Advanced Software Protection:
Integration, Research and Exploitation

D3.04

Intermediate 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 / D3.04 / 1.01
WP and tasks contributing:	WP3 / Tasks 3.1, 3.2, 3.3
Due date:	Oct 2015 – M24
Actual submission date:	28 November 2015
Responsible Organization:	UEL
Editor:	Paolo Falcarin
Dissemination Level:	Public
Revision:	1.01

Abstract:
This deliverable documents the tool support and the research undertaken in WP3 towards the end of year 2 of the project. The document starts describing the new version of ASPIRE client-server communication logic, and then progresses about different online code protections are documented, namely: code mobility, client-server code splitting, implicit remote attestation, anti-cloning, software diversity, and renewability.

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

Editor

Paolo Falcarin (UEL)

Contributors (ordered according to beneficiary numbers)

Bert Abrath, Bart Coppens, Bjorn De Sutter, Mohit Mishra (UGent)

Cataldo Basile, Alessio Viticchié (POLITO)

Mariano Ceccato, Andrea Avancini (FBK)

Alessandro Cabutto, Paolo Falcarin (UEL)

Jerome d'Annoville (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

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

Executive Summary

This deliverable reports the year two RTD progress in WP3 on the topic of online protection techniques. Six major sections report on the progress of the three tasks in WP3.

First, Task 3.1 (Client-Server Code and Data Splitting) is reported. UEL has developed the ASPIRE Client-side Communication Logic (ACCL) and the ASPIRE Server-side Communication Logic (ASCL), with the new feature of bidirectional communication based on Websocket protocol, and the related network APIs that will be used by all the online techniques developed in WP3, such as the different remote attestation techniques of Task 3.2, and the renewability techniques of Task 3.3 that will be developed in the third year of the project.

UEL and UGent have integrated the Code Mobility framework in the ACTC and accomplished a performance analysis of the framework in different network settings and configurations, which is reported in this deliverable; the overall Code Mobility framework has been described in a paper titled "Software Protection with Code Mobility", published in the ACM Proceedings of the Second Workshop on Moving Target Defense [Cab15], co-located with the 22nd ACM Conference on Computer and Communications Security (CCS-2015).

FBK has performed an empirical assessment to estimate the preliminary impact of applying Client/Server Code Splitting on two case study applications: the simple License Checker and the Diamante license-checker use case in ASPIRE.

Second, Task 3.2 (Remote attestation) is reported: POLITO reports on the detailed Remote Attestation framework architecture, with an additional analysis of the composability of remote attestation with the other ASPIRE protections. POLITO and UGent developed the static remote attestation framework based on binary diversification and the dynamic remote attestation based on invariant monitoring.

NAGRA describes its Anti-Cloning protection implementation, the server-side policy and the composability issues.

Third, Task 3.3 (Renewability) is reported; UEL designed the overall architecture for renewability in time and space (software diversity) that will rely on Code Mobility extensions, planned for the 3rd year of the project, and reported in deliverable D1.04-v2 (Reference architecture). Here we report on the process and decisions made in the consortium to agree on the common architecture and the decision on which protections can be made renewable in time.

UEL and FBK report on their work aiming at maximizing diversity among different diversified copies using a search-based approach on a set of versions generated on a cluster of machines. Different heuristics have been tested and initial results are reported.

UGent reports on its initial work on the new feedback-driven diversification, and their new crash-reporting framework for diversified binaries: dBp (delta Breakpad). It is the first practical solution to the problem of crash reporting for applications with fine-grained layout diversification. UGent's approach allows embedding a small amount of encrypted information in a diversified application that, when sent to a bug crash collector together with a crash report, supports the reconstruction of an accurate, human-readable stack trace without requiring any persistent storage of data about the diversified application on a server.

Table of Contents

List of Figures

List of Tables

Section 1 Introduction

Section Author:

Paolo Falcarin (UEL)

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 ASPIRE's Work Package 3, namely: Code Mobility, Client/Server Code Splitting, Remote Attestation, Anti-Cloning and Renewability.

The remainder of this text report first discusses in detail the ASPIRE Client Server Architecture, with more details on the ASPIRE Client-side Communication Logic (ACCL) and the ASPIRE Server-side Communication Logic (ASCL) and the implemented APIs, on which the online protection techniques rely for their network communications.

Some of the online protection techniques are integrated into the ASPIRE Compiler Tool Chain (ACTC), such as Code Mobility and Client/Server Code Splitting. Other techniques such as Remote attestation and anti-cloning will be integrated by M30, while all the diversity and renewability techniques will be integrated by the end of the project at M36.

Section 2 introduces the new version of the ASPIRE Client/Server Communication logic, allowing server-initiated transactions. Section 3 reports the updates on Code Mobility, while Section 4 on Client/Server Code Splitting. Section 5 details the design and initial implementation of Remote attestation, while section 6 describes the detailed design of anti-cloning. Finally, Section 7 introduces the extended architecture and plan for renewability, and the initial works on software diversity (renewability in space).

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 include the final WebSocket protocol design, which was drafted in the first version of D1.04 document (reference architecture) and finalized in D1.04 v 2.0 (M24).

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

Major updates to the ASPIRE Client-Server Architecture occurred to the WebSocket Protocol which has been extended to better support Remote Attestation and Client-Server Code Splitting techniques. The first one explicitly needs server-initiated communication while the latter needs low latency communication between client and server. WebSocket protocol can fit both those requirements.

An implementation for Android and Linux of the Client/Server Communication Logic has been delivered for use in the ASPIRE Build Virtual Machine by all partners.

In D1.04 is reported the overall design and in this document more details about the communication are given.

2.1.1 Bidirectional communication

The WebSocket protocol has been extended so that it can provide full bidirectional communication. By design WebSocket has a non-blocking behaviour and data delivery is managed by callbacks invoked by a service handler that runs in a separate thread. Therefor data flow has to be managed by the ACCL and ASCL components using synchronization primitives.

The WebSocket channel is initially opened from the client-side when a specific handler (callback function) for incoming data is set. The callback is implemented inside the protection technique code and is in charge for incoming payload managing. On the server-side a similar scenario is set up inside the ASCL component. The channel initialization event is signalled by the ASCL to the protection backend via a named pipe.

The five different messages that can be sent over the pipe are described in the following sections. Messages respect the format described in Table 1.

4 bytes	4 bytes	N bytes
Message ID	Payload Size	PAYLOAD

Table 1 - Messages format

where

- **Message ID** is the identifier of the current message
- **Payload size** indicates the size of the payload in bytes
- **Payload** is a buffer containing the original payload

Possible values for Message ID are:

Message ID	Message
0	Channel initialization
1	Payload ‘send’ received
2	Payload ‘exchange received
3	Connection closed

The ASCL never inspects incoming payloads content, its only duty is to deliver them to the service backend.

Channel initialization

This message is sent to the protection backend service to allow it to initialize internal structures. In this phase the protection backend should prepare for the eventual arrival of payloads coming from the client. The Application ID is passed as payload in this case and will be used to manage data sending to the client.

Payload ‘send’ received from client

This message is sent when an `acclWebSocketSend` operation is initiated on the client side. The operation terminates when the full payload is sent. No answer is expected from the backend service.

Payload ‘exchange’ received from client

This message is sent when an `acclWebSocketExchange` operation is initiated on the client side. A response respecting the response format described in Section 0 is expected from the service backend. As soon as the response is received the ASPIRE Portal packages and sends the content to the client.

Connection closed

This message is sent to the service backend when the connection is terminated by the client (`acclWebSocketShutdown` has been called on the client side) or lost.

Response format

Responses coming from the service backend follow the subsequent format:

4 bytes	N bytes
Response size	RESPONSE

where

- **Response size** is the length in bytes of RESPONSE
- **RESPONSE** is the buffer containing the response to be sent to the client

Server initiated communication

A service backend can send data to the client only after a channel initialization has been received. In that case the Application ID has to be used to identify the recipient of the payload.

4 bytes	4 bytes	N bytes
Communication Type	Payload Size	PAYLOAD

where:

- **Communication Type** can be 0 in case of 'send' communication or '1' in case of 'exchange' communication
- **Payload size** is the length in bytes of the payload buffer
- **PAYLOAD** is the buffer containing data to be sent to the client.

Section 3 Code Mobility

Section authors:

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

This section reports the work of Task 3.1 on Code Mobility: its minor implementation changes, and experimental performance analysis.

The results of this work has been reported in a paper titled "Software Protection with Code Mobility", published in ACM Proceedings of the Second Workshop on Moving Target Defense [Cab15], co-located with the 22nd ACM Conference on Computer and Communications Security.

3.1 Architecture

The architecture presented in the last report (D3.02) is still valid and probably will be stable until the end of the project.

3.2 Code Mobility Components

The design and implementation of Code Mobility components did not receive significant updates after the previous report.

3.2.1 Downloader

While targeting Android ARM it came out that the `posix_memalign` syscall is not available on all Android versions. Android NDK revision 10d solved this bug but in general we cannot rely on the presence of this function.

Since page aligned memory allocation is needed by Code Mobility the `memalign` syscall is now used instead. This is a minor difference between Linux and Android platforms.

3.3 Performance analysis

Our performance analysis was carried out on three case studies written in the C and C++ languages, taken from the SPEC CPU 2006 benchmark suite, namely libquantum, namd and milc. Tests were performed on a SABRE Lite i.MX6 board with a Quad-Core ARM Cortex A9 processor at 1 GHz clock speed, with 1 GByte of 64-bit wide DDR3 at 532 MHz.

To evaluate the steady-state overhead of the mobile code transformations, i.e., the performance overhead on an application in which all executed mobile code blocks have already been downloaded, we used a customized version of Diablo. It transforms the applications by applying the GMRT indirection (see D3.02 Section 3.2.1) and by making all mobile code offset-independent as described in D3.02 Section 3, but it actually skips the mobile code blocks dumping operation (it leaves them in the binary).

To evaluate the latency that the downloading of the blocks might incur, we tested four different network scenarios: localhost, LAN, WiFi, and 3G. In the localhost scenario, all components were configured such that the client and the Code Mobility Server reside on the same test machine: all communications took place locally, in order to exclude influence of network transmission delays and to collect a reference baseline for the other configurations.

In the LAN configuration, we tested the code on a 100 Mbps wired network; in the WiFi configuration we tested the code on a 54 Mbps wireless network, while in the 3G scenario we tested it on a HSDPA mobile network.

We measured the *latency*, i.e. the time required to establish a new TCP connection, whenever a new code block has to be downloaded; then we calculated the *blocks download time* to measure the time needed to download a mobile block on different network configurations. For the block download we made an arbitrary function mobile and measured the time needed to transfer it from the server to the client. The chosen function has a code footprint of 412 bytes.

Each experiment was repeated 500 times to collect data and we calculated average value and standard deviation of latency and time to download a mobile code block (see Table 2); for latency measures we run the code only 100 times. The last column of Table 2 represents the total execution time of a mobile version of the libquantum application. In this case we made a hot function mobile that represents by itself circa 50% of the executed operations.

		Latency	Block download	Libquantum 50% mobile
Localhost	Average	0.12	9.36	369.37
	Std Dev	0.03	6.63	66.28
	Overhead			+1.97%
LAN	Average	0.32	6.98	370.45
	Std Dev	0.02	1.46	65.74
	Overhead			+2,27%
WiFi	Average	3.43	29.64	401.56
	Std Dev	2.81	24.49	68.36
	Overhead			+10,86%
3G	Average	134.27	228.87	659.54
	Std Dev	119.58	154.44	173.42
	Overhead			+82,08%

Table 2 - Summary of Performance Overhead (in *ms*)

Since most of the overhead comes from downloading blocks, which happens only once per mobile code block in our current implementation, and because our Android boards are relatively slow, we used the test SPEC inputs in our experiments. As expected, the worst overhead (82%) is found in case of mobile network connection while in a LAN scenario the overhead is as low as 2%.

Table 3 shows the performance once all mobile code blocks have been downloaded, i.e., when the redirection via the Binder's GMRT table is applied to all the fragments of an application.

For each benchmark application scenario, the average total execution time and its standard deviation are provided, overhead is computed as the increment of execution time with respect to the original application, where no functions have been instrumented to become mobile. Each row indicates a different experiment with a significant percentage (20%, 50%, and 100%) of indirection/mobility, evaluated as the number of instructions executed in mobile functions over total number of executed instructions.

Execution time	Average	Std Dev	Overhead
libquantum			
original	362.23	63.11	
20%	363.18	67.93	+0.26%
50%	355.73	67.14	-1.80%
100%	394.80	62.06	+8.99%
milc			
original	85,697.45	29.98	
20%	85,417.24	46,73	-0,33%
50%	85,985.24	46.73	+0,34%
100%	88,557.82	133.17	+3,34%
namd			
original	92,729.70	107.89	
20%	93,403.56	124.05	+0.73%
50%	94,383.00	115.48	+1.78%
100%	95,503.73	119.98	+2.99%

Table 3 - Summary of Computational Overhead (in *ms*)

In both the 20% and 50% coverage example we can see that the overhead is extremely low and sometimes even less than zero, which means that the instrumented version of the application can run faster than the original one. This is probably due to the optimizations applied to the code by Diablo.

Only when 100% of the application's functions are made "mobile" forcing the indirection we can see a significant overhead occur.

3.4 Planning

M26

The Code Mobility framework will be updated to provide support for moving data structures (UEL-UGent). This update is necessary to implement renewability of other techniques such as NAGRA's WBC and SFNT's Soft VM. An update on the server side logic will be released in order to allow server initiated mobile code blocks expiration, deletion and update (UEL-UGent).

M27

Server-side support for renewability (UEL+ALL) will be released. This is update is needed by all RA techniques.

M26 –M28

Once code mobility will be able to support data transfer WBC tables and VM's bytecode can be made mobile, therefore:

Renewable WBC library with code mobility should be ready for M28 (NAGRA-UEL-UGent)

Mobile bytecode with code mobility should be ready for M28 (SFNT-UEL-UGent)

M27-M30

Once Code Mobility on server-side will be able to keep track of code versions of different clients running then it can be used by RA techniques.

Section 4 Client/Server Code Splitting

Section Authors:

Mariano Ceccato, Andrea Avancini (FBK)

The *client/server code splitting* protection (task 3.1) aims at increasing the security of applications by moving sensitive, attackable portions of the program code from an untrusted client *C* to a trusted server *S*, in order to prevent attacks from malicious users.

This Section reports updates on client/server code splitting. We progressed with the implementation of the protection, to produce tool SLP06 in the ACTC, beyond what was reported in M12. Moreover, we conducted an empirical assessment to verify the correctness of what implemented and to estimate the preliminary impact of applying client/server code splitting on the performance of two case study applications.

The implementation does not differ from what was presented in the previous deliverables, and so a brief recap will just be given. Instead, the experimentation will be discussed here.

The section is structured as follows: Section 4.1 briefly recaps client/server code splitting, focusing on the structure of the tool we implemented, with details on annotations, on barrier slice computation and on client and server generation. Section 4.2 presents our experimental framework, by introducing the research questions we posed, the metrics we measured and the case study applications we used. Section 4.3 describes the experimental procedure we adopted, Section 4.4 focuses on the results we obtained, Section 4.5 describes the current status of the tool, and Section 4.7 closes this part of the document with the work plan for the next months of the project.

4.1 Client-server Code Splitting

Client/server code splitting [ZHA03] is a protection technique conceived within the ASPIRE project. With this protection, parts of an application are moved on a secure server, where they can run in a trusted environment. The parts to move are those that are considered sensitive, attackable, to ensure that malicious users cannot tamper with the application to alter its intended behaviour.

The technique applies *barrier slicing* [KRI03] to identify the portions of the application to move, and a set of source code transformations to generate the new, protected client application and a new corresponding server part. Communication capabilities are included in client and server to exchange data when required. The protected client and the server will then execute the portions of code identified by barrier slicing in a synchronous way to preserve the original functionalities of the application.

The next paragraphs will give a brief recap on the tool we developed. Further details about the protection can be found in deliverables D3.01 (Preliminary Online Protections Report – M12) and D3.02 (Preliminary Online Protections Support – M18).

Structure of the tool: the tool we implemented for client/server code splitting works by combining GrammaTech CodeSurfer [GCS], to analyse the pre-processed code of the application to protect and to identify those portions of the code that require to be moved on the secure server, and the TXL transformation framework [TXL], to apply precise code transformation patterns to generate the new protected client application and the

corresponding server-side code.

The input of the tool is the pre-processed code of the application to protect while, as mentioned earlier, the generated outputs are the client protected by client/server code splitting and the server-side code that runs in synchronous with the protected client.

The client/server code splitting step is integrated in the source level part of the ASPIRE Compiler Tool Chain (ACTC) as SLP06 component.

Annotations for client/server code splitting: client/server code splitting is driven by code annotations to apply its protection. Annotations and their syntax for client/server code splitting are described in Section 4.8 of deliverable D5.01 (Framework Architecture, Tool Flow, and APIs – M9). Briefly, a splitting annotation specifies:

- a slicing criterion C, in form of set of statements and a list of sensitive variables;
- a set of barrier statements B, to block the propagation of data and control dependencies while calculating the barrier slice.

We call a splitting annotation expressed in this form an *annotation configuration*.

In this context, a sensitive variable is a program variable that influence critical parts of the program and thus more prone to be attacked by a malicious user.

Computation of the barrier slice: we implemented a custom backward slicing algorithm (barrier slicing), which runs on top of the CodeSurfer framework. The code in input is analysed by CodeSurfer to extract the system dependence graph (SDG) of the program to protect. Then, the slicing algorithm queries this data structure to calculate the portion of the code that must be move to the secure server, *the barrier slice*, with respect of the current annotation configuration and the code to protect.

As already mentioned, a barrier slice is based on the concept of backward slice. The backward slice *s* on a criterion C includes all the statements that directly or indirectly hold data or control dependencies on the sensitive variables at the statements in C. A barrier slice, instead, can be calculated by stopping the computation of a backward slice whenever the set of barrier statements B is reached.

The size of the code to move, in terms of number of statements, can vary according to several factors, like the distance between criterion and barriers, the data dependencies in the code, and the peculiarity of the source code itself.

Generation of client-side and server-side code: for the client side, code transformations remove any definition or use of a sensitive variable to protect. Synchronization and communication primitives are added to the code in order to communicate with the sliced code that runs on the server. At the server side, the sliced code is also modified to support communication and synchronization.

4.2 Experimental Framework

This section describes the experimental framework we built to support the analysis we conducted on protecting applications with client/server code splitting. We applied the protection to two case studies, to generate the corresponding protected applications and the server-side code. Then, we ran the applications to extract performance and communication metrics that help us answering the research questions we formulated.

Research questions: the goal of applying client/server code splitting is to reduce the attack surface of a program that can potentially be targeted by attackers. This, however, introduces modifications in the protected application that can have an impact on the overall performance, namely on execution time and memory occupation. The purpose of our empirical investigation is to answer the following research questions:

RQ1: What is the execution time overhead caused by client/server code splitting?

RQ2: What is the memory overhead caused by client/server code splitting?

Metrics: since client/server code splitting produces two distinct software artefacts as output, the protected client and its corresponding server-side code that communicate through the network, we distinguish between metrics that are related to the client and metrics that are related to the server.

More precisely, in order to answer the research questions presented above we measured the following metrics:

- Client
  - Execution time of the program;
  - Memory occupied by the program during its lifetime;
- Server
  - Execution time of the server;
  - Memory occupied by the server;
- Generic metrics
  - Number of sensitive variables to protect;
  - Total number of statements that compose the barrier slice, i.e. the number of statements that are moved on the secure server;
  - Total number of exchanged messages between client and server;

We used the Linux utility *time* to measure execution time and memory for both client and server programs. This command runs another program, and displays information about the resources, like memory and time, consumed by that program.

Information related to the number of sensitive variables to protect are extracted directly by the tool when splitting annotations are found in the code. The total number of statements in the barrier slice is calculated by a custom shell script after the computation of the slice itself.

The client generated by client/server code splitting is equipped with a communication library[1], to exchange values and to synchronize the execution of the sliced code on the server. Communication works in both directions, since server-side code also requires values coming from the client to keep the execution of the slice synchronized. For the experimental analysis, the communication library was instrumented to collect communication-related metrics (number of messages exchanged). With the instrumented version of the communication library, any execution of the protected application generates a log file that can be parsed to extract the metrics.

Subject applications: as case studies for the experimental analysis, we use a small license checker C application called *License Checker* and the program *TCAS* from the Software-artefact Infrastructure Repository (SIR, http://sir.unl.edu/portal/index.php).

The case study applications are listed in Table 4 and briefly described in the following paragraphs.

Application	LOCS	# of functions
License Checker	101	2
TCAS	173	9

Table 4 - Subject applications used in the experiment

[1] A custom communication library was used for the experiment. However, the tool is supported by the ASPIRE client/server communication logic

We selected applications that come with available test cases. Tests are fundamental to ensure that our protection, when applied, does not alter the correct behaviour of the applications.

License Checker: *License Checker* is a C program that checks if the license of a software component is still valid compared with the current date.

We can identify two sensitive variables an attacker can tamper with:

- The variable that holds the license emission date;
- The variable that holds the current date.

A malicious user might tamper with these two variables by, e.g., adding a value to the variable that stores the current date to fool the license check algorithm and to illegally validate his/her license, which would have been expired under normal circumstances. This kind of attack can be mitigated by applying client/server code splitting.

TCAS: *TCAS* (*traffic collision avoidance system)*, is a C program used for the purpose of aircraft collision detection that verifies statuses of planes according to several parameters that can be passed as input.
Also in this case, several variables can be identified as sensitive (for example, variables that hold values on the plane status), which can be protected by client/server code splitting.

4.3 Experimental Procedure

Configuration extraction for License and TCAS applications: In order to apply client/server code splitting, the code of the application to protect must be annotated with splitting annotations, as described in Section 4.1.

To test our tool in a more extensive way, we also implemented a *configurator extractor*, a CodeSurfer script written in Scheme that uses heuristics to automatically extract and define a set of configurations of barriers and slicing criteria, to be used with License Checker and the TCAS program. The source code of the two applications was subject to *configuration extraction*, in order to extract as many annotation configurations as possible. Here, the focus is not on the security of the application: in fact, some of the configurations we extracted can be trivial from the security point of view. Nevertheless, they are useful for estimating the possible performance degradation introduced in the case studies by client/server code splitting.

The computation of the configurations is performed by the Scheme script we developed on top of CodeSurfer. The script exhaustively extracts all the possible configurations, i.e. valid combinations of barriers and criteria, for each method/function that appears in the code. Each configuration is then converted into code annotations that are added to the source by means of a python script. Then, each splitting configuration produces a copy of the original application with the annotations included.

By means of configuration extraction, we generated 27 valid annotation configurations for License Checker, and 9 for TCAS.

Input values/scenarios: we defined different execution scenarios for the case studies. For License Checker and TCAS, they correspond to test cases available for the two applications.

