Advanced Software Protection:
Integration, Research and Exploitation

D1.06
ASPIRE Validation

Project no.:	609734
Funding scheme:	Collaborative project
Start date of the project:	1st November 2013
Duration:	36 months
Work programme topic:	FP7-ICT-2013-10
Deliverable type:	Report
Deliverable reference number:	ICT-609734 / D1.06 / 1.01
WP and tasks contributing:	WP 1 / Tasks T1.5
Due date:	October 2016 - M36
Actual submission date:	9 January 2016
Responsible Organization:	~~GTO~~ - UGent
Editor:	Bjorn De Sutter
Dissemination Level:	Public
Revision:	1.01

Abstract:
This document reports the results of the validation activities conducted in the ASPIRE project. It validates the foundations in the forms of the attack mode, the requirements and the reference architecture. It then assesses whether or not the developed technology can meet the previously expressed requirements, whether all developed protections are actually covered by the use case assets such that they are validated, whether the protections foreseen within the project in principle suffice to protect the use case assets, to what extent the integrated protections can be deployed on the use cases, what protection the actually provided, and what overhead they introduce. Also the ADSS has been validated, its effectiveness in helping software protection experts by automatically producing useful data (threats, risks, mitigations) and by identifying the golden combinations

Keywords:
coverage, requirements, correct deployment, evaluation, assessment, overhead, protection strength

Editor

Bjorn De Sutter (UGent)

Contributors (ordered according to beneficiary numbers)

Bart Coppens, Jens Van den Broek, Bert Abrath (UGent)

Cataldo Basile, Daniele Canavese, Leonardo Regano, Alessio Viticchié (POLITO)

Brecht Wyseur (NAGRA)

Mariano Ceccato, Roberto Tiella (FBK)

Alessandro Cabutto (UEL)

Werner Dondl (SFNT)

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

We first assess whether or not the used attack model (D1.02v2.1), the stated requirements (D1.03v2.1), and the used reference architecture (D1.02v2.1) are still up to date. We conclude that (i) the technology it develops to protect mobile applications still covers the relevant Man-At-The-End attacks; (ii) the proposed requirements stand; (iii) the ASPIRE reference architecture suffices to deliver the protections as specified in the DoW, and thus provides an excellent vehicle for providing adequate protection at least on the project use cases, and most likely beyond those cases to a broad range of mobile applications.

Next, we validate the capability of the overall ACTC design, the design of its protection components, and the ADSS design to meet all the requirements of deliverable D1.03 v2.1. The goal of this validation exercise is to answer the following question: Do the designs allow meeting the security requirements and recommendations of the use cases? We conclude that the project results meet almost all requirements. In particular, the most important ones are met.

Next, we check the coverage of the protections by the use cases. The goal is to verify that all protections developed in ASPIRE are actually evaluated, demonstrated and validated on at least one use case. We conclude that, with the exception of some data obfuscations and multi-threaded cryptography, all protections developed during the project are covered by the validation of the use cases.

Next, we validate the capabilities of the individual and combined protections to meet the security requirements of the use case. The goal is to answer the following question: Do the protections as they are designed in the reference architecture and in the rest of the project in principle allow us to meet the security requirements of the use cases. We conclude the following. The envisioned, useful protections for the NAGRA use case are available in the ASPIRE protection framework, with the exception of interprocedural data protections. Control flow tagging was delivered too late for deployment and validation on the use case, however. The envisioned, useful protections for the SFNT use case are available in the ASPIRE protection framework, with the exception of interprocedural data protections. The envisioned, useful protections for the GTO use case are available in the ASPIRE framework.

Next, we validate the deployment of the individual protections integrated into the ACTC on the assets in the use cases. The goal is to answer the following question: Can the research tools implementing the protections actually apply the protections to the code fragments corresponding to the assets? The reason this question needs to be answered is that in a project with limited resources and time, the research tools cannot be expected to handle all corner cases of the C programming language correctly. To allow validation and demonstration, they should at least handle all relevant uses of the C programming language in the assets in the use cases. We conclude that all the protections supported in the ACTC and listed as potentially useful for the use cases can be deployed on those use case, given a slight redesign.

Next, we validate the protection strength of the protections in isolation and in combination, based on the academic experiments, the tiger experiments, and the public challenge conducted in WP4, as well as on other validation experiments. We conclude that although many of developed and integrated protections still offer a large potential for improvement, those protections effectively delay attacks and increase the effort that attackers need to invest in identifying attack vectors; make it harder to exploit identified attacks at a large scale; and hence effectively reduce the profitability of engineered attacks. We also conjecture that the developed and integrated renewability techniques deliver improved protection, as they make the scaling up of attacks harder, as they can delay attack vector identification, and as they can help in raising the costs faced by attackers.

To a large degree, the project has hence achieved its goals to demonstrate that software-based protection techniques can deliver true protection.

Next, we assess the overhead caused by deploying the various protections. The goal is to validate that the amount of overhead introduced by the deployment of the protections can be controlled by the user or decision tools, and that the amount of overhead does not deem the protections unusable. We conclude that the overhead they introduce is limited and can be controlled by deploying the protections cautiously. In particular, on the project use cases, the protections could be deployed as foreseen by their developers and the project's security experts, with an acceptable overhead.

Finally, we validate the ADSS. The goals of the performed exercise are to validate that the ADSS can reason about the deployment of protections on software as complex as the project use cases, can take into account realistic attack paths as build from the ASPIRE Knowledge Base (AKB), and can come up with adequate (if not optimal) combinations of protections given the existing knowledge in the AKB. The ASPIRE consortium concluded that the ADSS has a very high potential even if it is not yet ready to be used to protect real applications. The ADSS can automate complex operations (like discovering attack paths, suggesting protections). It can also gather a huge amount of data useful for making decisions. Moreover, it proposes effective combinations of protections, even if it was not possible to validate these combinations as optimal. However, there is more work to do to improve the ADSS. The level of details of the output is not yet enough to be used in practice (e.g., attack paths description is too coarse grained) and there are still concerns about the presentation of the results, especially outside the ASPIRE project.

Revision History

Version 1.01 of this document fixes several small factual mistakes from v1.0 :

- The assessment of REQ-ASR-006 in Section 3 has been updated, as the necessary metrics have been developed for client-server code splitting (after the validation of the requirements effort was finalized).
- In Table 2, some cells have been corrected: the SoftVM has in fact been deployed successfully on the NAGRA use case, and the WBC protection not useful for the GTO use case, as the diversified cryptographic library protection was used there instead.

Note that all of these changes improve the outcome of the validation!

Contents

List of Figures

List of Tables

Section 1 Introduction

Section Authors:

Bjorn De Sutter (UGent)

An important chain of task dependencies in the ASPIRE project is the following:

1. Define use cases (D1.01, M3) and their security requirements (D1.02, M3 and D1.03, M6).
2. Define a reference protection architecture with which the requirements can be met on the use cases (D1.04, M9).
3. Develop the use cases (D6.01, M12).
4. Develop the many protection techniques (many deliverables in WP2 and WP3).
5. Design the ASPIRE Compilation Tool Chain (D5.01, M9).
6. Integrate the protection techniques into the ASPIRE Compilation Tool Chain (ACTC) and apply them onto the use cases (many deliverables in WP5).
7. Build decision support to automate the use of the ACTC, i.e., the selection of the available protections and their deployment, on the use cases, and to let users of the ACTC understand how, where and why to deploy certain protections.
8. Validate and demonstrate the protection of the use cases in terms of the stated requirements (D1.06 and D6.03, M36).

This validation consists of 6 major parts, corresponding to the next 6 sections in this report.

- **Section 2**: Validation of the used attack model (D1.02v2.1), the stated requirements (D1.03v2.1), and the used reference architecture (D1.02v2.1). The main question to answer is whether or not they are still up to date.
- **Section 3**: Validation of the capability of the overall ACTC design, the design of its protection components, and the ADSS design to meet the requirements of deliverable D1.03 v2.1. The goal is to answer the following question: Do the designs allow meeting the security requirements and recommendations of the use cases?
- **Section 4**: Coverage of the protections by the use cases. The goal is to verify that all protections developed in ASPIRE are actually evaluated, demonstrated and validated on at least one use case.
- **Section 5**: Validation of the capabilities of the individual and combined protections to meet the security requirements of the use case. The goal is to answer the following question: Do the protections as they are designed in the reference architecture and in the rest of the project in principle allow us to meet the security requirements of the use cases.
- **Section 6**: Validation of the deployment of the individual protections already implemented and integrated into the ACTC on the assets in the use cases. The goal is to answer the following question: Can the research tools implementing the protections actually apply the protections to the code fragments corresponding to the assets? The reason this question needs to be answered is that in a project with limited resources and time, the research tools cannot be expected to handle all corner cases of the C programming language correctly. To allow validation and demonstration, they should at least handle all relevant uses of the C programming language in the assets in the use cases.
- **Section 7**: Validation of the protection strength of the protections in isolation and in combination. Based on the academic experiments, the tiger experiments, and the public challenge conducted in WP4, as well as on other validation experiments, the provided level of protections are assessed.

- **Section 8**: Validation of the acceptable overhead caused by deploying the various protections. The goal is to validate that the amount of overhead introduced by the deployment of the protections can be controlled by the user or decision tools, and that the amount of overhead does not deem the protections unusable.
- **Section 9**: Validation of the ASPIRE Decision Support System (ADSS). The goals are to validate that the ADSS can reason about the deployment of protections on the project use cases, can take into account realistic attack paths as build from the ASPIRE Knowledge Base (AKB), and can come up with adequate (if not optimal) combinations of protections given the existing knowledge in the AKB.

Sections 2 to 9 contain clearly marked conclusions.

Section 10 brings all of the conclusions together in one concise conclusion section.

Section 2 Attack Model, Requirements, Reference Architecture

Section Authors:

Bjorn De Sutter (UGent)

At the end of the second year of the ASPIRE project, the consortium updated the attack model considered in the project (D1.02), the considered security requirements (D1.03), and the reference architecture of the protections to be deployed (D1.04).

The updated versions as revised after the year 2 review, i.e., D1.02 v2.1, D1.03 v2.1 and D1.04 v2.1 are the basis for the validation reported here. The validation itself is conducted on the three project use cases that were documented in D6.01 v2.1. The deployment of protections on the use cases as validated in this document will also be demonstrated in D6.03, as planned in D6.02.

Regarding that basis, the consortium is confident that

- the technology it develops to protect mobile applications still covers the relevant Man-At-The-End attacks;
- the proposed requirements stand;
- the ASPIRE reference architecture suffices to deliver the protections as specified in the DoW, and thus provides an excellent vehicle for providing adequate protection at least on the project use cases, and most likely beyond those cases to a broad range of mobile applications.

Section 3 ACTC Security Requirements Assessment

Section Authors:

Bjorn De Sutter (UGent)

In Q3/Q4 of 2015, the consortium spent two conference calls and one physical meeting session on the validation of the capability of the overall ACTC design and its ongoing implementation to meet the requirements of D1.03 v2.1. For each requirement and recommendation of that project, the consortium discussed whether or not the requirements are met. For each requirement, the discussion can have four outcomes:

- **OK**: We meet the requirement (sufficiently).
- **FAIL**: We do not meet the requirement, and there is a fundamental reason why the requirement cannot be met, even if more effort/engineering is invested to extend or revise the current design.
- **Compatible**: We do not meet the requirement currently, but there are no fundamental incompatibilities between our design and implementation results and the specified requirement. In other words, with additional engineering effort, it would be perfectly possible to meet the requirement. This outcome typically results from project management decision to prioritize on more scientifically relevant topics and more challenging problems within the project, given its limited scope and resources.
- **Open**: We are not in a position yet to draw final conclusions.
- **Confirmed as Infeasible**: In the case of recommendations, the fact that a recommendation rather than a requirement was put forward by the industrial partners already hints at their infeasibility. So in case we don't meet a recommendation, we report this as a confirmation, rather than as a failure (as suggested by the project's Advisory Boards).

Table 1 lists the results of the discussions, with some additional clarifications. Updated cells (compared to the intermediate validation of D1.05 v1.2) are marked with a *.

The conclusion of this list is that the project results meet almost all requirements. In particular, the most important ones are met.

Table 1 - Assessment of the ACTC in light of the ASPIRE security requirements

Requirement	Status	Additional Information
REQ-FSR-001	OK	ASPIRE's remote attestation protection provides this.
REQ-FSR-002	Compatible	Standard encryption can provide this.
REQ-FSR-003	Compatible	Annotations can easily be foreseen to indicate that dynamic remote attestation or other protections should never send privacy-sensitive data computer or stored in the application to the security server. With respect to the use of the application itself (e.g., the pure fact of launching the application and connecting to a server), lawyers have to answer the question what steps are needed to make the collection of such data legal, e.g., by means of a clear end-user license agreement.

REQ-FSR-004	OK	ASPIRE's anti-cloning protection provides this.
REQ-FSR-005	OK	Code mobility supports this.
REQ-FSR-006	OK	Renewed code is not stored persistently, so a new version must be downloaded, code downloaded during the execution will be checked by the remote attestation (using specific checks for mobile code, binder, and binder data).
REQ-FSR-007	Compatible	It is simple to update the ASPIRE reference architecture to include the transfer of a signature, and to perform verification in the Code Mobility Downloader or Binder. Schemes exist that do not require a shared secret, such as PKI (public key infrastructure)
REQ-FSR-008	Compatible	This requirement can be met easily by using encrypted communication schemes where a nonce is updated per transmission.
REQ-FSR-009	OK	Mobile code can be obfuscated by means of the ASPIRE binary control flow obfuscations, its anti-debugging technique, and by converting it to bytecode.
REC-FSR-010	Compatible	Easy to meet, but no engineering resources will be spent on this in the project, as enough infrastructure already exists outside of the project.
REC-FSR-011	OK	For the online protection techniques, we already have identification. Application authentication is delivered by means of the remote attestation. User authentication depends so much on the type of application and the content, service, or software provider that user authentication methods need to be provided as part of the application to be protected. ASPIRE technology will then protect the code in support of that user authentication.
REQ-NFS-001	Partial *	ASPIRE's anti-callback checks provide some protection, but they implement no full control flow integrity. Furthermore, the stripping of libraries combined with the application of control flow obfuscations and code layout randomization make the identification and lifting of API functions hard. During the tiger experiments and the public challenge, attackers did not try executing functions protected with anti-callback checks out of context. Either no out-of-context execution was tried, or, in one experiment, it was limited to the execution of exported APIs, which are by definition allowed to be executed out of context. *
REQ-NFS-002	OK	External APIs are left intact.
REQ-NFS-003	Partial *	Almost all security libraries are linked into the application/library to be protected, and protected as a whole during the binary-level processing steps of the ACTC. The only exceptions are the reaction logic and diversified cryptography. *
REQ-NFS-004	OK	Binaries/libraries produced by Diablo do not contain debug information.
REQ-NFS-005	OK	We only use BSD-licensed external code for the moment. This requirement is also enforced by the project's Consortium Agreement, so no partner is allowed to break it by silently introducing GPL-licensed software.
REC-NFS-006	Compatible	To save on engineering effort, the project currently reuses 3rd party software (cURL, OpenSSL, LLVM) as part of the protected applications. No

		protection depends specifically on these implementations of the required functionality, however, so it is perfectly possible to avoid the 3rd party software.
REC-NFS-007	Partial	This requirement is met with respect to cryptographic keys and their protection via white-box cryptography techniques. No keys have been extracted from white-box crypto code during the Tiger experiments or public challenge. ASPIRE's data obfuscation implementations come with some limitations, however, such as the fact that no data can remain encoded over function boundaries and that it does not apply to aggregate data structures. This makes them inapplicable to some of the use case assets. *
REC-NFS-008	Compatible	Currently, no protections check the integrity of hardcoded data. Currently, no hardcoded data in the use cases needs to have its integrity checked. The static remote attestation or offline code guards could be extended to check this, but it is not yet clear whether there will be enough resources in the project to implement this within the project.
REC-NFS-009	Partial *	In some of the tiger experiments, the control flow obfuscations pushed the attackers towards attack paths with dynamic techniques. In that regards, the protections were effective. When traces could be produced or debugging techniques deployed by the attackers, however, the code could still be reverse-engineered to some extent. In other tiger experiments, some of the control flow obfuscations were undone using static rewriting techniques. So their protection strength was weak at best. *
REC-NFS-010	Partial *	In the tiger experiments, tampering was not required in most cases. In one case, out-of-context execution was enabled by tampering with the initialization code of the anti-debugging technique. That code was not guarded, however, so this can be considered on overlooked case where protections had to be combined better. *
REC-NFS-011	Compatible	The delay data structures have been proposed to meet this requirement.
REC-NFS-012	OK	ASPIRE's anti-debugging provides the necessary protection.
REC-NFS-013	Partial	ASPIRE's anti-callback checks and remote attestation provide some of the required protection. The project consortium has decided, however, not to invest in protections to detect injected threads. Priority was given to demonstrate protections on dynamically linked libraries. In that context, thread counting and reliable checking of valid thread counts are considered to complex to be done within the project.
REQ-ASR-001	OK	In so far as profiling information can be presented beforehand or can be obtained automatically, with external tools/scripts.
REQ-ASR-002	OK	The ACTC configuration options and the source code annotations' expressiveness support this.
REQ-ASR-003	Compatible	Implementing versioning support properly requires a lot of engineering. To save resources, this is not done within the project. However, all development of the ACTC is compatible with proper versioning support.
REQ-ASR-004	OK	Only deterministic tools are used.
REQ-ASR-005	OK	Extensive logging is built into the ACTC and all its components.

