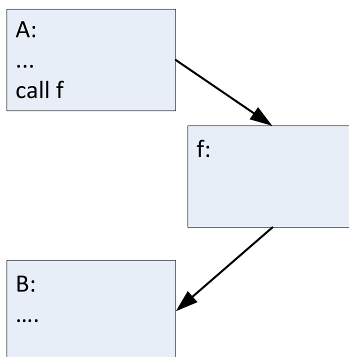

Figure 2 - Call to procedure *f* in the original control flow

Figure 2 shows the original control flow without mobile code. Procedure f is selected to become mobile. In the transformed program, shown in Figure 3, Diablo inserted a look-up table with procedure pointers. Look-up table accesses are depicted with dashed arrows whereas control flow transfers are depicted with regular arrows.

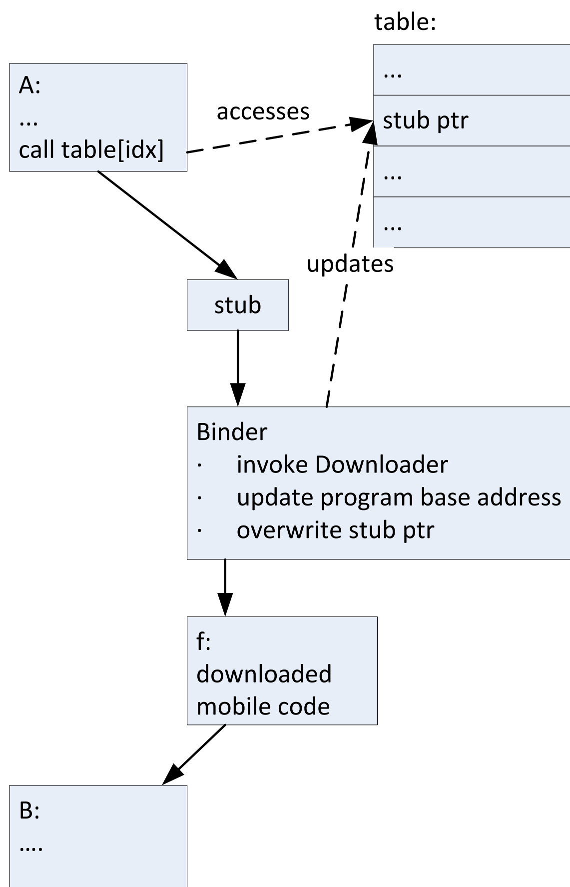

Figure 3 - Calling function *f* (passing through Code Mobility)

The pointers in the look-up table either point to stubs that invoke the Binder to start the mobile code downloading process, or they point directly to the already downloaded code. All calls to mobile functions are transformed into a code snippet consisting of a table lookup and an execution control redirection to the address loaded from the table.

Initially, when the called mobile function f() has not yet been downloaded and bound, the address in the look-up table is that of a stub that invokes the Binder. This process is shown in Figure 3. This stub calls into the Binder, providing as argument the index at which this stub is installed in the look-up table. This index is then used as an identifier of the mobile function to be downloaded. The Downloader component is then invoked to retrieve the mobile (PIC) version of the function's code body from the Code Mobility Server, and stores this code body in a dynamically allocated buffer.

Apart from being a PIC, mobile version of the original procedure body, this mobile code block is prepended with a small instruction sequence that ensures that the proper execution context is restored before the actual procedure body is executed. More concretely, when the original direct call to f() (as in block A in Figure 2) was replaced by a table look-up and an indirect call (as in block A in Figure 3) for which registers had to be freed by spilling them onto the stack, the registers need to be restored.

Whenever the mobile code body of f() needs to access non-mobile code or data, or other mobile code, for example to access statically allocated data or to invoke a non-mobile procedure, the mobile code needs to be able to compute the addresses of that code or data. Because the non-mobile client-side code can itself be PIC code that is relocated when it is first loaded (e.g., because the protected client-side software is a dynamically linked library and because address-space layout randomization is deployed on the client device), the mobile code needs to know the starting address at which the non-mobile code and data have been loaded to compute the necessary addresses. To this end, a data location is reserved in the mobile code block to contain this starting address. When the mobile block has been retrieved and stored in memory, the Binder fills in this value. The Downloader also sets the permissions of the memory page that contains the buffer to make it executable. The final layout of the mobile code block as allocated in memory is shown in Figure 4.

Finally, the Binder updates the entry in the pointer look-up table by overwriting the address of the stub with the address of the downloaded code, after which it redirects the control to this code, and normal code continues.

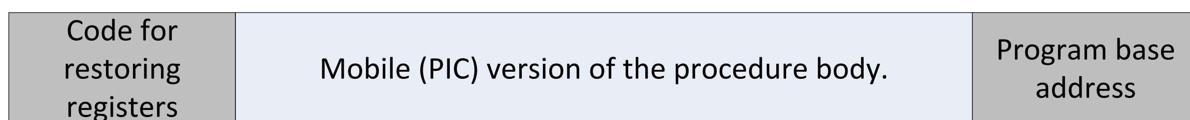

Figure 4 - Mobile function code layout

Subsequent calls to the already downloaded procedure f() then proceed as indicated in Figure 5. Since the Binder has already updated the pointer in the look-up table at the used index to let it point to the downloaded code, the inserted code snippet (in block A in Figure 5) now loads this procedure pointer and thus transfers control immediately to the previously downloaded mobile code. So for subsequent calls, the overhead is limited to the table look-up, and the necessary spilling and restoring of registers.

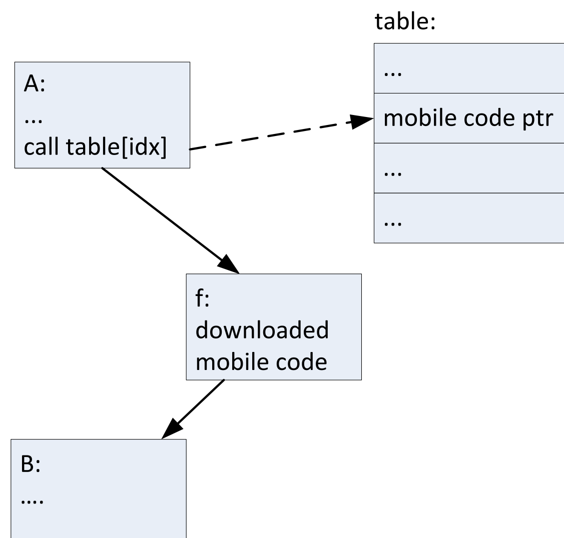

Figure 5 - Calling function *f* passing through already downloaded mobile code

This design in which mobile code is allocated in a dynamically linked Mobile Code Space limits the run-time overhead and provides stronger protection than the design as originally presented in D1.04 Section 3.4. First of all, the addresses at which the mobile code is downloaded will differ from one run of the program to another. This makes all kinds of dynamic attacks mode difficult. Secondly, almost all the necessary support is already available to support flushing of the Mobile Code Space. To do that, it will suffice to free the allocated memory of mobile code blocks, and to restore the addresses in the look-up table to their original values, i.e., the stub addresses. Once this is implemented (later in the project), it will allow us to (1) make sure that not all mobile code is present at once, (2) to let multiple different mobile code blocks occupy the same memory addresses during a single run of a program. The fact that addresses in the program's address space then no longer map onto instructions in a one-to-one mapping, also complicates many dynamic and hybrid attacks, e.g., because many tools such as IDA Pro are engineered around the central notion that every code byte and address corresponds to at most one instruction.

2.2.3 Mobile Code

Mobile code, i.e., instructions in the body of mobile functions, can refer to addresses in the non-mobile part of the application for two reasons:

1. Control flow is redirected to non-mobile code, such as when non-mobile functions are called from within mobile functions
2. Addresses of non-mobile code or data are computed

Because the non-mobile code can be loaded at arbitrary addresses, the mobile code needs to compute the addresses of the referred memory locations dynamically. We will support this by appending a placeholder to the mobile code, in which the Binder will write the address at which the client application is loaded. All instructions in the mobile code that refer to or compute addresses in the non-mobile code are rewritten to compute the addresses based on their offset relative to the start address of the client application, as it is stored in the placeholder.

2.3 Mobile bytecode

The client-side binary code splitting protection in ASPIRE is an offline software protection technique in which binary code is translated to custom bytecode, and a so-called SoftVM is linked into the client application to interpret the bytecode. This protection is currently being developed by the project partners SFNT and UGent.