For License Checker, we selected 2 scenarios, corresponding to licenses emitted on different dates. In one scenario, the emitted license was valid, while in the other the license was expired. For TCAS, we randomly selected 2 different test cases among the ones provided with the application.

For both License Checker and TCAS, we added an artificial loop of 1000 executions of the *main* function, to avoid that constant setup time required to start a process dominates the

actual execution time. With this modification, execution time can be measured in a more precise way.

For each annotation configuration, the protected application is executed along with the secure server, once per scenario. To make sure client/server code splitting preserves the original semantics of the programs under analysis, the output produced by the protected application is compared with the output of the original application. Eventually, we measured execution time and memory consumption on the protected client and on the server. During executions, communication-related metrics were also collected.

The experiment has been conducted on a Desktop machine equipped with Intel Xeon 3.3 GHz CPU (4 cores), 16 GB of memory, running Red Hat 6.5 64 bit. Both client and server executables were executed on the same machine.

4.4 Experimental Results

The Section reports the results obtained from the experimental validation of the client/server code splitting approach on the three case study applications presented earlier in the document. In particular, Section 4.4.1 presents the experimental results we obtained on the License Checker and on TCAS for execution time and memory overhead.

4.4.1 License Checker and TCAS

Execution time: The diagram in Figure 1 shows the client execution time for License Checker, when more and more messages need to be exchanged between client and the corresponding server. Execution time is on displayed on the y axis, expressed in seconds, while the amount of messages is displayed on the x axis.

As can be seen in the graph, execution time seems to show a linear trend. To verify the statistical significance of the observed trend we used the *Pearson correlation* test. The Pearson's correlation test computes the correlation coefficient ρ, a measure of the strength of the linear relationship between two variables. It ranges from -1 to +1, where the extremes indicate perfect (positive or negative) correlation and 0 means no correlation. Statistical significance is assumed when this test reports a p-value is <0.05 (we assume significance at a 95% confidence level, α=0.05).

For the case of Figure 1 we have a correlation coefficient ρ equals to 0.97, with a p-value < 0.01, which means that we have a statistical significant case. Then, we can say that, if the number of messages to exchange with the server increases, the execution of the protected application slows down.

The Pearson's correlation is computed between dependent and independent variables. We can identify the following independent variables:

- the number of sensitive variables that are protected by applying client/server code splitting;
- the total number of statements moved from client to server, which roughly corresponds to the size of the barrier slice that is computed by the protection;
- the total number of messages client and server need to exchange to execute correctly and to keep the execution synchronized.

Dependent variables, instead, are:

- execution time (client);
- execution time (server).

We computed Pearson's correlation for each couple of dependent variable/independent variable, for License Checker and TCAS.

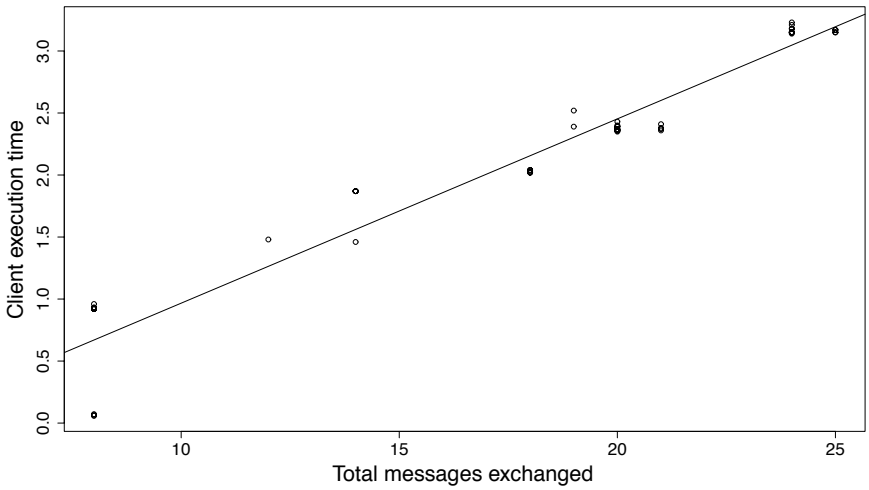

Figure 1 - Client execution time vs total messages exchanged (License Checker)

All the results can be found in Table 5. The case studies, License Checker and TCAS for both client and server programs, are reported on rows, while columns show:

- the Pearson correlation between the number of sensitive variables (independent variable) and execution time (dependent variable), with the p-value and the slope *m* of the interpolating line (column *Variables*);
- the Pearson correlation between the number of statements moved on the server (independent variable) and execution time (dependent variable), again with p-value and the slope m (column *Statements*);
- the Pearson's correlation between the number of exchanged messages (independent variable) and execution time (dependent variable), with p-value and the slope m (column *Messages*).

As mentioned earlier in the Section, the number of total messages the client application and the secure server exchange slow down the execution of the protected program. This trend is visible on both License Checker and TCAS, for client and server (see Figure 2 for TCAS at client-side). In case of License Checker, we can say that degradation is 0.148 second per each message exchanged (client), and 0.148 seconds for the server. For TCAS, degradation can be measured in 0.178 seconds per message in case of the client, and 0.178 seconds in case of the server.

For the License Checker, the number of the statements moved on the secure server and the number of the sensitive variables has also an impact on the execution time. Performance degradation can be estimated in 0.504 seconds per each sensitive variable (client), and 0.502 seconds (server). We have a degradation of 0.074 seconds per each additional statement at the client side, and 0.075 seconds at the server side.

TCAS does not show any statistically significant correlation between the number of sensitive variables and execution time, nor between the number of statements and the execution time.

Memory overhead: Figure 3 shows the server memory overhead for License Checker when more and more messages are exchanged by client and server (memory consumption is displayed on the y axis, while the messages are on the x axis). Memory overhead, similarly to what observed in case of execution time, seems to follow a linear trend in the number of messages exchanged. Also for the memory, we used the Pearson correlation test to verify the statistical significance of the observed trend.

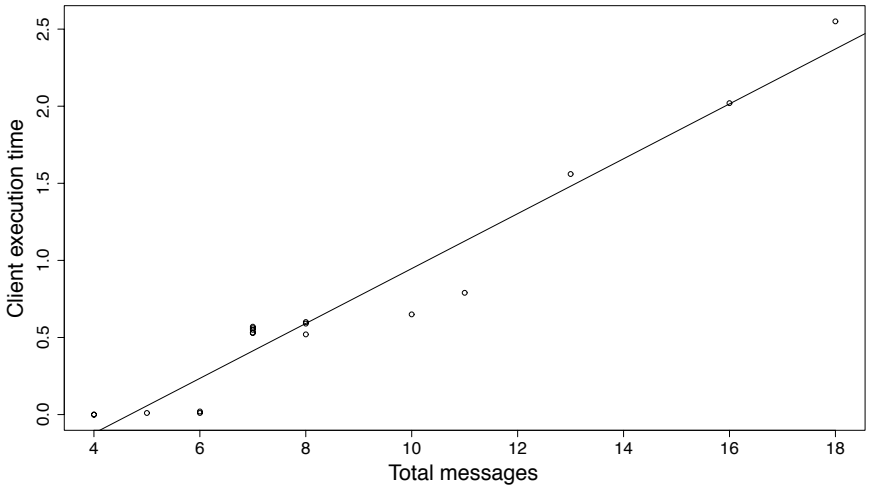

Figure 2 - Client execution time vs total messages exchanged (TCAS)

SUT		Variables			Statements			Messages		
		ρ	p-value	m	ρ	p-value	m	ρ	p-value	m
License Checker	Client	**0.81**	**<0.01**	0.504	**0.73**	**<0.01**	0.074	**0.97**	**<0.01**	0.148
	Server	**0.80**	**<0.01**	0.502	**0.73**	**<0.01**	0.075	**0.97**	**<0.01**	0.148
TCAS	Client	0.40	0.10		0.01	0.96		**0.97**	**<0.01**	0.178
	Server	0.40	0.10		0.01	0.96		**0.97**	**<0.01**	0.178

Table 5 - Pearson's correlation between variables/statements/messages and execution time

While the independent variables remain the same as in the case of the execution time, the dependent variables are:

- memory consumption (client);
- memory consumption (server).

Again, we computed the Pearson's correlation for each possible couple of dependent variable/independent variable.

For the case of Figure 3, we have a correlation coefficient ρ equals to 0.97, with a p-value < 0.01, which means that there is a statistically significant correlation between the total amount of exchanged messages (the independent variable) and the amount of memory consumed by the protected program at the server side (the dependent variable).

Table 6 reports the full results we obtained for memory overhead. The case studies, License Checker and TCAS, for both client and server programs, are reported on rows, while columns show:

- the Pearson's correlation between the number of sensitive variables (independent variable) and memory overhead (dependent variable), with the p-value and the slope *m* of the interpolating line (column *Variables*);

- the Pearson's correlation between the number of statements moved on the server (independent variable) and memory overhead (dependent variable), again with p-value and the slope m (column *Statements*);
- the Pearson's correlation between the number of exchanged messages (independent variable) and memory overhead (dependent variable), with p-value and the slope m (column *Messages*).

For License Checker, we have statistically significant correlation in all the cases. However, a relevant memory overhead can be observed only at the server side. It consists of 3.9 MB per each additional sensitive variable, of 735 KB per each additional statement, and 1.2 MB per each additional message.

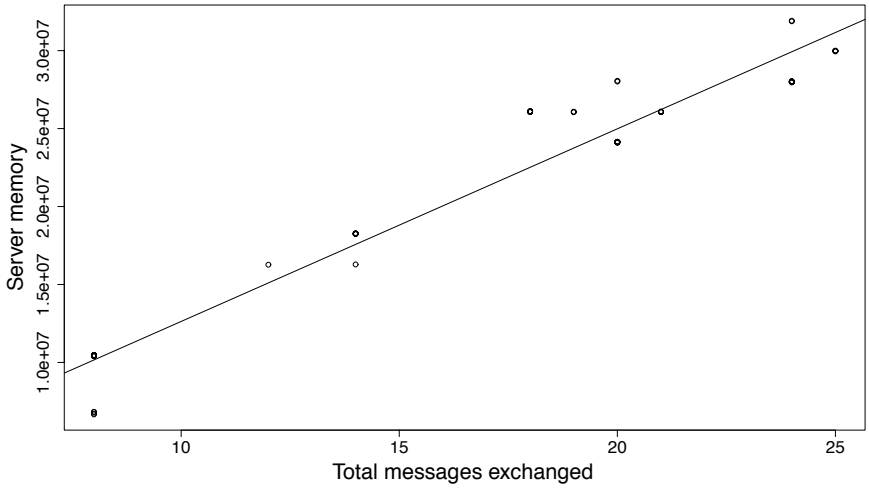

Figure 3 - Server memory overhead vs number of messages (License Checker)

At the client side, even if all the cases are statistically significant, we have a smaller memory overhead with respect to the server side. If execution time on client and server showed similar, almost equal, degradation, the memory overhead differs, in some cases, of three orders of magnitude. Figure 4 shows the memory overhead at the client side, when the number of statements that must be moved on the server increases. As can be seen, the graph in the Figure still shows a linear trend, but the slope of the interpolating line is visually less steep than it was in the other case. This suggests that increments in terms of memory overhead per each additional message are small. In fact, the memory overhead at the client side resulted to be quite small, 6 KB (1.2 MB at the server side) per each additional message. Also for the other metrics, the memory overhead at the client side reaches a maximum of 17 KB (3.9 MB at the server side) for each sensitive variable and 4 KB (735 KB at the server side) per each additional statement.

For TCAS, we identified only two statistically significant cases, between statements and memory at the client side, and between messages and memory at the server side. In both the cases, the overhead is negligible: 300 B maximum for each additional message at the server side, while degradation is practically 0 for each additional statement at the client side.

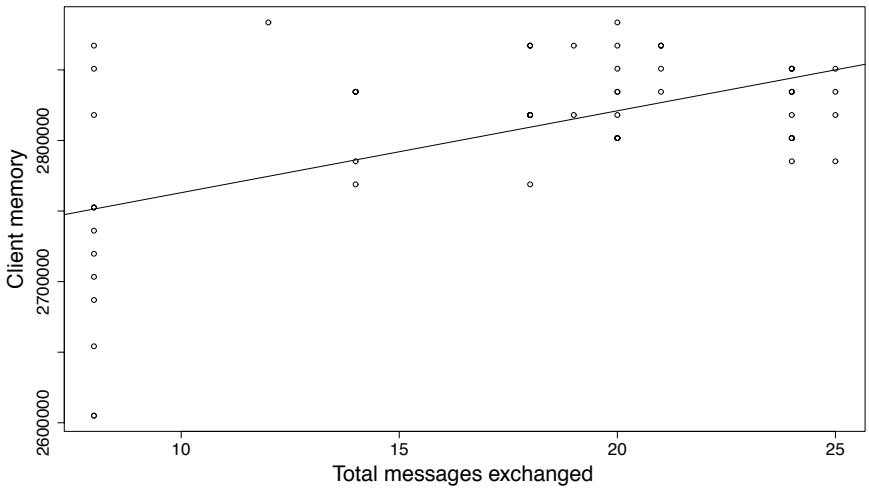

Figure 4 - Client memory overhead vs number of statements moved on the server (License Checker)

SUT		Variables			Statements			Messages		
		ρ	p-value	m	ρ	p-value	m	ρ	p-value	m
License Checker	Client	**0.42**	**<0.01**	17	**0.60**	**<0.01**	4	**0.56**	**<0.01**	6
	Server	**0.75**	**<0.01**	3902	**0.86**	**<0.01**	735	**0.97**	**<0.01**	1235
TCAS	Client	-0.12	0.64		**0.51**	**0.03**	~0	0.19	0.46	
	Server	0.46	0.05		0.02	0.94		**0.99**	**<0.01**	0.3

Table 6 - Pearson's correlation between variables/statements/messages and memory

4.5 Early experimentation on Diamante

We also conducted a preliminary experiment on the Diamante use case from SFNT (x86 version), taken from Work Package 6, with the goal of verifying if client/server code splitting can be applied on a large and complex application. In fact, the main Diamante source file consists of more than 1000 lines of C code, 10 times bigger than License Checker and TCAS, the case studies we used for the experiments presented earlier in the document.
In order to make client/server code splitting applicable on Diamante, we manually defined 11 annotation configurations. These annotations are intended for testing the client/server code splitting tool only, rather than representing a real effort to define annotations that are meaningful in terms of an asset security point of view. However, we followed SFNT guidelines about critical assets in Diamante to write the annotations, when possible.
For each annotation configuration, we applied our tool to produce the protected client-side application and the secure server. Then, the application is executed along with the secure server to check for potential issues introduced by the protection.
The results of the experiment are still being processed and evaluated.

4.6 State of the tool

- **Implementation of the tool**: the current version of the tool is 1.1.0.

- **Stand-alone testing of the tool**: the client/server code splitting tool was exhaustively tested on two C applications, License Checker and TCAS (Section 4.4.1). Tests on the ASPIRE use cases are still ongoing.
- **Integration of the tool**: the tool is integrated in the source level part of the ASPIRE Compiler Tool Chain (ACTC) as SLP06 component. Integration was tested on examples provided with the ACTC.
- **Technical details of the tool**: they are listed in Table 7. For each component/technology used to develop the tool (first column), a brief description is given in the second column (Description/Role). The third column (Lines of code) shows the number of lines of code developed for a specific component/technology.

Component/Technology	Description/Role	Lines of code
Shell scripts	Internal orchestration/coordination invocation of CodeSurfer and TXL programs	606
CodeSurfer programs	Perform extraction of information from the source code of the program to protect Written on top of CodeSurfer Programming language: Scheme	3406
Communication libraries	Allow communication between client and server components Programming language: C	861
TXL programs	Perform source code transformation Programming language: TXL	8375
Python scripts	Perform conversion of Pragmas Programming language: Python	110

Table 7 - List of components/technologies used in the client/server code splitting tool

4.7 Plan

At M24, the client/server code splitting tool is integrated in the ACTC.

The tool has been implemented and performance has been assessed on License Checker and TCAS. We started to evaluate the performance on SafeNet Diamante application. Further tests on the SafeNet use case are ongoing. Similarly, we will conduct the performance evaluation the ASPIRE use cases.

Outcome of the tests we want to conduct is fundamental for the activities we planned for the 3rd year of the project. In particular, we will focus on:

- Continuously improving the performance of the client/server code splitting tool, when applied;
- Studying the performances of the industrial case studies when client/server code splitting is applied.

Section 5 Remote Attestation

Section Authors:

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

This section covers the work performed in task T3.2. It describes:

- the updates to the remote attestation reference architecture compared to deliverable D3.02, which has been extended to support multiple clients, running multiple attestators of one or more attestation type;
- the updates to the static remote attestation, also presented in deliverable D3.02 which has been improved to supported diversified version of the static remote attestator for renewability purposes;
- the current status of the dynamic remote attestation development, that is first presented here;
- the current status of reaction mechanisms implementation, compared to deliverable D3.02, which has been detailed both for client and server components;
- updates to the delay data structures, which are now integrated in Diablo;
- the integration of the remote attestation protection technique into the ACTC.

Finally, this section presents the planning for the next months.

5.1 Remote attestation architecture and workflow

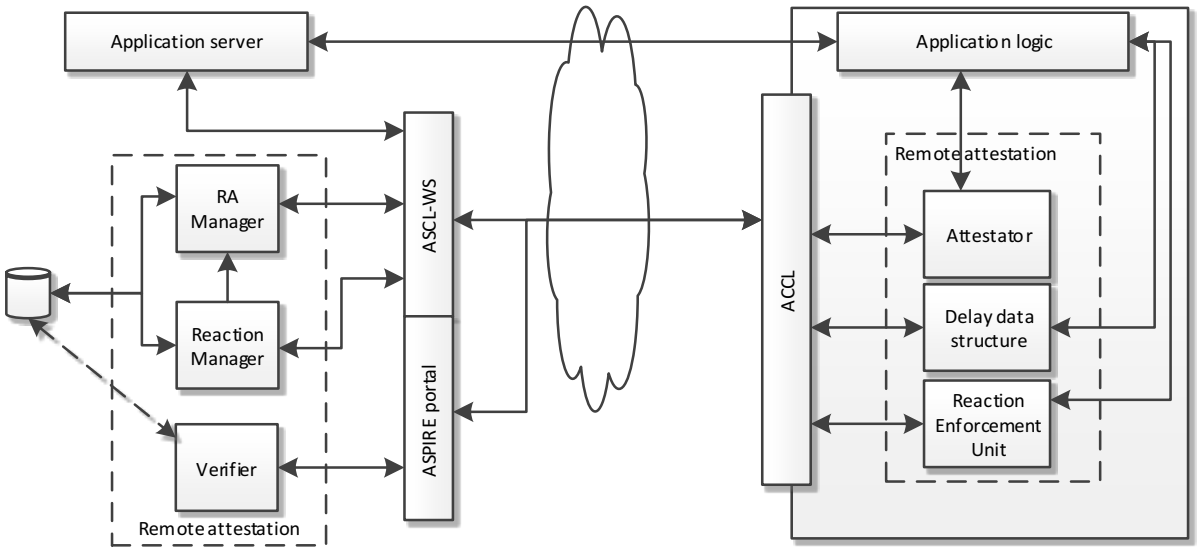

Figure 5 - Remote attestation reference architecture with a single client.

Figure 5 presents the M24 remote attestation reference architecture. Compared to the architecture in the deliverable D3.02, there are two major changes, all in the server side part.

Change 1: The reference architecture shows details about the server-side communication logic. As described in D1.04 v2.0, these are the communications to be executed:

- the RA Manager sends a client Attestator an attestation request;

- the client Attestator sends an attestation response to the Verifier;
- the Reaction Manager sends reaction reports that are written into the Delay Data Structure.

On one hand, the RA Manager and the Reaction Manager use the ACCL WebSocket Protocol. This component must initiate a communication with the clients they have to attest and to which they have to notify reactions. On the other hand, the Verifier, which only receives attestation responses and does not need to initiate a communication, can use the ACCL Simple Protocol if this can improve performances.

Change 2: The RA Manager and the Reaction Manager now have a direct communication.

As will be explained later in Section 5.1.2, the RA Manager exposes an API that can be used by the Reaction Manager to adapt the attestation frequency in case of suspicious clients (that may be already been compromised).

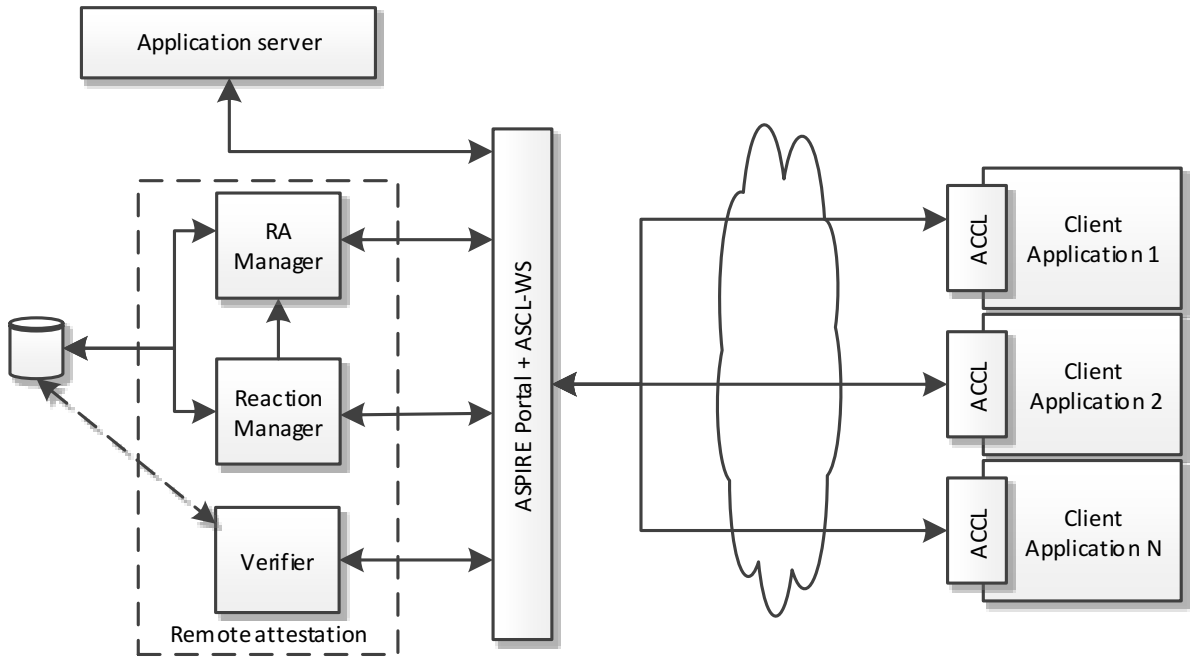

Figure 6 - Remote attestation reference architecture with multiple clients.