REQ-ASR-006	Partial *	A metrics framework has been proposed and support for a number of metrics has been implemented. This requirement is not completely met for one reason, however: due to insufficient resources, not all proposed metrics are yet supported in the prototype implementation of the ACTC. *
REQ-ASR-007	OK	The project Consortium Agreement also enforces this.
REC-ASR-008	Confirmed as Infeasible	As any (industrial) protection tool chain, the ACTC imposes several requirements and restrictions, and requires considerable effort by its users. ASPIRE and its ACTC are certainly not worse in than commercial solutions, and probably even better. That remains to be evaluated at the end of the project, however, when the ADSS is evaluated.
REQ-ASR-009	OK	All tools in the ACTC can handle shared libraries.
REQ-ASR-010	OK	Extensive happens on Android 4.0 devices.
REC-ASR-011	OK	We also test binaries on several versions of Linux and Android, including v.4.0, but also others.
REC-ASR-012	Confirmed as Infeasible	Some protections are by construction incompatible with requirements such as the fact that all code should be available statically (Apple Store).
REQ-ASR-013	OK *	The ADSS can take performance overhead into account when determining the best set of combinations. Moreover, in none of the project use cases, performance overhead was problematic or even significant. *
REQ-ASR-014	OK *	The ADSS can take the requirements into account when considering protections to deploy, and on the use cases, the user experience was not impacted at all by the selected protections. *
REQ-ASR-015	Compatible	This requirement might require some additional engineering for turning the ASPIRE research outcomes into robust industrial tools, but the ACTC and its components already produce extensive logs, and techniques are known and used to meet this requirement, such as the use of trees of pseudo-random number generators (PRNGs), such that each feature gets its own random number sequence independent of what other protections/features also request random numbers.
REQ-ASR-016	OK	The ADSS only relies on deterministic components.
REQ-ASR-017	OK	No persistent renewed code is supported in ASPIRE.
REQ-ASR-018	Compatible	Linear constraints in terms of protected application features can be supported with the ADSS and its optimization function. That covers most of those application features. Constraints relating to the development lifecycle, time-to-market, tool chain constraints, etc. can be handled with feature matrices in the ASPIRE knowledge base. The user can then specify his requirements and the ADSS can exclude problematic techniques with respect to the specified requirements.

Section 4 Coverage of ASPIRE protections

Section Authors:

Bart Coppens, Bjorn De Sutter (UGent), Mariano Ceccato (FBK)

The ASPIRE project is driven by its industrial partners, their use cases, and their use case security requirements. It is on those that the ASPIRE protections will be evaluated and demonstrated. The use cases were developed during the project, specifically for the project, and so are the protections. Some are developed during the project; some are even designed during the project. So it is only during the project itself that the authors of the use cases can make an estimated guess as to what implemented protections might actually be good candidates to protect the assets in the use cases and thus meet the security requirements.

To ensure that all protections designed, developed, implemented, and integrated during the project will actually be validated and demonstrated on at least one use case at the end of the project, we investigated the coverage of our protections.

This investigation coincided with the annotation of the use case source code by means of so-called protection annotations, as defined in deliverable D5.01 Section 2.3. These annotations specify in detail which protections need to be applied where in the use cases. While the ultimate goal of the project is not to depend on such manually added protection specifications, but rather build on security requirement annotations that simply mark assets and security requirements that the ADSS then converts to protection annotations, we do need such manual protection annotations at this point in the project.

As the ADSS is not ready yet, we need such annotations to actually trigger the deployment of the protections in the ACTC during its compilation and protection of the use cases, as needed for the intermediate validation reported in this document, as well as for the attack experiments that will be conducted on the use cases by the industrial partners' tiger teams during year 3. Furthermore, this manual protection annotation will allow us to experiment with the protections and to find a good trade-off between protection and cost, thus setting the bar for the ADSS in year 3.

4.1 Approach for selecting the protections

The industrial partners documented the assets in their use cases, the requirements on those assets, and the protections they considered potential candidates to protect the assets and to meet the security requirements. Each industrial partner then shared its document with the academic partners[1], after which they jointly discussed the candidate protections and their deployment on the use case, and ultimately annotated the use case source code. This effort was spread over time, and while there was some overlap in time, the three use-cases were to a large degree handled one after another.

[1] As was the case for their actual use cases, industrial partners did not share these asset description documents with the other industrial partners (i.e., to some extent their competitors) for reason of confidentiality. For that reason, no excerpts of those documents are included here. As an example, the document of NAGRA (numbered WD6.02) will be made available to the reviewers of the project. While we are of course open to answering questions regarding those documents during the project review meetings, we urge the reviewers to announce such questions before asking them, such that the present investigators of the other industrial partners can leave the meeting to respect the confidentiality of the documents' contents.

NAGRA Use Case. For the first use case, the asset description document WD6.02 was delivered in April 2015, and was first discussed between UGent and NAGRA during several conference calls. Jointly, they drafted the annotations for the selected protections in the use case source code, and then organized conference calls with the partners that contribute those protections to discuss the feasibility of the annotations, and to ensure that all involved partners have a common understanding and shared expectations about the applicability and the impact of the annotations on the assets and the use cases. Based on the feedback obtained during those conference calls, the annotations were then revised.

SFNT Use Case. For the second use case, SFNT started with adding comments in the source code to mark the assets in February 2015. Already for the second review demo preparation, academic partners worked with SFNT in April 2015 to add annotations. Later, SFNT extended and refined their asset description in the form of the internal working document WD6.03. Talks between NAGRA and SFNT investigators started in May 2015 around and during the two-day full consortium meeting in London (UK) with respect to the deployment of NAGRA's white-box cryptography (WBC) on the SFNT use case. These talks included (1) the precise forms of WBC for which NAGRA had to develop support in their WBC tools, and (2) the way in which SFNT had to update its code to make its structurally and syntactically fit for deployment of those WBC tools. Discussions also started with the academic partners to seek the best opportunities for their protections as they were continuously being integrated into the ACTC. SFNT then delivered a first set of source code annotations, which were discussed with academic partners during the October 2015 two-day full consortium meeting in Saint-Cyr-Sur-Mer (FR). At that time, additional annotations and applicable protections were proposed by some of the academic partners with regards to some of the protections that were still being integrated into the ACTC late in year 2 of the project.

GTO Use Case. GTO described the assets in its source code in a separate text file committed together with the use case, that was made available to the partners in February 2015. GTO also identified the assets in the source code by means of code comments, in June 2015. Around and during the meeting in Saint-Cyr-Sur-Mer (FR), GTO's principal investigator then discussed the candidate protections for its use case with the academic partners that deliver those protections and their tool support. During those discussions, the first concrete annotations were also proposed. A methodology has been put in place in January 2016 to enable partners to set protections independently. An extra pre-processing step generates the annotations in the source code according to a protection activation configuration set by the developer in a script. With this mechanism the developer doesn’t need to edit the source code of the use case but he can just activate protections in a script. None, one, several, or all protections specified for the use case can be set by configuration. The constraint is that a new protection annotation set in the use case must be placed in a macro with a specific syntax. If the developer is reluctant to this methodology, he/she can still specify no protection in the shell script and specify the ASPIRE annotations by hand in the C source file generated by the pre-processor. Based on this methodology, Gemalto has gradually provided new protections annotations all through the year but partners were also able to test their protections independently. The annotations for the DCL protection for example has been provided very late in the project in August 2016 when the protection has been fully integrated in ACTC but the related macros in the use case had no impact on other partner tests and protection configurations.

Once the consortium thus obtained a clear view of the annotations and protections that would be used for the NAGRA, SFNT and GTO use cases, the whole consortium had a plenary meeting in year 2 where we discussed the coverage of the protection techniques as applied to all three use cases. This coverage was then again reviewed in M35 in the light of the availability and compatibility of protections at the end of the project.

4.2 Coverage

The result of this discussion is summarized in Table 2, which shows which protection techniques will be applied to which use cases.

- Green cells indicate that the annotations for this use case contain the protection and that we evaluated the deployment of the technique on the use case.
- Yellow cells indicate that the use case owners and the protection technique owners do not think that the specific protection can or should be applied to that use case. In other words, yellow indicates that a protection is not applicable because it does not relate to the type of assets and security requirements of the use case, or because other protections are considered a better alternative.
- Red cells indicate that, even though we think that the protection could be useful in a use case, practical reasons limit the applicability of that technique.
- Orange cells indicate that a technique developed by one partner cannot be deployed on a use case of another partner due to legal limitations. According to the Consortium Agreement, GTO simply does not have access to the code supporting the protections as delivered by NAGRA and SFNT. Similarly, the reaction mechanisms are not applicable to the NAGRA use case because access rights limitations. In some cases, the academics (who had access rights to all techniques and use case) could deploy the techniques themselves, but in other cases help would have been needed from the use case owner because of the complexity of deploying a technique on a piece of software as complex as their use case, and the legal arrangement prohibited providing such help.
- Brown cells indicate that the partner developing a technique delivered the automated tool support too late to the other partners to allow for a validation on a use case.
- Finally, grey cells mark techniques that were researched in the project, but that were found to be incompatible with the ACTC.

The fact that XOR masking, residue number coding and the merging of scalar variables have limited applicability is due to limitations in how these protection techniques are implemented and also due to the fact that the use cases do not contain patterns where the current versions of these obfuscation algorithms apply. A first limitation of these protections is that they work on intra-procedural data flow and they do not support inter-procedural data flow, i.e., data flow across function calls. Furthermore, the current implementation supports scalar types, but it does not support aggregated data types such as arrays. During the consortium meeting in Saint-Cyr-Sur-Mer, it became clear that only inter-procedural deployments of these obfuscations on aggregate data are useful in the NAGRA and SFNT use cases. FBK assesses the extra work needed to support both features of inter-procedural protection and of aggregate data structures is infeasible within the project duration and resources. Still, one algorithm available in the data obfuscation ACTC plug-in can be applied to, and validated on, all the industrial case studies, i.e., static-to-procedural conversion applies to all the case studies.

The tool support for dynamic remote attestation became mature only late in the project. Moreover, it requires advanced tracing capabilities that are not yet available for Android ARM platforms. The tracing capabilities are only available for x86 Linux at the moment. As the command-line version of the SFNT use case can be deployed on the x86 Linux platform, that version is used for validating dynamic remote attestation.

Finally, multi-threaded cryptography is marked in grey because, as documented in Deliverable D2.08, this protection proved to be not fit for automated deployment with the ACTC: The technique inherently targets online applications, and deploying the technique requires changes to the size of the client-server communication payloads, which in turn requires updates to the server-side software. Automatically transforming both the server-side and the client-side of an application is out of the ASPIRE scope, however, if feasible at all. See also the reporting thereon in deliverable D1.05.

The conclusion of this coverage analysis is that, with the exception of some data obfuscations and multi-threaded cryptography, all protections developed during the project are covered by the validation of the use cases.

Table 2 - Coverage of the protection techniques, including renewability techniques in the bottom rows

	NAGRA	SFNT	GTO
XOR masking	🟥	🟥	🟥
Merge scalar variables	🟥	🟥	🟥
Residue number coding	🟥	🟥	🟥
Static to procedural conversion	🟩	🟩	🟩
Multi-Threaded Cryptography	▨	▨	▨
Reaction Mechanisms	🟧	🟧	🟩
White-Box Cryptography	🟩	🟩	🟨
SoftVM (Client-side splitting)	🟩	🟩	🟨
Anti-Debugging	🟩	🟩	🟩
Call Stack Checks	🟩	🟩	🟩
Offline Code Guards	🟩	🟩	🟩
Binary Code obfuscations	🟩	🟩	🟩
Client-server Code Splitting	🟨	🟩	🟨
Code Mobility	🟩	🟩	🟩
Static Remote Attestation	🟩	🟩	🟨
Dynamic Remote Attestation	🟥	🟩 (x86 Linux only)	🟥
Control Flow Tagging	🟫	🟨	🟩
Anti-Cloning	🟩	🟧	🟧
Diversified Crypto	🟧	🟧	🟩
Native Code Diversification	🟩	🟩	🟩
Bytecode Diversification	🟩	🟩	🟨
Renewable White-box Cryptography	🟩	🟩	🟨
Diversified Mobile Code Blocks	🟩	🟩	🟩
Renewability Manager	🟩	🟩	🟨

Section 5 Use case requirements assessment

Section Authors:

Bjorn De Sutter, Bart Coppens (UGent), Werner Dondl (SFNT), Jerome d'Annoville (GTO), Brecht Wyseur (NAGRA)

The next part of our validation is to check whether the security requirements as expressed by the industrial partners on their use cases can in principle be met with the protections as designed, developed, implemented and planned to be implemented during the project.

In the consortium, this part of the validation is done by means of reports that

- summarize the annotations added to the source code;
- discuss discrepancies between the protections requested in the asset description documents (discussed in Section 4.2) and known features of the chosen protections.

For the same confidentiality reasons already discussed in Section 4.1, these reports are not shared with all industrial partners. They are hence not included in this deliverable either.[2] Instead they are only summarized in this document, to the point of allowing us to draw conclusions.

5.1 NAGRA use case

In the NAGRA use case, NAGRA specified security requirements on seven assets[3], and proposed potential protections to meet those requirements. The consortium assessed the availability of the necessary protections and their strength as evaluated in the tiger experiment as summarized in Table 3.

On top of the first layer of protections listed in Table 3, anti-debugging needs to deployed to protect all the assets directly, to protect the listed proposed types of protections, and to prevent alternative attack paths that could be used to circumvent the listed protections, such as control flow obfuscations, which only provide real protection against static attacks. Anti-debugging was available and applicable as required, and did prove to prevent dynamic attacks (tracing and live-debugging) in the tiger experiment.

From this assessment, we conclude that the envisioned protections for the NAGRA use case are available in the ASPIRE protection framework, with the exception of interprocedural data protections. Control flow tagging was delivered too late for deployment and validation on the use case, however. In Section 7, the actual level of protection provided by the actually deployed protections will be assessed.