A possible way to apply Code Mobility to a SoftVM-protected application is to make the bytecode mobile. Bytecode is actually produced by a software component called X-Translator. This component is invoked during BLP02 phase of ACTC to produce BC03: bytecode and stubs .o files. See Section 9.3.1 of deliverable D5.01 and Section 3 of deliverable D2.03 for further details.

2.3.1 Required design changes

Due to its design constraints the bytecode is not relocatable so, even if mobile, the bytecode should be executed from a fixed location.

For that reason, the standard X-Translator component has to be modified so that it can emit a sort of placeholder code (filled in by the appropriate quantity of random data) instead of actual bytecode. The placeholder will then be linked into the protected application in step BLP03 of the SoftVM protection instead of the bytecode (see deliverable D5.01 Section 9.4).

As shown in Figure 6 the bytecode itself is stored in a repository together with some additional information:

- An application-wise unique randomly generated identifier;
- The bytecode runtime offset address;
- The bytecode length.

The complete repository is marked with a unique application identifier and kept for future use (bytecode delivery) by the Code Mobility Server.

Stubs invoking the SoftVM must then call into the Code Mobility Binder component as shown in Figure 7. This Binder component is not the standard one already descripted before but it is a specific “SoftVM-aware” version able to retrieve bytecode instead of native code. Moreover the downloaded blocks have to be placed in the right fixed position in the code section, where they will overwrite the aforementioned placeholder data. After that overwriting took place, the control flow can proceed as in the original SoftVM approach.

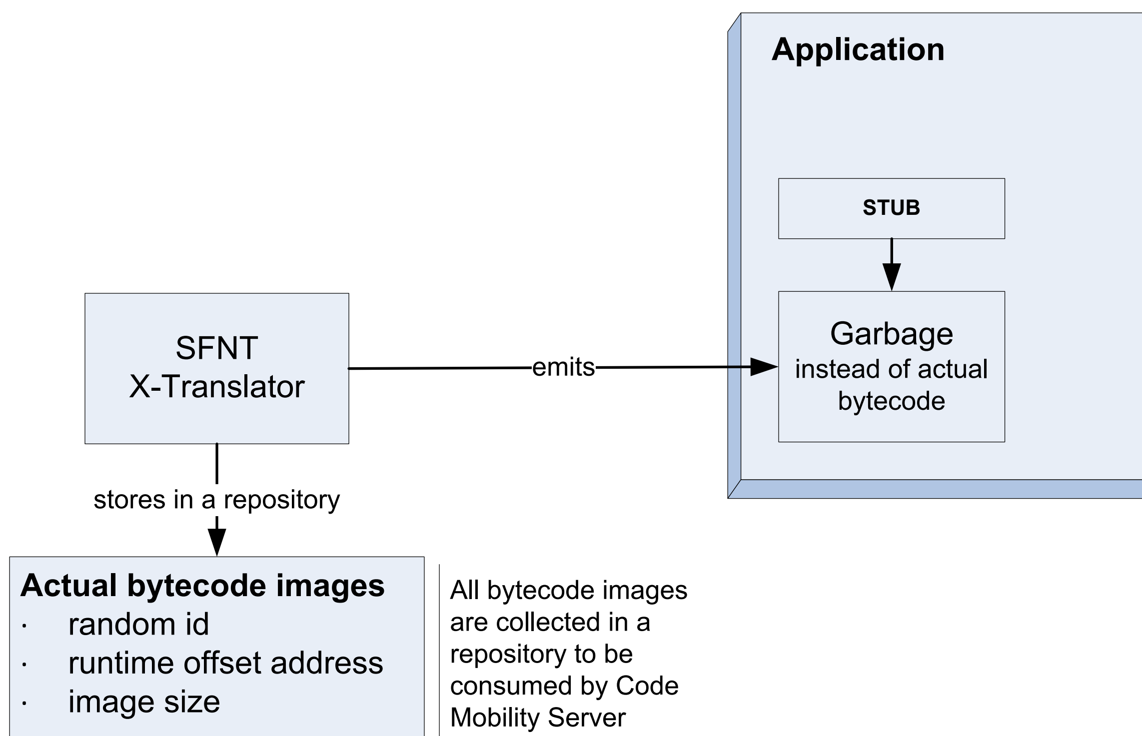

Figure 6 - Bytecode dumping process

```
STUB

…

Binder_SoftVM(application_uuid, 32_bits_chunk_id)

…

vmExecute (...)

...
```

Figure 7 - Modified stub

2.4 Planning

The next months will be dedicated to work on the refined design and implementation. Under these considerations, we envision the following implementation plan:

- **M18**:
  - Complete implementation (still not integrated in the ACTC)
  - Experimentation on simple examples and on the available use cases developed by industrial partners (deliverable D3.02);
- **M24**:
  - First integration in the ACTC (D3.03 and D3.04);
  - Improvement of code mobility framework, still not integrated in the tool chain;
  - Integration in the ASPIRE Decision Support System (ADSS);
- **M30:**
  - Complete integration in the ACTC (deliverables D3.05 and D3.06).

Section 3 Client/Server Code Splitting

Section Authors:

Mariano Ceccato, Andrea Avancini (FBK)

As software developers, one of the most difficult challenges is ensuring that a malicious user cannot tamper with our application to alter its regular behavior. Users of applications that run on client devices must be considered a primary source of threats, since they could be interested in making the program execute in a way that might give them benefits of various types. As client users, attackers have full control of the application's execution environment and they have the possibility to use any sort of methods to attack the program, from running dynamic or static analysis tools to reverse-engineer the entire application.

To reduce the attack surface that can be potentially targeted by attackers, this document describes how to implement the so-called *client/server code splitting* [ZHA03] as protection technique within the ASPIRE project. The goal of this protection technique is to remove portions of the application that are considered unsafe, attackable by a malicious user, and to move them on a secure server where they run in a safe, trusted environment. To be deployed, this technique requires the identification, by applying *barrier slicing,* of the part of the application that needs to be moved, i.e., split off, as well as the transformation of the original code to make the new client and the trusted server communicate for synchronization and the exchange of data.

This section is structured as follows: Section 3.1 introduces the problem of protecting the code of an application from malicious users, while client/server code splitting is described in Section 3.2. Section 3.3 depicts the design of the protection and some implementation details, while Section 3.4 proposes the intended work plan.

3.1 Problem Definition

An attacker may want to alter a target application by applying several different attacks, with the ultimate goal of gaining personal benefits. When the software under attack runs on a client machine, the attacker can use every technique and any tool to tamper with the application without restriction.

```
1       year2 = tm.tm_year+1900;
2       for (i = ref; i < year1; i++) {
3               if (i % 4 == 0)
4                       dd1 += 1;
        }
5       dd1 = calculate_original(dd1);

6       dd2 = 0;
7       for (i = ref; i < year2; i++) {
8               if (i % 4 == 0)
9                       dd2 += 1;
        }
10      dd2 = calculate_current(dd2);
11      if (dd2 - dd1 > 30)
12              printf("Fail\n");
        else
13              printf("Ok\n");
```

```
1       year2 = tm.tm_year+1900;
2       for (i = ref; i < year1; i++) {
3               if (i % 4 == 0)
4                       dd1 += 1;
        }
5       dd1 = calculate_original(dd1) + CHEAT;

6       dd2 = 0;
7       for (i = ref; i < year2; i++) {
8               if (i % 4 == 0)
9                       dd2 += 1;
        }
10      dd2 = calculate_current(dd2);
11      if (dd2 - dd1 > 30)
12              printf("Fail\n");
        else
13              printf("Ok\n");
```

Figure 8 - Running example of C code before and after an attack

In the running example of Figure 8 we have a piece of C code taken from a non-protected

application that has been chosen as a case study to test the preliminary implementation of client/server code splitting. The case study, called license checker, is a small routine devoted to check the validity of a software license number in order to activate or deactivate a software component. The serial number contains the date corresponding to when the license was emitted, and its validity is meant to expire 30 days after the emission.