Figure 6 shows the remote attestation reference architecture when multiple clients are connected to the main server. No changes are required to the server side architecture, as the ASCL already contains the features to support multiple clients. However, as it will be evident later in this document, the server-side components have been designed to maintain state information for all the clients that need to be protected with RA.

5.1.1 Verifier

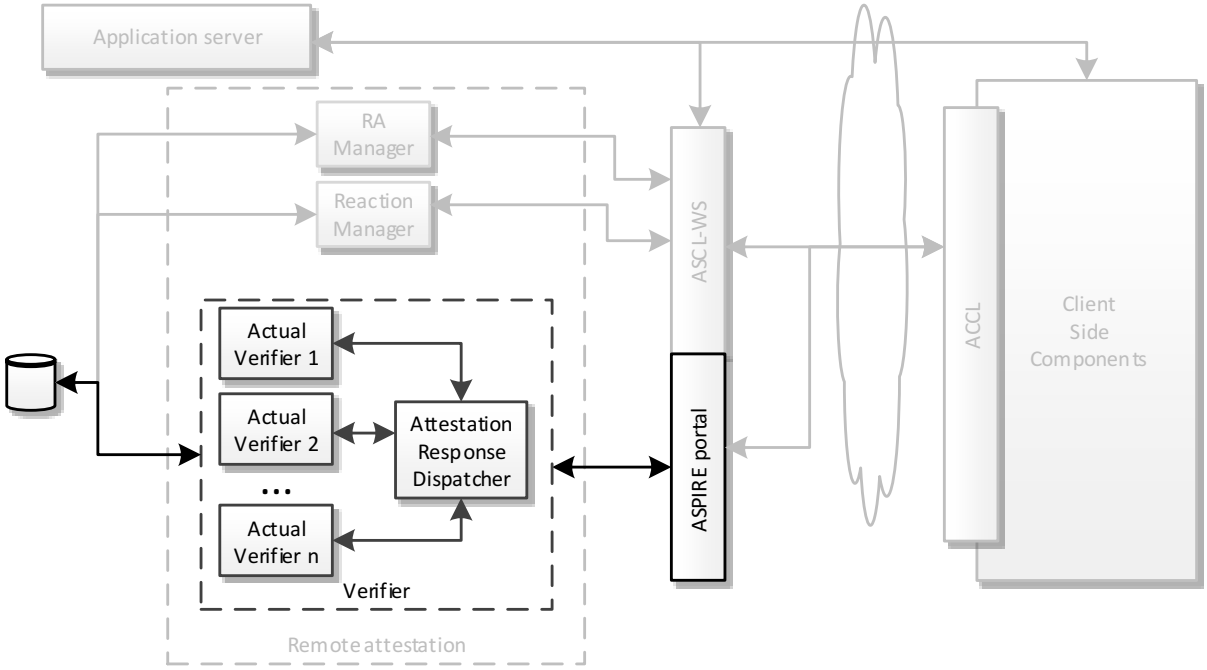

Figure 7 - Architecture of the Verifier.

Figure 7 shows the architecture of the Verifier, which is composed of an Attestation Response Dispatcher and several Actual Verifiers (Actual Verifier 1 to Actual Verifier n). Several Actual Verifiers are needed because we support more than one client, and each client may have been protected with different RA techniques. More precisely, one client may have been protected with zero or more attestation techniques, and several clients may use the same attestation technique. Therefore, the Attestation Response Dispatcher forwards attestation responses to the proper Actual Verifier.

5.1.1.1 Attestation Response Dispatcher

This element dispatches the attestation responses received from the client Attestators and routes them to the right Actual Verifier. The Attestation Response Dispatcher receives the attestation responses from the ASCL. Then, it deduces the association between the client and the Actual Verifier to use by reading information from the ASPIRE DB. When a client is not associated to one and only one Actual Verifier, it is able to determine the Actual Verifier to which the response needs to be sent by analysing the information in the attestation response. Then, it instantiates the proper Actual Verifier and passes it the request.

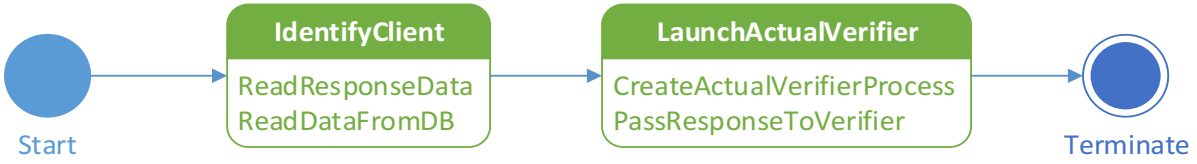

Figure 8 - Attestation Response Dispatcher workflow (as a Finite State Machine).

Referring to Figure 8 the dispatcher works as follows:

- It receives an attestation response by the ASCL (ReadResponseData);
- It receives client identification (clientID) data by the ASCL (ReadResponseData);
- It reads client information from the ASPIRE DB to determine the actual verifier to use (ReadDataFromDB);

- If the Actual Verifier is not univocally determined, it unpacks the response, reads attestation response data, to univocally determine the Actual Verifier (ReadDataFromResponse);
- It launches the Actual Verifier process as an independent process (CreateActualVerifierProcess);
- It passes the attestation data to the Actual Verifier (PassResponseToVerifier);
- It terminates its execution.

5.1.1.2 Actual Verifier

The Actual Verifier is a component that is able to emit a verdict on the correctness of an attestation response received from a client. Each Actual Verifier reads attestation request information from the DB in order to get the original information sent by the RA Manager and to estimate the delay between when the attestation is sent and the response received. Each Actual Verifier uses one or more ad hoc tables in the DB to store technique-specific information. The verifier, in order to achieve its goal, exploit two kinds of data from DB:

- the information about the request made by the manager;
- the pre-computed attestation data.

The former is used to deduce the nonce and the prepared data relative to the request made and sent by the RA Manager. The latter one is combined with the application id and the nonce to produce the expected attestation response to be compared with the returned one. For example the static RA Actual Verifier uses two DB tables, namely ra_request and ra_prepared_data, whose structures are reported in Table 8 and Table 9.

Table 8 - Structure of ra_prepared_data table.

Column name	MySQL type	C type	Description
id	bigint	uint64_t	Record id and primary key
application_id	bigint	uint64_t	Id of ra_application record the current record is associated
nonce	tinyblob	uint8_t *	Nonce value
data	longblob	uint8_t *	Attestation data generated by the nonce value

Table 9 - Structure of ra_request table.

Column name	MySQL type	C type	Description
id	bigint	uint64_t	Record id and primary key
prepared_data_id	bigint	uint64_t	Id of ra_prepared_data record that contains the prepared data used for the request
product_id	bigint	uint64_t	Id of the ra_product record for which the request was generated
send_time	timestamp	time_t	The time when the request was sent

response_time	int	uint32_t	Number of seconds between when the request was sent and the response received
status	bigint	uint64_t	Id of ra_status record that describes the request current state
validity_time	smallint	uint16_t	Maximum response_time for a response to be valid

It is worth noting that for performance optimization and scalability purposes, the proposed architecture allows the use of more than one Actual Verifier for the same type of remote attestation.

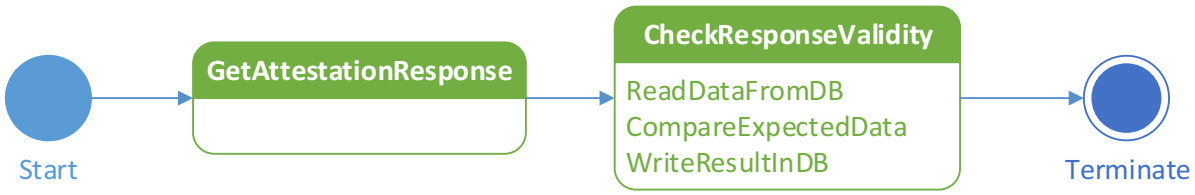

Figure 9 - Actual Verifier workflow (as a Finite State Machine).

The Actual Verifier is launched by the dispatcher. Referring to Figure 9:

- It reads an attestation response passed from Attestation Response Dispatcher (GetAttestationResponse);
- It performs the attestation response verification, that is, it compares data received in the attestation response with the expected values (CompareExpectedData), calculated with or without pre-computed data taken from the DB (ReadDataFromDB);
- It writes the verification result into the DB (WriteResultInDB);
- It terminates its execution.

5.1.2 RA Manager

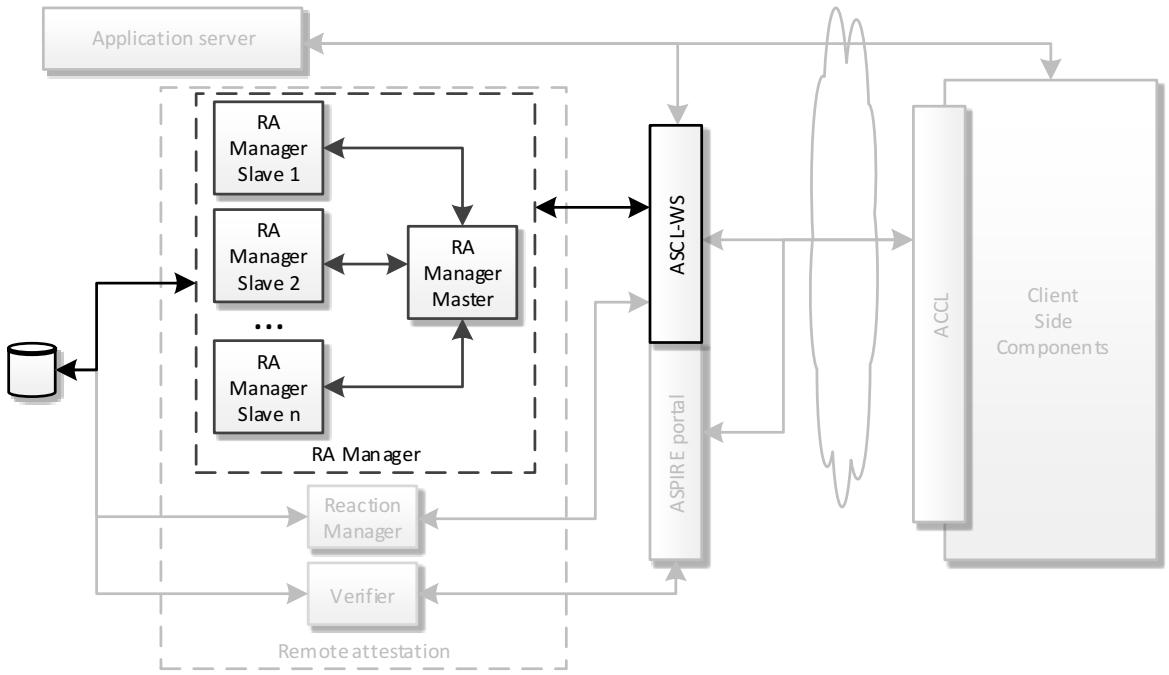

Figure 10 - Architecture of the RA Manager.

The RA Manager is the server side-remote attestation component in charge of sending the attestation requests to clients.
Therefore, the RA Manager uses one RA Manager Master, which is able to manage connection and disconnection of clients, and several RA Manager Slaves, which actually generate and send attestation requests to the clients they were/are assigned to by the RA Manager Master.

5.1.2.1 RA Manager Master

The RA Manager Master is the component invoked by the ASCL WebSocket Protocol when a client connects to the ASPIRE Portal or when one of the connected clients performs the shutdown. When a client connects, the RA Manager Master reads from the ASPIRE DB information about the attestation frequency and the RA protection techniques the client implements. Based on this information, it estimates the effort required to attest the client and assigns the client to the proper RA Manager Slave. After having passed the responsibility of the client to one of the RA Manager Slaves, the RA Manager Master will ignore all other communications from the client until the client performs the ACCL WebSocket Protocol Connection Closed function. When a shutdown is caught, the RA Manager Master notifies the proper RA Manager Slave that it must stop to serve the client.

Theoretically, RA Manager Slaves can be instantiated when needed by the RA Manager, i.e., when the resources needed to process a newly connected client are not enough. This architecture is therefore prone to scale well also in the cloud. However, in the current implementation, which is based on a single server, the RA Manager Master initiates all the RA Manager Slaves when it starts. The number of the initiated RA Manager Slaves depends on the number of cores/threads available at the server CPU.

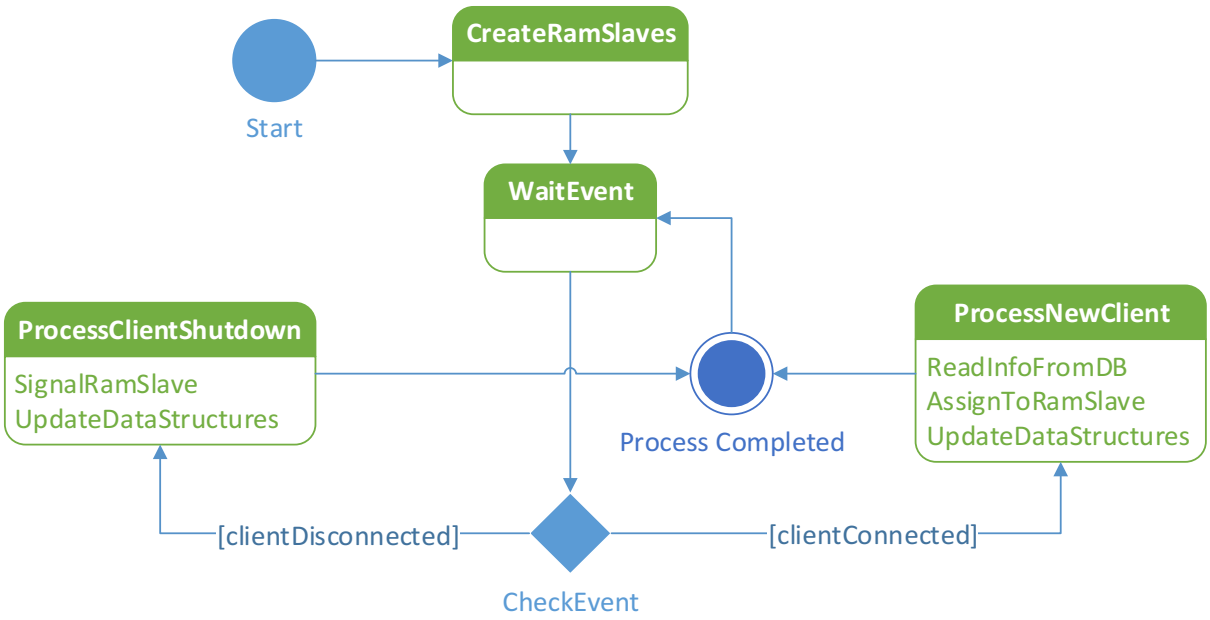

Figure 11 - RA Manager Master workflow description (as a Finite State Machine).

Figure 11 shows the workflow of the RA Manager Master, which is composed of the following operations:

- Start
- Creates and initiate RA Manager Slaves as threads (CreateRAMSlaves);
- Wait for an event thrown by ASCL-WS, infinite loop (WaitEvent)
    - If event = clientConnected: process and assign new client to slave (ProcessNewClient)
        - Get attestation frequency and RA technique the client uses (ReadInfoFromDB);
        - Select the RAM Slave which the client must be assigned to (AssignRAMSlave);
        - Record client RAM Slave association in its internal states (UpdateDataStructures);
    - If event = clientDisconnected: signal proper slave to not serve the client anymore (ProcessClientShotdown)
        - Notify shutdown to the proper client (SignalRAMSlave);
        - Remove the entry associated to client from the internal state (UpdateDataStructures);
- Return to WaitEvent state;

5.1.2.2 RA Manager Slave

The RA Manager Slave is the component that actually prepares and sends attestation requests. It manages a set of clients. The clients to manage are decided by the RA Manager Master that assigns them when they connect to the ASPIRE portal. The RA Manager Master also calls the ASCL WebSocket Channel Initialization function.

This components sleeps until a new attestation need to be prepared and sent or a new client is assigned by the RA Manager Master.

The next attestations to send are stored in a queue, which is ordered based on the time when the RA Manager Slave needs to start preparing them. If the RA Manager Slave wakes up because there is a new attestation to process, it reads the information on the next client to attest from the queue, complements this information with data taken from the DB, prepares

the attestation request and sends it. The operation that the RA Manager Slave will actually perform depends on the protection technique. Then, before going to sleep, the RA Manager Slave establishes the new time when the client will be attested and inserts this info in the queue.

More in detail, the queue contains a set of queue elements. Queue elements hold two data: the client *c* to attest and relative time *t*. That is, it stores the number of seconds *t* to wait before attesting the client *c*. This interval (*t*) is intended as the number of seconds to wait after the element before client *c* in the queue is attested. Each client appears in the queue just once.

Hence, each element in the queue specifies how much time the RA Manager Slave has to sleep after its predecessor is served.

Moreover, the first queue element represents the next client to be attested and holds the information about the time that the RA Manager Slave has to sleep before sending a request to that client.

This way, when the RA Manager Slave sleep is interrupted because of a new client connection, it can correctly update the queue by comparing the time slept and the time it should have slept.

Figure 12 shows an example of the status of the queue in a generic moment during the RA Manager Slave execution. Assume the RA Manager Slave began its sleep at t=x. Then, it is supposed to sleep 3 seconds as specified by the first element in the queue. When that sleep ends, at x+3s, App3 is served. After that, App3 is scheduled again. Suppose that App3 needs to be attested again after 5 seconds. Then the updated form of the queue is the one presented in the figure at t=x+3s. Notice that the queue keeps its consistency, i.e., App3 is served at t=x+3s, App1 is served at t=x+7s and App2 is served at t=x+9s.

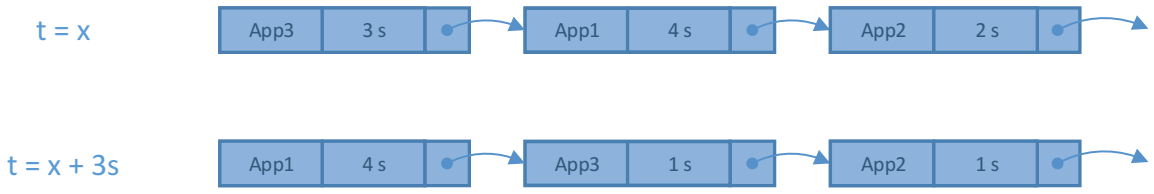

Figure 12 - Example of the client attestation scheduling.

Every client is associated to an attestation frequency value. This value is intended as the average time between two attestations. Attestations must not be sent at fixed schedule, as an attacker can restore a correct copy of the application, serve the attestation, and restart the tampered version of the application. Therefore, the RA Manager Slave randomly selects the time to the next attestation so that the average time between two attestations is close to the frequency value. The frequency value can be changed, directly in the DB, by the RA Manager or the Reaction Manager.

Attestation requests are atomic operations that cannot be interrupted when a new client connects. The new client will be served after the attestation request is sent.

When the RA Manager Slave receives a new client to process from the RA Manager Master, it verifies whether the client needs to be attested at start-up. In this case, it prepares and sends a new attestation. If the client does not need to be attested at start-up, it reads frequency information from the DB, defines the time when the client will be attested and insert in the attestation queue this information.

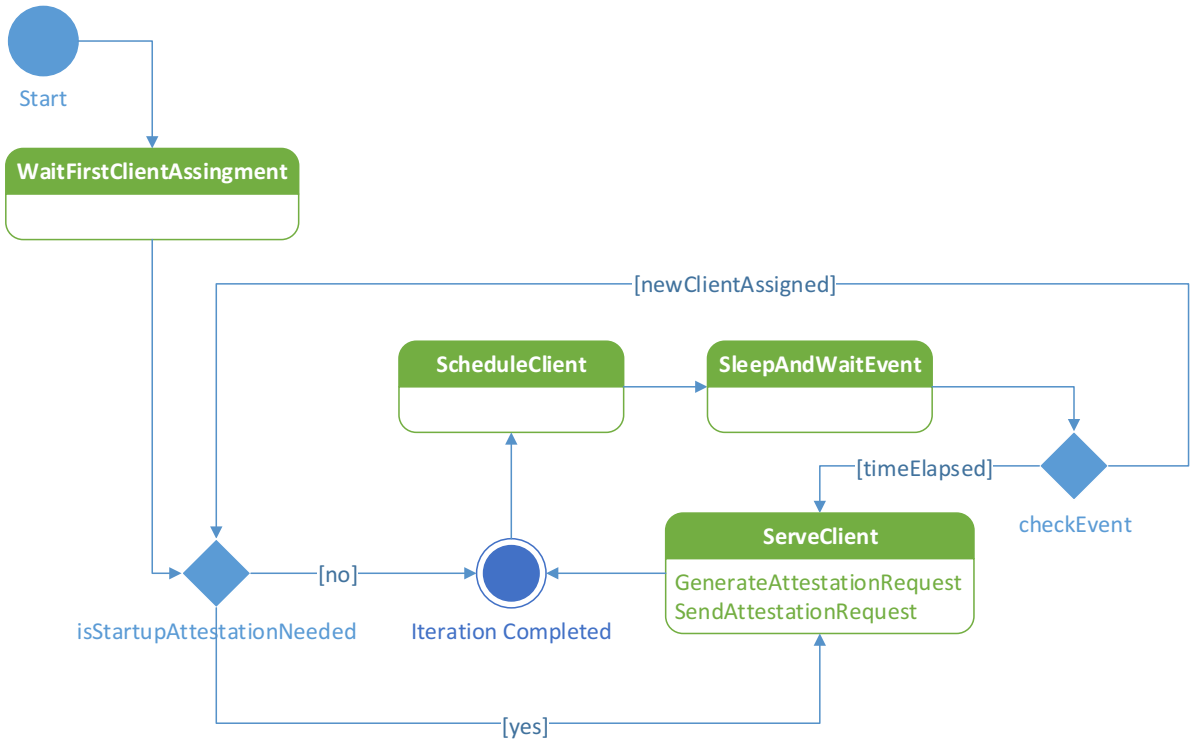

Figure 13 - RA Manager Slave workflow description (as a Finite State Machine).

Figure 13 shows the workflow of the RA Manager Slave, which is composed of the following operations:

- Start, launched by master;
- Waits to be assigned to a client (WaitFirstClientAssigned);
- When the first client connects, if it needs to be attested at startup
  - Attest the client (ServeClient)
    - prepare the attestation request (GenerateAttestationRequest);
    - send the attestation request (SendAttestationRequest);
  - Schedule the time when to attest the client (ScheduleClient)
- Enter the main loop, that is, sleep until the first client has to be served or new client is assigned (SleepAndWaitEvent);
  - if event = newClientAssigned
    - if it needs to be attested at startup, attest the client (ServeClient)
  - if event = timeElapsed
    - attest the client (ServeClient);
  - Schedule the time when to attest the client (ScheduleClient)

5.1.3 Attestator and application logic

The Attestator actually performs the attestation operations required by the technique it implements. There are no changes to this component in the reference architecture. However, since the last deliverable we have a more precise definition of the ASCL_WS, so we report in this section the interactions with this communication component.