[2] One such report will be made available to the project reviewers as an example, to allow them to assess the activities and progress in the project. But we again ask to take care not to disclose information from those documents to industrial partners that do not have the necessary access rights.

[3] The assets are identified and discussed in more detail in internal project documents. Because of their sensitive nature, they are only numbered here.

Table 3 - Availability proposed protections NAGRA use case

Asset	Security Requirements	Proposed Types of Protections	Validation Conclusion
1	• Confidentiality	• WBC (strongest) • Data obfuscation (weaker alternative)	The strongest proposed protections and their required features are available in the ACTC. The WBC protection technique has been applied, and has was not broken within the provided time-frame of the Tiger Team experiment.
2	• Integrity • Prevent Forgery	• Code flow integrity checks (strongest) • Data obfuscation (weaker alternative) • RAM-2-RAM encryption[4]	The strongest proposed protections are available. The lightweight form of anti-callback stack checks (D1.04 v2.1) might be too lightweight, however. The Tiger Team experiment is inconclusive in this regard; the experts were not obstructed by the callback stack checks technique in their executed attack path. The requested interprocedural data obfuscations are not available, as discussed in Section 4.2.
3	• Integrity	• Anti-tampering techniques • Data obfuscation	Most proposed protections are available, only the interprocedural data obfuscations are missing. During the Tiger Team experiments, no attack paths featuring tampering were executed.
4	• Confidentiality	• WBC (strongest) • Data obfuscation (weaker alternative)	Fixed-key WBC has been deployed to protect this asset. This asset has not been attacked during the Tiger Team experiment.
5	• Execution Correctness	• Control flow protections (obfuscations and anti-tampering) • Data obfuscations	Most proposed protections are available. Control Flow Tagging was not available in time for validation on the NAGRA use-case. The interprocedural data obfuscations are missing. During the Tiger Team experiments, no attack paths featuring control-flow tampering were executed. After observing the control flow obfuscation, the attackers changed their approach to use dynamic attacks.
6	• Integrity	• Code obfuscations	All proposed protections and their required features are now available in the

[4] In "RAM-2-RAM encryption", data is stored encrypted when it resides in memory, i.e., in between operations on it during the program's execution. That form of protection is not foreseen in the DoW, and resources are lacking to support it in addition to the techniques that are mentioned in the DoW.

	• Confidentiality	• Anti-tampering	ACTC. The CF Tagging technique, however, was not available in time for validation on the NAGRA use-case. After observing the control flow obfuscation, the attackers changed their approach to use dynamic attacks.
7	• Unique execution (no cloning)	• Anti-cloning techniques	All proposed protections and their required features are/will be available in the ACTC. The validation of this protection was out of scope of the Tiger Team experiment because this technique mitigates exploitation of the attack rather than breaking it (once).

5.2 SFNT use case

SFNT specified requirements on eight assets[5], and proposed protections to meet those requirements. The consortium assessed the availability of the necessary protections and their strength as evaluated in the tiger experiment as summarized in Table 4. For several assets, the overall flow of code and data through the main library functions, of which the bodies mainly consist of sequences of calls into internal procedures, needs to be protected. Control flow protections are available to do so, but interprocedural data obfuscations are not available, as discussed in Section 4.2.

From this assessment, we conclude that the envisioned protections for the SFNT use case are available in the ASPIRE protection framework, with the exception of interprocedural data protections. In Section 7, the actual protection provided by the actually deployed protections will be assessed.

Table 4 - Availability proposed protections SFNT use case and conclusions tiger experiment

Asset	**Security Requirements**	**Proposed Types of Protections**	**Validation Conclusion**
1:	• Confidentiality	• WBC	The proposed protection is available in ACTC. The WBC protection technique has been applied, and has was not broken within the provided time-frame of the Tiger Team experiment.
2:	• Confidentiality	• Code flow integrity checks (strongest) • Control flow obfuscation • Data obfuscation	All proposed protections are available, except for the potentially useful interprocedural data obfuscation. The tiger team was able to reconstruct the control flow, nevertheless they were not able to recover the asset in the given time.
3:	• Execution correctness • Confidentiality	• Code flow integrity • Anti-tampering techniques	All proposed protections are available, except for the potentially useful inter-

[5] The assets are identified and discussed in more detail in internal project documents. Because of their sensitive nature, they are only numbered here.

		• Data obfuscation • Remote attestation • Client-server code splitting	procedural data obfuscation. During the tiger experiments, no attack path was found in the given time.
4:	• Confidentiality	• Call stack protection • Code flattening • Data obfuscation	All proposed protections are available, except for the potentially useful inter-procedural data obfuscation.
5:	• Execution Correctness	• Control flow protections (obfuscations and integrity checks) • Data obfuscations • Remote attestation • Client-server code splitting	All proposed protections are available, except for the potentially useful inter-procedural data obfuscation. Despite the fact that the tiger team was able to recover the control flow, no way was found to take advantage and recover or tamper the logic.
6:	• Execution correctness • Confidentiality	• Code obfuscations • Anti-tampering • SoftVM • Remote attestation	All proposed protections and their required features are/will be available in the ACTC. During the Tiger Team experiments, it was possible to recover some SoftVM bytecode involved in the encryption. Nevertheless, the tiger team was not able to pursue that attack path further.
7:	• Execution correctness • Confidentiality	• Code flattening • Code guards • Remote attestation • Client-server code splitting	The proposed mechanisms are available. Despite the fact that the tiger team was able to recover the control flow, no way was found to take advantage and recover or tamper with the mechanism.
8:	• Execution correctness	• Code obfuscations • Anti-tampering • Remote attestation	The proposed protections are available in the ACTC. During the Tiger Team experiments, no attack paths featuring tampering were executed.

5.3 GTO use case

5.3.1 Use case specificities

A first preliminary remark is that the One-Time Password (OTP) application has been implemented for the purpose of the project with the intent to leave a wide attack surface to enable protections to be applied on the code. It is a significant application that provides a feature required for eBanking Authentication but some doors are left open in the specification that help the attacker. For example, the Personal Identification Number (PIN) is passed by the server to the application to enable the authentication control. A professional application would avoid this and would not store the correct PIN value in the client.

The logic of the application is that it first needs to set some data from the application server in a provisioning phase. After this step the application can run locally in its so-called generation phase, without any connection to the server, in order to generate as many passwords as required by the user. Online protections are not applicable in this generation phase because it adds an unrequired connection constraint that could prevent the user to get its password.

Using online protection would also add a performance penalty to the application. So the online protections can only be applied to the provisioning phase.

The consequence of these two remarks are that the OTP use case is difficult to fully protect and some doubts expressed in the Table 5 should be interpreted with this in mind.

About the relevance of this use case in the project, Eric Ahlm and Ant Allan wrote in "Market Trends: User Authentication Providers, Worldwide, 2015" that “User authentication is one of the older markets in information security, filled with large, longstanding providers; but it continues to expand with new and emerging providers. Gartner examines the trends driving this market expansion.” This quote is extracted from “Roundup of Identity and Access Management Research”, 3Q15, Published on December 4th, 2015 by Gartner. Consequence of this quotes of these Analysts is that User Authentication at large is a relevant use case.

In the same document there is a quote from the same author Ant Allan about the choice of a replacement for Incumbent One-Time Password Tokens: “Many enterprises are seeking replacements for incumbent OTP hardware token deployments for a variety of reasons. In all cases, they must understand their needs clearly, the suitability of alternative authentication methods, and the costs of and opportunities for switching vendors.” Consequence of this quote is that generating an OTP using a mobile application without hardware token is requested by the market. This is precisely what the use case is addressing and what has been shown during the third review of the project in January 2016 with an OTP generated by pure software after a provisioning step done from a remote server.

5.3.2 Use case protections

With this OTP use case, GTO specified security requirements on five assets, and proposed potential protections to meet those requirements. The consortium assessed the availability of the necessary protections and their strength as evaluated in the tiger experiment as summarized in Table 5.

From this assessment, we conclude that the envisioned protections for the GTO use case are available in the ASPIRE framework. Because of the nature of this use case, online protections cannot be applied (otherwise the standard conditions of use would not have been respected). In Section 7, the actual protection provided by the actually deployed protections will be assessed.

Table 5 - Availability proposed protections GTO use case

Assets	**Security Requirements**	**Proposed Types of Protections**	**Validation Conclusion**
1: PIN	• Code integrity	• Code Guards	The protections proposed are available, keeping in mind that a patient attacker would probably always be able to achieve its goal and bypass the authentication control. This asset was retrieved by the Tiger Team, first through brute force attack and confirmed with a side channel attack.
2: Master Key	• Confidentiality	• Diversified Crypto Library • Client-server code splitting • Data obfuscation	There are several options to protect this major asset. All proposed protections and their required features are available in the

		• Binary obfuscation	ACTC. Diversified Crypto has been preferred for the validation because it is the strongest protection. When Data obfuscation is used, the Master key is still exposed to an attacker lurking. The protection on this asset has not been broken during the Tiger Team experiment.
3: Device Key	• Confidentiality	• Client-server code splitting • Diversified Crypto	The proposed protection and their required features are available in the ACTC even if in practice it imposes a connection with the ASPIRE server that is troublesome for the user because of the induced time penalty. As an alternative, diversified crypto protection has been extended to protect application key. This asset has been retrieved by the Tiger Team.
4: Storage Key	• Confidentiality	• Client-server code splitting • Code obfuscation • Anti-debugging	All proposed protections and their required features are available in the ACTC even if it imposes a connection with the ASPIRE server. The same restriction applies as for the previous asset with regards to the online protection. It is not realistic due to the intrinsic offline nature of the OTP generation part of the use case. Code & Data obfuscation protection can be used here as well as a weaker alternative. The asset has been retrieved by the Tiger Team through side-channel attack. The Anti-debugging protection has not been broken but circumvented since a weak point was found in the application data storage mechanism.
5: Application code	• Integrity • Application correctness	• Code Guards • Call stack check • Control Flow Tagging combined with a reaction	All proposed protections and their required features are available in the ACTC. Control Flow Tagging might be used either offline and online. The offline form is used in practice. Call stack check and control flow tagging were not applied during the experiment. Tiger team did not mention having been hindered by integrity checkings.

Section 6 ACTC protection deployment assessment

Section Authors:
Bjorn De Sutter, Bart Coppens (UGent), Werner Dondl (SFNT), Jerome d'Annoville (GTO), Brecht Wyseur (NAGRA)

In this final section, we assess to what extent the already supported protections in the ACTC effectively are applicable to the use cases. We evaluate whether or not they are applied as intended, i.e., as requested in the source code annotations and given the current limitations of the protection tools integrated in the ACTC. In other words, we check whether the necessary source code transformations and binary code transformations are applied, and whether the produced application still executes correctly. We checked this for individual protections as well as for compositions of protections.

In this intermediate validation, we do not yet assess the run-time overhead or the level of protection achieved. The latter will be done once the tiger experiments (WP4, Task 4.4) have been run in the industrial partners.

The validation experiments have been conducted mostly by UGent: reproducing all protections outside the labs of the contributors of those protections increases the quality of the performed assessment.

The project partners that contributed protections already integrated in the ACTC intensively supported this effort by debugging their contributed tools to enable their deployment on the use cases. The industrial partners NAGRA and SFNT also assisted when changes to their use cases where needed to make some of the tools applicable.

With respect to the integrated protections techniques marked as integrated in the ACTC (bold) in Table 2, we note that the integration of remote attestation was not yet mature enough to be validate on applications as complex as the project use cases. For that reason, remote attestation has not yet been validated.

6.1 NAGRA use case

6.1.1 Methodology

The NAGRA use case consists of an Android media player and two libraries to be used by the Android DRM framework. Only the libraries need to be protected in ASPIRE. As a consequence, the NAGRA use case can only be validated on Android.

On this use case, we performed the following validation checks (besides validating the correct execution of the protected programs).

- For the source-level protections (static-to-procedural data obfuscation, WBC, anti-cloning, offline code guards), we checked that the source code was transformed as intended.
- For the binary-level protections (control flow obfuscations, call stack anti-callback checks, anti-debugging, SoftVM, code mobility, offline code guards), we checked that the code was actually transformed as intended.
- We checked that the SoftVM is actually deployed in the protected application by letting it produce log messages.
- We checked that the anti-cloning server detects the execution of multiple clones on clients.

- For remote attestation and code mobility, we verified that the server side is actually triggered by a connection initiated from the client.

6.1.2 Results

We can confirm that the media player, and hence the libraries, still work when deploying the combination of the following protections as specified in the use case annotations:

- Binary obfuscations, incl. CFG flattening, offline code guards, opaque predicates, and code layout randomization
- Fixed-key WBC and dynamic-key WBC
- Static-to-procedural data obfuscation
- Anti-callback stack checks
- Offline code guards
- Code mobility
- Anti-debugging
- SoftVM
- Static Remote Attestation
- Anti-Cloning

On top, the following forms of renewability have been deployed successfully:

- Renewable WBC
- Native code diversification
- SoftVM bytecode diversification
- Mobile code block diversification
- Renewability Manager (managing diversified mobile code blocks)

The deployment of these techniques on the use case can therefore be considered a success.

There is one caveat, however. In the original use case implementation, the two plug-ins (DRM and crypto) had to be loaded into the same mediaserver process. As the anti-debugging protection only operates correctly when only one protected component is loaded into a process, the use case design had to be altered slightly. Rather than linking the crypto-plug-in dynamically to the DRM-plug-in, it is linked statically.

6.1.3 Conclusions

We can conclude that all the protections supported in the ACTC and listed as potentially useful in Section 5.1 can be deployed on the NAGRA use case, given the slight redesign.

6.2 SFNT use case

6.2.1 Methodology

Two versions of the use case exist. First, we have a command-line riddle application that is dynamically linked to the libdiamante.so shared library that is to be protected by the ACTC. This version can be compiled and evaluated on both Android and Linux. Secondly, there is a Java (Dalvik) riddle app that includes the libdiamante.so library. In the latter case, the library also includes some JNI wrapping functionality, but apart from that, the libraries to be protected are the same in both versions. Obviously the Dalvik version only runs on Android.

On this use case, we performed the following validation checks (besides validating the correct execution of the protected programs).

- For the source-level protections (static-to-procedural data obfuscation, WBC, offline code guards, and client-server code splitting), we checked that the source code was transformed as intended.

- For the binary-level protections (control flow obfuscations, anti-callback stack checks, anti-debugging, mobile code, and SoftVM), we checked that the code was actually transformed as intended.
- For the online client-server code splitting and code mobility protections, we also checked that the server side is actually triggered by a connection initiated from the client, and that the application only works when the server is responding correctly.
- We checked that the SoftVM is actually deployed in the protected application by letting it produce log messages.
- For remote attestation, we verified that the server side is actually triggered by a connection initiated from the client.
- On the command-line version of the use case, we checked that the anti-debugging is actually providing protection in two ways: by checking that the protected application fails to execute correctly when it is launched from within a debugger that attaches to the application process, and by checking that it is not possible to attach a debugger to the application process once it is launched outside a debugger. As exactly the same functionality is protected in native library embedded in the Dalvik app, we can conclude that the functionality is also protected in that version of the use case.

6.2.2 Results

When we applied protections individually on the Linux and Android versions of the use case, all tests succeeded.