Sensitive variables (i.e., the ones that can be attacked) are those that hold the license's emission date and the current date. An attacker may want to tamper with these values to make an expired license last longer. The code fragment on the left of Figure 8 is the original code of this simple license checking algorithm: The purchasing date is calculated and stored in variable *dd1* at line 5, while the current date, after being computed, is saved in variable *dd2* at line 10. The two variables are then compared at line 11 to perform the temporal check. The attacker could try to tamper with the definition of variable *dd1* at line 5 by adding a proper value CHEAT to fool the license check algorithm and to illegally validate his/her license, which would have been expired under normal circumstances. This tampering is shown on the right of Figure 8. We can define those variables that influence the intended behavior of an application when they are attacked by malicious users as *sensitive*, like variables *dd1* and *dd2* in the example. It are then these variables that must be protected.

3.2 The Protection

The protection proposed in this section, client/server code splitting, is intended as a means to protect software by identifying portions of its code in which sensitive variables reside, by slicing these portions away, and by moving them from an untrusted client *C* to a trusted server *S*, in order to prevent any tampering attack.

Moving entire functions is not always feasible, either because a function might be larger than the portion of code that needs to be transferred, but also because of the side effects that might arise. For example, the function to move might need to modify some global variables that should remain on the client. This means that also data and control dependencies should be carefully taken into account, to ensure that original functionalities are unmodified in the new, protected configuration.

Under these assumptions, program slicing can be considered a suitable approach and, more specifically, a *barrier slicing* [CEC07] algorithm can be applied to limit the portion of code to be moved onto the newly created server. Furthermore, moving a portion of the client on the server introduces the need for the two new components to communicate, i.e., to exchange requested values and to synchronize the execution of the sliced code. Communication works in both directions, since the server also requires values from the client to keep the execution of the slice synchronized.

3.2.1 Background

A barrier slice is based on the concept of (backward) slice [WEI81]. Let C = (x, V) be a slicing *criterion*, where *x* is a statement in a program P, usually expressed as the number of line of code that identifies the statement in the source, and *V* is a subset of variables in P. A backward slice *s* on a given criterion, represented by a program variable at a specific statement, can be computed as a sub-program P' which is equivalent to the original program P with respect to the criterion. The slice *s* on criterion C includes all the statements that directly or indirectly hold data or control dependencies on the variables in V at statement x.

A barrier slice [2] is the slice on the program P, where some statements are considered as "barriers": the computation performed at those statements is considered not relevant for the slice itself, so the barriers are excluded from the slice and they block the propagation of data or control dependencies when the slice is calculated. A barrier slice can be computed by

stopping the computation of a regular backward slice whenever one of the barrier statements is reached.

As proposed by Krinke et al. [KRI03], let the SDG of the program P be the *system dependence graph* of P, a representation of the program where nodes represents statements and predicates, while edges carry information about control and data dependencies between statements. Then, the barrier slice Slice#(C, B) of an SDG G = (N, E) for the slicing criterion C ⊆ N with the set of barrier nodes B ⊆ N consists of all nodes on which a node n ∊ C (transitively) depends via an inter-procedurally realizable path that does not pass a node of B, as follows:

$$Slice\#(C,B) = \left\{ m \in N \;\middle|\; \begin{array}{c} p \in m \longrightarrow_R^* n \wedge n \in C \\ \wedge\; p = \langle n_1 \ldots n_l \rangle \\ \wedge\; \forall\, 1 \leq i \leq l : n_i \notin B \end{array} \right\}$$

Example On the left, Figure 9 contains a simple snippet of C code. It takes two variables, *x* and *y*, and it performs few mathematical operations. By applying the barrier slicing algorithm presented earlier on the code in Figure 9, with variable *x* at line 6 as slicing criterion C and statements at line 1, 4 as set of barriers B, we obtain the barrier slice consisting of the red nodes of the SDG of the program depicted on the right of Figure 9. The slice is expressed in terms of nodes and edges of an SDG, while red and blue arrows represent data and control dependencies respectively. Dependencies with black crosses are those that are not traversed because of the presence of the barriers. If we consider the code, the criterion is represented by statement *x--* (line 6), on variable x, while the barriers are statement *x++* at line 4 and statement *x = 1* at line 1. To calculate the barrier slice, we start from line 6 (the statement is included in the slice) and we traverse any control or data dependency that reaches the statement in a backward fashion. Since line 5 holds a control dependency on statement at line 6, statement *while(x > 0)* is also added to the slice. From this point, we can still traverse other dependencies (for example, *x++* at line 4 holds a data dependency on line 5 and the same does statement at line 1), but since the barrier is reached, dependencies are not further propagated through line 4 (and line 1). So eventually, the statements at line 5 and 6 constitute the resulting slice.

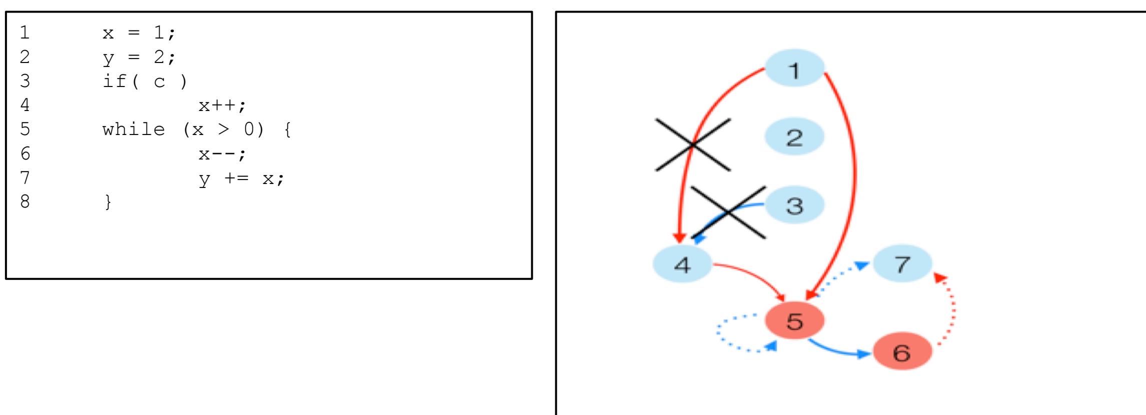

Figure 9 - C code snippet and the SDG with the barrier slice (nodes in red).

3.3 Design

3.3.1 Client/Server

We can define the program *state* of a program P as a map *s* : *Var* → *Values* that associates each variable in P with a possible value. Let *Non-sensitive* ⊆*Var* be the subset of the variables in P that are considered *already secure* by construction, like those variables that are not involved in sensitive computations, while *Sensitive* = *Var*\ *Non-sensitive* is the subset of the *security critical* variables, those variables that can be tampered by an attacker to interfere with the normal behavior of P.

The original application P runs in an untrusted client as depicted in Figure 10, exposing its sensitive variables, i.e., the red shape in the figure, to attackers. The green part, the non-sensitive state, represents the program variables that are insensitive to attackers, since they cannot be tampered with or their malicious modifications does not have any role in altering the regular behavior of the application.

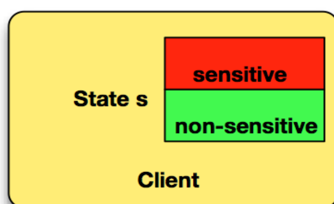

Figure 10 - Original application to protect with ASPIRE

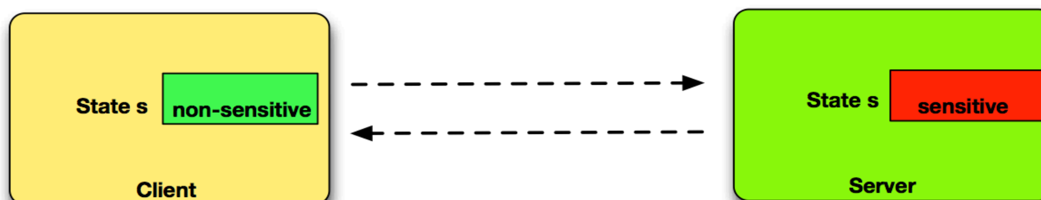

Figure 11 - New client/server application after splitting

The sensitive state of the application must be protected. By applying client/server code splitting, the new application P' is generated with the creation of a new client *C* by removing the sensitive part of the original client application P and by moving it on a trusted server *S* as shown in Figure 11. In this new architecture, the client still contains the subset of the non-sensitive variables, while its original sensitive set of variables is moved onto the server to be handled in a trusted environment. Any reference to sensitive variables is removed from the client.

Since values of sensitive variables are still required for the computation, a communication protocol must be established between client and server to ensure the correct behavior of the application. To facilitate this, we introduce a new component, the *code splitting manager* (Figure 12), which is linked into the protected client application and which is responsible for managing the coordination required between the new client application and the server on which the sliced code is moved, such that both sides' executions work synchronized.