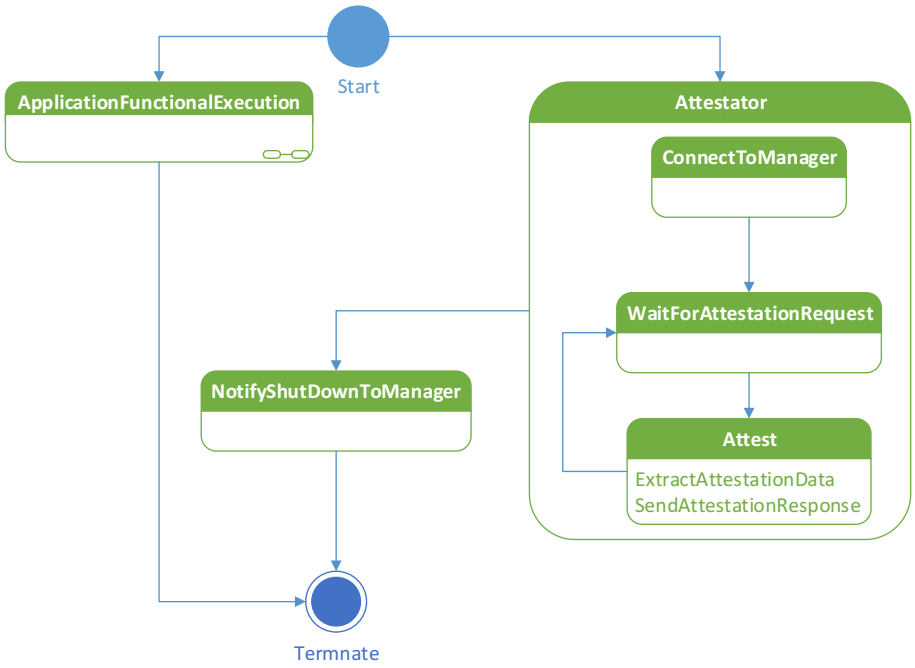

Figure 14 - Client application workflow description.

Figure 14 presents the workflow performed by the Attestator within the application. The workflow splits in three phases:

1) Start
   - The application is launched by the user. This starts
     - the application logic (ApplicationFunctionalExecution) and
     - the Attestator (Attestator);
2) Attestator execution
   - The Attestator connects to the ASPIRE portal via the ACCL WebSocket Protocol (ConnectToManager)
   - Main loop, wait for incoming attestation requests (WaitForAttestationRequest)
     - Parse the Attestation request (ExtractAttestationData);
     - Prepare attestation response and send data to server via the ACCL (Attest).
3) Terminate
   - when the application is closed (e.g., by the user)
     - The Attestator interrupts any operation and sends shutdown via the ACCL WebSocket Protocol (NotifyShutDownManager);
     - The ApplicationFunctionalExecution state terminates.

5.1.4 ASPIRE Database

No changes to the ASPIRE DB have been made compared to D3.02.

We only agreed that, to ease development and integration, individual techniques may use different tables, as was foreseen but not explicitly state in D3.02. Therefore, we have a set of global RA tables (used by the RA Manager, Reaction Manager, and all the Actual Verifiers) and one or more RA-specific tables per RA technique. All these tables (global and RA-specific) need to be created in the DB when the server is instantiated.

5.1.5 Delay data structures

One of the main concerns for any code integrity protection is to hide the link between the detection of an integrity violation and the reaction. If an attacker tries to tamper with the code, and the program fails as a consequence, it should be hard for the attacker to trace the

symptom (abortion, crash, ...) back to the cause (failed integrity check), such that he cannot easily work around the failed attestation.

In this context, delay data structures are data structures used to keep track of failed checks in a *covert* manner until a reaction is triggered sometime later in the program execution. To improve the stealth of the delay mechanism implemented with these data structures, their definition should not be hardcoded in a tool chain, as it would be all too easy for attackers to learn to recognize them.

Consequently, we aim to implement tool chain support for what we call *flexible delay data structures*. Flexible in this context means that users of a software protection tool chain should be able to supply any suitable data structure they have defined themselves. By choosing new data structures every time they protect a program, the users can prevent that attackers attack their program on the basis of knowledge learned from the protected programs of other users.

In this **completely novel** approach, the user of the protection tool should provide the data structures and an API to access them, with implementations of the corresponding API functions. The tool chain itself should then be able to integrate the data structures and the API functions into the program to be protected, and inject the necessary operations as defined in the API to store (following a performed check) and retrieve (to decide (with some delay) on a reaction) predicates in the data structure.

Even better, it should be possible for the user to instruct the protection tool to use data structures that are already available in the original program itself. If the API functions are then invoked as part of the original program and as part of the protection, it will be much harder for an attacker to identify the (newly inserted) invocations to the data structures as part of a protection scheme. In essence, the code then used for the protection has semantic relevance in the original program, and can hence not easily be omitted from the program by an attacker with advanced skills and tools such as the ones by Debray *et al.* described in the updated Section 4.4.4.2 of deliverable D1.02 v2.0.

To provide the necessary tool support for integrating this protection in the ACTC, UGent already implemented an architecture in Diablo (the link-time rewriter used for all binary-level protections in the ACTC) that allows users to define custom data structures, define functions on those data structures, and define how these interact to define predicates, which can then be injected automatically in the binary. These predicates can be used to store information about detected tampering.

First, we researched and specified a meta-API that allows users to define a set of functions that change and query these data structures, and to define the properties that the arguments to these functions must take to store true or false predicates in the structures. By allowing the user to choose, with a large degree of freedom, how true and false are stored in the data structures and how those properties are encoded in the invocations of the functions that store the predicates and that retrieve them, we aim to further improve the stealth and non-learnability. Obviously, if the inputs and outputs of the API functions would be simple boolean values, it would be all to obvious what their purpose is in the protected program.

The flexible delay data structures and functions defined for these can be defined with an XML file. For example, a set of functions on a linked list can be defined as follows:

<datastructure>

```
  <name> Linked list </name>
  <struct> struct linkedList </struct>
  <file> objectfile </file>
  <init>
    <function>
      <name> initLinkedList </name>
```

```
      <parameters> </parameters>
      <return> struct linkedList * </return>
    </function>
  </init>
  <predicate id=”1”><name>RA_failed</name></predicate>
  <predicate id=”2”><name>code_guard_failed</name></predicate>
  <transformations>
    <function>
    <name> setTampered </name>
    <parameters>
      <parameter>
        <name> var1 </name>
        <datatype> struct linkedList * </datatype>
      </parameter>
      <parameter>
        <name> var2 </name>
        <datatype> int </datatype>
      </parameter>
    </parameters>
    <return> void </return>
    </function>
  </transformations>
</datastructure>
```

This defines a data structure `struct linkedlist`, to which Diablo can initialize a pointer with the function `initLinkedList`. Furthermore, Diablo now has knowledge that there are two predicates defined on this data structure: `RA_failed` and `code_guard_failed`, and that there is a function `setTampered`, which transforms the data structure.

In order to provide maximum flexibility for the users of the ASPIRE tool chain in specifying the effects the defined functions, the semantics of the functions is described in a Domain Specific Language. For example, a user can specify the semantics of `setTampered` as follows:

```
setTampered var1 var2
| var2 > 0 => RA_failed=T
| var2 < 0 => code_guard_failed=T
| var2 == 0 => code_guard_failed=T && RA_failed=T
```

This specifies how the effect of calling `setTampered` depends on the integer value of its second argument: if it is larger than zero, `RA_failed` is set to true and the value of `code_guard_failed` is kept unchanged. If the second argument is equal to zero, both predicates are set to true. Diablo can then automatically inject code to set the correct arguments, depending on the situation.

Diablo will also link in the API functions if they are not yet present in the program itself.

At the start of the development of the necessary Diablo support for these flexible data structures, in Q4 2014, the anti-tampering techniques in ASPIRE were not mature enough yet for integration in the ACTC, so UGent decided to research the use of the data structures in another protection scenario, in which it could pursue the research independently of the other ASPIRE partners. This other scenario consists of opaque predicates, as documented in Section 5.1.2 of deliverable D2.08.

In year 3, the current framework for defining data structures and predicates thereon will be used to support delay data structures as follows: we will store the result of a (failed) anti-tampering check in one of the predicates that we associate with a delay data structure. We will define a data structure with the following properties:

- a predicate with associated query and setter functions;
- a function that changes the state of the predicate, while keeping the value of the predicate.

An example of such a data structure can be a vector; a predicate defined on it can be whether the length of that vector is odd or even. Increasing or decreasing the length of the vector by 2 keeps the value of this predicate unchanged.

Our plan is then to support delay data structures with this frameworks by letting Diablo do the following:

- Inject such a data structure in the binary.
- Choose a value for a predicate defined on this data structure that will correspond to 'no tampering detected', and inject code that initializes the data structure to this value when the protected program is loaded.
- Insert calls to the function that updates the state but keeps the predicate value throughout the program. Updating the state throughout the program means that a reverse engineer cannot simply ignore this data structure, but instead has to analyse its use globally, thus increasing the complexity of an attack.
- On the locations where anti-tampering verifications occur, use the results of these verifications to set the value of the predicate: if the check failed, the predicate is updated to the value that corresponds to 'tampering detected'; otherwise the predicate value is not modified.
- Insert calls that query the predicate throughout the program. Furthermore, code is injected to compares the resulting Boolean value to the 'no tampering detected' value. Program execution will continue in the correct way if this query returns the correct value. Otherwise, control can redirected to the Reaction Manager, which can trigger a number of different reactions.

What remains is then defining a relevant data structure and the appropriate functions that are relevant for (at least) one of the use cases; and extending Diablo to also inject calls throughout the program that do not update the predicate, and to integrate this with the anti-tampering code. This can then be easily integrated with the Reaction Manager by having it query the predicate.

5.2 Reaction

The Reaction service is the part of the Remote Attestation mechanism that triggers the action to be done when verdicts made by the Verifier need a reaction to be set. This Reaction Service is depicted in Figure 6 by the Reaction Manager. The Reaction Manager itself is made of several components described in this section. The novelty brought by the Reaction Manager is that based of verdicts some rules enable to graduate the reaction according to the configuration made for an application. Actions set by the Reaction Manager range from a notification to the Application Server to sabotage notification to the appropriate component on the device.

The Reaction mechanism is used jointly with the Remote Attestation Manager. The Verifier component of the Remote Attestation Manager sets attestations results in the ASPIRE DB. The purpose of the Reaction Manager is to query those results and based on rules and policies to send reaction notifications. Figure 15 shows the main components on the server side and on the client side.

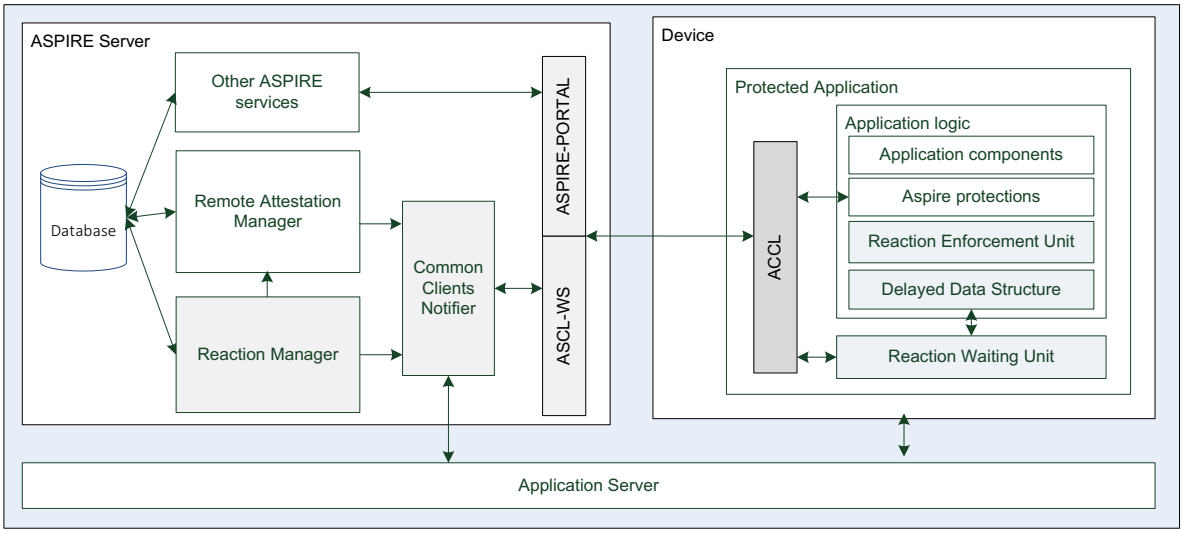

Figure 15 – Reference architecture of the Reaction Mechanism

The Reaction Manager itself is made of several components to decouple the database queries, the reaction logics and the notifications. The component on the server side that handles the communication with the various devices is shared with the Remote Attestation manager.

5.2.1 Server-side component

The role of the Reaction Manager is to trigger reaction actions based on verdicts set by the Remote Attestation Verifier component in the ASPIRE DB. Those actions are notifications to be sent either to the Application Server or to Client Applications and possibly to the RA Manager. As shown in Figure 16, the Reaction Manager unit is made of four components: the Reaction Issuer, the Reaction Manager Engine, the Notification Dispatcher and the Common Clients Notifier.

The role of these four components of the Reaction Manager is described below.

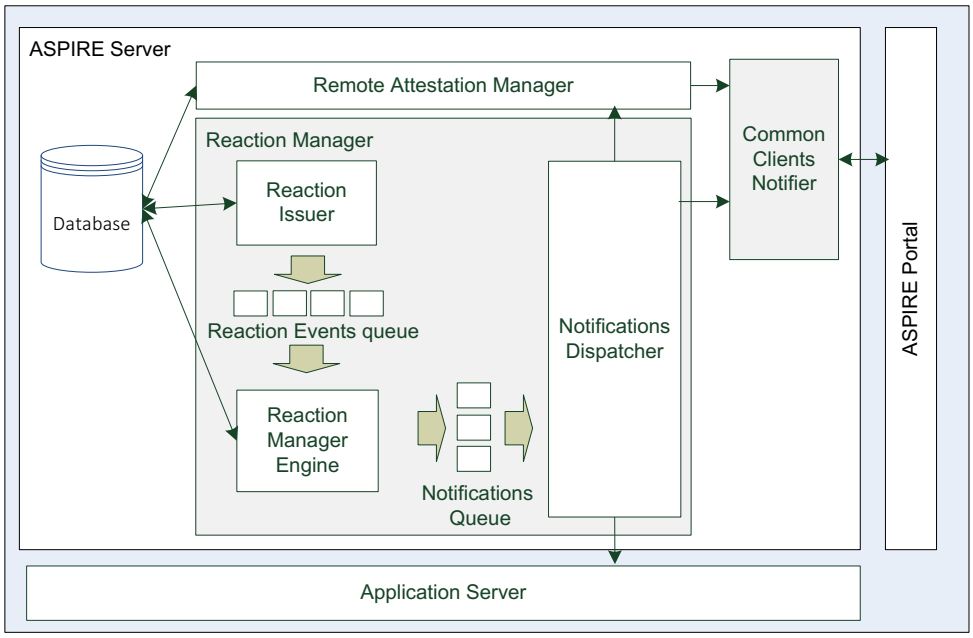

Figure 16 – Reaction Manager

5.2.1.1 Reaction Issuer

The Verifier component of the Remote Attestation writes the results of the verifications in the ASPIRE DB. The Reaction Issuer has rules to decide if a reaction has to be set for an occurrence of application on a device based on the history of the verifications.

In case a reaction is required then a Reaction Event is created and put in the Reaction Events Queue. This queue is processed by the Reaction Manager Engine.

5.2.1.2 Reaction Manager Engine

The Reaction Manager Engine processes the Reaction Events created by the Reaction Issuer in the Reaction Events Queue. It contains the various reaction strategy logics and based on these policies and the Reaction Event created by the Reaction Manager it creates Notification Events in a Notification Events Queue.

A notification can be a reaction decision to be sent to the client or a decision to suspend a delayed destruction previously notified. For the same reaction the Reaction Strategy may decide to put several Notifications in the queue: one reaction Notification to be sent to the client and a Notification to the RA Manager to increase or decrease the frequency of attestation.

5.2.1.3 Notifications Dispatcher

The Notifications Dispatcher processes the Notification Queue. Based on the type of Notifications it may notify the Application Server, notify the Remote Attestation Manager or notify the Common Clients Notifier. The purpose of this component is to decouple the reaction logic and the notifications processing.

5.2.1.4 Common Clients Notifier

The Reaction Manager needs a protocol to communicate with its client side. In an ideal world the reaction mechanism would rely on opaque data structures that would be conveyed by the Application Server to the client application on an answer to a client request through the application protocol. Such a design would avoid a dedicated Reaction Client component waiting for possible reaction requests sent be the reaction mechanism. Unfortunately, such design is much too invasive for the application, so a dedicate channel has to be provided.

The Common Clients Notifier maintains the connections with all the client applications to allow notifications to be sent to clients. When notified by the Notification Dispatcher, the Common Clients Notifier sends the notification to the Reaction Manager Proxy located on the client side. Like in other components of the server the ASCL WebSocket library is used to enable this feature. This component is shared with the RA Manager.

5.2.2 Client-side components

On the client side a component needs to set the connection with the Reaction Manager located on the server and to get the reaction notification if it is sent by the Reaction Manager. The reaction notification must be stored under a Boolean form in a data structure to enable reaction to be triggered.

This data structure is the Delayed Data Structure (DDS) described in Section 5.1.5 and that is depicted as the Remote Attestation reference architecture of Figure 5. A reaction component such as the Software Time Bombs is the Reaction Enforcement Unit in the figure. A component needs to handle the communication with the Reaction Manager. This component is the Reaction Waiting Unit in Figure 17.

The challenge with this architecture is to find a way to implement the Reaction Waiting Unit in such a way that it cannot be removed by the attacker. Indeed this component is not required to enable the application to work as it should. It is even exactly the contrary: the Reaction Waiting Unit is there to convey the data that will provoke a failure in the application.

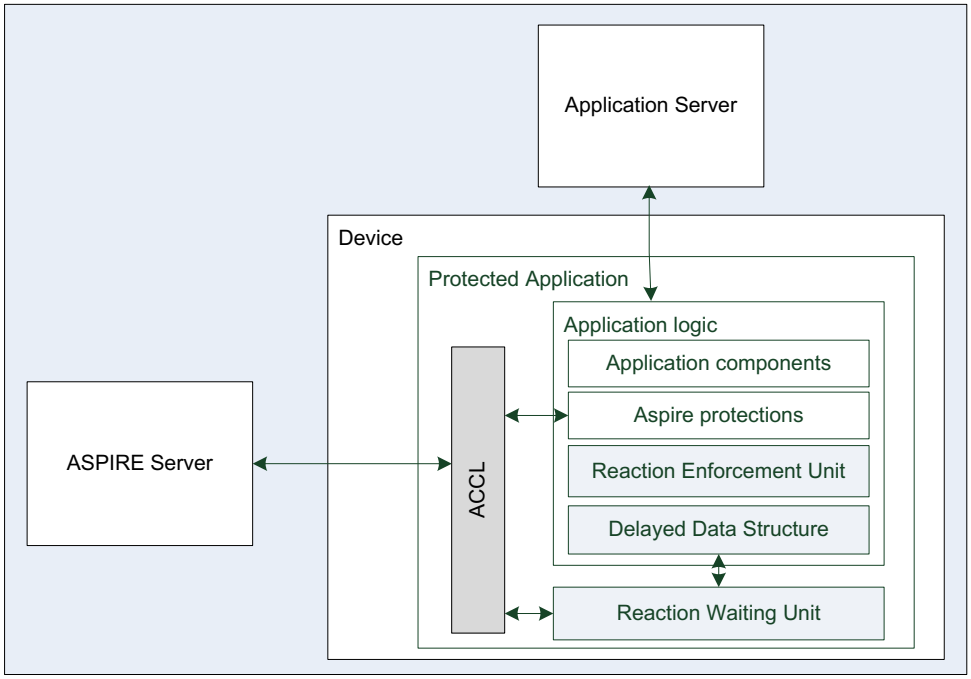

Figure 17 – Reaction components on the application client side

5.2.2.1 Reaction Waiting Unit

The WebSocket protocol enables the Reaction Manager to notify the Reaction Waiting Unit out of a dedicated request sent by the client. Still, it supposes an initial handshake at the initiative of the Reaction Waiting Unit to set the bi-directional communication.

The Reaction Waiting Unit cannot run in the main thread as it handles the communication with the server. It must run in a separate thread.

The only way to avoid this thread to be stopped by the attacker would be to hide its inherent protection nature into a useful service server like a synchronized time service or something similar that has some importance for the application. This service would be attached in some way with the client application to prevent its removal.

The real mandatory feature implemented by the Reaction Waiting Unit is to set the DDS when the Reaction Manager sends a notification.

5.2.2.2 Reaction Enforcement Units

Reaction Enforcement Units are the implementation of the reaction mechanism inside the application. Reaction Enforcement Units would actually start their action based on a query on a predicate in Delayed Data Structures. An example of a reaction is described in the next section.

5.2.2.3 A Reaction Enforcement Unit implementation: Software Time Bombs

Software Time Bombs is a Reaction Enforcement Unit. It is a client side technique that enables to sabotage the application with a time delay to prevent the attacker to relate the application degradation effect to the reaction cause.

In case of attack detection or because of an abnormal behaviour detected in the application a reaction mechanism is required to stop the execution. If the application is stopped right after the detection, there is a high probability that the attacker will make the connection between

the detection mechanism and the reaction mechanism. To mitigate the capacity of the attacker to locate and to neutralize the reaction phase it must be de-synchronized from the detection phase. The Software Time Bomb tries to address this issue by proposing a reaction mechanism that will cause a delayed crash of the application when it is triggered.

The logic of the Software Time Bomb can be depicted with the following pseudo-code. The predicate is there to illustrate the logic but it is not part of the reaction mechanism.

```
int main() {

  structure *critical_pointer;
  int timebomb = 0;

  /* .... */

  while (true) {

    if(!predicate)
      timebomb=1;

    /* ...
       logic of the application using the structure
       pointed by critical_pointer
    */

    timebomb *=2
  }

  /* ... */
}
```

This protection will not work if it is implemented as-is in source code because of the overflow protection of the C language. Therefore, the implementation of the Software Time Bomb should be done at a lower level.

The favourite way in ASPIRE to implement such mechanism is through the Diablo framework because it operates at binary level and exposes adequate functions to manipulate the code. Before implementing the Reaction Mechanism with Diablo a technical study has been done and is reported in this section.