Also we applied combinations of protections on the Linux and Android, all tests succeeded. So we can now protect this use case with the following combination of protections:

- Static-to-procedural data protection
- Fixed-key WBC
- SoftVM
- Anti-callback stack checks
- Binary obfuscations, incl. CFG flattening, offline code guards, opaque predicates, and code layout randomization
- Offline code guards
- Client-server code splitting
- Code mobility
- Anti-debugging
- Static remote attestation

On top, the following forms of renewability have been deployed successfully:

- Renewable WBC
- Native code diversification
- SoftVM bytecode diversification
- Mobile code block diversification
- Renewability Manager (managing diversified mobile code blocks)

As for the composability of the protections (see also D1.04 Reference Architecture v2.0 Section 5), the correct deployment of the above combinations shows that they are composable at the application level. With respect to composability at the source code fragment level, we have observed only one issue: due to a bug in Diablo (the tool used for the binary code rewriting), fragment-wise composition of two forms of code obfuscations (code flattening and branch functions) with code mobility and with anti-debugging had to be disabled for the time being. The composition of these protection techniques has been tested on both the Linux version and on the Android version. Compared to the intermediate validation reported in D1.05 v1.2, client-server code splitting now also works in combination with the other online techniques.

6.2.3 Conclusion

We can conclude that all the protections supported in the ACTC and listed as potentially useful in Section 5.2 can be deployed on the SFNT use case.

6.3 GTO use case

6.3.1 Methodology

The GTO use case consists of an Android Java application for one-time password (OTP) generation. It contains a native shared library that needs to be protected in ASPIRE. As a consequence, the GTO use case can only be validated on Android.

The use case was available for partners to test their protections since early January 2016. Scripts are available to configure and compile the application. The way to compile the code is documented and is available on the server together with the use case source code. At least three partners have used the environment so far:

- UGent helped to test the code mobility on the use case because they already have the adequate server environment.
- FBK used it to test the behaviour of different data types in Data Obfuscation.
- POLITO compiled and analysed the GTO use case with the ADSS to perform the validation of the effectiveness of the decision support on this use case. POLITO identified the assets, collaborated with GTO to determine their importance, identified all the attack paths against the assets, and determined the golden combination and the Layer 2 Protections.

There are some intrinsic difficulties with this use case that may have prevented partners to use and evaluate the OTP use case. A minor difficulty is that it needs an application server in addition to the ASPIRE portal. To minimize the work to be done a server has been deployed over the Internet. For partners using a board or an Android device with local WiFi connection without Internet access the source code and the .war of the http server have been delivered since year 1 on the project's svn server.

The other difficulty is that the use case has been designed to be run on a device with some user interaction that is not convenient in the daily life where automatic tests are preferred.

To perform the actual validation, a specific test has been derived from the use case.

The motivation is that the password generated by the application is inherently different for each invocations of the application and there is no way to check the correctness of the password since using back-office infrastructure such as a database and authentication would have significantly impact in terms of effort and would have made the test scenario more complex to partners.

The specific test reproduces the same condition for the generation of a specific password value by capturing all properties of the application in initialized values taken as input of the derived application. The data taken from the property file are dropped and the current context is taken from initialization values inserted by hand in the code. With these initializations the specific test always generates the same password value otherwise the logic of the code has been unexpectedly tampered, which means that a protection applied to the code has abusively changed the logic.

Another property of this specific test is that it skips the user interaction part of the application implemented in Java in the full use case and can be run directly on an Android device. Note that in this latter test condition the device needs to be rooted to run a pure C application.

On this use case, we performed the validation checks of the protections listed below. In most cases the validation has been done by checking the correct execution of the application. It is briefly described when specific actions have been done in addition.

- Static-to-procedural data protection
- Anti-callback stack checks
- Binary obfuscations
- Offline code guards
- Diversified Crypto: verified that the master key is no more exposed in binary code
- Code Mobility
- Anti-debugging: verified that the password is not generated correctly if the attached process running the debugger is kill by hand
- Control Flow tagging
- Software Time bomb: verified that a reaction actually occurs when portion of the code controlled by Control Flow Tagging protection has been hacked.

To make the data obfuscation tool work on the use case, its source code had to be rewritten slightly, e.g., to remove a cast that the tools could not handle correctly.

6.3.2 Results

We can confirm that the OTP generator still works correctly either when running the full use case on an Android device or by running the specific test as described in Section 6.3.1 when the following combinations of protections is deployed:

- Data obfuscation (static-to-procedural)
- Binary code obfuscation
- Anti-callback stack checks
- Offline code guards
- Diversified Crypto Library (DCL)
- Code mobility
- Anti-debugging
- Offline Control Flow Tagging combined with Offline Reaction
- Software Time Bombs

Code mobility and online control flow tagging have also been applied but separately even if combined with most of the offline protections listed above. As already mentioned, applying online techniques to the GTO use case is very artificial and even if it makes sense for the sake of unitary testing it is far from a realistic condition of use and cannot be kept in the golden combination of protections for this OTP use case.

6.3.3 Conclusions

We can conclude that all the protections supported in the ACTC and listed as potentially useful in Section 5.3.2 can be deployed on the GTO use case.

Section 7 Security Assessment

Section authors:
Bjorn De Sutter, Bart Coppens (UGent), Brecht Wyseur (NAGRA), Jerome d'Annoville (GTO), Mariano Ceccato (FBK), Cataldo Basile, Alessio Viticchié (POLITO)

In this section, we assess the actual protection effectiveness of the protections that were deployed on the use cases. The inputs for this assessment come from four sources:

1. the tiger experiments conducted by the three industrial partners, of which the detailed designs and results are also reported in deliverables of WP4;
2. the academic experiments conducted by the four academic partners, of which the detailed designs and results are also reported in deliverables of WP4;
3. the project's public challenge, of which the design and results are also reported in deliverables of WP4;
4. internal assessments conducted within the participating companies (but outside the tiger experiments and hence outside the project);
5. input gathered from brainstorms involving the project's principal investigators, its advisory board members, and third parties with which concrete protections were discussed, e.g., during the two SPRO workshops organized by the project consortium.

For each protection, we discuss its impact on the attack effort. We do so for each protection on isolation and, where applicable, for synergistic combinations of protections. The discussions also include lessons learned, e.g., to improve the deployment of the protections.

7.1 White-Box Cryptography

In all experiments involving white-box cryptography, a practical form of the developed schemes was used, i.e., a form that is (provably) not completely secure but that comes with acceptable overhead.

In the DRM tiger experiment at NAGRA, the code implementing the white-box cryptography was identified correctly in the collected traces. The attackers observed that the code did not reveal keys right away, but lacked time to attack the identified implementation. So the protection proved effective in hiding the embedded keys for at least the duration of the experiment.

In the software licensing tiger experiment at SFNT, the crypto code was identified using static attacks, but the embedded keys were not extracted. Due to a lack of additional time, no conclusive attack on the identified code was executed. So the protection proved effective in hiding the embedded keys for at least the duration of the experiment.

In the OTP tiger experiment at GTO, white-box cryptography was not evaluated for Intellectual Property protection reasons.

In the academic experiments, white-box cryptography was not evaluated.

In the public challenge, one attacker succeeded in inverting the performed encryption using brute-force techniques to solve the one challenge (out of eight challenges) that focused on white-box cryptography. This exploitation of a known weakness [Wys13] of the used white-box encryption scheme points to the challenge being designed suboptimally. The attacker was not able (and did not try) to extract the key that was protected by the use of the white-box scheme. In that regard, the white-box cryptography proved to work.

Despite the use of practical forms of WBC, which are not supposed to provide full, indefinite protection, they have proven to withstand at least the limited-duration attacks conducted in the project.

7.2 Diversified Cryptography

For the purpose of the Tiger Team experiment, the security assessment of this protection has been done by an external team that is specialized in this type of analysis.

The initial goal of the protection is to hide a Master key and the process of derivation and indeed this Master Key has not been extracted during the experiment. Note that there were constraints set for this assessment and that with more time and resource the Tiger Team would most probably succeed in the attack. A sub-goal is to protect the derived key and the cryptographic operations on this derived key and with this regard the assessment has pointed out some significant weaknesses in the implementation of the protection.

There is some flexibility with this protection that enables to embed some application-oriented code. This code lies encrypted in the library and the assessment shows that the decryption of this code before being executed is breakable.

We can conclude that this protection is good enough to protect a master key for non-critical application but that the protection of the diversified keys and the code that uses these keys needs to be aligned on more solid protections.

7.3 Data Obfuscations

In the tiger experiments, the few deployed data obfuscations were not observed by the penetration testers. This is mostly due to the obfuscations not being deployed on the assets under attack in the experiments.

In the academic experiments, we observed that the adoption of data obfuscation makes the source code more difficult to understand and, even if attacks might remain still possible, they require more effort and more time. Data Obfuscation caused a dramatic impact on success rate and on the time necessary to complete a successful attack. The odds of a successful attack to code obfuscated with data obfuscation decreased on average by a factor in the interval of [3.3-4]. The time required to port a successful attack increases by a factor in the interval [2-3.5].

In the public challenge, two challenges were protected with data obfuscations (once with and once without anti-debugging). In the version without anti-debugging, a successful attack was able to reverse-engineer the code, and hence comprehend even the protected code, using a debugger to study the code in action. In the version with anti-debugging protection, he studied the code statically to uncover the keys, reusing his knowledge (comprehension of the code) obtained from the dynamic attack on the version without anti-debugging.

We can conclude that data obfuscations effectively delay attackers, but also that in case dynamic attacks (debugging or trace collection) are possible, those obfuscations' protection is weakened.

7.4 Client-Side Code Splitting (SoftVM)

In the DRM tiger experiment at NAGRA, the SoftVM and the bytecode were not attacked, as the deployed dynamic techniques were able to locate the relevant crypto code in traces without having to analyse or comprehend the trace parts corresponding to bytecode interpretation.

In the software licensing tiger experiment at SFNT, the tiger team discovered the use of plain LLVM bitcode as the bytecode format. They were able to do so because the linked-in LLVM-based interpreter included enough strings to infer its origin. It then required very little effort to disassemble the bytecode using existing LLVM tools. This result was expected by the principal investigators of the project consortium. The solution is to use renewable (i.e., diversified) bytecode formats, for which the necessary support was contributed to the project by SFNT. It was contributed too late for inclusion in the tiger team experiments, however.

In the academic experiments, client-side code splitting was not deployed or evaluated.

In the public challenge, an attacker also discovered that LLVM bitcode was present. He studied the LLVM source code and discovered that in a debug-enabled build, a variable controls the logging of interpreted operations. He then forced the logging of operations, and reverse engineered the logged LLVM bitcodes.

We can conclude from our experiments that the use of a SoftVM significantly delayed attackers. However, because of the discussed shortcomings, overcoming the protection was still relatively easy. In order to make this protection effective, it is necessary to use a custom bytecode format and to ensure that the interpreter contains no logging or debugging functionality (incl. any strings that might allow an attacker to identify the used format or interpreter). Reusing publicly available virtual machines can hence never be considered safe.

7.5 Control Flow Obfuscations

In the DRM tiger experiment at NAGRA, the presence of control flow obfuscations and the experience of the pen testers pushed the attackers towards dynamic attack paths. The tiger team considered the presence of control flow obfuscations a burden for static analysis. As dynamic attack paths could be blocked with the anti-debugging protection, control flow obfuscations could be considered effective in the sense that they certainly made attacks harder.

In the software licensing tiger experiment at SFNT, the pen testers wrote scripts to undo the opaque predicates and branch functions. The ability to do so resulted from the simplicity and inflexible implementation of the used forms of those protections. This is not surprising, as control flow obfuscations were included in the project primarily to study how to compose those techniques (which produce very irregular control flow graphs) with other protections being deployed, not for pushing the state of the art or even for reproducing the state of the art. Code flattening was not circumvented. While it made the control flow look more complex, the tiger team considered it as not hampering code comprehension too much.

After undoing the obfuscations, the resulting code was not executable (for use in dynamic attacks). This was due to incomplete implementation, however, not due to anti-tampering detecting the tampering.

In the OTP tiger experiment at GTO, the cyclomatic number of the portion of the code protected by obfuscation is very low and hence this protection is not fully effective. Still, the Tiger Team switched to dynamic analysis because static analysis was difficult mainly because of the binary obfuscation.

In the academic experiments, control flow obfuscations were not studied.

In the public challenge, all eight challenges were obfuscated with opaque predicates, control flow flattening, and branch functions. Also several of the linked-in components implementing other protections were obfuscated. An attacker used pattern matching to undo the branch functions insertion as well as to remove opaque predicates. He conceded that building the custom tools to undo the obfuscations were the most time-consuming part of his attacks.

We can conclude that the control flow obfuscations hamper and delay attackers to some extent, and that they push attackers of the easiest paths of static attacks towards the use of more advanced dynamic attack techniques. The latter has if fact always been the prime goal of control flow obfuscations. We also have to conclude, however, that at least the deployed forms of control flow obfuscation did not prove to be unbreakable obstacles, even when only static attacks are used. To prevent pattern-based attacks and attacks based on symbolic execution, the deployment of the tech-

niques needs to be moved to a higher level of complexity. Several ways can be imagined to achieve this, i.e., to hide code patterns in harder to analyse, global control flow.

7.6 Multithreaded Crypto

Research in the second year revealed that this protection could not be deployed automatically by the ACTC. Beyond the intermediate validation reported in deliverable D1.05, its protection strength has hence not been validated on the use cases, in the tiger experiments or in the public challenge.

7.7 Anti-Debugging

In the DRM tiger experiment at NAGRA, we learned that a lack of anti-debugging protection allowed the pen testers to easily collect execution traces in which the relevant cryptographic code under attack could be identified. When our anti-debugging protection was deployed, no traces of API calls coming from the Android DRM/media servers could be collected. Thus, the anti-debugging protection proved highly effective. One can expect that with considerable effort tools can be engineered to collect such traces even in the presence of the anti-debugging, but at least with existing tools it was not possible.

In the software licensing tiger experiment at SFNT, the tiger team was able to debug API calls to the protected library out of context, despite the available protection. The attackers relied on mistakenly unremoved symbol information to identify the initialization routines of the anti-debugging technique and to alter the software to skip their execution. This did not allow them to collect complete traces, as the application is then non-functional, but they could then execute API calls out of context to study the execution of limited parts of the protected software.

Two lessons can be learned from this: First, the anti-debugging technique should be actively used in all relevant code (i.e., all code of which the tracing or debugging can help attackers), not just in the surrounding code or in the code that is guaranteed to be executed before the relevant code fragments in the normal execution of a program. Secondly, the anti-tampering techniques should be extended to detect tampering with the initialization code of other protection techniques, such as anti-debugging, to prevent that attackers can partially circumvent those protections, as was done for anti-debugging.

In the OTP tiger experiment at GTO, the tiger team investigated a use case version protected with anti-debugging. They noticed the ptrace-based nature of the protection, and acknowledged that the protection was sophisticated and strong. With their full system emulator, they would believe they would have been able to collect traces, but live debugging a single process with such an emulator would have been cumbersome and certainly require more time and effort than live debugging a single process.

In the academic experiments, anti-debugging was not studied.

In the public challenge, which involved very small binaries, an attacker was able to reconstruct the original control flow (i.e., undo the redirections through the self-debugger) and data flow after manually analysing the functionality of the transformed code with IDA Pro and the HexRays decompiler. He then omitted the self-debugger initialization routines, and was again able to attach a debugger. The manual effort was doable because the attacker could easily spot the redirections because all of them were implemented through simple breakpoint instructions.

In the end, we can conclude that the anti-debugging technique proved to be quite effective in preventing live debugging and the collection of whole traces under normal execution circumstances. Attack paths on software protected with this technique proved to be significantly delayed (i.e., when the protection was undone manually to

enable live debugging) or blocked (i.e., when the attackers were pushed to other attack paths). To prevent out-of-context debugging and trace collection, a more complete combination with anti-tampering techniques needs to be developed. To make the undoing of the control flow redirections consume more effort, two reinforcements are necessary: (i) using more stealthy control flow redirection mechanisms (i.e., exception inducing instructions), and (ii) use more complex schemes to encode the link between those exception inducing instructions and the continuation points in the self-debugger. Of course, such undoing of anti-debugging could also be hardened by ensure that tampered anti-debugger code is detected with anti-tampering techniques.