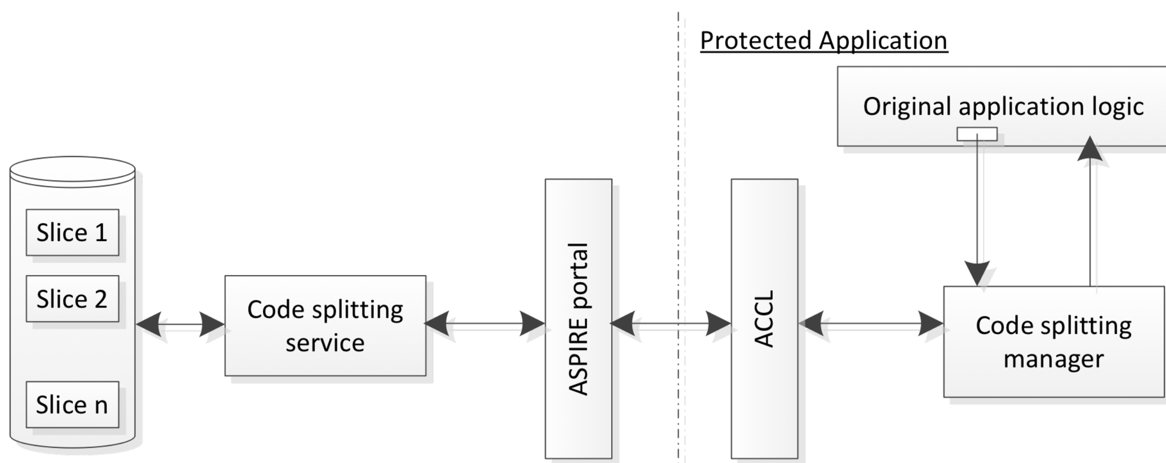

Figure 12 - Architecture for client-server code splitting

The new component exposes an API that will be used by the protection technique. At various places in the protected client application, calls to library functions are introduced by the protection technique in such a way that the now protected client application operates as intended.

At the server side, a component implements a main thread that manages connections and any message from and to connected clients. The sliced code is executed in an auxiliary thread that is launched from the main one whenever a client connects. Function calls from communication API are also injected into the sliced code at the server side.

Calls to library functions can be summarized as follows:

- Client-side:
  - sync() synchronizes with the server by sending the current execution point reached. It replaces any definition of sensitive variables in the original code.
  - ask() sends a message to the server, requiring a value that is necessary for the progress of the computation in the client. This call replaces uses of sensitive variables in the original application.
  - send() forwards data that cannot be moved from the client to the server, when the server requires those data for the execution of the sliced code. Possible values of this kind are those that come from barriers or other variables that do not belong to the slice.
  - waitForValue() waits for a value that must be sent by the server.
  - exit() notifies the server that the client ends its computation.
- Server-side:
  - checkSync() synchronizes the execution with the client at the server side, like the corresponding sync() at client-side. This call anticipates any definition of a sensitive variable in the sliced code.
  - waitForValue() waits for a value coming from client, typically values of barrier variables or other variables that are not ported on the server; This call replaces any use of this kind of variables.
  - sendValue() sends a value to the client. This call replaces any use of a sensitive variable in the sliced code.

A detailed description of the architecture can be found in deliverable D1.04, Section 3.3.3.

3.3.2 Structure of the Tool

The tool that implements client/server code splitting works in several steps. Figure 13 shows

a high-level view of its structure. It combines Grammatech CodeSurfer [GCS] and the TXL transformation framework [TXL] to analyse the source code of the target application, to identify the portion of the code that requires to be moved onto the secure server and to apply code transformation patterns to generate the new protected application. Input of the tool is the resulting pre-processed code after the application of the other source-level protections (SCzz in Figure 13), while its outputs are the client protected by client/server code splitting (SCzz+1), and the server-side code that will run on the secure server (SCzz+2). In the figure, components are numbered zz because the client/server code splitting step is not yet integrated in the source level part of the ASPIRE Compiler Tool Chain (ACTC) (see Section 7 of deliverable D5.01 for further references).

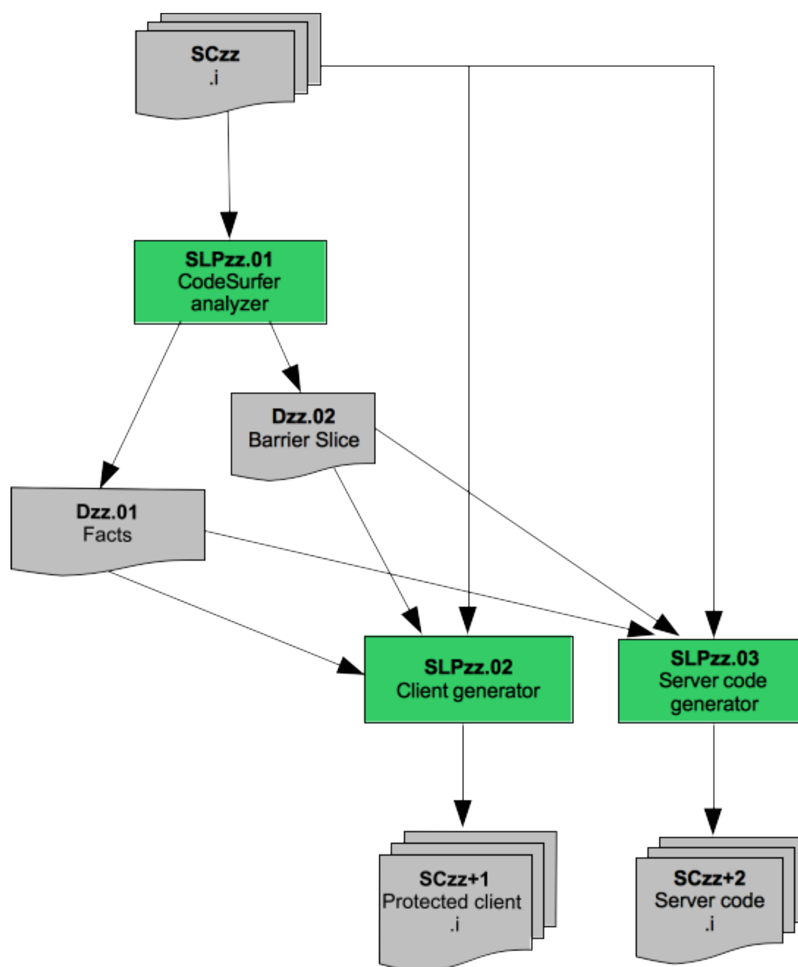

Figure 13 - Tool flow for client/server code splitting

3.3.2.1 Annotations for Client/Serve Code Splitting

As mentioned in Part 1 and Section 7.01 of deliverable D5.01, the user first has to annotate his/her source code. Client/server code splitting annotations specify a slicing criterion C as a set of statements and a list of sensitive variables on which to apply the barrier slicing algorithm, and a set of barrier statements B that blocks the propagation of the dependencies while calculating the slice.

After their extraction by components of the main ASPIRE tool flow (such as SLP04 in the ACTC design presented of D5.01), annotations are manipulated by the client/server code splitting tool to derive a set of fact files to be used to apply the protection.

For each annotation, criteria are encoded as line numbers corresponding to each statement in the code block that is indicated by the annotation itself. Barrier statements are also represented as line numbers, while sensitive and barrier variables are reported as specified within the annotation, by the use of their identifier in the code plus an integer as index. The integer associated with the variable will be used by TXL in the creation of the new client application and the server-side code.

Annotations and their syntax for client/server code splitting are described in Section 4.8 of deliverable D5.01.

3.3.2.2 SLP01 – CodeSurfer Analyzer

Component SLPzz.01 is implemented by means of GrammaTech CodeSurfer, which provides a framework on top of which custom data flow analyses can be implemented.

Starting from the information extracted from annotations, we implemented a custom backward slicing algorithm to precisely calculate the portion of the code that represents the barrier slice with respect of the current annotation configuration and the code to protect.