In a first step the Software Time Bomb has been prototyped using gcc inline assembly. This feature enables to:

- insert the Software Time Bomb reaction mechanism directly in the C source code,
- prevent the optimizer phases of the compiler to modify the Software Time Bomb logic,
- explicitly set the registers used by the assembly sequence. No need to take register saves/restores into account,
- use an *lvalue* in the C code that is defined in the assembly sequence

Some prototyping has been done with code insertion done with scripts, out of the ACTC. In the following code the annotation indicates that the loop can be run twice:

```
int j = 0
while (j<3) {
    #pragma ASPIRE begin protection(bool_assertion,count_pass,max=2)
    printf("j = %d\n",j);
    #pragma ASPIRE end
    j++
}
```

Then assembly code has been generated to trigger the Time Bomb:

```
int j = 0
while (j<3) {
/* ASM TEST bool_assertion,count_pass,max=2 */
__asm__ __volatile__          ("ldr  r3, =c2\n    "
"ldr    r2, [r3, #0]\n  "
"add    r2, r2, #1 \n   "
"str    r2, [r3, #0]"
: :
:"r2","r3","memory")
__asm__("cmp r2, #2\n"
"ldrgt r5, =tb\n"
"ldrgt r6, [r5, #0]\n"
"addgt r6, r6, #1\n"
"strgt r6, [r5, #0]\n"
"str r2, [r3, #0]"
: :
: "r2","r3","r4","r5","r6");
    printf("j = %d\n",j);
    j++
}
```

After this loop, the Time Bomb variable is equal to one. Some other code has to be inserted in the code to multiply the Time Bomb variable by a factor to make the degradation effect happen sooner.

These direct replacements have enabled to validate the approach but the assembly code is a pattern that can be spot by the attacker. Using Diablo would enable to interlace this reaction code with the application to prevent automatic detection. A nice aspect of the assembly in-lining is the register management done by the compiler while in Diablo some extra store and load will be required that can also be spotted by a static analysis of the code by the attacker.

5.2.3 Plan

The detail design of the Reaction Server side will be done in Q4, 2015 with a split between the generic part and the database queries that depends on Attestations. The implementation will be done in Q1, 2016.

On the client side the Software time Bombs will be implemented in Q1, 2016.

5.3 Static remote attestation

The static remote attestation technique is defined in D3.02. There are no specific changes to the static remote attestation reference architecture compared to the one presented in D3.02, it only reflects the changes to the RA Manager and Verifier (general changes that apply to all techniques) to support multiple clients, with several attestators.

5.3.1 Diversification

Static remote attestation uses three components to actually generate (and verify) attestation. These components can be customized.

Attestation responses are obtained as:

Hash(nonce || attestation_data || ID)

attestation_data are obtained by means of a random walk in the code areas to attest. The random walk is driven by the nonce, which is random. Currently, we have implemented two random walk algorithms: the one presented in the Section 5.3.2 of D3.02, and a new one, based on the Goldbach conjecture, which is described below.
The value of the nonce is used to determine the areas to attest and other parameters passed to the random walk. The algorithm that determines how the parts of the nonces are used to determine the areas and the parameters values is named *nonce interpretation algorithm.* Nonce interpretation functionality is divided in two libraries, the nonce encoding library and the nonce decoding library. Currently, 5 different nonce interpretation algorithm have been implemented. Finally, we make available 5 different Hash algorithms: BLAKE2, MD5, SHA1, SHA256, SHA512.
We recall here that information about the code areas to attest is stored in an Area Data Structure (ADS). There is one API to read and one API to write ADS data. Therefore, given the memory layout of the application to protect, the actual ADS representation to be merged with the application depends on the selected ADS API. Currently, only one API is implemented.
Overall, we have 4 hash algorithms, 5 nonce interpretation algorithms, and 2 random walk algorithms; therefore, we can generate 40 variants of the remote attestation that can be used for diversification purposes. Moreover, since diversified versions have different performance and security, the ADSS can choose the diversified version to use when selecting the golden combination of protection.

5.3.1.1 Goldbach random walk

The random walk version based on the Goldbach conjecture takes its parameter from the nonce embedded in the RA request received by the Attestator. The basic parameters for this random walk are:

- the Area index *k*=nonce[(31-x+1)-31] is interpreted as a x+1 bits integer (from 0 to 2^x); x depends on the number of areas to attest determined by the annotations;
- the Base *b*=nonce[0-3] is interpreted as a 32 bit binary integer;
- The size N of the attested memory area;

The nonce is a 32 bytes array; buffer[i] refers to the value of the i-th byte.Starting from those parameters the Goldbach random walk makes some further preliminary adjustments. It splits the memory area to be attested into two sub-buffers called left-buffer and right-buffer. The size of the two buffers are determined as the Goldbach partition of the memory area size as follows:

$$(N_l, N_r) = G(N)$$

Where:

- N_l is the left-buffer size;
- N_r is the right-buffer size;
- $G(x)$ is the Goldbach partition of x defined as:

$$G: 2\mathbb{Z} \rightarrow P \times P$$

$$G: \{y, z \in P, x \in 2\mathbb{Z} \mid y + z = x \ \wedge (y, z) = \min\{|y - z|\}$$

If N is an odd number, i.e., it does not belong to $2\mathbb{Z}$, N-1 is considered as the size of the memory area to be attested.

Then two new base values are computed:

- left-buffer-base as $b_l = b \bmod N_l$
- right-buffer-base as $b_r = b \bmod N_r$

The size of the buffer that will contain the final extracted data is $\mathrm{B} = \max\{N_l, N_r\}$

Then the random walk is actually performed as follows:

```
for (i=0 ; i < B ; i++)
        attestation_data[i] = M[o + b_l^i mod N_l]+ M[o + N_l + br^i mod N_r]
```

The random walk performed in this way extracts at least N-1 bytes from the attested memory area and produces a buffer, containing the attestation data, whose size is approximately N/2.

Notice that also if the memory areas are scrambled and randomized they are treated as straightforward arrays of bytes thanks to the API described in Deliverable

5.3.2 ADS and integration with Diablo

There are no updates to the ADS creation and change management in Diablo compared to D3.02.

5.3.3 ACTC integration

The static remote attestation technique is currently being integrated in the ACTC.

There are two points in the ACTC workflow where operations related to the remote attestation are performed.

1) Source Level. First of all, there is the source level RA component, which consumes static RA annotations passed using a JSON file.

Depending on the annotation, it adds the following information, (in deliverable D3.02 we have defined this information *static remote attestation descriptor*):

- Hash function to use
- Nonce encoding API implementation
- Nonce decoding API implementation
- ADS API implementation
- Attestation data preparation code

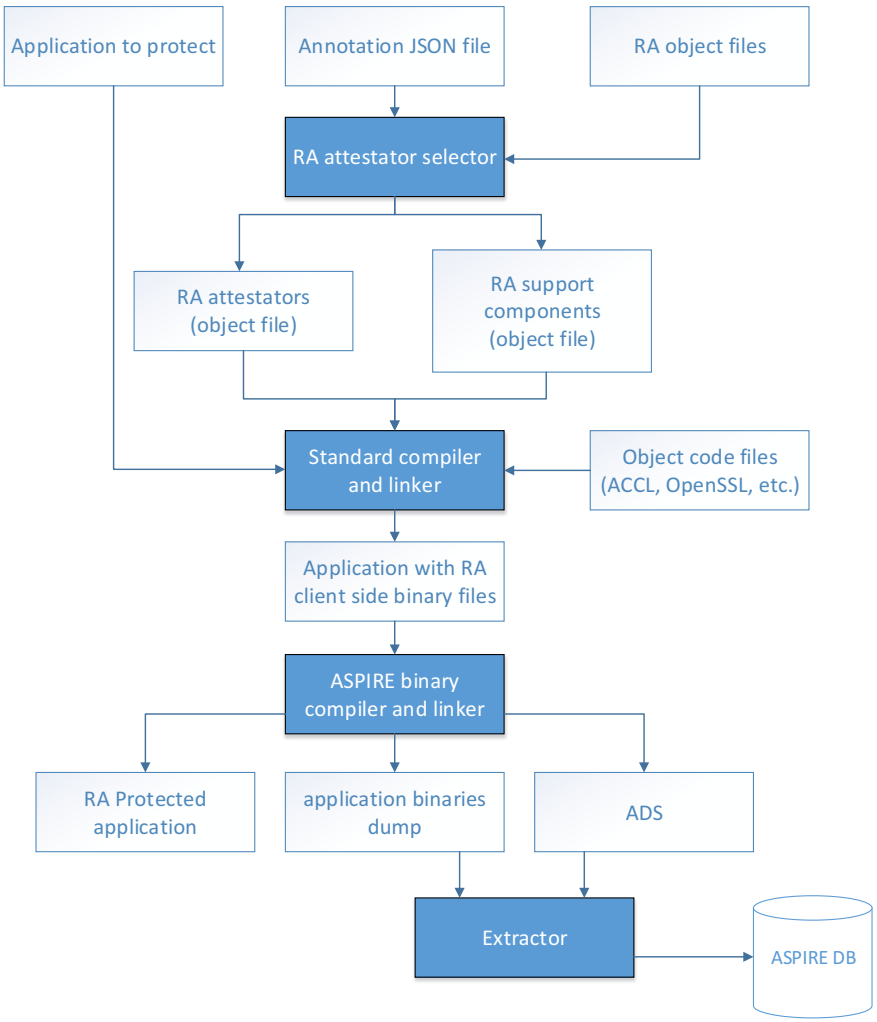

Figure 18. Client-side protection.

Figure 18 presents the snapshot at M24 of the steps and the components needed to protect an application with remote attestation. We assume that the original application source code is annotated to indicate the areas to be watched by the remote attestation. It is also assumed that the application has AID (Application ID) that allows the ASPIRE portal to identify requests from an application running on a remote device[2].

The first step is a selection of the RA object files to be included in order to produce the protection required by the annotations. This step is performed by a simple component, the RA Attestator Selector.

The RA Attestator Selector reads the JSON file containing all the annotations and determines the object code to select. The annotation provides the names of RA techniques, the diversification parameters, and some optional customization parameters. Examples of customization parameters are the average time between two attestations.

Then, starting from the annotation information, the RA Attestator Selector determines the proper RA attestators to be used, selects their relative object files among all the pre-compiled attestators object files and makes them available to next ACTC steps as a unique object file.

[2] For more details on the definition and use of the AID see the internal wiki https://aspire-fp7.eu/wiki/actc-aid.

Additionally, the RA Attestator Selector delivers another object file which implements functionalities needed by the attestators. C

2) Binary level. In particular, the client-side application that also includes the RA code may be rewritten at binary level by the ASPIRE binary compiler. RA code is independent of any type of transformations performed at binary level, however, since the transformations certainly change the areas to attest, the binary obfuscator (i.e., Diablo) keeps track of the changes so that the RA code is able to attest them correctly.

Therefore, the Diablo-based binary phase described in Section 5.2.2 of deliverable D3.02 is executed. When rewriting code according to what requested by annotations of other protection techniques, Diablo, has been modified and keeps track of the changes to the areas to protect with remote attestation.

Therefore, Diablo is able to build the ADS, inject it into the application binary and rewrite remote attestation code to link it to the ADS. It also outputs to a file the ADS. Both the application binaries and the ADS are used by the extractor to fill the ASPIRE database with the precomputed attestation data.

5.3.4 Composability

Static remote attestation determines whether the application code that is running on the client is the same as the one distributed by the application developer or whether it has been modified. The areas to attest are annotated. Information about areas to attest is made available to the remote attestation technique as a JSON file.

Attestation responses are obtained as:

Hash(nonce || attestation_data || ID)

The attestation_data are obtained by a nonce-driven attestation data generation algorithm that takes as input the nonce and the memory area to attest. It is therefore important to ensure that the information about the memory areas to attest are maintained up to date when the Actual Verifier has to perform the comparison. Memory areas are described in the ADS as ordered sequences of memory blocks.

Therefore, for the sake of composability, other techniques can be divided in *techniques that may modify memory areas* to attest *and techniques that do not modify memory areas* to attest.

Techniques that do not modify memory areas. Techniques that do not modify memory areas to attest are composable with this remote attestation technique, also on the same application part/asset.

Techniques that may modify memory areas. Techniques that may modify memory area can be made composable by keeping track of the changes that are performed to memory areas to attest. This is what diablo actually does when it applies layout randomization and obfuscation prior to ADS generation. For static remote attestation, it is not needed to know the transformations applied, it is only important to know the exact location of memory areas, so that both the client-side Attestator and the server-side Actual Verifier (and Extractor) read the same memory parts when computing the attestation_data content.

Composability with other local techniques. If another source level technique (executed before this remote attestation technique) inserts source code that requires integrity checks from this technique, it must include RA annotations in its source code. The same consideration applies to binary level techniques: they have to update the JSON file with the annotations concerning memory areas to attest before any binary level transformation is applied.

Composability with other online techniques. If client-server code splitting is applied to some of the fragments that need to be attested, it is important to keep track of the parts of those memory areas that remain on the client. If an entire memory area is moved on the server, the corresponding annotation must be removed from the JSON file. Renewability is much harder to support. Even if it is theoretically possible, it requires a lot of engineering effort. Indeed, when a code block is replaced, it is necessary to check if some of the memory areas to attest is affected by the change. If so, it is necessary to inform the Verifier of the changes in the client binaries so that the attestation_data computed (or pre-computed) on the monitored client. Moreover, the DB must store precise info on the code blocks that are currently deployed. For preemption, all the version of code blocks should be made available to the Extractor to precompute valid (nonce, attestation_data) pairs.

5.3.5 Legal issues

We remark here that static remote attestation is compliant with REQ-FSR-003, “Legal compliance with respect to privacy regulations for the collection of data shall be respected” as it only processes code memory areas, application and client identifiers.

5.4 Dynamic remote attestation

We present here the achievements for another remote attestation technique: the invariants monitoring remote attestation. This technique is currently in the development phase, therefore it is an isolated prototype. It is not integrated in the M24 ACTC architecture.

Invariants are predicates defined over the application variables. It is assumed that invariants remain true during the whole program execution. Invariants are quite difficult to determine, unless the programmers explicitly mark them or a formal model of the application is available.

The best way to have a consistent set of invariants is to use likely invariants, which are predicates built over execution traces: all the values assumed by the variables in the execution traces satisfy the likely invariants.

Likely invariants may be false, but the probability of false positives is reduced if enough traces are collected. Given another set of traces, it is possible to have a different set of likely invariants.

This remote attestation protection is built on the assumption that, if an attacker tampers with the application, some of the likely invariants will no longer be satisfied. Therefore, the purpose of the Attestator is to collect the variable values to send to the Verifier. The Verifier then checks if the likely invariants are satisfied or not. The scenario is made more complicated as variables are not always available in memory. Therefore, the RA Manager cannot precisely ask for the variables it needs for specific variables and the Verifier must progressively collect the variables’ values the Attestator can send and look for the invariants that can be evaluated.

Moreover, recursive and nested functions need to be processed at compile time in order to carefully diminish the quantity of data exchanged between the client and the server. Indeed, an invariant is valid for a given execution point and does not depend on the functions previously called to reach that point. Then, for a recursive function, an invariant can be evaluated just on the last function call ignoring the previous ones.

Following the basic workflow of the general RA architecture, the invariants monitoring RA Manager sends the Attestator an attestation request and the Attestator sends the Verifier the corresponding attestation response.

The attestation request contains the following information:

- Nonce, a random value that adds freshness

The attestation response contains

- A sequence of (variable ID, value) pairs;
- The hash of the content computed.

Likely invariants have been used in literature to implement remote attestation techniques. More information can be found here [Kil09] [Sad09]. The prototype being developed in ASPIRE follows the basic theoretical principles, that is, it assumes that an application is tampered if some invariant is no longer satisfied. However, the ASPIRE ambition is to automatically protect any application for which a set of invariants is known.

5.4.1 Determining invariants

The dynamic remote attestation technique which we are developing is based on likely invariants. To deduce a valid set of invariants can be very difficult, so we decided to exploit an external tool, namely *Daikon* [ERN07]. Daikon is a consolidated and tested tool for the dynamic extraction of likely invariants from application executable. This tool is able to achieve its goal my monitoring the execution of an application and inspecting the values of its variables.

Then, it is been designed a tool able to invoke Daikon and manipulate its output in order to obtain a useful and usable representation of invariants and properly store them into the DB. This tool is called the Invariants Extractor. Consequently, when an application has to be protected with Dynamic RA it is necessary to put it through a training phase. This phase aims at extracting as many invariants as possible and makes them available for their evaluation at runtime.

5.4.2 Variables identification

In order to verify invariants, it is necessary to:

- identify each variable involved in each invariant;
- access and extract each variable value involved in each invariant during the execution of the application.

To satisfy the former need, we identify each variable by its name and the name of its scope, i.e., the function/method or nothing if a variable is global. Hence, for each application protected with Dynamic RA, the ASPIRE DB holds a set of data that specifies the identification of all the program variables.

The latter point has been achieved by performing the following workflow:

1) The program is compiled with *DWARF* [DWF] debug symbols.
2) The binary file is then analysed by parsing the DWARF symbols using the *libdwarf* library [LDW], inside the *info* section of DWARF, the *DW_AT_LOCATION* field specifies a relative location of the variable at runtime. Each variable can reside in a memory location, in a register or its value can be defined as a constant depending on the actual execution point.
3) The library *libunwind* [LUW] is used to extract information about the program execution. It allows deducing the possible call-stack configurations and the references to be used as bases for the relative locations obtained in the previous point.

This procedure is performed during a preliminary training phase that allows knowing the location of each variable at run time.. After the training phase, the program is re-compiled without any debugging symbol, the extracted information is encoded in binary form,embedded in the final binary, and accessed through an API.

Notice that, after the preliminary phase the application can be recompiled without any debug symbol, this does not invalidate the data taken. Indeed, the structure of the executable binary file is not influenced by the DWARF symbols.

5.4.3 Composability

Invariant monitoring remote attestation determines whether the application that is running on the client is the behaving as expected by checking predicates, the invariants, built on the application variables.

Therefore, for the sake of composability, other techniques can be divided in *techniques that may modify application variables and techniques that do not modify application variables*.

Techniques that do not modify variables. Techniques that do not modify variables are composable with this remote attestation technique; even if more techniques on the same code fragment. This consideration also applies to techniques that do not modify any of the variables that are used in any predicate. However, it would depend on the current enforcement of each techniques and it cannot be used to infer general composability properties.

Techniques that may modify variables. Techniques that may modify variables can be made composable by keeping track of the changes that apply to variables. Since invariants are formulas, if variables are split, transformed, relocated, etc. it is important that the Verifier knows the transformation in order to update the invariants formulas. Moreover, the Attestator must be informed of the variable IDs. That is, a new set of identifiers must be issued and reported in the annotations.

Composability with other local techniques. If another source level technique (executed before this remote attestation technique) inserts source code that requires integrity checks from this technique, it must add in the annotation JSON file the variables to monitor and invariants formulas. The same consideration applies to binary level techniques: they have to update the JSON file with the annotations concerning the variables to monitor and add the invariants involving them.

Composability with other online techniques. If client-server code splitting is applied to some of the fragments that include variables to monitor, the server-side component must make available to the Verifier the value of these variables to perform the verification of the invariants involving them. An API should be provided to ask the variable values. Feasibility and opportunity would be verified during Y3.

Also for invariants monitoring, renewability poses serious integration and engineering issues, even if it is theoretically possible to integrate them. Indeed, it is necessary to inform the Verifier of the changes to the monitored variables when generating alternative code blocks (e.g., by diversifying piece of code) so that the Verifier can update on the fly the invariants monitoring them. A simple example of integration between renewability and remote attestation will be provided during the third year of the project. However, the engineering effort to (1) abstractly model the changes operated to the code when performing diversification and (2) translate them into formulas to be composed with the original invariants, are probably too demanding for the current resources, but a simple integration of these two online techniques will be tried as it would be relevant to prove composability properties in ASPIRE.

5.4.4 Other legal issues

We remark here that dynamic remote attestation based on likely invariants may violate REQ-FSR-003, “Legal compliance with respect to privacy regulations for the collection of data shall be respected” as it collects data from user variables. However, by proper labelling variables (e.g., with annotations) as sensitive, it is possible to exclude them from the collection. Moreover, user-related information is usually not useful for calculating invariants and it is unlikely that are selected by tools like Daikon.

5.5 Plan

These are the four main activities to be performed in the remote attestation T3.2 in WP3:

1. Implementation and improvement of the static remote attestation techniques (done);
2. Design and implementation of the dynamic remote attestation techniques (done);
3. Development of the tool chain component that automatically protects the application with remote attestation and integration into the ACTC (only done for static RA);
4. Design and implementation of the implicit remote attestation (under development).

The static remote attestation has been extended to support the diversification need for the renewability. Updates are expected on this topic during Y3. Moreover, we will continue the strong interaction with the static and remote code guards.

Dynamic remote attestation design phase has been terminated. During Y3 we will complete the implementation of the automated protection technique, the validation of the approach, and the integration into the ACTC remote attestation tool chain component.

Implicit remote attestation is currently in the design phase, issues need to be solved to complete it. We will deliver it during Y3. Nevertheless, the architecture and workflow proposed in D3.02 are still valid.

Section 6 Anti-Cloning

Section Author: Brecht Wyseur (NAGRA)

6.1 Introduction

Anti-cloning (AC) is a special-purpose remote attestation technique that aims to allow a trusted server to efficiently detect concurrent clones that are connecting to this trusted server. This is part of the work envisioned in Task 3.2. as described in the ASPIRE Description of Work and has been introduced in the updated ASPIRE Deliverable D1.04 “Reference Architecture”, Section 4.6.

We further worked on maturing the idea beyond conception in Year 2. In this section, we explain more hands-on details and the implementation phase that we started. Delivery of this technique is envisioned early in Year 3 of the ASPIRE project.

As described in the ASPIRE Reference Architecture, this technique aims to mitigate cloning attacks by associating a tag to each application instance and changing this tag regularly during the entire lifetime of the application. That means that this tag needs to be persistently stored in between different executions. This is achieved by storing the tag value into a dedicated file that will be persistently stored along with the application.

Figure 19 depicts the anti-cloning concept. The original application holds an initial tag value “x”, which will evolve over time. It will evolve into the values “y” and subsequently “z” after an interaction with the trusted server. The trusted server holds the corresponding value in its database and checks that upon each connection, the received tag value is consistent with what is stored in its database. If at a certain point in time, a clone is made from the application (which necessitates that the dedicated tag file is cloned as well), the tag value will be updated by which ever instance (the original or the clone) makes the first interaction with the trusted server. The other instance will be de-synchronized upon its next interaction and as a result the trusted server can conclude that a cloning event took place. Note that the server cannot distinguish between an original application and a clone; this is an inherent problem in software-only solutions.