7.8 Offline Code Guards

In the DRM tiger experiments at NAGRA and the software licensing tiger experiment at SFNT, offline code guards were deployed, but did not hamper the attackers, either because they did not try to tamper or because the tampered code (initialization routines of anti-debugging) was not protected with the code guards.

A lesson to be learned is that this protection technique clearly needs to be able to cover those initialization routines. With the current tool flow design, where only code can be guarded, and then only code in the .text section of the protected binary or library, this is not yet feasible.

During the OTP tiger experiment at GTO this protection has been circumvent by using a custom operating system that enable to collect run time traces and the team did not report a specific delay in their attack strategy due to the code guards protection. This analysis should be mitigated by the fact that if the Tiger Team successfully retrieved most assets in a lab environment they have not published an attack to be deployed on the field where the Guards protection might be much more effective thanks to the integrity checks done on the code.

In the academic experiments, offline code guards were not studied.

In the public challenge, all eight challenges were protected with offline code guards. When an attacker tried debugging some challenges by means of breakpoints, the guards hindered him as software breakpoints were detected by the guards. He was able to work around them in two ways. First, he identified the location in the program where the hashes were computed, and then made sure to delay in setting of software breakpoints until after those computations. This way to use breakpoints was cumbersome, however, and made him loose considerable time. So secondly, he spent time reverse-engineering the rather simple hashing function used in the experiment such that he could more freely deploy software breakpoints again.

Offline code guards proved to be useful to delay certain tampering attacks. We can therefore conclude that offline code guards can be useful, but that they have to be applied carefully. Some lessons are (i) to use complex, and multiple hashing functions that cannot easily be reverse-engineered, (ii) let guards guard themselves and each other (this is actually conventional wisdom). Probably the level of protection can be improved by randomizing which guards are executed first, such that the few hardware breakpoints at an attacker's disposal cannot cover all of them.

7.9 Anti-Callback Stack Checks

In the academic experiments, anti-callback stack checks were not studied.

In the public challenge, all challenges were protected with anti-callback checks, but no out-of-context executions were tried by attackers that reported their attack method.

7.10 Control Flow Tagging

This protection has been provided on M34 but needed to be integrated with software time bombs to be effective. Final integration in ACTC has only been delivered in M35, so no assessment was possible in the various tiger experiments and in the public challenge.

From internal analysis one remark regarding the security assessment is that this protection is a good candidate for the layer 2 protection provided by ADSS to blur the static analysis detection of the protection. Reliability of this protection fully relies on the effectiveness of the reaction mechanism whose assessment in done in Section 7.15.

7.11 Code & Data Mobility

In the academic experiments, no code or data mobility was evaluated.

In the tiger experiments, only the code mobility version was evaluated in which downloaded mobile blocks are never flushed. One team noticed that not all code was present in the static binary. When no anti-debugging protection was present, they could still easily disassemble the mobile code, however, by letting IDA Pro disassemble the code present in a core dump. IDA Pro did not automatically detect the mobile code as a separate function, however, so it can be assumed that attacks requiring the collusion (and hence diffing, e.g., with the BinDiff plug-in on top of IDA Pro) of core dumps would be hampered.

In the public challenge, the one attacker that succeeded in attacking many versions with offline protections, did not succeed in running a version with the code mobility protection, let alone attack it.

We can conclude that code mobility delays static attacks by preventing straightforward uses of disassemblers. In case live debugging is possible, code mobility provides little protection by itself, however. Combining it with anti-debugging makes the protection much stronger, as does the flushing of downloaded code, because it then requires the attacker to take dumps at the right time, not just late in the program execution.

7.12 Client-Server Code Splitting

In the academic experiment with we observe that code splitting we observed that the adoption of client-server code splitting was effective in making the source code more difficult to understand and, when attacks are still successful, they require more time. The adoption of this protection, caused an increase in the time by a factor 1.4.

In the SFNT tiger experiment, the attackers observed that the application needed to connect to a server to execute correctly. The protection hence delayed the attackers as they were setting up their attack environment. During the actual attack, they focused on other assets, and were hence able to neglect the split code.

The academic experiments provide quantitative evidence for the shift in the economic convenience of code attacks when attackers deal with split code.

7.13 Remote Attestation

7.13.1 Static RA

No RA was evaluated in academic experiments.

In the tiger experiments, we observed that tracing tools such as Valgrind, which inherently rewrite the code to instrument it, do not suffer from anti-tampering techniques, because read accesses are still performed on the original code sections, not on the rewritten sections. We also observed that static code attestation does not suffice to detect tampering in .init sections, and hence does not prevent tampering with the initialization routines of other protections (in casu anti-debugging). The reason is that such tampering can be implemented simply by overwriting code pointers, rather than code.

In the public challenge, the one attacker that succeeded in attacking multiple challenge did not attack versions protected with online protections, so he did not attack RA.

We can conclude that static RA, while potentially delaying certain tampering attacks, does not provide protection against other forms of tampering. To make the protection stronger, it needs to be extended to at least cover the initialization routines and data of binaries/libraries.

7.13.2 Dynamic RA

Dynamic RA was delivered too late in the project to be included in controlled academic experiments, in the tiger experiments or in the public challenge.

During other evaluations and the development of the technique, the technique could be used to protect software in which a clear link between program invariants and attack methods/targets could be identified. In general, this identification is anything but simple.

7.14 Anti-Cloning

Anti-cloning has not been evaluated in the academic experiments, in the tiger experiments, or in the public challenge, because it was unrelated to the assets and security requirements on which those experiments focused. Moreover, anti-cloning is a protection that aims at preventing wide-scale attack exploitation, rather than attack vector identification. The experiments in the project all focused on attack vector identification.

Internally, and outside the ASPIRE project, in a validation setup that differed significantly from the typical ASPIRE setup because of the aforementioned difference in aims, NAGRA did validate the effectiveness of the contributed anti-cloning protection.

The conclusion was that the anti-cloning technique is highly effective. In theory, and with massive effort (and hence at high cost), the technique might be circumvented by highly motivated, expert attackers. But even under such attacks, anti-tampering techniques can help in making the anti-cloning technique sufficiently robust, such that the attacker's cost is even further increased.

7.15 Reaction Mechanism - Software Time Bombs

This protection has not been evaluated during the experiments and the challenge due to its late availability.

An internal analysis highlighted that this protection relies on calls to a centralized function that would benefit from obfuscation techniques to prevent the attacker to connect the detection and the reaction parts.

A first recommendation is to inline the call to the function to avoid static detection. Another way to reinforce the protection is to split the various reaction techniques used in the reaction function into small functions that can be either inlined in turn or replicated as small functions many times in the application with variations of the code. These obfuscations shall be combined with a large variety of delay data structures to make them stronger.

7.16 Renewability Techniques

No renewability techniques were evaluated in the academic experiment, in the tiger experiments, or in the public challenge, for two main reasons:

1) The renewability techniques matured late in the project in WP3 (as foreseen in the DoW) and were hence not ready to be evaluated in the mentioned experiments.
2) The renewability techniques, like anti-cloning, aim at mitigating wide-scale deployment of attacks rather than attack vector identification. All security assessments conducted within the project focused on the latter however.

Still, some relevant security observations can be made for several renewability techniques.

- With respect to native code diversification, UGent has already demonstrated its impact on static and dynamic collusion attacks [Cop13a,Cop13b]. In this project, the diversification techniques were ported to the ARM Android platform and integrated and composed with the other protections. The security assessment in the referenced publications hence still stands.
- In deliverable D2.01, a security assessment was presented for the white-box cryptography. This included a theoretical evaluation of the cost of brute-forcing WBC solutions. In scenarios such as live event video streaming, WBC keys can be renewed at a chosen frequency. Furthermore, in such scenarios the transmission delay can be estimated that viewers of stolen streams can tolerate. From the key renewal frequency, the estimated "tolerable" delay, and the theoretic brute-force costs, the required hardware investment can be estimated, thus resulting in a quantifiable protection level.
- The latest version of the client-side code splitting technique (i.e., SoftVM) contributed to the protect by SFNT supports bytecode and SoftVM diversification. Moreover, the used SoftVM is a custom one, not a reused open source VM or interpreter. This ensures that attackers have to relearn the bytecode format for every software instance they would attack.

The consortium is hence confident enough to conjecture that the developed and integrated renewability techniques deliver improved protection, as they make the scaling up of attacks harder, as they can delay attack vector identification, and as they can help in raising the costs faced by attackers.

7.17 Conclusions

Overall, the ASPIRE consortium concludes that, although many of developed and integrated protections still offer a large potential for improvement, those protections

- effectively delay attacks and increase the effort that attackers need to invest in identifying attack vectors;
- make it harder to exploit identified attacks at a large scale;
- and hence effectively reduce the profitability of engineered attacks.

To a large degree, the project has hence achieved its goals to demonstrate that software-based protection techniques can deliver true protection.

Section 8 Overhead Assessment

Section authors:
Bjorn De Sutter, Bart Coppens, Jens Van den Broeck, Bert Abrath (UGent), Brecht Wyseur (NAGRA), Jerome d'Annoville (GTO), Mariano Ceccato, Roberto Tiella (FBK), Alessandro Cabutto (UEL), Cataldo Basile, Alessio Viticchié (POLITO)

In this section, we discuss and quantify run-time overheads (in the form of execution slowdowns, increased memory usage, and communication bandwidth) for the protections developed and integrated in the project.

For each of the protection techniques discussed here, the contributing partners performed their own experiments, using their own methodology.

8.1 Code Mobility

The content of this section was originally published in D3.04 Section 3.3

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 6); for latency measures we run the code only 100 times. The last column of Table 6 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 6 - 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 7 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 7 - 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.

8.2 Static Remote Attestation

Our assessment has been performed on a case study written in C language, namely Bzip2 compiled in the CPU SPEC version. The test environment is the following.

- Server-side: Linux based machine with i7-3610QM @ 2.30GHz and 6GBytes RAM
- Client-side: Nitrogen6_MAX i.MX6 board with ARM-Cortex A9 @ 1GHz and 4GByte DDR RAM.

We selected three code regions associated to the whole code of three functions. They have been annotated and then protected with remote attestation. The resulting sizes of the protected regions in the final binary are reported hereafter:

Function	**Bytes**
BZ2_bzCompressInit	883
BZ2_bzCompress	560
BZ2_bzDecompress	605

The three regions have been split into three block each by Diablo. The tests have been performed by make the Bzip2 application compress a file with size 1.8GB in order to make the execution long enough to observe remote attestations.

The manager is a 795 LOC application written in C that uses a 631 LOC DB access library that wraps the MySql Connector/C library.

The verifier is a 190 LOC application written in C that uses a 631 LOC DB access library that wraps the MySql Connector/C library. In addition, it includes a combination of RA fundamental blocks that depends on the protection configuration. Then, the actual total length of the verifier code is variable.

The attestator is a 308 LOC application written in C that also includes a combination of RA fundamental blocks that depends on the protection configuration.

8.2.1 Time evaluations

We report here the execution times and overhead we measured associated to each component of the remote attestation infrastructure. The results reported in this section have been obtained by observing a sample of 200 remote attestation request-response sessions.

Manager. We observed that the time needed by the manager to generate and send an attestation request is 46μs with standard deviation of $6.1 \cdot 10^{-5}$. It includes the time to access the ASPIRE database and retrieve a valid nonce, the time to create the message, to send it, and the time to actually send the message to the attestator (i.e., invoking system call). The measured time is independent of any parameter that may be used to configure the technique.

Verifier. The average duration of an attestation verification has been measured as 282μs with standard deviation equal to $4.7 \cdot 10^{-5}$s. It includes the time to access the ASPIRE database, read the prepared data and compute the hash on the prepared data. The measured time has been evaluated as independent of any parameter associated to the technique.

Attestator. The average duration of an attestation on the client side has been measured as 10ms with standard deviation of $2.5 \cdot 10^{-3}$s. It includes the time to access the Attestation Data Structure, read the prepared data and compute the hash on the prepared data. The measured time is linearly dependent of the number of blocks and the overall size of the attested area.

8.2.2 Technique overheads

In this section we report the measured overheads introduced by the remote attestation. The results reported in this section have been collected on a sample of 50 application executions.

8.2.2.1 Time overhead

We measured the time needed to execute the unprotected version of the application (the aforementioned Bzip2) and then we compared the result with the protected one. Collected execution times are in Table 8.

Table 8 - Measured time overhead.

Version	**Average execution time**	**Standard deviation**	**Attestation performed**
Unprotected	1769s (29min and 29s)	10.57s	N/A
Protected	1797s (30min and 53s)	9.27s	177

Then we observed that the average time overhead due to the protection technique is about 1.56%.

8.2.2.2 Application size overhead

We did not empirically evaluate the application's binary size overhead because it can be precisely computed theoretically.

We know that the technique generates an object file (linked with the application ones) that includes the object files related to each declared attestator. The size of a single attestator object file is 18kB. Then the size (in bytes) of the final object file is given by:

$$S_{At} = 18|A| \text{ kB}$$

where $|A|$ is the number of the attestators that need to be used to protect the application.

The technique produces also a binary data structure, the ADS, for each attestator that will be injected in the final application. The size (in bytes) of the ADS associated to the generic i-th attestator depends on how many code areas are monitored by the attestator and how many blocks stand in each area according to the following formula:

$$S_{ASD}^{i} = 28 + 6|R_i| + 12 \sum_{j=0}^{|R_i|} |B_{ij}|$$

where $|R_i|$ is the number of code areas recorded in the ADS (to be monitored by the attestator), $|B_{ij}|$ is the number of memory blocks reported in the ADS for the *j*-th code region of the i-th attestator.

Then the total size (in bytes) of all the ADSs is given by:

$$S_D = \sum_{i=0}^{|A|} S_{ADS}^{i}$$

In the end, the application final binary size is enlarged by a quantity of bytes given by:

$$E = S_{At} + S_D$$

8.2.3 Application memory overhead

The technique allocates a set of runtime data structures to keep track of the code areas to attest. This is actually the only source of static memory overhead. As we did for the application size, we did not empirically assess the memory overhead since it can also be computed theoretically. The memory consumption related to the technique directly depends on the number of monitored code areas and on how many blocks compose each area.

The runtime static memory usage (in bytes) related to the technique is given by the following formula:

$$M_s = \sum_{i=0}^{|A|} \left[62 + 42|R_i| + 12 \sum_{j=0}^{|Ri|} |B_{ij}| \right]$$

Where:

- $|A|$ is the total number of attestators injected in the application;
- $|R_i|$ is the total number of code areas monitored by the i-th attestator;
- $|B_{ij}|$ is the total number of code blocks inside the *j*-th code region of the *i*-th attestator.

Moreover, we have to consider the dynamic memory overhead, due to the temporary data needed during the computation of the attestations, the received nonces and the prepared data. Also this value can be computed without an empirical evaluation. In fact, the temporary memory usage to perform an attestation can be statistically estimated using the following formula:

$$M_d = \frac{\overline{|R|}}{\bar{p}}|A|$$

Where:

- $\overline{|R|}$ is the average attested code areas size (in byte) evaluated over the all the areas attested by all the attestators;
- $\bar{p}$ is the average interval between subsequent attestations (in seconds) evaluated over all the attestators;
- $|A|$ is the number of attestators in the protected application.

8.3 Data obfuscations

The assessment of the overhead introduced by state-of-the-art data hiding techniques was presented in Deliverable "D2.01 Early White-Box Cryptography and Data Obfuscation Report". In this section, we report about overhead possibly introduced by advanced techniques developed in the Year 2. These techniques use opaque constants to hide obfuscation parameters and are extensively discussed in Deliverable D2.08 ASPIRE Offline Code Protection Report.