The code in input is analyzed by CodeSurfer to extract the system dependence graph (SDG) of the program to protect. The slicing algorithm queries this data structure to extract the barrier slice. The algorithm is implemented in a CodeSurfer script written in the Scheme language.

```
procedure calculate-slice (criterion C, barrier B)
  set-of-vertices := vertices-of (C)
  set-of-barriers := vertices-of (B)
  slice := vertices-of(C)
  while not (is-empty(set-of-vertices))
        predecessors := predecessors-of-vertices (set-of-vertices, DATA-CONTROL-DEPS)
        filtered-predecessors := predecessors \ set-of-barriers
        set-of-vertices := filtered-predecessors
        slice := slice ∪ filtered-predecessors
  }
```

Figure 14 - Pseudo-code of the barrier slicing algorithm

Figure 14 shows the pseudo-code of the barrier slicing algorithm we implemented. It starts by taking slicing criterion C and barrier B as parameters and iterates until no valid predecessors for the current nodes can be found in the SDG of the program. When this condition is met, the slice is returned as a set of nodes of the SDG. The nodes are then converted and stored as a list of line numbers indicating all the statements that belong to the slice. It is indicated as Dzz.02 (Barrier slice) in Figure 13.

To apply the code transformation in the following stages correctly, other information needs to be retrieved at this stage. In fact, the sliced code requires the careful handling of all those pieces of code that reside outside the slice but influence or are influenced by the slice itself. We implemented additional CodeSurfer scripts to perform extraction. Data extracted, indicated as Dzz.01 (Facts) in Figure 13, are:

- Definitions (e.g. assignments) of:
    - Sensitive variables that belong to the slicing criterion;
    - Barrier variables that are not in the slice but their value is required for computation;

   - Input variables that cannot be ported to the server;
- Uses of:
   - Sensitive variables in the portion of code to be sliced;
   - Barrier variables in the portion of code to be sliced;
   - Input variables (like the ones provided by the user or taken from an input file);
   - Any variable that has an inter-procedural dependence with other parts of the application that are not in the slice;
- Def/use chains (i.e., a definition *d* of a variable *x* and the set U of uses of that variable *x* reachable from definition d without any other redefinition);
- Pointers, in order to deal with the presence of aliases;
- Declarations of:
   - Any user-defined function that is called in the slice;
   - Any variable that is defined in the slice;
   - Any variable that is used in the slice.

3.3.2.3 SLP02, SLP03 – Client Generation and Server Code Generation

Figure 15 shows the C code of Figure 8 with the addition of the code annotations introduced by the user

```
1       _Pragma("ASPIRE begin protection(barrier_slicing, barrier(year2), label(s1))")
2       year2 = tm.tm_year+1900;
3       _Pragma("ASPIRE end")
4       for (i = ref; i < year1; i++) {
5               if (i % 4 == 0)
6                       dd1 += 1;
        }
7       dd1 = calculate_original(dd1);

8       dd2 = 0;
9       for (i = ref; i < year2; i++) {
10              if (i % 4 == 0)
11                      dd2 += 1;
        }
12      dd2 = calculate_current(dd2);
13      _Pragma("ASPIRE begin protection(barrier_slicing, criterion(dd1, dd2), label(s1))")
14      if (dd2 - dd1 > 30)
15              printf("Fail\n");
        else
16              printf("Ok\n");
17      _Pragma("ASPIRE end")
```

Figure 15 - Annotated running example

According to the annotation syntax, annotations in Figure 15 indicate:

- At line 1, the presence of a barrier B on variable *year2* at line 2;
- At line 13, variables *dd1* and *dd2* at lines 14, 15, 16 represent the slicing criterion C for calculating the barrier slice. Variables *dd1*, *dd2* are then sensitive variables, so any reference to these two variables must be removed from the code of the protected application. Statements 14, 15, 16 are also part of the barrier slice, so they are extracted and modified accordingly to produce server-side code.

Component SLPzz.02 and component SLPzz.03 are responsible for the generation of the protected client application and for the generation of the server-side code to run the slice, resp. The two components apply on the pre-processed source code that still contains annotations in C format, while facts (Dzz.01) and barrier slice (Dzz.02) extracted with CodeSurfer are used as input to perform source code transformations. Client generation (SLPzz.02) and server code generation (SLPzz.03) are implemented in TXL. They apply code transformations to produce the protected client application and the server-side sliced code. The following paragraphs report some of the code transformations we have implemented.

Remove definitions

Figure 16 shows a portion of code taken from the running example (Figure 15) that must be transformed. In fact, we have one of the sensitive variables of the program, *dd2*, which is defined twice, at line 8 and at line 11 respectively.

By applying TXL transformation routines for client and server we obtain the code shown in Figure 17.

```
8       dd2 = 0;
9       for (i = ref; i < year2; i++) {
10              if (i % 4 == 0)
11                      dd2 += 1;
        }
```

Figure 16 - Code containing definitions of sensitive variable dd2

```
8       sync (1);
9       for (i = ref; i < year2; i++) {
10              if (i % 4 == 0)
11                      sync (2);
        }
```

```
8       checkSync (1);
9       dd2 = 0;
10      for (i = ref; i < year2; i++) {
11              if (i % 4 == 0) {
12                      checkSync (2);
13                      dd2 += 1;
        }
```

Figure 17 - Protected code for the example of Figure 16

For client-side code, on the left in the figure, calls to function sync() (lines 8, 11) replace the definitions of variable *dd2*, which disappear from the code of the protected client. In case of the server-side code, on the right, calls to function checkSync() are introduced at line 8, 12, to anticipate any sensitive variable definition, which are kept in the code. Function sync() and checkSync() are intended to synchronize client and server: the former notifies the server about the execution point the client has reached (indicated by the integer parameter of the function), while the latter checks the execution status at server-side to keep the two executions aligned.

Remove uses

```
13      _Pragma("ASPIRE begin protection(barrier_slicing, criterion(dd1, dd2), label (s1))")
14      if (dd2 - dd1 > 30)
15              printf ("Fail\n");
        else
16              printf ("Ok\n");
17      _Pragma("ASPIRE end")
```

Figure 18 - Code containing uses of sensitive variables dd1 and dd2

```
14      if (ask (1, 2) - ask (2, 1) > 30)
15              printf ("Fail\n");
        else
16              printf ("Ok\n");
```

Figure 19 - Protected code for the example in Figure 18

The annotated code of Figure 18 also needs to be modified by the tool, since it contains references to sensitive variables *dd1* and *dd2*, and it is part of the barrier slice. By applying code transformation patterns, we obtain the protected code depicted in Figure 19.

Each occurrence of the sensitive variables *dd1* and *dd2* are substituted by calls to function ask() (Figure 19) which accepts two parameters: an integer label that is used for synchronization similarly to what is done with synch(), and the integer associated to the variable reference that has to be removed. This information is contained in the fact files collected when annotations are extracted. At server side, the sensitive variables are replaced

by calls to function sendValue() (Figure 20), which waits for value requests coming from the client and answers with the proper value.

In both of the cases, annotations are removed from the resulting code.

```
14     if (sendValue (1, 2) - sendValue (2, 1) > 30)
15             printf ("Fail\n");
       else
16             printf ("Ok\n");
```

Figure 20 - Protected code for the example of Figure 18 (server side)

Transformation of barrier variables

Transformation patterns also apply in case of barrier or other variables. For barrier variables, lets consider the fragment of annotated C code of Figure 21.

```
1      _Pragma("ASPIRE begin protection (barrier_slicing, barrier (year2), label (sl1))")
2      year2 = tm.tm_year+1900;
3      _Pragma("ASPIRE end")
```

Figure 21 - Annotated code for barrier variables

The annotation indicates a barrier variable, *year2*. The statement at line 2 then blocks the propagation of the dependencies when the barrier slice is calculated, and it is not included in the slice itself. This means the server will require the value of *year2* if a use of that variable is present in the sliced code. Our code transformation produces the code of Figure 22.

```
1      year2 = tm.tm_year + 1900;
2      send (1, 1);
```

```
1      waitForValue (1, 1);
```

Figure 22 - Protected code for the example of Figure 21

In case of the protected client (Figure 22, left) a call to function send() is inserted after the statement marked as barrier by the annotation. Like ask(), function send() takes two integer parameters: one for synchronization and one to indicate which variable needs to be sent to the server. The statement annotated as barrier is not part of the slice, so it is not present at server side. The server code, anyway, requires the value of the barrier variable *year2* to execute properly, so a call to function waitForValue() is introduced in place of the original statement. This function stops the execution of the sliced code until the required value is available.

Also in this case, annotations are removed from the protected code.