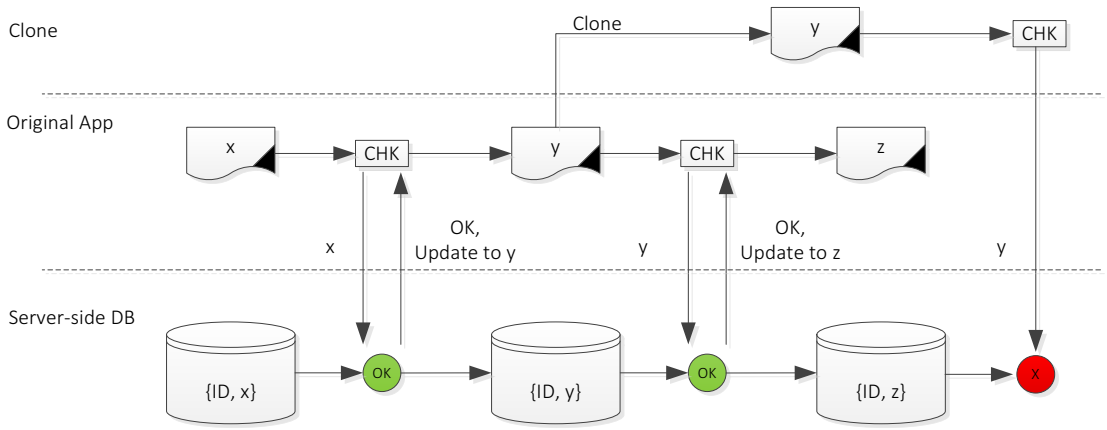

Figure 19 - Anti-cloning concept

6.2 Design

6.2.1 Architecture

A high-level architecture design has been presented before in the ASPIRE Deliverable D1.04 “Reference Architecture”. For the sake of clarity, we depict the anti-cloning workflow diagram again in Figure 20.

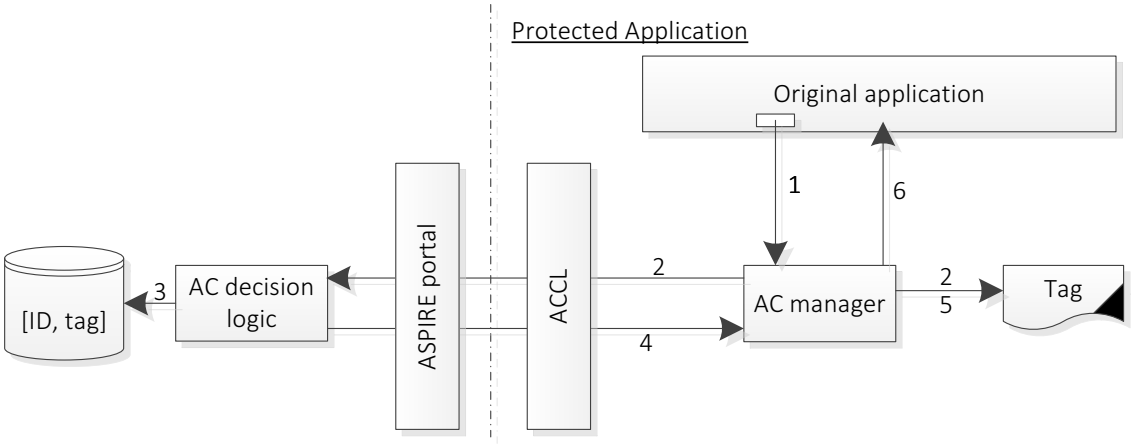

Figure 20 - anti-cloning workflow diagram

6.2.2 Implementation details

The main functionality of the anti-cloning technique will be implemented in a dedicated library that will be statically linked with the application. This library exposes a simple API comprising just the function that invokes the process as presented in Figure 20. The library will be written in C, and will invoke the ACCL as described in the updated ASPIRE Reference Architecture.

The AC feature can be invoked from anywhere in the original application. It has the best effect when invoked just prior to the need for fetching some assets from a server; an event where the server can verify if the status of the application that is connecting is still OK. These events are indicated with the annotations as described in the ASPIRE Deliverable D5.01 “ASPIRE Framework Architecture Tool Flow and APIs” and updates in the working document WD5.02.

To deploy the AC feature in an application, it suffices to translate the AC annotations into an AC function call. Thus, the AC feature will be implemented as a source-to-source step that will be integrated into the ACTC.

6.2.3 Server-side policy

At server side, the AC decision logic will be implemented as a python script. It will store the tuple { application ID, tag } into a database and upon each connection of the application, it will verify the consistency of the tag presented by the application with respect to the value stored in the database, and if OK, proceed to update the tag to a new random value.

If the value is not OK, there are many different reactions possible, depending on the business value of the application or the risk that a service provider is willing to take. We consider complex reaction logics out of scope for ASPIRE and will implement a simple policy. Namely, when the value that the application presents to the AC decision logic does not correspond to the expected value that is stored in the database, the application ID will just be flagged. This signifies that the AC decision logic considers that the application has been cloned and thus that there are multiple instances with that identifier. Reaction logic will not be

implemented, but instead the AC decision logic will have an interface that allows other techniques or the original server to verify if a certain application ID is flagged or not.

6.3 Composability

The AC technique operates by verifying the value of the tag that is sent to the server-side support and updating this tag regularly. Only the value of the tag is important and therefore any further modification of the application that persists in this semantic behavior is OK for composability. This in general is any protection technique, since SW protection techniques preserve semantical behavior.

Composing AC with additional techniques to strengthen this may be of interest as follows:

- To protect the tag value itself while in storage in the tag file. This could be achieved by encrypting it at storage-time with a unique application secret, and decrypting it at read-time with the same secret.
- To ensure that the AC manager code itself cannot be lifted or to ensure that the application control flow is preserved. This is in particular of interest to prevent a type of proxy attacks which attackers may try to build to enable larger-scale exploitability of the attacks.

6.4 Plan

There are 2 major next steps that are scheduled early Year 3 to finalize the implementation of the anti-cloning technique.

First, there will be a focus on the integration steps, where the source code rewriting operations will be written; that is to translate the AC annotation into a AC function call. This allows for testing the integration into the ACTC early on.

Secondly, the AC manager library itself will be implemented.

Section 7 Renewability

This section briefly reports the work of the project consortium in Task 3.3.

Section 7.1 describes the design process of the Renewability Architecture, and the third year plan to achieve it (both fully described in the new version of D1.04 - Reference architecture).

The initial works on software diversity (renewability in space), are reported in the next three subsections: Section 7.2 describes the initial experiments to maximize diversity among different diversified copies, Section 7.3 describes diversification based on crash reporting, and Section 7.4 introduces Feedback-driven diversity.

7.1 Design of ASPIRE renewability techniques

Section Author: Paolo Falcarin

In order to meet the different requirements of the DoW, we have to take into account the properties of the available offline and online protections, their technical constraints, and the resources of the different partners involved.

The Code Mobility framework will be extended in the 3rd year of the project to implement the renewability framework, by adding new technical features identified during this analysis phase.

Then the consortium has discussed the different requirements of the online protections and which can be made renewable, and which additional technical features each protections would require to be made renewable.

For example, Time-limited White Box Crypto can be made renewable, provided that the Code Mobility framework will be extended to support Data Mobility, a new technical feature which will enable to make data mobile along with code blocks.

On the other hand, the VM bytecode (now embedded in the binary code) can be considered data that could be possibly loaded at run-time from a server. Again, the Data Mobility feature would help to realize this scenario as well. However, making the SoftVM bytecode diversified, and then renewable, would require a re-design and a new implementation of the current SoftVM: this would require a lot of resources and cannot be achieved in the rest of the project.

Therefore, the Data Mobility extension of the Code Mobility prototype (described in D1.04 v2 Section 5.2.2.1) will allow SoftVM bytecode mobility and WBC data table run-time renewability (D1.04 v2 - Section 5.3).

Other techniques, such as dynamic remote attestation, delayed tamper response, CFG tagging, anti-cloning, and time-bombs cannot be made renewable because of conflicting technical constraints, as they all access to the memory, expecting a particular code layout, that instead will be dynamically changed by the Code Mobility framework.

We will try to integrate Code Mobility and static remote attestation in order to make the attestors renewable.

Another effort for the third year of the project will be devoted to the implementation of the different server-side components that will manage the diversified code blocks both at server and at client side, and their interaction with Diablo (as a diversifier) and the Static Remote Attestation service.

The Renewability Manager component is introduced to manage the different lifecycle of the various client applications and to orchestrate the renewability schedule of each mobile block

depending on different and configurable renewability strategies described in D1.04 Reference Architecture v2.

7.2 Experiments to Maximize Diversity

Section Authors: Paolo Falcarin, Alessandro Cabutto (UEL), Mariano Ceccato (FBK)

Software Diversity is a protection aiming at reducing the exploitation of reverse engineering attacks; instead of delivering the same program to all the users, different versions are delivered to different users, so that a successful attack on one code instance cannot be easily exploited on a diversified code instance.

Ideally, all the deployed versions should be "enough" different from one another, such that an attack developed on one version cannot easily replayed on another version; the information learned from one copy should not be used to implement automated attacks on other versions, or at least reducing the success rate on a very limited number of versions.

In this sub-section, we start tackling the problem of searching for the best subset of diversified versions of the same program, which is the subset of versions that are the pairwise most different from one another; we resort to different configurable algorithms of Java code obfuscation to generate several different versions of the same program. Then, we resort to search based heuristics to find an optimal set of different versions, different enough to be deployed.

To compare the code of such versions we used *gzip* similarity as first and simple metric and we are planning to use more appropriate metrics in the last year of the project. However, our search-based approach is general and the metric function can be easily replaced by the implementation of more advanced code similarity metrics. We started working on Java bytecode diversity and we report about the application of hierarchical clustering and hill-climbing heuristics.

7.2.1 Problem Formulation

We define the distance between two diversified versions of the code by using a similarity metric ranging from 0 to 1, where two identical versions have distance equals to zero and two totally different versions have distance equals to 1.

Assuming that we can generate N diverse versions of the same code, we then need to deploy M versions (with M less than N) to M different users at most. We set a constraint that the distance (similarity metric) between each couple of chosen versions in the set M must be greater than a threshold T.

We can model the problem with a complete graph of cardinality N where each node is a version, and each arc is labelled with the distance between two nodes (versions).

We extended the tool AntiCopia [Fre07], to perform the similarity analysis (with the NCD metric) on different versions of a simple Java application created by setting the parameters of Zelix obfuscator [Zel] to different values. The graph distance histogram of Figure 21, centred on the original program *v0*, shows how some diversified versions are still close (similar) to the original program while others are more distant (different). It is also visible on the left a small cluster of versions that are very similar among each other, but very distant to the original program *v0.*

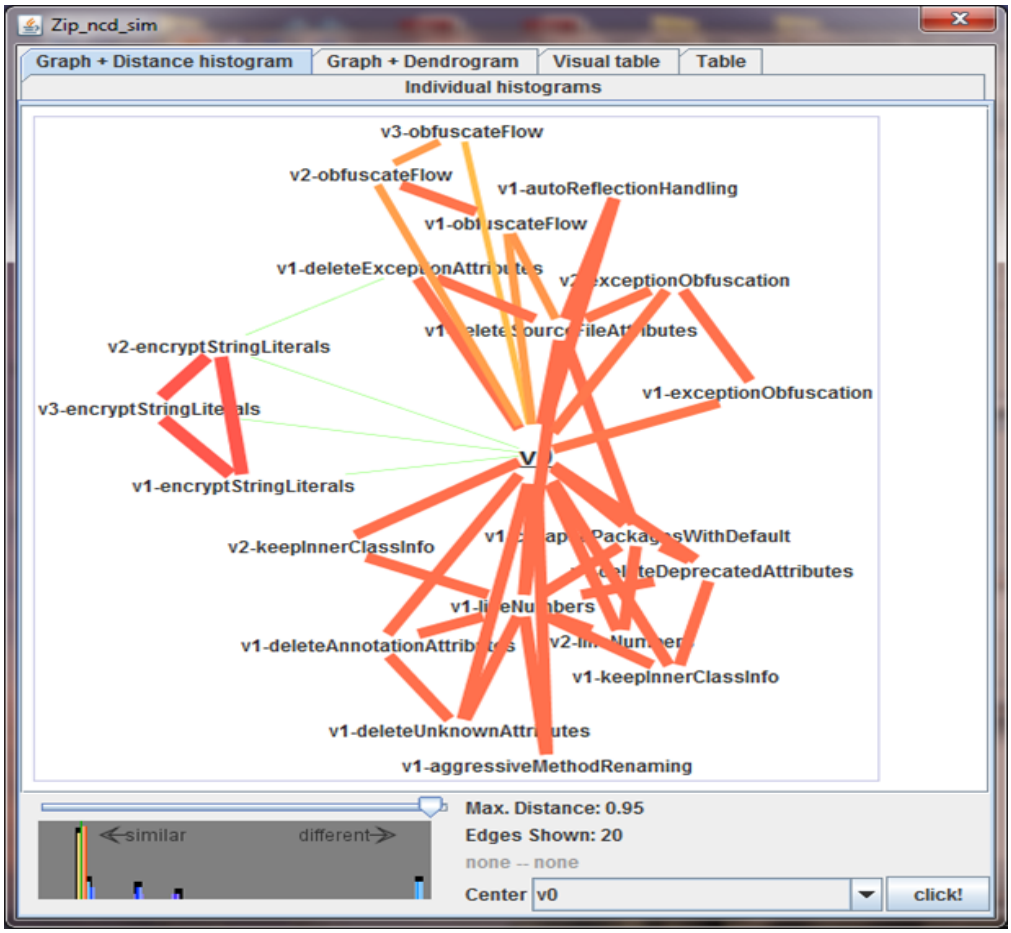

Figure 21 – Distance Histogram of diversified versions

From this observation, we decided to formulate the software diversity problem as a clustering problem.

The objective is to find in the original N-complete graph, where each arc represents the similarity between the corresponding nodes (versions), the clusters of versions that are very similar among each other. So, after identifying clusters of similar versions, the final set of versions to deploy can be selected by taking just one element from each high-similarity cluster. In general, if N clusters have been identified and M versions must be deployed, (M / N) versions per cluster can be chosen to minimize the pairwise similarity of the versions to be deployed.

Let the distance metric *D(v1,v2)* approximate the actual similarity between two versions v1 and v2 assume values in the interval [0,1], and given a partition of all the available versions into similarity clusters, we define the intra-similarity A_i of the cluster I as the average similarity of all the pairs of elements in the cluster:

$$A_i = \frac{\sum_{v1,v2} D(v1, v2)}{\mathrm{n}(\mathrm{n}-1)/2}$$

We define the intra-similarity between two clusters C_i and C_j as the average similarity of the versions in the two clusters (where v1 belongs to C_i and v2 belongs to C_j):

$$E_{i,j} = \frac{\sum_{v1,v2} D(v1, v2)}{|C_i||C_j|}$$

Our objective is to search for a clustering configuration with clusters that contain elements as similar as possible (high intra-similarity) and low similarity between elements from different clusters (low inter-similarity). Thus we define the overall *similarity quality* among the clusters as the average intra-similarity minus the average of all the inter-similarity (where k is the number of clusters in the partition to evaluate):

$$SQ = \frac{1}{k}\sum_{i=1}^{k} A_i - \frac{1}{\frac{k(k-1)}{2}}\sum_{i,j=1}^{k} E_{i,j}$$

At this stage, the software diversity problem can be expressed as a search problem, aiming at finding the clustering partition that maximise the value of SQ.

7.2.2 Similarity metric

To detect similarity between two candidates we decided to use the NCD method [Cil05] using gzip as compression algorithm applied as follow:

$$S_{i,j} = \frac{G(ij) - min\{G(i), G(j)\}}{\max\{G(i), G(j)\}}$$

where *G(x)* denotes the length of the code version *x* compressed by gzip and *G(xy)* the compression of the concatenation of code versions *x* and *y*.

Similarity values $S_{i,j}$ obtained by applying the NCD formula on the different N versions are stored in an NxN matrix: since $S_{i,j} = S_{j,i}$ and $S_{i,i} = 1$ we can consider the upper triangular matrix and exclude its diagonal, hence the final number of values to be stored is $\frac{N-1}{2}$. In our experimental implementation we stored these values in a SQLite database.

7.2.3 Clustering algorithms

This subsection will describe the three different heuristics utilized to obtain clusters of similar versions; we implemented three different clustering algorithms, namely Hill Climbing, Hierarchical Clustering, and Genetic Algorithm.

7.2.3.1 Hill Climbing

Hill climbing is one of the most widely used local search algorithm [Min04]. Hill Climbing starts from a randomly chosen initial solution in the search space and works to improve the solution. The neighbourhood of the current solution are investigated and a new solution is taken among those that improve the current solution. This new solution will replace the old one and become the new current solution. Again, the neighbourhood of the current solution will be explored to find a better one, if an improving solution is found then it will be the current solution and so on, the process continue until no improved solution is found for the current solution until the search budget finishes.

This improvement of solutions is linked to the metaphor of the climbing of hills in the landscape of a maximizing objective function [Min04]. In this landscape, peaks characterize solutions with locally optimal objective values. It is possible to have more than two solutions that improve the current solution. Hill Climbing uses different strategies to select one solution among neighbours' solutions that improve the current solution. In a steepest ascent climbing strategy, all neighbours are evaluated, with the neighbour offering the greatest improvement chosen to replace the current solution. In a random ascent strategy (sometimes referred to as first ascent), neighbours are examined at random and then one neighbour that offers an improvement is chosen.

In most cases, Hill climbing generates or finds the neighbour solutions by applying mutation or making small change to the current solution, as a result, hill climbing gives fast results. However, search could give sub-optimal results, since, hill climbing starts from a random position (solution) in the search space and accepts any solution which improves it and this

might lead hill climbing to get trapped in local optima and unable to explore other more promising areas of the search space. Hill climbing is starting solution (position) dependent in the search space, and hence, there are some meta-heuristic search techniques that extend it. A common extension to this algorithm is to incorporate a series of restarts involving different initial solutions to sample more of the search space and to minimize the problem of getting trapped in local optima as much as possible.

7.2.3.2 Hierarchal Clustering

Hierarchal Clustering is a greedy algorithm that tries to find a good partition in the search space. This algorithm starts from the initial configuration where each candidate is assigned to a different cluster. At each step, the inter-similarity between each cluster is computed and the two most similar clusters (those with highest inter-similarity) are merged to form a single cluster. This process is iterated and at each step the number of clusters decreases by one. The iteration terminates when the partition contains only one big cluster where all the elements belong to that cluster. During this process the Similarity Quality value is recorded at each step; the one with highest SQ represents the final sub-optimal solution.

7.2.3.3 Genetic algorithm

Genetic Algorithms are adaptive heuristic search algorithm based on the evolutionary ideas of natural selection and genetic recombination, and use historical information to direct the search into the region of better performance within the search space. Likewise to the natural genetic evolution, GA encodes candidate solution as individual (chromosome); each individual has a genetic component or property know as gene that can be mutated or altered during the genetic evolution.

GA evolves a set of candidate solutions rather than just one single solution. Those set of solutions will be referred as population. GA generate randomly the initial population, and this enables the algorithm to have many starting points, as a result GA samples more of the search space than local search algorithms [15]. Sometimes, that initialization may be ”seeded” in the area where optimal solutions are likely to be found.

In each generation (iteration), the fitness of every individual in the current population is evaluated. Most of the time, the fitness of an individual is the value of the objective function in the optimization problem being solved, or the value scaled in some way. These fitness values are the main deciding factor for an individual to undergo recombination. Recombination is a mechanism of exchange genetic information between individuals to “breed” new individuals and possibly mutates them. Based on the fitness value, GA apply recombination procedure under some selection strategies such as Roulette wheel selection, Tournament selection, Linear Ranking selection, etc., and produce new individuals which will become the population of the next generation. This process continues until the termination condition is reached or the allocated search budget finished.

7.2.4 Target of experimentation

For ease of implementation in our experimentation we decided to target Java applications. Nevertheless, the whole approach is valid in general and could be applied to binary applications also by selecting a proper similarity metric for binary code.

Java is a high level programming language and keeps a lot of information from the original source code after compilation. Java programs are distributed over the Internet as bytecode exposing them to a high risk of reverse engineering attacks. Automatic code obfuscation is currently one of the most adopted approaches to protect Java applications from reverse engineering attacks [Col97] by reducing code comprehension.

Obfuscation in this case is a source code protection technique: it generates a semantically equivalent program, which is harder for a malicious human being to understand and analyse.

7.2.4.1 Obfuscation Tool

Many open source and commercial obfuscation tools are available. For our experiments we selected the Zelix Klassmaster™ [Zel] toolkit because it provides several configurable obfuscation techniques. This gives us control on the complexity of the obfuscated output and an easy way to produce diversified applications. Analysing the documentation of the Zelix toolkit, we identified a total of 13 distinct obfuscation configurations:

1. deleteSourceFileAttributes
2. deleteDeprecatedAttributes
3. deleteAnnotationAttributes
4. deleteExceptionAttributes
5. deleteUnknownAttributes
6. aggressiveMethodRenaming
7. keepInnerClassInfo
8. obfuscateFlow
9. encryptStringLiterals
10. collapsePackagesWithDefault
11. lineNumbers
12. exceptionObfuscation
13. autoReflectionHandling

Each parameter controls a different obfuscation algorithm and these parameters can assume different values. Every combination of those parameters corresponds to a diversified obfuscated variant. Among these 13 parameters, 8 parameters support binary value, other 3 parameters can have 3 possible values each and remaining 2 parameters can have 4 possible values each. Therefore, we could theoretically generate up to 28*33*44 = 110,592 distinct obfuscated candidates using this tool.

However, with 110,592 candidates we have 6,115,239,936 $\left(\frac{N\cdot(N-1)}{2}\right)$ distinct pairwise similarity values to consider. Moreover, some of these parameters do not to bring any benefit in terms of diversity, so we decided to exclude four parameters (the ones with less diversity impact) from our analysis. Finally, for our study we will have at most 24*33*44 = 34,848 distinct candidates heading to 607,174,128 pairwise similarity values to consider.

7.2.4.2 Case studies

We chose some case studies from very popular (in terms of user installations) Android applications coming from the Google Play Store:

- Google Chrome (a browser)
- SkypeWifi (an IM application)
- Opera Mini (a browser)
- GoTetris (a game)
- Twitter (a social network)
- ESFileExplorer (a utility)
- WordFriends (a game)
- Contacts (a utility)

For each application the following process has been followed:

1. APK file download from the application store
2. Java source code extraction
3. Source code diversification using Zelix Klassmaster with different obfuscation combinations
4. Similarity Matrix generation by comparing each diversified version
5. Clustering algorithms execution over the Similarity Matrix

7.2.4.3 Experimental Results

The process described in the last section is extremely time-consuming. We run some experiments to benchmark how long it takes to process the similarity matrix using the clustering algorithms. The computation has been done on machines equipped by Intel Xeon quad-core blades, 8/12 cores, with 20GB of RAM. Our clustering implementation does not optimize the amount of used RAM, in fact the whole Similarity Matrix is loaded to allow faster and easier clustering operations.