The rest of this section is structured in two subsections that describe (a) a set of requirements techniques based on opaque constants must satisfy, and (b) an empirical evaluation to assess if those requirements are met.

8.3.1 Opaque Constants Overhead Requirements

In techniques based on opaque constants, additional code is inserted to a program to compute those constants at execution time. When this additional code is complex, the obfuscated code might suffer sensible runtime overhead and performance degradation. Despite different execution contexts might pose different constraints to execution time, obfuscation should be in general lightweight and it should not impact too much the program execution speed. For example, when needed, an opaque value should be computed in polynomial time. Thus, the first requirement is:

Requirement Req$_1$: *Computing the opaque constant at execution time should be fast.*

The second requirement is that it should be computationally cheap to apply the obfuscation. The obfuscation should be based on a problem that, despite it is hard to solve by the attacker, it should be easy to construct and verify by the obfuscating tool. To assess that the obfuscation is correctly applied, the obfuscating tool should not solve the same problem as the attacker. Thus, the second requirement is the following:

Requirement Req$_2$: *Constructing the opaque constant at obfuscation time should not be hard.*

8.3.2 Empirical Validation

We performed an empirical evaluation to validate the execution time overhead involved with the usage of opaque constants (both at runtime and obfuscation time).

8.3.2.1 Research Questions and Variable Selection

We formalized our evaluation goals in the following research questions:

- **RQ$_1$**: How long does an obfuscated program take to compute the value of an opaque constant based on k-clique at runtime?
- **RQ$_2$**: How long does a state-of-the-art SAT solver take to check satisfiability of a 3SAT formula?

The first question is intended to quantify the performance degradation due to program obfuscation and study how long it takes to compute an opaque constant at runtime. The second research question aims at assessing the computational effort required by the defender to generate a 3SAT problem which is needed to make a constant opaque (Req_2). To answer these research questions, we will consider the following metrics on a number of experiments:

- **ETIME**: Execution time for a program, the user time reported by the Linux's tool time and converted in seconds.
- **NVARS**: Number of propositional variables in a 3SAT problem.
- **PSAT**: Fraction of satisfiable problems in a given set of 3SAT problems.

8.3.2.2 RQ_1: Time to Compute Opaque Values at Runtime

In this experiment, we measure how long an obfuscated program takes to compute the value of an opaque constant based on k-clique. We base this empirical assessment on a program that just calls 1,000,000 times a function *f* that computes an opaque constant value of 16 bits. The experiment is repeated with an increasing number of propositional variables NVARS. We start with NVARS = 4 and we increase this value in steps of size 4, until 40 propositional variables. For each value of NVARS, 10 different random 3SAT formulas have been generated. To minimize random error, the measurement of the execution time ETIME is repeated 25 times. Thus, in total, we collect 250 measurements of ETIME for each value of NVARS. Figure 1 reports the boxplot of the execution time ETIME for increasing values of NVARS. It can be noted that the computation of 1,000,000 opaque constants on average takes from 4.13 seconds for formulas with 4 propositional variables up to 4.48 seconds for formulas with 40 propositional variables.

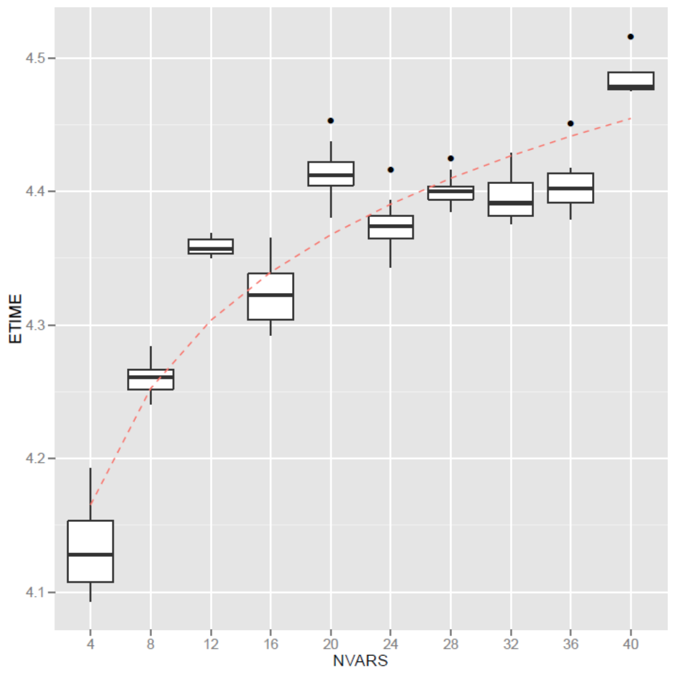

Figure 1 - Execution time taken by the obfuscated program to compute 1,000,000 opaque constants. Opaque constants are based on a k-clique problem defined by 3SAT formula in **NVAR** propositional variables. The red line represents the log model

The trend of ETIME seems to suggest a logarithmic dependency with NVARS. To validate this observation, we fit experimental data with the subsequent log model: ETIME = a + b*log(NVARS).

Table II show the result of the model estimated with a linear least squares regression. The first column reports the name of parameter that is estimated (a and b). The second column reports the estimated value for the parameter. Then the third and fourth columns report, respectively, the standard error and the p-value of the t-test. As we can see, the parameters can be estimated with a very large confidence. The estimated log model is shown in Figure 1 as an interpolating red dashed line.

Table 9 - Estimated coefficients of the logarithmic model for the execution time taken to evaluate 1,000,000 opaque constants

Parameter	**Estimation**	**Std. Err.**	**P-value**
a	3.99126	0.04789	4.79e-13
b	0.12560	0.01607	5.17e-05

Based on this experiment, we can answer to research question RQ_1 in this way:

*A program obfuscated with opaque constants based on k-clique is affected by a very low runtime overhead. The runtime overhead increases logarithmically with the number of propositional variables used in the 3SAT formula according to this model: ETIME = 3.99 + 0.13*log(NVARS).*

8.3.2.3 RQ_2: Time to Verify 3SAT

In the experiment we measured how long it takes to Yices[6], a state-of-the-art SMT solver, to check if a 3SAT formula in NVAR propositional variables and 4.3*NVARS is satisfiable. For NVAR running from 50 to 350, we generate 100 random 3SAT formulas and for each formula we execute Yices to check if it is satisfiable or not. We collect execution time from the execution log Yices produces when the tool is run with the flag (show-stats). We take NVARS≥50 to avoid time measures be shorter than the measurement accuracy. Table 10 show aggregated statistics for collected data. Mean of ETIME ranges from 1.3e-4 for NVARS = 50 to 828.2 for NVARS = 350. PSAT ratio estimates the probability of generating a satisfiable 3SAT problem and ranges from 0.60 for 50 variables to 0.30 for 350 variables.

Table 10 - Satisfiability ratio (PSAT), mean and standard deviation of execution time for Yices solving random 3SAT problems in NVARS variables (for NCLS/NVARS=4.3).

NVARS	**PSAT**	**MEAN(ETIME)**	**SD(ETIME)**
50	0.60	0.00013	0.00019
100	0.51	0.00176	0.00143
150	0.49	0.01134	0.00645
200	0.45	0.09320	0.06800
250	0.37	1.09245	0.74141
300	0.37	25.42116	26.68942
350	0.30	828.21444	837.64963

[6] Yices, http://yices.csl.sri.com/

Based on the empirical evidence, we can answer to RQ_2 in this way:

The time a state-of-the-art SAT solver takes to decide a 3SAT formula with a number of variables NVARS less than 200 is negligible. Furthermore, the probability of generating an (un)satisfiable 3SAT formula is close to 50%.

8.3.2.4 Considerations about Requirements

Here we discuss the results of the empirical investigation with respect to the requirements of opaque constants defined above.

Requirement Req_1: The computation of the value of an opaque constant based on k-clique is quite fast (see RQ_1) and it grows logarithmically with the problem size. So, we can claim that requirement Req_1 is fully met.

Requirement Req_2: Our tool took milliseconds to generate the opaque constant code, so we can claim that requirement Req_2 is also fully met, because the time taken to obfuscate constant values is short.

Thus, we can formulate the subsequent statement:

Opaque constants based on the k-clique problem are fast and scalable to compute at execution time (Req_1); and fast to generate with our approach (Req_2).

8.4 Client-server code splitting

The content of this section is partially copied from the Section 4 of Deliverable D3.04

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[7], 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 11 and briefly described in the following paragraphs.

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

Table 11 - Subject applications used in the experiment

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.

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

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.

8.4.1 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.

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.

8.4.2 Experimental Results

Execution time: The diagram in Figure 2 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 number 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 2 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.

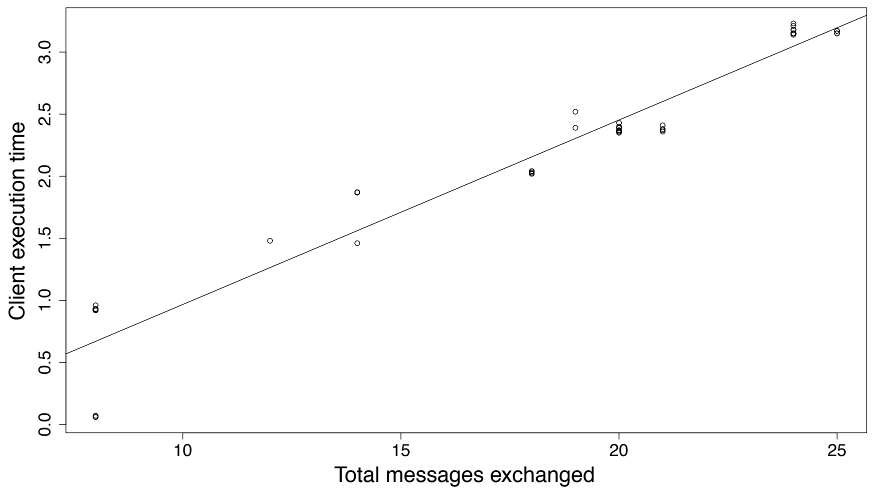

Figure 2 - Client execution time vs total messages exchanged (License Checker)

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.

All the results can be found in Table 12. 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 execution time (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 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, 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 3 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.

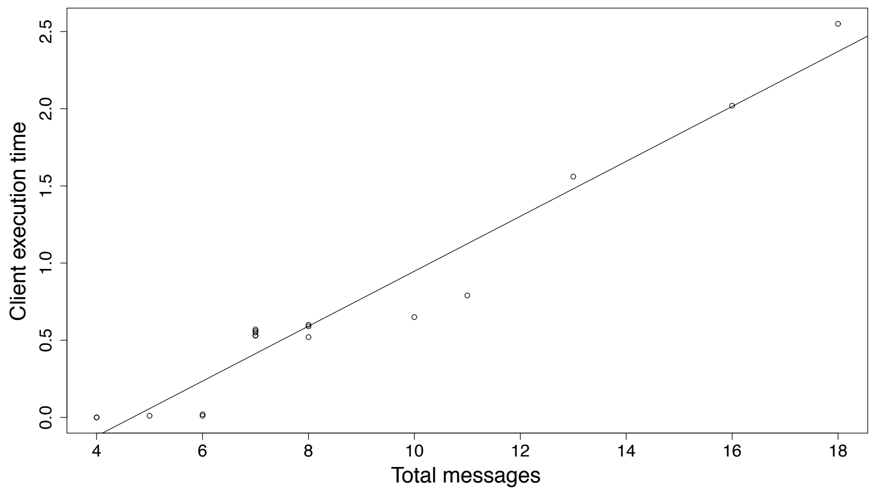

Figure 3 - Client execution time vs total messages exchanged (TCAS)

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.

		Variables			**Statements**			**Messages**		
SUT		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 12 - Pearson's correlation between variables/statements/messages and execution time

Memory overhead: Figure 4 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.

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 4, 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 13 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.

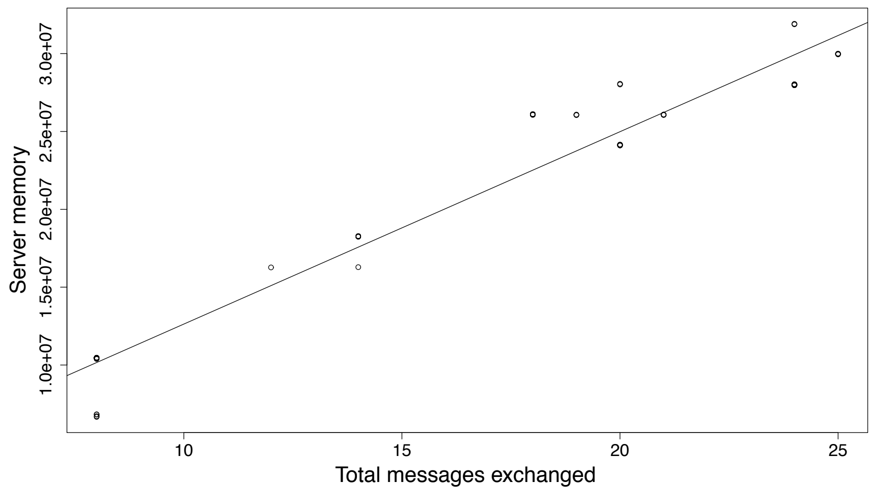

Figure 4 - Server memory overhead vs number of messages (License Checker)

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 5 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.

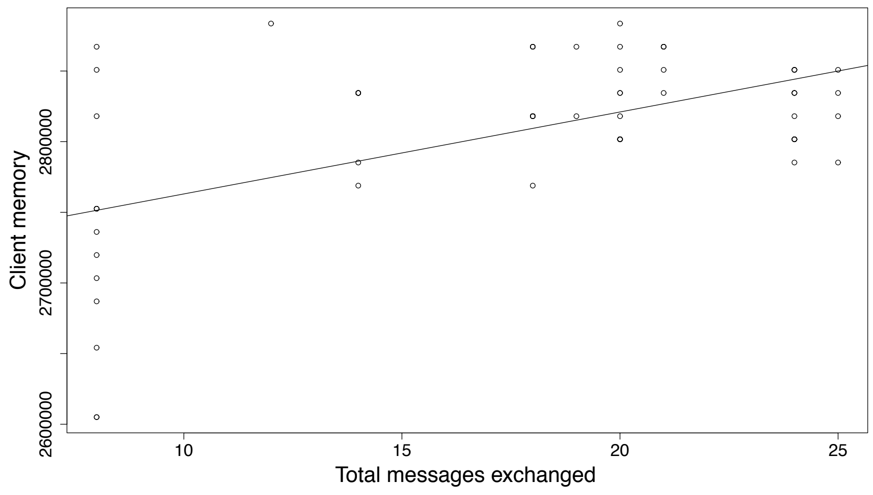

Figure 5 - Client memory overhead vs number of statements moved on the server (License Checker)

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

8.5 White-box cryptography

Deliverable D2.01 contains a theoretical analysis of the overhead of provably secure WBC schemes.

In the project, we evaluated less secure, but practical versions of WBC AES implementations.

The size of the fixed-key a dynamic key AES WBC is dominated by the lookup tables that are introduced. Both come with 167936 bytes of lookup tables.

In addition, for the dynamic-key AES WBC case, there is also an impact on the size of the obfuscated key, which is 176 bytes rather than 16 bytes.

The performance impact is rather small, because these AES WBC implementations have been optimized for use for a content descrambler which cannot suffer much performance impact.

This low overhead was affirmed during measurements on the Linux version of the SFNT use case. When we executed the content of one use case scenario step (asking for the solution of a riddle, in which case the computations are mostly concentrated in four invocations of cryptographic primitives to be protected with WBC) in a loop, a slowdown of only 8% was observed when the WBC was actually deployed instead of the standard crypto implementations.

There remains one caveat of course: as soon as other protection techniques are deployed on the white-box code, performance may further degrade.