At the conclusion of the transformation, the modified client (SCzz.02) and server-side code (SCzz.03) are generated. Note that the output of the tool is still pre-processed code.

3.3.3 Current Implementation

At the current stage of the implementation, the tool prototype is working but it is not yet integrated into the ACTC. The responsibility for the orchestration of the various components of the tool is on a shell script, which takes the source code to protect as input and applies the protection. An execution of the tool generates two new executables, a protected client application and a server application equipped with the sliced code. The server application intends to simulate the ASPIRE framework until this will be available.

As a case study for testing the current implementation, we use the simple license checker presented earlier. To check the correctness of our tool, we created a regression test suite with different annotation configurations. It consists of several different configurations of

barriers and criteria. For each annotation configuration, the tool produces a protected client and a server, which runs the sliced code. The original, non-protected application is run several times with different inputs and its output is compared with the output obtained by the protected applications.

3.4 Plan

Required code transformations, along with a simulator of the communication infrastructure, have been implemented and tested on the license checker presented earlier in the deliverable, but tests on larger examples are necessary to check the validity of the current implementation.

Under these considerations, we envision the following implementation plan:

- **M18**:
    - Regression testing and complete implementation (still not integrated in the ACTC)
    - Experimentation on simple examples and on the available use cases developed by industrial partners in time for deliverable D3.02;
- **M24**:
    - First integration in the ACTC for deliverables D3.03 and D3.04;
    - Evolution of client-server slitting to boost performances, still not integrated in the tool chain;
    - Integration in the ADSS;
- **M30:**
    - Complete integration in the ACTC for deliverables D3.05 and D3.06.

Section 4 Implicit Remote Attestation

Section Authors:

Cataldo Basile (POLITO)

Implicit Remote Attestation (IRA) aims at overcoming the current limitations in the Remote Attestation (RA) techniques. IRA will be developed in Task T3.2 of WP3, a task that started only very recently, at M10. Therefore we will present here the general protection principles, preliminary design ideas, and a development plan. No actual results or an IRA implementation are available yet.

The basic RA architecture, characteristics and techniques are presented in deliverable D1.04. In Figure 23, we repeat the RA reference architecture.

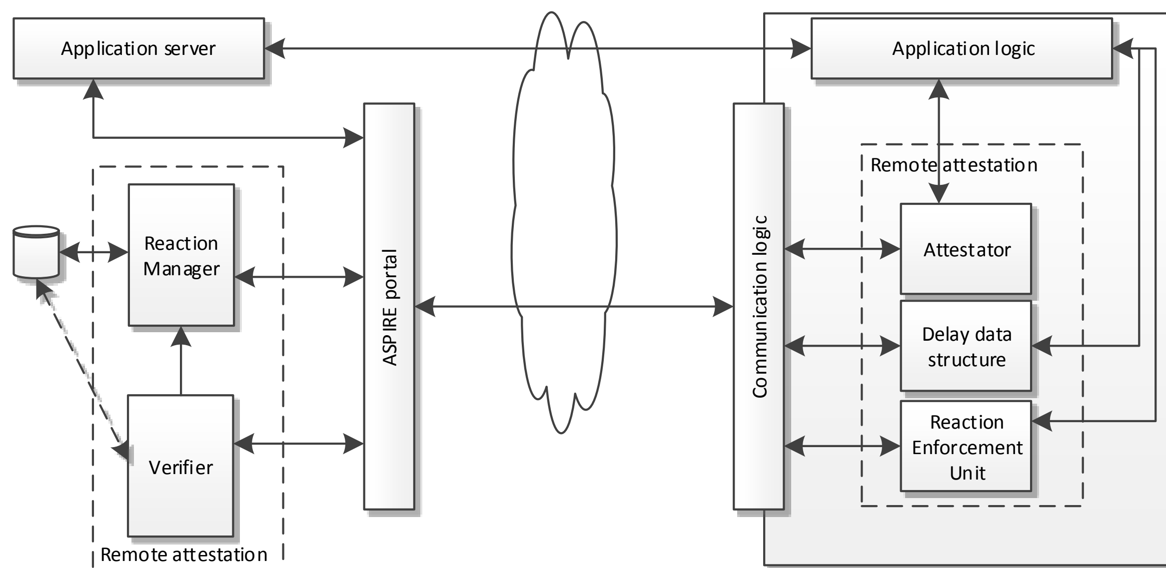

Figure 23 - Remote Attestation, reference architecture.

Together with the Communication logic, the client side components needed to implement RA are the Attestator, the Delay Data Structures, and the Reaction Enforcement Unit.

The RA objective is to check the integrity of the application to protect by asking the client-side Attestator to produce an Attestation Report. The Attestation Report is verified at the server side by a Verifier. In case of attestation failures, the Reaction Enforcement Unit must render the application unusable (according to a strategy defined by the Reaction Manager). The Reaction Enforcement Unit is controlled by means of the Delay data structures, a software-based covert communication channel between the Reaction Manager and the Reaction Enforcement Unit. The Attestator is in charge of collecting and maintaining Attestation Data and computing the Attestation Report when the server sends an Attestation Request. Depending on the type of technique used (static, dynamic, temporal), the Attestator may perform sensitive operations that can be used by the attacker to spot the Attestor code. For instance, static techniques make anomalous accesses to code memory pages, dynamic techniques read several variables values, and temporal techniques execute heavily optimized pieces of code. The knowledge of the Attestator code can be exploited to mount several attacks. For instance, an attacker can use it to read and store valid Attestation Data

(used with server nonces to compute valid Attestation Reports) or to directly forge fake Attestation Reports.

The same consideration applies to the Reaction Enforcement Unit: it performs operations on code sections to render the code unusable, as foreseen by graceful degradation, time bombs, etc. An attacker can look for these operations to try to locate and defeat the Reaction Enforcement Unit.

Even if in practice RA works very well and, especially when coupled with renewability, allows the achievement of very good protection levels, being able to (theoretically) spot important protection components is a weakness that we want to address with IRA.

IRA will be developed to avoid the presence of Attestator and Reaction Enforcement Unit at the client side by exploiting the following ideas:

- To avoid the presence of the Remote Enforcement Unit, IRA will work with applications that are not independent of the server, i.e., IRA does not work with stand-alone applications. In case of applications that depend on the server, the detection of tampered application instances can be punished in a simple yet effective way: disconnecting the client from the server.
- To avoid the presence of the Attestator, IRA uses a server-side Verifier that is able to inspect the executed server-code to obtain the Attestation Data that the (client-side) Attestator would have collected after a server request.

There are two main pre-conditions to use IRA: (1) the application must be composed of a client and a server, with the client not executable without interacting with the server, and (2) the server-side code must contain enough information to allow the IRA Verifier to reach a verdict about the application integrity.

Since concentrating on natively distributed applications would restrict too much the range of applicability of this technique, and, more important, it is not expected that the server-side code is in general sufficient to reach the verdict, we will develop an IRA Protection Tool that will instrument the application to make IRA possible. The IRA Protection Tool will take as input an application's source code (distributed or not) and will produce another application, divided in a client and an IRA-ready server, that also contains the code of the IRA Verifier. The IRA Protection Tool will not actually split the application, it will output annotations for the Client/Server code split service developed in Task T3.1 of WP3. The annotations will indicate how to split the application. More details on the IRA Protection Tool design will be presented in Section 4.1.

The important research issues to solve to make IRA usable in practice is identifying a set of properties/peculiarities the remote IRA Verifier can observe from within the server-side application to establish the client-side application integrity. Research is ongoing to determine these properties. Currently, we have identified two approaches (decision on usability still pending): CFG-based IRA and remotely watched invariants monitoring.

CFG-based IRA consists in assigning different paths in the CFG of the application to different labels. The set of labels can be populated using application-specific characteristics (e.g., high-level purpose or function performed) or using business-related information (e.g., licence needed to execute it). A possible application scenario for CFG-based IRA is to protect applications that have different license profiles (e.g., a free standard version and a premium version that enables several other features) whose executables contain the entire application logic. By moving selected parts of labelled paths onto the server, IRA allows monitoring of the execution of functions that are not allowed by the licence type, hence separating tampered applications from original ones.