On the average the overall process (diversification, Similarity Matrix generation and clustering) lasts one to two months. The expected exponential behaviour of the elaboration time is reported in Table 10. We decided to consider acceptable an elaboration time of few hours, therefore considering our data from the experimental analysis the maximum number of versions to be considered is 2048 and 512 for Agglomerative Clustering and Hill Climbing/Genetic Algorithm respectively.

Table 10 - Clustering algorithms execution time

	Elaboration time [s]		
n. of candidates	**Agglomerative clustering**	**Hill Climbing**	**Genetic Alg.**
2	0,47	1,48	3,65
4	0,47	1,49	4,21
8	0,49	1,71	4,36
16	0,49	2,29	6,11
32	0,51	4,36	6,87
64	0,56	42,56	39,35
128	0,74	194,29	187,17
256	1,92	1430,01	929,58
512	13,91	5535,60	4045,74
563	18,44	6273,39	4895,41
614	23,62	7143,16	5621,25
666	31,33	9171,58	6733,61
717	39,29	10553,68	8165,70
768	47,75	9773,24	9932,01
819	58,10	13002,69	10522,26
870	70,99	20018,86	12005,55
922	85,87	13661,24	13930,47
973	102,10	23333,18	15647,54
1024	118,64	23920,31	17534,31
2048	1048,45	174567,59	57993,37
4096	18813,49	-	-

7.2.5 Future work

The experiments on software diversity started earlier than planned, before M18 when task 3.3 was supposed to start. Some progresses have been made in the second year, but we had to prioritize the implementation and integration of Code Mobility and the ACCL architecture for all the online protections.

To optimize the clustering processing time we plan to reduce the search space by excluding from the search all the obfuscation configurations that are not effectively contribute to maximize similarity. In the next months, we will concentrate on defining and applying a strategy to achieve this goal.

7.3 Practically useful Diversification: Crash Reporting

Section Author: Mohit Mishra, Bjorn De Sutter (UGent)

One of the forms of renewability researched in the ASPIRE project is code diversification in space. By giving different users a different binary, attackers cannot easily deploy attacks on a wide range of machines. Their potential gains hence diminish, potentially up to the point where they will be no longer interested in investing in the identification and engineering of an attack. We point the reader to Section 2.2 of D1.02 for a more extensive discussion about the economic model behind this reasoning.

When academics present new code diversification schemes, industrial vendors and developers, as well as maintainers of major open source projects, typically welcome the protection it provides, but they almost immediately express reluctance because of the supposed costs the schemes would impose on the software development, maintenance and customer support. Amongst others, they are afraid that the code diversification might introduce bugs, and that it would make it very hard to interpret bug reports, e.g., in the form of crash reports.

Simplistic solutions such as permanently storing debug information for all diversified versions are infeasible, because full debug information is many times bigger than the code and data in a binary executable, be it an application or a library. So permanently storing it would be very space consuming. Moreover, for client-side install-time diversification approaches, e.g., as could be implemented in the Android Run-Time (ART) when bytecode is compiled into native code upon its installation on an Android device, storing the debug information on a central server is totally impractical. The alternative solution of rebuilding a software version and its debug information on the fly when a crash report comes in is impractical as well, because it requires the precise reproduction of the developer's build environment in the bug tracking & crash collection environment. Furthermore, if the diversification also involves private encryption keys and seeds, rebuilding a software version might introduce security threats when third parties get involved. An additional complication is that debug information is often sensitive information that should not easily fall into the hands of attackers. Amongst others it allows them to work around the hurdles introduced by diversification.

To overcome this problem in the context of the ASPIRE project, where diversification plays an important role and industrial partners are interested in practically deployable solutions, we set out to develop dBp, short for delta Breakpad. It is the first practical solution to the problem of crash reporting for applications with fine-grained layout diversification.

Our approach allows us to embed a small amount of encrypted information in a diversified application that, when sent to a bug crash collector together with a crash report, supports the reconstruction of an accurate, human-readable stack trace without requiring any persistent storage of data about the diversified application on a server.

We also developed some minimal adaptations to compilers in order to let them introduce enough diversification in the code and stack layout of binaries to raise the bar for code injection and code reuse exploits significantly, but without unacceptably inflating the amount of information that needs to be embedded in the application and sent to the crash collector.

With this line of research, we do not solve all problems immediately, and for the time being, we are only able to support relatively simple forms of diversification. But we consider it an important first step. Moreover, the protection tools used in our research are precisely those used to implement the renewability protections in ASPIRE's ACTC, being the compiler (LLVM) and the binary rewriter Diablo.

To optimize the exploitability of the results, we focus on BreakPad, an existing crash reporting tool and library developed by Google, and integrated into many real-life software products, such as Chrome.

7.3.1 Approach Overview

Figure 22 presents an overview of the dBp approach. dBp looks much more complicated than Breakpad, as can be expected from an approach that has to handle multiple versions of binaries, but in fact it remains very simple.

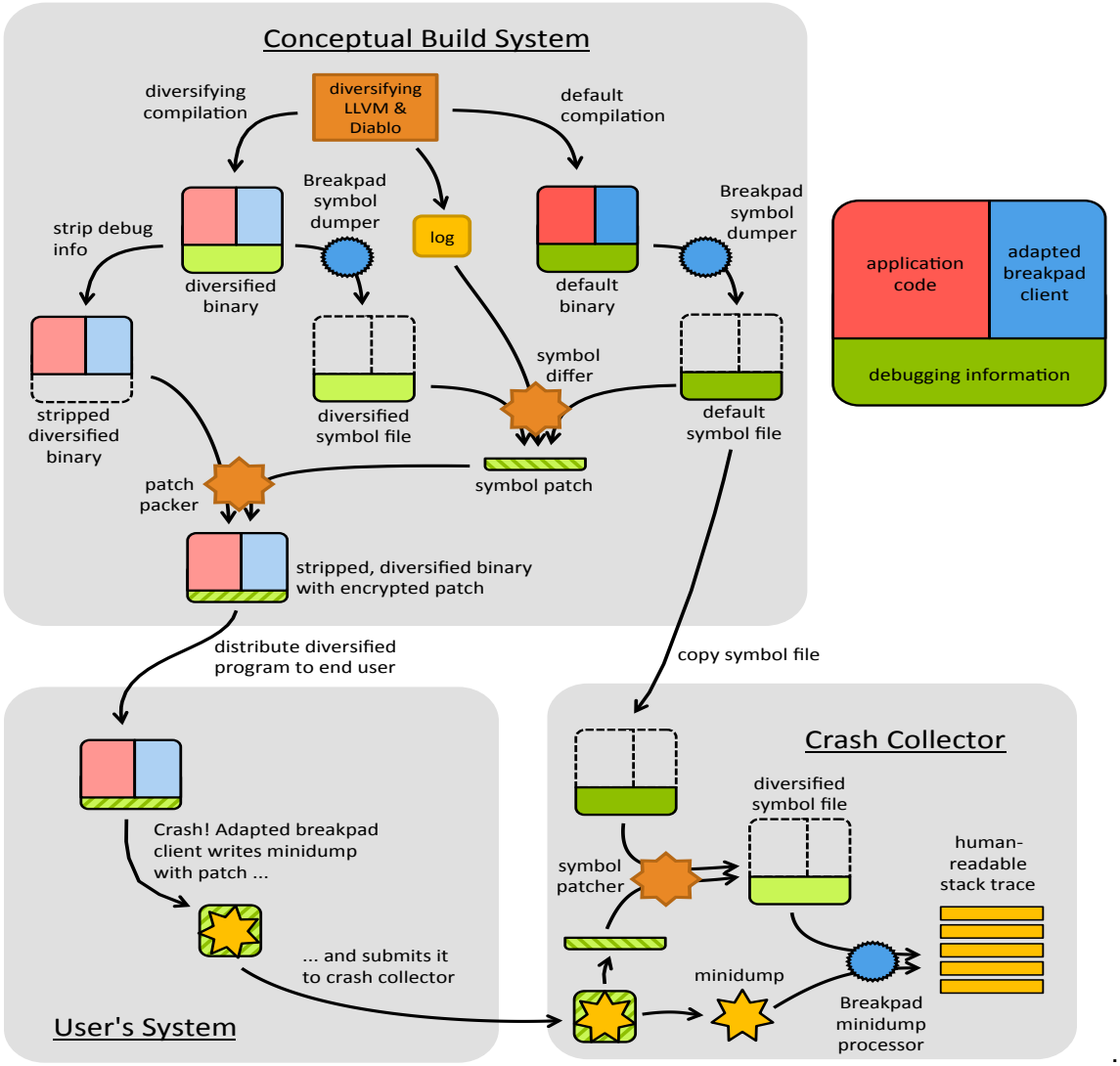

Figure 22 - Overview of dBp as an extension of Google Breakpad for reporting crashes of diversified binaries

The big picture is that an end user runs a diversified binary on his system (bottom left the figure). When it crashes, the embedded Breakpad client catches the signal received from the OS, and writes what is called a minidump to disk before actually halting the binary's execution. The minidump file format as developed by Microsoft is similar to core dump files, but much smaller, better documented, and less OS-specific. A minidump contains

- A list of the executable and all shared libraries loaded into the process when the dump was created.
- A list of threads in the process, together with all threads' (complete) stacks and processor register contents. Complete stacks are included because the applications typically do not contain debug information to analyse the stack.
- Some more system information, incl. the processor and operating system versions, as well as the reason for the crash.

In dBP, the Breakpad client is adapted to send not only the minidump to a crash collector server (bottom right of the figure), but to send a small data section of the binary along with it. In a custom symbol patch format, this data section contains all the necessary data for the crash collector to interpret the threads' stacks and registers contents in order to produce a complete, human-readable stack trace.

The crash collector server persistently stores debug information of the default version of the binary. This debug information is stored in a so-called symbol file, an ASCII text file that is both human and machine readable, with lines delimited as appropriate for the host platform. Breakpad comes with symbol dumper utilities for the major OSs, which simply extract the necessary information from the DWARF or STABS sections in ELF files or from stand-alone PDB files). The symbol file contains a list of functions, a mapping between instruction addresses and source line numbers, and a description of all stack frames and their uses in the binary.

When receiving a crash report, the server will first apply the symbol patch to the default symbol file. The result is a symbol file that corresponds to the diversified binary from which the crash report was generated. This symbol file and the minidump are then fed to the standard Breakpad minidump processor, which produces a human-readable stack trace, i.e., a trace that reports the state of the crashing application in terms of source code line numbers.

The top part of shows the adapted build system. On the right of this part, the standard Breakpad symbol dumper flow is shown to generate the symbol file to be stored persistently on the crash collector server. In our approach, this symbol file is extracted from the default binary. This is a binary generated without diversification, but with some slightly changed default compiler settings, to which we will come back in later sections.

On the left of the shown build system, the diversified binary is generated. Along with the diversified binary, some information is produced in a log that describes to some extent the randomization that took place. From the diversified binary, a symbol file is extracted, just like it was done for the default binary.

Based on the default symbol file, the diversified symbol file, and the log, a custom symbol differ we developed then generates a symbol patch. This is in fact nothing more than a compact script to translate the default symbol file into the diversified one.

This patch is then packed into the diversified binary from which all debug and symbol information was stripped as shown on the very left. The stripped, packed binary is then distributed to the user, ready to crash.

Whereas the original breakpad does not require any symbol or debug information to remain in the binary distributed to the end user, dBp does require the patch to be packed with the binary. As discussed in later sections, the patch does not contain any references to symbols. Still, it does leak information regarding the applied diversification, and hence it potentially aids attackers in circumventing the diversification. This can easily be solved, however. When

a developer or vendor embeds the Breakpad client into the application, he already chooses a crash collector, with which we can assume he has a trusted relationship. Using a public key encryption scheme, the patch packer can encrypt the packed patch with the public key of the crash collector, such that only the collector can decrypt the patch, prior to applying it to the default symbol file.

7.3.2 Currently Supported Code and Stack Layout Diversification

A key aspect to make the dBp approach practical is to keep the size of the packed patches small, while still offering sufficient diversification to actually raise the bar for attackers.

In this paper, we consider three static forms of layout diversification that have previously been presented to mitigate code injection and code reuse attacks to a large degree. They do not offer full protection, but they do offer significant protection at an acceptable cost.

7.3.2.1 Random No-op Insertion and Function Shuffling

Randomized no-op insertion [Hom13] and function shuffling aim at diversifying offsets within binaries' static code sections [Kil06]. Their goal is to prevent the leakage of useful information by reducing the amount of deduction an attacker can make about the location in memory of one code fragment once he has learned the address of another fragment. Whereas randomized no-op insertion typically only randomizes the lower significant bits in the displacements between two code fragments, function shuffling, i.e., the reordering of functions in the binary, also randomizes the higher bits in the displacements, incl. the sign bit.

At first sight, inserting no-ops and function shuffling seem to introduce no problem for generating the symbol patches in our approach. To allow the symbol patcher to patch the line number information in the symbol file when no-ops have been inserted, it suffices to store where the no-ops were inserted. For function shuffling, it suffices to store the start addresses of all functions in the diversified binary, in the order in which they appeared in the default binary.

In practice, this simple solution only holds for small enough binaries on RISC architectures. For larger binaries, and for binaries stored on CISC architectures, the insertion of no-ops and the shuffling of functions can result in displacements in the diversified binary that become too large to be stored within the bytes foreseen for them in the default binary. As a result, branches and other instructions that include code offsets (such as PC-relative addresses) as part of their instruction encoding can either require more bytes, as happens frequently on the x86 architecture, or require multiple instructions instead of one, as can happen on the ARMv7 architecture. In the case of x86 code, this is easily solved by recording not only the places where no-ops have been inserted, but also the places where instructions require more bytes to be encoded. In the case of ARMv7, trampolines (also called veneers) might be inserted, and to compute a correct symbol file patch, we also need to keep track of the locations where those trampolines have been inserted.

On architectures like the ARMv7, that lack a so-called global pointer, another secondary effect of inserting no-ops is that literal pools get moved around. These pools are small blocks of data in the code section that get accessed via PC-relative loads to produce addresses and large constants in registers. When the PC-relative offset grows because a no-op is inserted in between the load instruction and the accessed literal pool, the offset can become non-encodable as an immediate operand. Generating correct code is easy in that case: It suffices to move the literal pools around in the code section, if necessary splitting them into smaller pools, and to insert additional branches in the code to jump over the pools where necessary. But of course the resulting changes in the code layout again need to be tracked for correctly patching the default symbol file into the diversified one.

Keeping track of all the locations where secondary effects occur in an existing compiler backend, assembler and linker requires quite some engineering, simply because those tools

are not designed and engineered to minimize differences between similar code versions. Instead they are engineered to optimize code as much as possible or wanted by the programmer.

To avoid this large engineering effort in our research project, we have instead opted to perform the no-op insertion and the function shuffling in a post-pass tool in the form of a simple link-time rewriter based on the Diablo link-time rewriting framework that has been under development for over more than a decade in our research group. Tools based on Diablo by default produce as output a mapping between instructions' addresses in the input binary and in the output binary, which greatly simplifies the generation of compact symbol file patches.

Fundamentally, however, nothing prevents us of implementing similar tracking in a compiler suite like Clang/LLVM. For a real production build system, that would probably be even better.

7.3.2.2 Random Stack Padding

By inserting randomized amounts of stack padding in functions' stack frames, we randomize the displacement in those frames between the location of the stored return addresses and the locations of buffers that might be overflown. This makes it harder for an attacker to predict where to put the malicious payload in an overflowing buffer, and hence reduces his chances of success [For97].

Figure 23 compares simplified stack frame layouts in diversified and undiversified binaries.

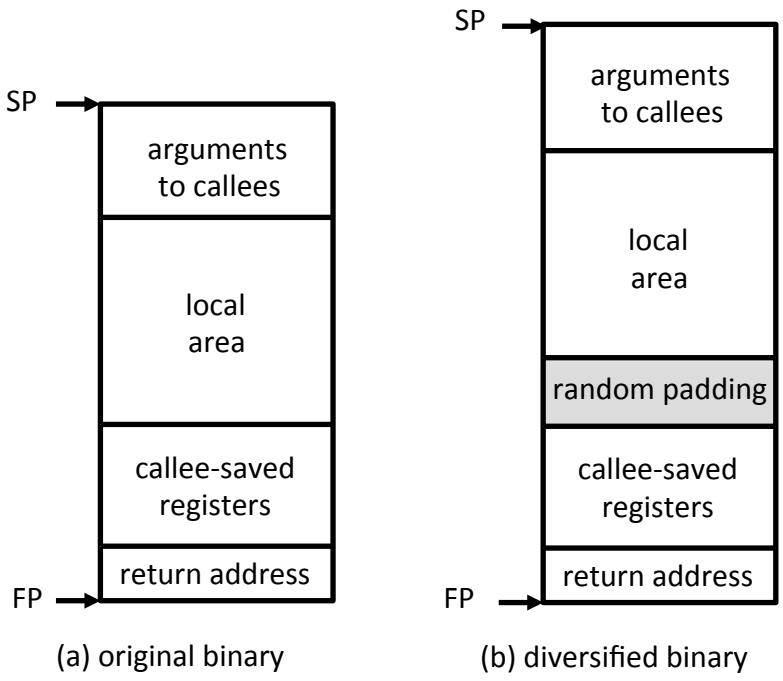

Figure 23 - Stack frames in original and diversified binaries

It is important to note that changing stack frames in a post-pass tool such as link-time rewriter is much harder to do than inserting no-ops or shuffling instructions. The reason is that a complete understanding of the stack frame and the assumptions that the compiler has relied upon to generate optimized code, is impeded by a lack of precise enough alias information. Relying on debugging information is also not guaranteed to be correct, for example if the debug information is inaccurate because of extensive optimizations performed by the compiler. For this reasons, we decided to implement the random stack padding in the

LLVM backend, rather than in a Diablo-based link-time rewriter. So random stack padding will be inserted before no-ops are inserted and before functions are shuffled.

To understand what changes in the code might result from the padding added by a compiler, the following observations need to be made.

- The offsets of memory entries (m-entries) in the local area of the stack frame relative to the frame pointer (FP) change because of the inserted padding.
- The offsets of the m-entries in the local area relative to the stack pointer (SP) do not change.
- In functions that call other functions with a variable number of arguments (a.k.a. vararg functions) the offset between m-entries in the local area and the SP is not constant because the SP changes during the function's execution. So in those functions, compilers tend to use the FP to access local data on the stack.
- When the offsets encoded as immediate operands in memory accesses change, this can cause non-trivial secondary changes in the generated code. This is particularly the case when an offset has grown so much as a result of the padding, that it can no longer be encoded as an immediate operand in its FP-relative load or store instruction. In that case, the address of the accessed entry (or its FP-relative offset) is first computed and stored in a register. This increases register pressure, as a result of which the compiler will often even schedule the code differently. Moreover, if multiple such computations take place in a small code fragment, the compiler might even deploy common-subexpression elimination to optimize the computations. On ARMv7, such larger side-effects of increased offsets are not rare, because relatively few bits are available to encode offsets in memory access instructions, and because of the architecture's peculiar way of encoding offsets as 8 consecutive bits that can be rotated over a 5-bit amount. It can, by the way, also happen that the diversified binary at some point requires fewer instructions than the default binary, e.g., because an original offset 0x3ff0 cannot be encoded in an immediate operand, but the bigger offset 0x4000 after padding can.
- In many cases, the compiler can choose between accessing data in the local area via the FP or via the SP. Compiler back-ends like LLVM's llc then choose the most efficient form, for which they compare the offset to the SP with the offset to the FP. As the latter changes when padding is inserted, this optimization frequently introduces more differences between default and diversified binaries than strictly necessary.
- It occurs that function prologues set up the stack frame without performing explicit SP increment instructions. In that case, adding stack padding requires inserting such instructions, rather than simply adapting their immediate operand.
- Even if only the immediate operand of a SP increment instruction needs to be adapted, this can again result in an operand that can no longer be encoded in a single instruction, thus again causing additional instructions to be inserted. However, as the prologue code is typically inserted very late in the back end code generation, this does typically not result in other instructions being reordered.

Given these observations, our "implementation" in LLVM 3.6.2 consists of three rather trivial patches:

1. An additional command-line option enables randomized stack padding insertion. The range of padding can be specified, as well as a random seed.
2. In the prologue & epilogue emitter, random padding is inserted in functions that have a stack frame, i.e., mostly non-leaf functions. The padding is at least 8 bytes, even for the default binary. This reduces the number of cases where additional instructions have to be inserted in the prologue of the diversified version compared to the default version, as discussed in the last but one bullet above.
3. In the ARM backend, a one-line patch disables the optimization in which accesses via the FP are selected over accesses via the SP depending on the offsets. This patch

also reduces the number of changes introduced in the generated code as a side-effect of the padding insertion.

Together, these patches implement randomized stack padding, while minimizing the collateral damage in the form of secondary changes and side-effects. These patches also in no way limit the compiler capability to omit frame pointer when useful.

The patches cannot prevent that more complex code differences between default and diversified binaries will occur, so in general, our symbol differ and symbol patcher need to be capable of handling more extensive code differences. This is discussed in more detail in the next section.

7.3.3 Compact Symbol File Patches

As explained in the previous section, our prototype implementations supports three forms of diversification, in two compilation steps: the LLVM backend first applies stack padding, after which the Diablo-based link-time rewriter performs no-op insertion followed by function shuffling. In this regards, our experimental build system differs somewhat from the idealized visualization in Figure 22. Instead, it looks more like the one in Figure 24.

The figure shows that we actually do not construct one symbol file patch, but two. The first covers the distance between the default binary and the one in which LLVM randomly padded the stack frames. The second covers the no-op insertion and the function shuffling. On the crash collector server, these two patches will be applied one after the other, in the same order.

To ease our research, all unencrypted, uncompressed patch files are simple human-readable ASCII text files. With more but relatively simple engineering, smaller patch sizes can be obtained. So any sizes we will report are upper bounds on what could be achieved with a more mature implementation.