8.6 Anti-Cloning

Adding the anti-cloning library increases the footprint of a binary or library with +- 1kB.

Performance impact is dependent on the network latency and frequency of calls to the AC API. We refer to Sections 8.1 and 8.4 for extensive evaluations of the overhead of connections to the security servers.

8.7 Anti-Debugging

The performance overhead of this technique is heavily dependent on which parts of the application are protected (and how often they are executed). Therefore we can't simply give an overhead for this technique. On the other hand, we do know that the overhead is introduced by the transformations of control flow and memory accesses. All control flow and memory accesses happen through a single instruction, but we can measure the execution time introduced by the transformation. On our ARM board, the transformation of control flow introduced an overhead of 1.7 ms, that of memory reads and writes 3.4 μs and 2.3 μs respectively.

We also give a general equation for the total size overhead of the technique:

$$\begin{aligned}&Total\ Size\ Overhead\ (bytes) = size(DebuggerObject) + 4\times \\ &\quad (NrOfTransformedMemoryAccesses\times \\ &\quad (InstructionsPerMemAccessTransformation - 1) + \\ &\quad NrOfTransformedInvocations\ \times (InstructionsPerInvocationTransformation - 1) + \\ &\quad NrOfTransformedExit\ \times (InstructionsPerExitTransformation - 1) + \\ &\quad (InstructionsPerEntryTransformation - 1)\end{aligned}$$

8.8 Offline Code Guards

The performance overhead of this technique is dependent on the size of the regions to be guarded, and the frequencies with which they are attested and verified. These frequencies are determined by the hotness of the code in which the associated annotations are placed and thus it is hard to give a meaningful performance overhead.

The size overhead per attestator can be estimated using the following equation:

$$\begin{aligned}Total\ Size\ Overhead\ (bytes) = size(AttestatorObject) + size(ADS) + 8\times \\ (nr\_of\_verifier\_annotations + nr\_of\_attestator\_annotations),\end{aligned}$$

where the size of dependent on the number of verifier/attestator annotations because each of these annotations requires inserting an invocation (usually consisting of 2 instructions).

8.9 Control Flow obfuscations

Figure 6 shows the overhead of applying the branch function insertion and opaque predicate insertion obfuscations, using either the original, stochastic method or the profile-guided method (where the X percent least frequently executed blocks are selected per function) on the bzip2 SPEC2006 benchmark. The x-axis indicates the percentage of transformed code blocks, and the y-axis the execution time overhead. As can be seen, the stochastic method already introduces an overhead when only 10% of the blocks are transformed, compared to 0% overhead from the profile-guided approach. The profile-guided approach consistently produces less overhead, except for when all code blocks are obfuscated, in which case both methods produce the same result.

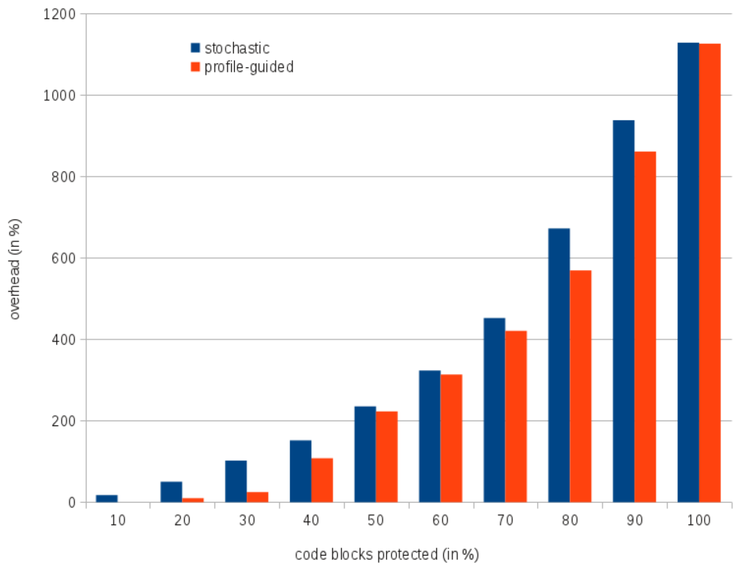

Figure 6 - Overhead comparison between stochastic and profile-guided obfuscation

As for size overhead we can say that opaque predicates introduce on average 62 new instructions (that is, 248 new bytes). Branch functions and function introduce a number of new instructions per BBL in the function that is transformed (the percentage of BBLs that is transformed can be set using $percent\_apply$. We thus have the following equation

$$Total\ Size\ Overhead = percent\_apply \times NrOfBblsInFunction \times 10 \times transformation\_cost.$$

This $transformation\_cost$ is equal to 10 for function flattening and 7 for branch functions.

8.10 Anti-Callback Stack Checks

We evaluated this technique by inserting a check in every function of the bzip2 SPEC2006 benchmark. The performance overhead was 0.25%, the size overhead was 2%.

8.11 SoftVM

For this technique, we study the performance overhead empirically. We applied the protection technique to the NAGRA use case on different regions. We observed that dependent on the size and the hotness of the protected regions the overhead varied significantly. In particular, when we moved the entire decryption to the SoftVM, the overhead became so large that the video played at less than 1 frame per second. The region we eventually decided to protect with the SoftVM was a compromise between overhead and security. With this region, the overhead is small enough that the user experience is not affected.

We also tested the overhead on the bzip2 SPEC2006 benchmark by translating as much as a possible of the application into the SoftVM bytecode. Whereas the unprotected version of the application would finish in less than a minute on our setup, the protected version ran for over two days before finishing. It is thus clear that moving code on a critical path to the SoftVM should be avoided.

8.12 Diversified Cryptography Library

This protection is implemented with a call to the DCL library that has a significant memory and CPU overhead. In this section, the Android Monitor tool of Android Studio is used to visualize this overhead. The OTP use case has been used to check the overhead.

The device used is a Samsung S3 (GT-I9300 model). On more powerful devices like Samsung S4, S5, or S6, the application footprint is too small to have significant diagrams.

The first two diagrams in Figure 7 and Figure 8 show the overall consumption at the end of the provisioning phase and the generation of several OTP values.

The DCL protection sets the device key in the library after the derivation and this induces much more processing compared to the provisioning without DCL applied where there is only a key derivation and the storage of all protected data.

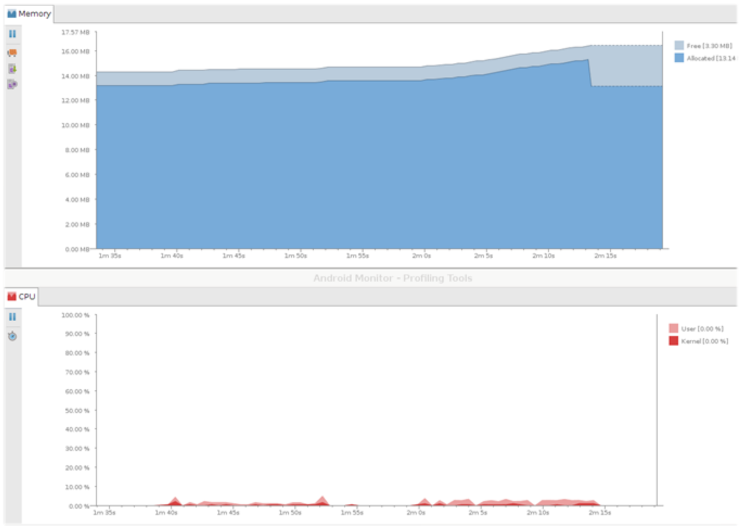

Figure 7 – Memory and CPU consumption with DCL protection applied

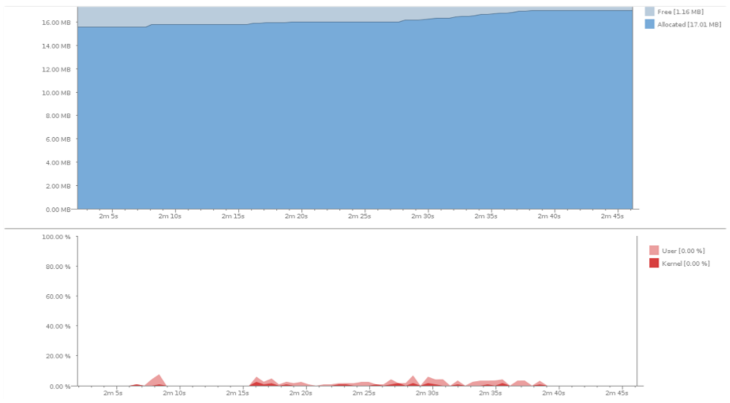

Figure 8 – Memory and CPU consumption without DCL protection applied

There is some GUI involved in the use case; this is why the interval between the two groups of data differ in both diagrams.

By zooming on the CPU diagram the DCL library it is easier to check the differences. The red part is the kernel mode and the pink part represents the CPU consumption in user mode. In Figure 9 the DCL overhead clearly appears.

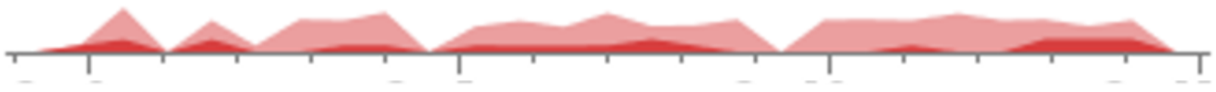

Figure 9 – CPU consumption with DCL applied

One might be confused by the overall shape difference on the right side of the curve but during the measurement, several One Time Password (OTP) have been generated and the number is not the same in the two cases.

Some image manipulation has been done that gives the effect that there is more CPU consumed when DCL is not applied in Figure 10. These diagrams are just there to show the effect of the protection without giving precise CPU consumption values.

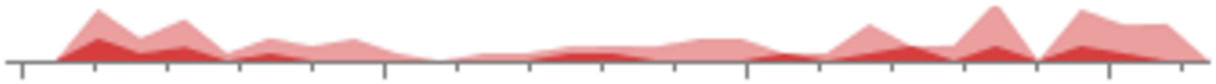

Figure 10– CPU consumption without DCL applied

8.13 Control Flow Tagging

The overhead caused by this protection is insignificant. For a gate annotation set in the source code, a few assembly instructions are added in the function and the CPU overhead can hardly be measured. For one gate, one counter has to be statically added, which means 4 bytes are allocated in the data segment.

The verifier is generated in a separate function but again it only contains the logical expression. The most noticeable code is the extra call generated to call the verifier function.

This protection can take advantage of the ADSS level 2 protections that aim to add extra invocations to protections to fool the attacker. For Control Flow Tagging (CFT) this would mean more gates in the code dispatched all over the application. The overhead would then slightly increase but again, considering the very light footprint of this protection the memory and time overhead will not be significant.

8.14 Conclusions

From the various assessments of the protection overheads, we can conclude that the overhead they introduce is limited and can be controlled by deploying the protections cautiously. In particular, on the project use cases, the protections could be deployed as foreseen by their developers and the project's security experts, with an acceptable overhead.

Section 9 ADSS Validation

Section authors:
Cataldo Basile, Daniele Canavese, Leonardo Regano (POLITO)

The ultimate goal of the ASPIRE Decision Support System (ADSS) is to identify the golden combination, a combination of protections that, once applied on the listed assets in the application to protect, provides the best mitigation of the risks against the assets, thus the best level of protection.

Therefore, validating the ADSS will consist of the analysis of the golden combination (together with the best *n* combinations, named golden combinations) to determine if they are:

- *Correct*, the individual protections in the golden combinations protect the asset,
- *Valid*, there are no issues in the simultaneous use of techniques in combination,
- *Effective*, the use of combinations of protections is effective to protect the assets, and
- *Optimal*, under the experts' experience and best practice the golden combinations are (among the) best way(s) to mitigate risks against the assets.

However, there are additional data, which are produced by the ADSS, that deserve a validation because they are (or may be) useful for software protection experts:

- the identified attack paths;
- the suggested protections; and
- the Level 2 Protections (L2P).

These data need to be validated as well. More information on the ADSS workflow, the artefacts produced and the processes to obtained them are available in the deliverable D5.11.

We remark that the validation will also focus on the L2P, which have been discussed and added into the ADSS scope in the Lausanne meeting at NAGRA in March 2015, and developed by POLITO during the last 18 months. L2P protections are the protections that are deployed not to directly protect the assets but to render more complex to the attackers the identification of the actual assets.

Indirectly, the validation is used to confirm the validity of the methods used by the ADSS to find the golden combinations:

- the ASPIRE Security Model, the ASPIRE Knowledge Base (AKB), and the inference engines performed by the Enrichment Modules;
- the Protection Instantiations, which are precise ways to use a protection technique;
- The models for estimating effectiveness and overheads of the application of combinations of techniques;
- The (game theoretic) optimization model used to find the golden combinations.

Furthermore, we aim at determining the user friendliness of the ADSS and how helpful it is, i.e., the suitability for everyday tasks of software protection experts, by checking:

- how meaningful are terms and data organization (in the ADSS Report);
- if the data they find in the report are enough to evaluate the decision process performed by the ADSS;
- if the types of data presented by the tool and the report are complete.

We present in Section 9.1 the methodology to perform the validation, then we will discuss individually the results of the validation in each of the three ASPIRE use cases, that is, the validation of the use of the ADSS on the NAGRA, SFNT, and GTO use cases.

9.1 Information and validation methodology

There are two sources of information to perform the ADSS validation:

- *expert judgments at the industrial partners*, namely NAGRA, SFNT, and GTO experts. Experts have concentrated on their own use case, because they know it well and can express concerns on the ADSS results, and because of IPR issues. Experts involved in the validation of the ADSS can be further divided in experts that actively participated in the ASPIRE project, the *internal experts*, who also helped in tuning the initial executions of the ADSS, and experts who were only involved in the validation of the final results, named *external experts*.
- *reports from the tiger teams* that were involved in the experiments that aimed at attacking ASPIRE protected use cases. Also in this case, there are three reports, one for each use case and only visible to the use case owner and to the academic partners.

9.1.1 Experts judgement collected from internal experts, the protection owners, and the project coordinator before the validation

These inputs have been collected during the three years of the project from the internal experts (and some other experts that were in the end not available for the validation) and from the protection owners:

- the Protection Instantiations;
- the asset in each use case;
- weights associated to each asset.

Protection Instantiations (PIs) contain information about different ways to use a protection technique, for all the ASPIRE techniques. That is, a PI is a precise way to apply a protection that also includes information about the parameters to use when invoking the tool that deploys the protection. PIs also report (1) the methods to estimate the effectiveness of that protection (used exactly as required by the PI) if it is used to mitigate risks against a given security property of the assets, and (2) the methods to estimate the overhead added by the protection on different area (computation, network, binary size, server-side overheads). Both effectiveness and overheads are also characterized by functions that depend on several parameters and allow a precise estimation of the effects of the application endowed by the PI. The parameters used by current PIs are different types of metrics and constants. When actually applied on an asset, PIs become *instantiated PIs*. The information contained in the Protection Information has been acquired by the POLITO team with a set of interviews with the protection owners. Data extracted with these interviews have been updated as soon as new improvements were available during the ADSS design stages. Data from protection owners have been completed to completely fill in the data in the AKB required to perform reasonings.

The assets in each use case application have been collectively identified by the entire ASPIRE consortium when preparing the tiger team experiments. To that purpose, the Consortium manually selected the protections to enforce on each application part (code and variables) and manually derived the protection-specific annotations. The POLITO teams, starting from the analysis of these protection-specific instantiations, derived a first list of assets. This list has been validated with the internal experts on several occasions, by email and face to face in the Lausanne and Munich meetings and, only for NAGRA, in the Torino meeting held in July 2016.

The weights associated to the assets have been preliminarily estimated by the POLITO team, based on the protections (types and number) applied to the application parts in the use cases' source code.