The second approach investigates the possibility of performing invariant monitoring on the server-side code. When invariant monitoring is deployed, the Attestator collects and sends the server the values of a set of variables that are used to compute the invariants. The list of

the variables whose values needs to be collected is requested by the server at every Attestation Request. The server may also ask for values of variables that are not actually used in invariants to make it harder for the attacker to guess the invariants. However, an attacker does not necessarily need to guess the invariants, which can be a very difficult task. Alternatively, the attacker can spot the Attestator code, e.g., by means of a dynamic analysis tool. He can then inspect the code of an untampered application and save the values of the variables sent to the server. After having built a database populated with the saved variables' values, he can modify the Attestator code to send the server the variables values taken from the database (instead of reading the variables from the memory). With *remotely watched invariants monitoring*, the application (original or after the instrumentation with the IRA Protection Tool) is split into a client and a server such that the variables of the invariants to monitor are passed to the server and used in some non-trivial) computation. This obliges the attacker to send the server the real variable values, as otherwise it is not guaranteed that the program can continue to work correctly. In short, IRA renders the previous attack based on a database useless.

Another research issue to address is how to determine the minimum quantity of information to move on the server to (1) allow an IRA verdict, and (2) to make it impossible for an attacker to build a fake server. Indeed, if the server-side component performs trivial tasks, an attacker could easily implement a fake version of the server, thus avoiding the problem of being disconnected. Moving more computations onto the server decreases the overall performance of the application, however, due to network latency and potential server overloading. This decision is a typical optimization decision that we will leave to the ADSS.

4.1 Design

The preliminary IRA Protection Tool is presented in Figure 24.

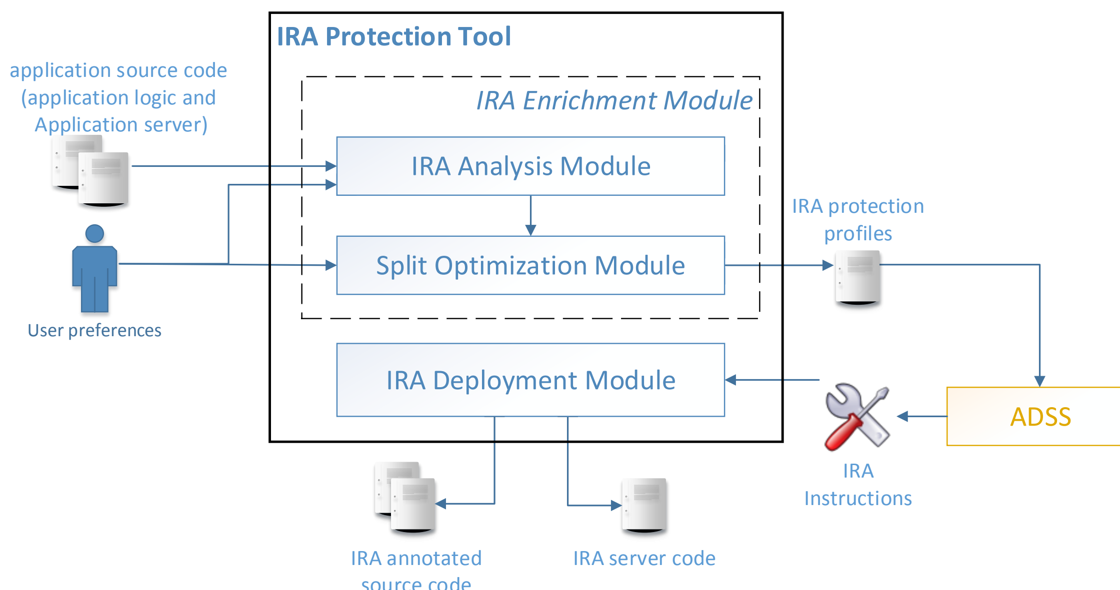

Figure 24 - The architecture of the IRA Protection Tool.

The IRA Protection Tool includes an *IRA Analysis Module* that will determine the possibility to protect the application with one or more IRA approaches. The IRA Analysis Module is based on compiler tools that are used to generate abstract representations of the application, and on ASPIRE-developed models that will check the abstract representations and identify suitable IRA approaches. The user may constrain the search by limiting the IRA approaches to check. The IRA Analysis Module produces a report that includes all the discovered pairs

(IRA approach, split points). The IRA approach also implicitly indicates the IRA Verifier to deploy on the server.

Following guidelines in D5.01, the IRA Protection Tool will not pass all the identified (approach, split point) pairs to the ADSS. The reason is to make the decision process in the ADSS that will select the golden configurations more manageable. Therefore, using terminology borrowed from the optimization field, we plan to pass to the ADSS only the elements on the Pareto front (or a subset of them). These elements will be passed as protection profiles together with the value of all the metrics needed to the ADSS to evaluate the impact. This operation is performed by the Split Optimization Module. The union of the IRA Analysis Module and the Split Optimization Module will form an Enrichment Module that will be used during the ADSS Combination Generation phase.

Once the ADSS has selected the golden configuration, the Tool Flow Bridge (an ADSS component) will instruct the IRA Protection Tool on the selected profile (if any). The actual IRA enforcement consists in (1) generating a set of annotation (to be added to the original source code) to indicate where to split the application source code, and (2) the source code of the IRA Verifier that will be deployed on the server.

4.2 Plan

The next 6 months will be devoted to build the theoretical model for IRA enforceability. We will perform the following two research-intensive tasks.

First, we will analyse sample applications (taken from the ASPIRE use cases and from repositories of open-source software) to determine properties/peculiarities the remote IRA Verifier can watch from within the application server to establish the application integrity. Several applications with different characteristics will be analysed by means of static and dynamic analysis tools, and other reverse engineering tools (e.g., generators of likely invariants). We will also generate and save several abstract representations of the selected applications. The aim is to use these abstract representations to identify new remotely watchable application properties/peculiarities and to validate the applicability of our ideas on CFG-based IRA and remotely watched invariants monitoring. The work will concentrate on the abstract representations (rather than on the applications) as our aim is to build the theoretical framework, determine the pre-conditions to apply IRA and evaluate the impact of alternative splits. We may need to simulate the execution of simplified versions of the applications (split in a client and a server) that only convey the aspects that are interesting for IRA enforcement.

Second, we will work on the definition of the optimization problem that will be used to determine the minimum portion of code to split. This phase will address the identification of the optimization parameters, thresholds, etc. and relations to metrics (described in D4.02). This second research effort will be complemented with the definition of criteria to generate protection profiles to be used by the ADSS (as explained in deliverable D5.01).

Moreover, the research will focus on the identification of potential issues and incompatibilities with the techniques used in initial ACTC flow as presented in deliverable D5.1. Solutions to potential issues will be incorporated in the Preliminary ASPIRE Online Protection Tool Chain (in deliverable D5.05). Several techniques may impact this technique. For instance, obfuscation and other transformations can change the abstract representations of the application and render the IRA deployment impossible. For instance, CFG flattening might make the use of CFG-based IRA impossible. Analogously, likely invariants identified on the original applications need to be updated after the use of data hiding techniques. Therefore, we will investigate the impact of IRA on the order of execution of the ACTC components.

Section 5 Anti-Cloning

Chapter Authors:

Brecht Wyseur (NAGRA)

5.1 Introduction

An intrinsic difference between hardware and software is that software can easily be replicated. In a white-box attack context, which is the targeted attack context for the ASPIRE project, an attacker has full access to the software applications, and can thus clone the software to run an identical instance in different, and possibly multiple locations.

Mitigating such attacks can only be done by 'locking' the software with some component that cannot be cloned, such as a hardware building block or a remote server. The latter is the approach that is conceived in this anti-cloning (AC) technique that we are developing for the ASPIRE project.

The basic idea of this technique is to assign to individual software application instances a unique value that is tracked by the ASPIRE server, and to let this value evolve in an unpredictable way. While two application instances may be identical at some moment in time (i.e., when cloned), once one of them connects to the ASPIRE security server and is forced to update its value, the two instances will be desynchronised.

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

The design and development of this AC technique is part of Task T3.2 of WP3. At this point, the technique is only at its conception phase. A detailed architecture design and development are scheduled at a later phase in the project as described in the ASPIRE DoW. Nevertheless, in this document we already present a preliminary design description. Details and progress on the development will be reported in the later deliverables of WP3.

5.2 Design

5.2.1 Architecture

A preliminary design description has been provided in deliverable D1.04 (ASPIRE Reference Architecture), in which the client-side components (tag and AC manager) and server-side components (the AC decision logic) were introduced.