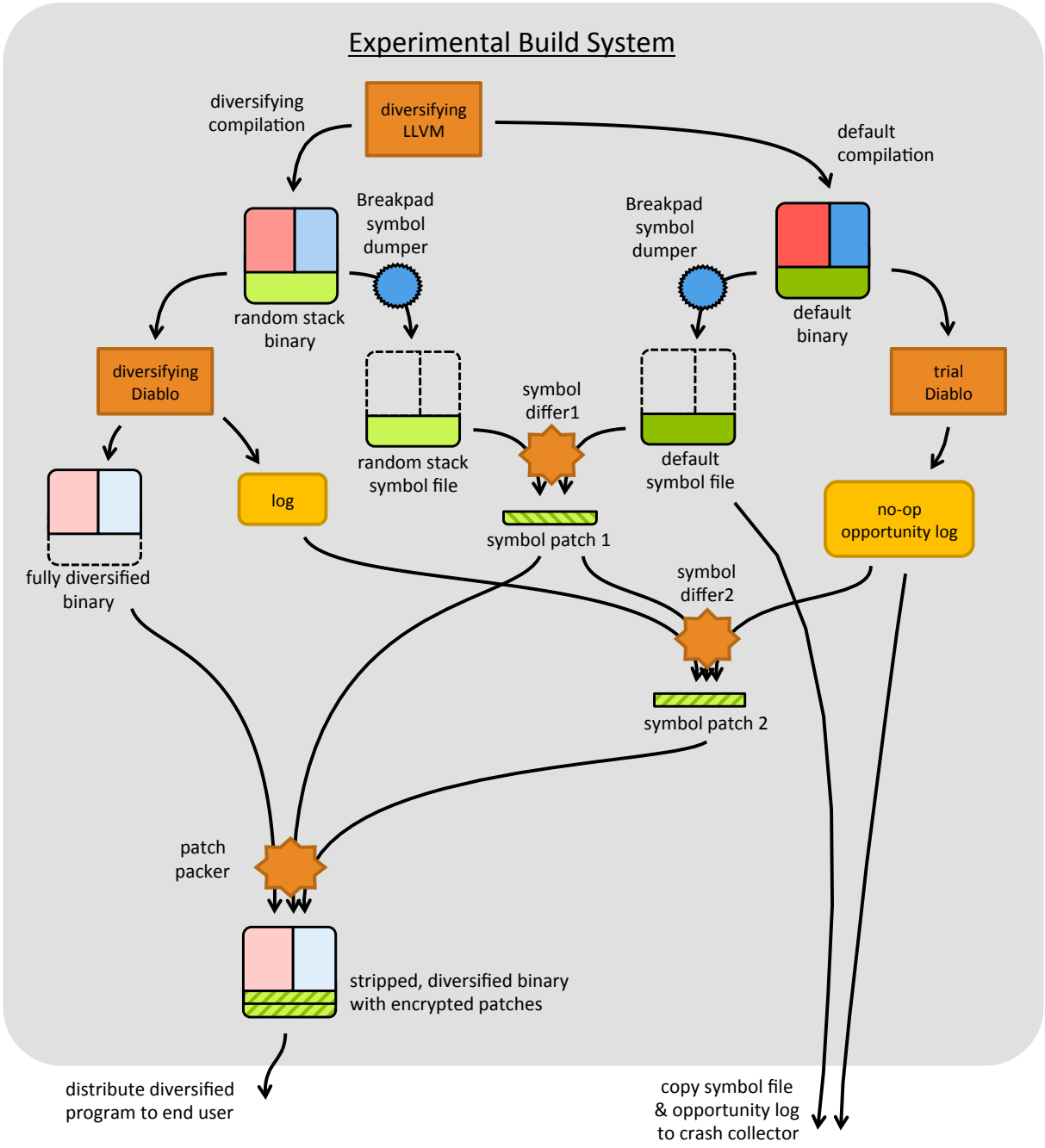

Figure 24 - Our prototype build system on top of two diversification tools similar to those in the ACTC

7.3.3.1 Patch file 1: stack padding

As discussed in Section 7.3.2, the injection of stack padding can cause additional instructions to be injected and existing code to be reordered. Because our LLVM patches aim at limiting the secondary effects of randomized stack padding, such occurrences are rare. In particular the reordering happens very rarely. So it is not problematic if we do not describe the necessary patching for those cases in the most compact form.

To generate the first patch, we developed a custom differ (called differ 1 in Figure 24) that diffs the two symbol files obtained from the two outputs of LLVM, i.e. the default one and the partially diversified one. Those symbol files contain two parts of interest.

Description:

```
FUNC address size parameter_size name
address size line filenum
```

Example excerpt:

```
FUNC 157c 34 0 google_breakpad::LineReader::PopLine
157c 4 113 4
1580 30 116 4
FUNC 15b0 38 0 sys_close
15b0 4 2979 16
15b4 1c 2979 16
15d0 10 2979 16
15e0 8 2979 16
FUNC 15e8 5c 0 google_breakpad::PageAllocator::FreeAll
15e8 4 142 13
15ec 8 142 13
```

Figure 25 - Source line mapping in the symbol file

Description:

```
STACK CFI INIT address size reg1: expr1 reg2: expr2 ...
STACK CFI address reg1: expr1 reg2: expr2 ...
```

Example symbol file excerpts:

```
STACK CFI INIT 1bdc f0 .cfa: sp 0 + .ra: lr
STACK CFI 1be0 .cfa: sp 8 + .ra: .cfa -4 + ^ r11: .cfa -8 + ^
STACK CFI 1be4 .cfa: r11 4 +

...

STACK CFI INIT 28a4 f8 .cfa: sp 0 + .ra: lr
STACK CFI 28ac .cfa: sp 20 + .ra: .cfa -4 + ^ r4: .cfa -20 + ^
             r5: .cfa -16 + ^ r6: .cfa -12 + ^ r7: .cfa -8 + ^
STACK CFI 28b4 .cfa: sp 904 +
```

Corresponding assembler code excerpts:

```
<function1>:
     push    {fp, lr}
     add     fp, sp, #4
     sub     sp, sp, #16
     ...
<function2>:
     push    {r4, r5, r6, r7, lr}
     cmp     r3, #0
     sub     sp, sp, #884    ; 0x374
     ...
```

Figure 26 - Stack walking information in the symbol file

The first part describes the mapping of function symbols (using FUNC m-entries) and the mapping of instruction chunks to source code files and line numbers (using m-entries starting with the chunk's address). An example is depicted in Figure 25. Our differ walks over all functions in this part of the two symbol files. Per function, it first pair-wise compares all the chunk m-entries. It is immediately clear whether or not instructions were added or deleted in a function because of the stack padding. If not, no information whatsoever needs to be stored for the function. If instructions were only inserted or deleted, a difference shows up in the second column. In that case, we record which entry was changed and how much. Furthermore, we check that the offsets between the addresses of consecutive m-entries are consistent in the two symbol files. If they show differences, we also track that. We need to track both the changes in size and in offset because of the potential presence of padding no-ops and literal pools that the compiler might have inserted in between code fragments.

When code has been reordered in a relevant manner, i.e., in a way that impacts this part of the symbol file, this is obvious from the third and fourth column in a function's entries. In that very rare case, our differ produces a more verbose diff record that simply states which lines to replace, insert, and remove to convert the default symbol file entries into the diversified ones.

All collected information is stored in the symbol patch in order. To apply the patch, the patcher executed on the crash collector simply walks through the default symbol file and updates each entry on the fly, keeping track of the shifts in absolute addresses.

The second part of the symbol file describes how to walk the stack. Using post-fix expressions on symbolic register names (r3, r11, lr,...), this part describes code chunks and how to compute certain stack-related values when execution has reached a point in a given chunk. The "registers" of which the values can be computed are the canonical frame address (.cfa), the return address (.ra), and the values of callee-saved registers in a function's caller. The two example excerpts in Figure 26 illustrate an interesting point. The first three entries relate to function1 that has a FP, and for which register r11 is reserved on ARMv7. The expression for .cfa on the first line encodes that on entry to function1, the SP still points to the start of the function's stack frame. The second line clarifies, amongst others, that after the push instruction, two callee-saved registers can be found on the stack, and that the SP now points 8 bytes beyond the start of the frame. The third entry, corresponding to the program point following the add instruction that sets the FP, indicates that the start of the frame can be computed by adding 4 to the FP.

In the example, function2 does not have a FP. As a consequence, even after the sub instruction that allocates the local area on the stack, the start of the stack frame has to be computed by adding 904 to the SP.

When random padding is inserted, the stack part of the symbol file changes in three ways. First, the addresses and sizes of the code chunks in the STACK entries can change. We keep track of those like we did for m-entries in the line number part. Overall, such changes are rare.

Secondly, the numeric constants in the post-fix expressions can change. With our implementation of randomized stack padding as discussed in Section 587.3.2, such changes are limited to STACK entries such as the last one in Figure 26 that define the .cfa in terms of the SP. Such changes are obviously easy to check. If the .cfa is defined in terms of the FP, the values do not change as a result of stack padding, because the padding does not alter the location of the FP in the stack frame, which is specified by the ARM EABI. Concretely, this means that such changes only occur when the software was compiled with the -fomit-frame-pointer option enabled. In that case, all diversified functions with a local stack area and a fixed SP (i.e., no varargs) will result in at least one entry in the symbol patch.

7.3.3.2 Patch file 2: no-op insertion and function shuffling

As discussed in Section 7.3.2.1, no-op insertion only requires us to track where no-ops, trampolines, literal pools, and jumps over literal pools have been added or removed from the code. Function shuffling only requires us to store the starting addresses of the shuffled instructions. All of this information is obtained trivially from the instruction address mapping that Diablo produces.

It is important to realize that the insertion of the mentioned instructions and data, as well as the shuffling of functions only influence the instruction addresses that occur in the symbol file, not the source line numbers or the stack properties. Literal pools' addresses do not occur in the symbol file. Inserted no-ops and inserted jumps over (moved) literal pools can be mapped onto the same source line as the chunk they are added to, so the effect on the symbol file merely involves increasing the size of a chunk, and shifting remaining chunks down in the address space.

In dynamically linked binaries or libraries, trampolines need to be inserted very rarely. If some are needed, they can most often be inserted right next to the branch or call instruction that requires the trampoline. This is the case, e.g., when the 20 offset bits in a conditional branch do not suffice to reach the target, but the 24 bits in an unconditional branch do suffice. In that case, the trampolines can simply be accounted in the symbol patch as increases in the original branch's chunk. In the extremely rare event that a trampoline is inserted somewhere else in the binary as a side effect of the no-op insertion, a completely new entry needs to be inserted into the default symbol file to translate it into the diversified symbol file.

So in almost all cases, and in fact throughout all of our experiments, only the locations of inserted no-ops, moved function entry points and moved data pools needs to be stored to enable correct patching. This obviously contributes to the compact encoding of a patch.

Still such a patch can grow quite big. If no-ops are inserted at a high frequency, listing all of them takes considerable space. To avoid this growth, we will try not to list all locations where no-ops were inserted in the diversified binary. Instead, we will store just enough information to allow the crash collector to recreate this list.

To recreate the list, the crash collector needs to be able to replay the randomized no-op insertion that was performed on the random stack binary on the very left of Figure 24. To replay this, the crash collector needs four pieces of information.

1. The collector needs to know the seed with which the pseudo-random number generator (PRNG) was initialized that controlled the randomized no-op insertion into the random stack binary. This seed can be stored in the second symbol file patch, such that it will be sent to the crash collector as part of the crash report.
2. Obviously, the crash collector needs to have the same PRNG. This can be achieved by using a custom, deterministic one that can easily be reproduced on the crash collector server.
3. To replay the invocations of the PRNG, the collector also needs to know the list of all locations at which the randomized no-op insertion probabilistically decided whether or not to insert a no-op by invoking the PRNG.
4. Finally, to replay those exact decisions, the crash collector needs to know all parameters involved in every decision at every location. If, for example, the no-op insertion was based on profile information to minimize the run-time overhead, the no-ops are inserted probabilistically based on the execution counts of program locations. In that case the crash collector needs to get the profile information, the threshold values used to distinguish hot from cold code, and the probabilities with which no-ops were inserted at hot and cold code locations. Like the used PRNG seed, those probabilities and the used threshold values can differ from one diversified copy to another. So we store those in the second symbol patch in the diversified binary, such that they are sent to the crash collector as part of the crash report.

As for the profile information, we believe it is realistic to assume that the same profile inputs will be used for every diversified instance: Doing otherwise would require too many resources to re-perform the profile runs, and too many different inputs.

Still, we face a problem with respect to the profile information of item 4 above and with respect to item 3, the list of all locations where the no-ops were inserted probabilistically: The instructions for which that information needs to be available are those of the random patch binary, not those of the default binary. And as we already discussed in Section 7.3.2 , the randomized stack padding insertion may have significant side-effects in the code, in particular in additional instructions being inserted or instructions being removed and in rare cases even in reordered code.

Because the stack padding gets inserted very late during LLVM's backend code generation, however, all of those code changes are only local. They are confined to individual basic blocks and do not impact the structure of the functions' control flow graphs. So at least the

basic block execution counts remain constant for all binaries, only the addresses and sizes of basic blocks vary.

Furthermore, as discussed in the previous section, the crash collector has already used the first symbol patch file to translate the addresses of code chunks in the default binary to the addresses of the corresponding chunks in the random stack binary.

This means that almost all information necessary to compute the addresses and execution counts of the program points that were considered for inserting no-ops during the no-op insertion in the random stack binary can be obtained from the default binary (in what we call the no-op opportunity log), symbol patch 1, and the no-op insertion parameters and PRNG seed that will already be passed as part of symbol patch 2.

This information might not be completely accurate because the mapping from code chunks to basic blocks is not a perfect one-to-one mapping, and because the precise instruction reordering during the randomized stack padding is not reconstructed with symbol patch 1, but for almost all functions in all our test programs, it allows a completely accurate prediction of the randomized no-op insertion points on the random stack binary, and hence for a perfect replay of the no-op insertion.

To fix the rare cases where the prediction is wrong, it suffices to add a small amount of additional information to symbol patch 2.

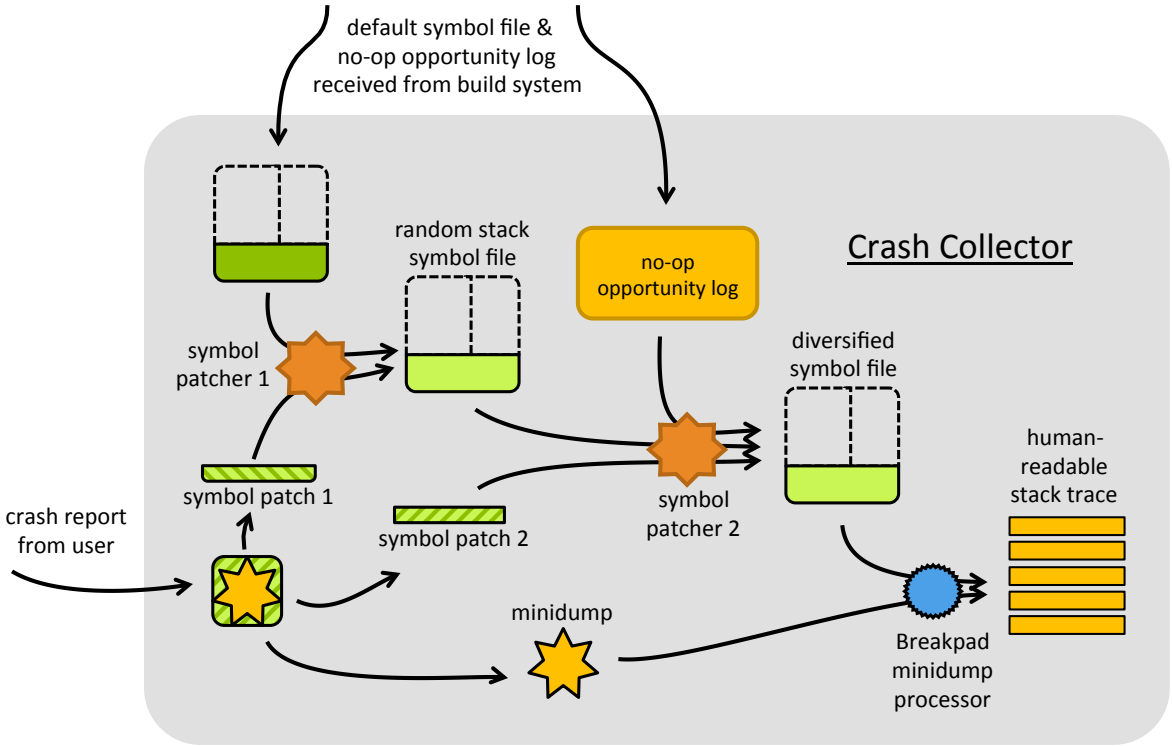

Figure 27 - Our prototype collector

7.3.3.3 Overview: Patch Generation and Symbol File Reconstruction

Figure 24 and Figure 27 present a complete overview of the patch generation and patch consumption. On the right of Figure 24, the "trial" run of Diablo is shown that mimics the no-op insertion decision process on the default binary, and produces the no-op opportunity log that contains the relevant program points and, if necessary, the relevant basic block execution counts. This will be send to the collector server, together with default symbol file. For each diversified binary, the first symbol patch is reconstructed by symbol differ 1 as

discussed in Section 7.3.3.1, based on the default symbol file and the random stack symbol file.

Symbol differ 2 uses the log of the diversifying Diablo run, symbol patch 1, and the no-opportunity log to create symbol patch 2. This patch contains

- the seed and any other parameters fed to Diablo to control the no-op PRNG, item a first patch part to correct the predicted list of program points and profile information for which the PRNG is invoked,
- a second patch part to translate changes in symbol file addresses due to function shuffling and due to side effects of the no-op insertion and of the shuffling.

On the crash collection server, the two patches and the minidump are extracted from the crash report. Then symbol patcher 1 applies symbol patch 1 to the default symbol file to reconstruct the random patch symbol file.

Symbol patcher 2 then first uses symbol patch 1 to convert the no-op opportunity log of the default file to a rough approximation of the list of decision points in the random stack binary. Then the patcher applies the first part of symbol patch 2 to reconstruct an accurate list of those points and the corresponding profile information. This accurate list, together with the PRNG seed and the other parameters in symbol patch 2 then enable symbol patcher 2 to replay the whole no-op insertion, with the exception of some rare decisions inside basic blocks that cannot be replayed completely accurately as indicated above.

Next, the patcher uses the second part of symbol patch 2 to correct the rare occasions where the replay wrongly predicted the inserted no-ops, to model the side-effects of the no-op insertion and the address permutations following from the function shuffling. The result is a complete mapping of addresses in the random stack binary to addresses in the fully diversified binary.

Symbol patcher 1 then uses this mapping to convert the already reconstructed random stack symbol file into the diversified symbol file matching the diversified binary, as shown to the right of Figure 27.

Finally, the standard Breakpad minidump processor then uses this diversified symbol file and the minidump extracted from the crash report to generate the human-readable stack trace.

7.3.4 Experimental Evaluation

At the time of writing of this document (Oct 2015), we are still finalizing the full implementation. Early experiments indicate that (a) the crash reporting works correctly on diversified binaries, (b) the amount of information that needs to be send to the crash report server together with the minidump remains in the order of a few kilobytes, even for large programs such as the GCC compiler. In later deliverables, more extensive results will be presented.

7.4 Feedback-driven diversification with minimal performance overhead

Section Author: Bjorn De Sutter (UGent)

As described in the DoW, UGent contributes its background IP regarding binary code diversification to the ASPIRE project. By contrast to the previous section, that IP focuses mostly on diversification in time: UGent's approach aims for generating one binary for a piece of software, such that that binary differs enough from all previously distributed binary versions of the same software to fool popular attacker diffing tools such as the combination IDA Pro - Bindiff. This approach has been described extensively in a couple of journal papers [Cop13a,Cop13b], so we will not discuss them in detail here.

In order to reuse that IP in the context of the ASPIRE project, however, the existing background IP first needs to be ported to ASPIRE's target platform. Whereas the original

implementation of UGent's IP targeted Linux/x86, ASPIRE targets Android/ARM for its demonstration and validation purposes.

For this porting effort, UGent has built on its already refactored, ported, and extended binary control flow obfuscation background, as described extensively in ASPIRE deliverable D2.06. This is possible because the transformations applied to generate diversity overlap to a large degree with the obfuscating transformations.

In addition to the porting, we also created a number of new scripts to automate the iterative diversification, i.e., to repeatedly invoke the diversifier and the diffing tool on which feedback the next iteration relies.

7.5 Renewability Plan

We envision the following implementation plan for the third year of the ASPIRE project:

- **M26:** creation of DB for storing different code blocks
- **M27**: Extension of the Code Mobility tool to transfer data blocks
- **M28**: first implementation of the Renewability Manager
- **M29**: testing of the approach with WBC and bytecode VM
- **M30**: initial implementation of renewable RA.
- **M30:** Complete integration of RA in the ACTC for deliverables D3.05 and D3.06.
- **M33**: Integration with Diablo for diversity and Renewability support in ACTC
- **M36**: Renewability on use cases (Nagra or SFNT).

Section 8 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

[Bin07] M. Binshtok, R. I. Brafman, S. E. Shimony, A. Mani, and C. Boutilier. Computing optimal subsets, volume 22. 2007.

[Cab15] Alessandro Cabutto, Paolo Falcarin, Bert Abrath, Bart Coppens, Bjorn De Sutter, Software Protection with Code Mobility. In Proc. of 2nd ACM Workshop on Moving Target Defense (MTD 2015), Oct. 2015 - pp 95-103.

[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.

[Cil05] R. Cilibrasi, P.M.B. Vitanyi, Clustering by compression, IEEE Trans. Inform. Theory, 51:12(2005), 1523–1545.

[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.

[Col97] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. 1997.

[Cop13a] Bart Coppens, Bjorn De Sutter, Jonas Maebe. Feedback-Driven Binary Code Diversification. ACM Transactions on Architecture and Code Optimization. Vol. 9 Nr. 4, Art. 24, January 2013

[Cop13b] Bart Coppens, Bjorn De Sutter, Koen De Bosschere. Protecting your software updates. IEEE Security & Privacy. Vol. 11 No. 2, pages 47-54, March-April 2013

[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.

[For97] S. Forrest, A. Somayaji, and D. Ackley, “Building diverse computer systems,” in Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI), ser. HOTOS ’97. Washington, DC, USA: IEEE Computer Society, 1997, pp. 67–72.

[Fre07] Manuel Freire, Manuel Cebrian, and Emilio del Rosal. AC: An integrated source code plagiarism detection environment. Arxiv preprint cs.IT/0703136, abs/cs/0703136, 2007.

[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].

[Hom13] A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz, "Profile-guided automated software diversity," in Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), ser. CGO '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 1–11

[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.

[Kil06] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning, "Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software," in Proceedings of the 22Nd Annual Computer Security Applications Conference, ser. ACSAC '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 339–348

[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.

[Min04] P. McMinn. Search-based software test data generation: a survey. volume 14, pages 105–156. Wiley Online Library, 2004.

[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.

[Pra11] K. Praditwong, M. Harman, and X. Yao. Software module clustering as a multi-objective search problem. volume 37, pages 264–282. IEEE, 2011.

[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.

[Zel] Zelix Klassmaster Java obfuscator. On-line at http://www.zelix.com/klassmaster/

[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.