The approach described above translates in practice as follows:

1. From anywhere in the program, the client-side AC manager can be called. That is, the AC manager will be invoked via a call to the AC_check_status() function.
2. Subsequently, the AC manager reads the tag value and communicates this to the server-side decision logic.
3. The server can choose different responses. It can choose to acknowledge reception and return control to the original application logic; or it can choose to enforce an update of the tag value by providing some such new value.

4. In case of a tag update, the local tag will be updated, and the new value will be communicated to the server to acknowledge that the update was successful (step 2).
5. Subsequently the server can decide again to either return control back to the application or again enforce an update (step 3), and so forth.

5.2.2 Annotations

The AC mechanism is integrated in ASPIRE protected applications and invoked by a dedicated function call to the AC manager API that is inserted into the application during the source-to-source translation. This call to the AC manager initiates a synchronization event between the ASPIRE protected application and the ASPIRE security server. This event can be invoked at any point in time during the execution of the ASPIRE protected application and from any place in the application. When invoked, the AC mechanism does not need any parameters; only the hooks from where it will be invoked need to be defined. These have to be specified by means of annotations, as also described in deliverable D5.01 Appendix B.13.

The annotation is defined as

```
<PROTECTION_NAME> ::= anti_cloning
```

The ACTC will translate this annotation into a call to AC_check_status(). This function will return a status code, which could be used to trigger some client-side response. In general, no particular client-side action needs to be taken; it will be the server that will use the AC status information to decide whether or not it will grant access to a service request. Nevertheless, it one can envision that in subsequent versions of the ACTC, different actions are introduced based on the status response, as presented in the code snippet below.

```
status = AC_check_status()
switch ( status )
{
      case TIMEOUT: …; break;
      case NOT_OK: …; break;
}
```

5.3 Plan

The AC technique is part of RTD Task T3.2. The task itself is planned from M10 to M36. While the basic conception and high level architecture description have been done (as described in this deliverable and the ASPIRE Reference Architecture), the fine-grained details of the design and the implementation of this technique are only planned at a later stage of the task. Given priorities on other techniques (such as Task T2.2 that terminates earlier) and resource planning, NAGRA does not plan to start the implementation of this technique in the next 6 months, but rather towards the end of year 2 or beginning of year 3 of the ASPIRE project.

Section 6 List of Abbreviations

AC	Anti-Cloning
ACTC	ASPIRE Compiler Tool Chain
ADSS	ASPIRE Decision Support System
ASPIRE	Advanced Software Protection: Integration, Research and Exploitation
BCxx	Binary code document nr. xx
BLCxx	Binary-level configuration file nr. xx
BLPxx	Binary-level software processing step nr. xx
Dxx	Datum produced or used by the ASPIRE ACTC identified with nr. xx
Dx.y	ASPIRE deliverable # y in work package x, y is a two digit number
DoW	Description of Work
IRA	Implicit Remote Attestation
SCxx	Source code document nr. xx
SLCxx	Source-level configuration file nr. xx
SLPxx	Source-level software processing step nr. xx
SDG	System Dependence Graph
RA	Remote Attestation
WP	Work Package

Bibliography

[Cec07] M. Ceccato, M. Dalla Preda, J. Nagra, C. Collberg, and P. Tonella. Barrier Slicing for Remote Software Trusting. In Proceedings 7th IEEE International Working Conference on Source Code Analysis and Manipulation, 2007, pp. 27–36.

[Cec09] M. Ceccato, M. Dalla Preda, J. Nagra, Ch. Collberg, and P. Tonella. Trading-off Security and Performance in Barrier Slicing for Remote Software Entrusting. Automated Software Engineering 16(2), 2009, pp. 235–261.

[Cha02] H. Chang and M.J. Atallah. Protecting Software Code by Guards. In Proceedings of the ACM Work-shop on Security and Privacy in Digital Rights Management, 2001, LNCS 2320, pp. 160–175.

[Che02] Y. Chen, R. Venkatesan, M. Cary, R. Pang, S. Sinha, and M.H. Jakubowski Oblivious hashing: A stealthy software integrity verification primitive. Revised Papers from the 5th International Workshop on Information Hiding, 2002, pp. 400–414.

[Col08] C. Collberg, J. Nagra, and W. Snavely. bianlian: Remote Tamper-Resistance with Continuous Re-placement. Technical Report TR08-03, Department of Computer Science, University of Arizona, 2008.

[Col09] C. Collberg, J. Nagra, Surreptitious Software: Obfuscation, Watermarking, and Tamper-proofing for Software Protection. Addison-Wesley, 2009.

[Col12] C. Collberg, C. Martin, J. Myers, and J. Nagra. Distributed application tamper detection via contin-uous software updates. In Proceedings of the 28th Annual Computer Security Applications Conference, 2012, pp. 319–328.

[Fal06] P. Falcarin, R. Scandariato, and M. Baldi. Remote trust with aspect oriented programming. In Proceedings 20th International Conference on Advanced Information Networking and Applications, 2006, pp. 451–458

[Fal11] P. Falcarin, S. Di Carlo, A. Cabutto, N. Garazzino, D. Barberis. Exploiting Code Mobility for Dynamic Binary Obfuscation. In Proceedings IEEE World Congress on Internet Security, 2011

[Few08] Fewer, S., Reflective DLL Injection. Tech. rep., Harmony Security, 2008.

[Gil11] B. Gilbert, R. Kemmerer, C.Kruegel, and G. Vigna. Dymo: Tracking Dynamic Code Identity. In Pro-ceedings of the Symposium on Recent Advances in Intrusion Detection, 2011, LNCS 6961, pp. 21–40.

[GCS] Grammatech CodeSurfer – Code understanding tool for C/C++ source code. [Online] http://www.grammatech.com/research/technologies/codesurfer [Accessed: 23 Oct 2014].

[IDA] IDA Pro Disassembler - multi-processor, windows hosted disassembler and debugger. [Online] http://www.hex-rays.com/idapro/ [Accessed: 10 Oct 2014].

[Jak01] M. Jakobsson and M. Reiter. Discouraging software piracy using software aging. In Proceedings 1st ACM Workshop on Digital Rights Management, 2001.

[Kil09] C. Kil, E. C. Sezer, A. Azab, P. Ning, and X. Zhang. Remote Attestation to Dynamic System Proper-ties: Towards Providing Complete System Integrity Evidence. In Proceedings of the 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, 2009, pp.115–124.

[KRI03] Jens Krinke. Barrier slicing and chopping. In SCAM, pages 81–87, 2003.

[Mad05] M. Madou, B. Anckaert, B. De Sutter, and K. De Bosschere. Hybrid static-dynamic attacks against software protection mechanisms. In Proceedings of the 5th ACM Workshop on Digital Rights Man- agement, 2005, pp. 75–82.

[Mon99] F. Monrose, P. Wyckoff, and A.D. Rubin. Distributed Execution with Remote Audit. In Proceedings Network and Distributed System Security Symposium, 1999, pp. 103–113.

[Raj08] H. Rajan and M. Hosamani. Tisa: Toward Trustworthy Services in a Service-Oriented Architecture. IEEE Transactions on Services Computing 1(4), 2008, pp. 201–213.

[Sca08] R. Scandariato, Y. Ofek, P. Falcarin, and M. Baldi. Application-Oriented Trust in Distributed Com-puting. In Proceedings of the 2008 Third International Conference on Availability, Reliability and Security, 2008, pp. 434–439.

[Sch12] Patrick Schulz, "Code protection in android," 2012. Online at https://net.cs.uni-bonn.de/fileadmin/user_upload/plohmann/2012-Schulz-Code_Protection_in_Android.pdf

[Sch12b] J. Schiffman, H. Vijayakumar, and T. Jaeger. Verifying system integrity by proxy. In Proceedings of the 5th international conference on Trust and Trustworthy Computing, 2012, pp. 179–200

[TXL] The TXL transformation framework. [Online] http://www.txl.ca [Accessed 23 Oct 2014].

[vOo05] P.C. van Oorschot, A. Somayaji, and G. Wurster. Hardware-Assisted Circumvention of Self-Hashing Software Tamper Resistance. IEEE Transactions on Dependable and Secure Computing 2(2), 2005, pp. 82–92.

[WEI81] Mark Weiser. Program slicing. In Proceedings of the 5th International Conference on Software Engineering, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[ZHA03] Xiangyu Zhang and Rajiv Gupta. 2003. Hiding program slices for software security. In Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization (CGO '03). IEEE Computer Society, Washington, DC, USA, 325-336.