In the preliminary phases, which in practice lasted the whole project duration, the POLITO team has taken advantage of the much better knowledge of the software protection field of the coordinator and his group at UGent, and of the experts at NAGRA. This (almost) continuous feedback resulted in a face to face meeting in March 2015 in Lausanne, with several experts from NAGRA and satellite companies, and with a face-to-face meeting in July 2016 in Torino, with the explicit purpose of validating the ADSS tool, the extracted information, and to discuss how to improve it in the short term, i.e., for this validation, and on the long term, to achieve a better impact.

The UGent team and the coordinator were also involved in the several design aspects, in the validation of the ideas and approaches before the implementation, the collection of the preliminary experts' judgements (weights, impact of attacks, effectiveness of protections) and in supporting the activity that lead to the definition of conflicting vs. cooperating protections (i.e., the synergies). In practice, the initial human knowledge, represented according to the security model, in the AKB was collected cooperating with UGent.

9.1.2 Validation with experts

All the people involved in the validation of the ADSS result were required to be knowledgeable in software protection and to be very well informed on their use case. We have performed several tasks immediately before the validation, some tasks were performed also longer before, but we have repeated them in order to have the most up to date information. Then we have performed a first iteration to collect feedback and comments. The iteration consisted of sending the experts the ADSS report and side information to process in the proper way the report, and an open questionnaire indicating the feedback needed for the validation. The POLITO ADSS teams has addressed all the feedback that were in order, and prepared the second iteration. Then the POLITO ADSS team has also addressed as many issues as possible after the feedback after the second iteration.

The following preliminary tasks have been performed before the validation:

1. Internal experts have been interviewed to validate the general information obtained by the protection owners on each technique marked as usable in their use case. The questions included suitability, opportunity, and effectiveness of each protection technique for their use case. Moreover, we asked about the estimations on overhead.
2. The assets, identified by analysing the annotations inserted into the applications to protect for preparation of the tiger team experiment, have been explicitly confirmed by the internal experts. That is, internal experts have been interviewed to confirm the status of asset (and add new assets if needed) and to assign proper weights in order to determine the importance of the assets and, thus, to estimate the consequence if they are compromised.
3. Internal experts have been interviewed to confirm preferences on the protection techniques to use and other suggested modes of operation (types of attackers to consider, thresholds for overheads, etc.). Moreover, we have asked about the overhead thresholds, which are the maximum allowed degradation that protections can impose, on size overhead, network overhead, and computation overhead.
4. The ADSS has been prepared to be executed on the use cases. This task required the proper configuration of the ACTC to be executed on each use case, to set up the metrics framework, and the verification that the CDT (and CodeSurfer) analysis tools were working properly, etc.
5. The ADSS Reports for the three use cases have been generated.

The following tasks have been performed with internal experts:

6. Internal experts have been asked to further validate the assets list.
7. Internal experts have been asked to validate the attack paths determined with the ADSS on their use case.

8. Internal experts have been asked to validate the golden combinations determined with the ADSS on their use case. Moreover, they have been asked to validate the L2P.
9. Internal experts have been asked to provide feedback on how to improve the results of the ADSS.
10. Feedback from internal experts have been incorporated and induced changes in the model used by the ADSS, in the information provided by the Protection Instantiations, and modifications to the tool.
11. A new ADSS report has been generated to circulate to the external experts for each one of the three use cases.

The following tasks have been performed with external experts:

12. External experts have been asked to further validate the assets list.
13. External experts have been asked to validate the attack paths determined with the ADSS on their use case.
14. External experts have been asked to validate the golden combinations determined with the ADSS on their use case. Moreover, they have been asked to validate the L2P.
15. External experts have been asked to provide feedback on how to improve the results of the ADSS.
16. Feedback from expert experts have been incorporated and induced changes in the model used by the ADSS, in the information provided by the Protection Instantiations, and modifications to the tool.

Information asked to the experts

Experts received the ADSS report computed by the ADSS on their use case. We preferred to give them the report as it contains (almost) all the information that can be accessed from the ADSS and with this approach we did not require them to install and configure both the ADSS and the ACTC, which may require hours of effort.

The information was collected in form of open questionnaire. Guidelines to answer the relevant information were provided in this form:

Some facts and requests:

The Assets are the ones the internal experts have confirmed in face to face meetings.

The Assets' weights are the ones that internal experts have decided for their use cases.

General

Any feedback on the report formal and visualization options is welcome

ASSETS / APPLICATION PARTS

You should say us if the list of the APPLICATION PARTS is useful or you would like to see it differently or different information

ATTACK STEP / ATTACK PATHS

1) You should VALIDATE if the attack steps are meaningful

2) For all the attack steps, you should VALIDATE if the Suggested Protections are correct, sound, proper, effective and if the EFFECTIVENESS estimated is correct.

3) You should say us if the data showed in the report are useful or you would like to see different information or the same information shown differently.

GOLDEN COMBINATIONS

1) You should VALIDATE if the golden combinations (L1P) are meaningful.

NOTE: we tried to pick the n best in different regions of the solution space, so they differ a bit and may be slightly different starting point, if one wants to use them as a starting point for the manual protection, they are not completely different though.

2) For all the 10 golden combinations, you should notify us if you noticed something strange in the use of techniques, some association of techniques which is strange for experts, the use of techniques for assets in anomalous ways.

3) You should say us if the data showed are useful or you would like to see it differently or different information.

LEVEL 2 PROTECTIONS

1) You should VALIDATE if the level two protections (L2P) are meaningful.

2) You should NOTIFY us if you noticed something strange in the use of techniques, some association of techniques which is strange for experts, the use of techniques for assets in anomalous ways, some cases where you wouldn't extend/randomly pick other areas.

3) You should say us if the data showed are useful or you would like to see it differently or different information.

4) You should tell us if the estimated maximum degradation thresholds (overheads sections of the L1P and L2P parts) are reasonable.

After the questionnaires were answered, we have performed several interactions for two purposes: allowing us to clarify the meaning of their answers, explaining them the unclear details, technicalities about the ADSS, and confirming proposed reactions to identified issues.

9.1.3 Validation with tiger team reports

All the three tiger team experiments have produced (more or less) detailed reports. These reports conveyed information on the attack steps and strategies attackers have tried to compromise the assets with applications that were previously protected with the ASPIRE protections. These reports have been analysed in order to validate the ADSS output, in particular, we have evaluated the attack paths the attackers have conducted and checked if a corresponding attack path was produced by the ADSS (attack path discovery Enrichment Module), and we have also double checked if the suggestions proposed by the tiger team were also given by the ADSS.

9.2 NAGRA Use Case

Internal expert: Brecht Wyseur.

External expert: none.

9.2.1 Experts judgment

After the first iteration, the NAGRA internal expert reported the following information:

- *assets*: the selection looked right, only the printBuf function was not expected to be protected.
- *from application parts to assets*: the expert noted that it would be interesting to see if this could be derived “automagically” from the list of application part, as that would be a sort of cross-check. Our opinion is such a task would require some static analysis and will easily be incomplete. Assets can be probably guessed in some way, but to make guesses reasonably complete and statistically significant, too much effort is needed and it is not clearly feasible. Moreover, during the interviews with the experts in the Lausanne meeting (March 2015) and the advisory board meetings, the experts wanted to have control on the assets.
- *General consideration on the report information*: quoting “I think this is very useful”. However, he saw improvements on the presented information. The "Cost" column in the attack path table has been marked as ambiguous, as it should be clear that it is

the attackers benefit. This has been solved before submitting the second version of the report for the external experts.

- *attack paths discovery*: this task has been defined as very meaningful for NAGRA. In their job, when designing a product, they perform a full threat analysis, which is a huge amount of work because it is about identifying assets, required properties, and then think about all possible kind of attack scenario's. The work needs to be as exhaustive as possible, and every time the product changes (sometimes even for small changes), that may drastically impact the threats. Therefore, a way to automate this task, which presents the different attack paths, can speed up a lot this kind of exhaustive work. To the best of the expert's effort, the attack paths looked exhaustive.
- *doubts on suggested protections*: the expert wanted to know if suggested protections were already filtered from the ones that do not meet the overhead constraints. The ADSS has a configuration option to this purpose. The ones in the provided report were already filtered.
- *doubts on the meaning of the golden combinations*: quoting: "Is this a minimal protection, or the optimal? Or can this already be different options of combinations?" We added more information in the report to avoid this kind of ambiguity.
- *techniques not in the report*: the expert noted that several techniques were not yet supported by the ADSS.
- *protections that protect other protections*: all the protections are on the original application assets. The expert suggested to extend the approach and apply protections on other protections. We started this work that is not finished yet, as explained in D5.11.
- *golden combinations*: the combinations reported are meaningful. The expert proposed to analyze the combinations and see if there is room for improvement in the golden combinations. This information has not reached us during the validation.
- *clustering protections*: since many of the golden combinations on the NAGRA use case only differ on a few protections, golden combinations could be presented by highlighting the differences, e.g., form the golden combination.
- *"Kept protections" in L2P*: the expert proposed to remove the protections marked as "kept" in the L2P. However, the L2P protections report all the protection techniques to enforce, not only the delta. This has been clarified in the next versions of the tool and of the report.
- *L2P are useful*: the expert evaluated the proposals in the L2P tables as interesting things to reflect upon. Quoting "Probably I would apply almost all of them, if that keeps the final artefact within the specified bounds."

9.2.2 Tiger team analysis

The NAGRA tiger team presented a very detailed report on the operations they conducted in order to compromise the use case assets. These operations formed a very complex attack path, which is much more detailed than the attack paths that are represented into the ADSS. However, by abstracting from all the practicalities they had to face in order to adapt tools and circumvent protections, the main phases (attack steps) can be described as:

1. Static analysis;
2. Dynamic analysis to extract traces;
3. Analysis of the traces to locate and compromise the (mainly cryptographic) assets.

This attack path was also identified by the ADSS.

9.2.3 Conclusions

The expert reported that the attack path discovery is very useful and to the best of his check, the output exhaustive. Nonetheless, he highlighted the fact that the attack path are too coarse grained to make very fine tuned decisions. This is also confirmed looking at the tiger team report. Having the possibility to explode the ADSS attack paths into sets of more precise alternative applications of the attack paths, may lead, through appropriate weighting and estimation of the difficulties and effectiveness of all of the, can lead to more precise decisions on the techniques to apply. However, by protecting against the high-level ADSS attack paths, all the attack vectors are mitigated, only with less precisions.

Moreover, he reported that the L1P and L2P protections are useful and, to the best of his check, correct. The expert did not report nor had the time to check if better combinations were possible, to prove or confute optimality.

However, he also reported that several ambiguities and details that could be improved to make the report more precise and useful for software experts.

Moreover, the proposed investigation of protections of protections is an interesting aspect we are exploring.

9.3 SFNT Use Case

Internal expert: Werner Dondl.

External expert: Andreas Weber.

9.3.1 Experts judgment: first iteration

After the first iteration, the SFNT internal expert reported the following information:

- *missing suggested protection*: some attack paths resulted as unblockable, given the protection techniques available at the ADSS. However, the expert's feeling was that some protection technique would have helped. In particular, the expert pointed out that the "dynamicallyLocate" step leading to the attacks was mitigated by anti-debugging. Indeed, this comment resulted in a major change in our algorithm (different evaluation of the impact of protections in intermediate steps toward the actual goal = compromise an asset) and in a different classification of anti-debugging, obtained through interactions with the protection owners (UGent).
- *wrong suggested use of protection*: the ADSS proposed anti-debugging as a technique to mitigate attack paths including the steps "staticallyLocate" and "staticallyChange". The expert suggested that the technique was not appropriate. This comment has been double checked with the protection owner and other partners, the ADSS was right.
- *overheads*: the expert was expecting estimations concerning the difficulties and times to mount the attacks and much more information on the overheads introduced by protections (quoting "in performance or memory usage/increase of code size etc. I think that would be quite interesting"). The overheads in the current version of the ADSS are rather approximated; however, we planned to report more data on overheads both in the tool and in the reports.
- *potential incoherence in data types associated to some assets* (variables): the use of "int_hex" was expected only to handle base64 and potentially conflicting with SoftVM. The data types were thus all checked.
- *not self-explanatory data in the report*: the golden combinations are rated by a "score" whose unit is not clear. The expert suggested to map into some "Protections Euros/Dollars" or in "time needed to break". We were not able to address this very meaningful comment, even if we planned to do it after the project end.
- *overheads of golden combinations*: as for single protections, the expert was expecting overhead data associated to the golden combinations.

At the second iteration, we have the feedback of the external expert, who reported:

- *Missing data in the visualization*: the expert noted that only 1/4 of the attack paths were visible. In the end, after our analysis, it was most probably a problem of his browser. From the visible attack paths, he noted that one could only see that "dynamicallyLocate" is made more difficult by "anti-debugging", a "right but not really exciting information". He was not able to see "All the Application Parts" thus he was not able to judge them, since only a few standard data types were visible in the first page of the table. Moreover, he was also confused by the dynamic objects that were used to show one golden combination per page, again, we fear because of his browser visualization issues.

- *Attack paths visualization*: a picture of the Petri net of the attack paths would have been useful. The table with the attack paths should also have a visible grid. The IDs associated to attack paths were considered unneeded. However, the attack graph is available from the tool and not in the report. We added a graph that shows all the attack paths but, since we planned to evaluate the ADSS Full separately from the ADSS Light, the Petri Net features were disabled.
- *Level 2 protections*: he suggested to omit the "kept" protections from the list. However, the L2P protections report all the protection techniques to enforce, not only the delta. This has been clarified in the last versions of the tool and of the report.

9.3.2 Tiger team analysis

The tiger team report did not provide a very precise attack path, rather a set of attack steps, in separate excel files. We can confirm that all the attack steps are used by the engine that performs the attack path discovery. Moreover, several attack paths were able to cover all the possible variants. However, even if we were not able to validate the attack paths, we guess it was covered.

9.3.3 Conclusions

This validation confirmed that the ADSS has potential but it needs a much closer involvement of the protection experts to perform the fine tuning of the decisions made. With the help of the experts we were able to fix issues in the decision process.

The attack paths were evaluated as very likely exhaustive. Unfortunately, the level of detail of the attack paths discovered was not able to map the actual operations performed by the attackers in the tiger team. The attack paths precisely convey information on the main activities and on the general strategy, but the attack paths discovered by the ADSS do not cover all the (possibly complex) activities the attackers have to face to perform them (writing scripts, adapting tools, etc.).

The golden combinations were declared as free from inconsistencies but they were not marked as unrealistic, however, there was not a precise commitment to declare them as optimal or to strongly support them.

Experts expressed the need for more information to better characterize the decisions, especially on the overheads, and provided suggestions on how to improve the visualization.

9.4 GTO Use Case

Internal expert: Jerome d'Annoville

External expert: Philippe Smadja

9.4.1 Experts judgment: first iteration

After the first iteration, the GTO internal expert reported the following information:

- *unbalanced recommendations*: due to not properly specified input weights assigned to the assets that lead a not balanced coverage of the critical code. The internal expert assigned weights ranging from 1 to 5000, therefore assets with weight around 100 vanished in the list of the most important assets. We learned that even if weights are integer numbers that can be freely assigned to assets, differences of orders of magnitude of magnitude can confound the algorithms and the expectations of experts.

- *useful asset and application part visualization*: the analysis pointed out some functions (e.g., in Utils) that were not selected as assets thus not protected. The assets and application parts tab has been marked a relevant output.
- *missing expected protections*: No guards protection in the golden solutions. No Data Obfuscation recommendations. No remote attestation in the recommendations. At the moment of the first report, these techniques were not supported by the ADSS. This created confusion as the expert was expecting all the ASPIRE supported technique and this was not evident from the tool.
- *some protections are over-represented*: while obfuscations can be widely applied, some protections like SoftVM or anti-debugging should be scarcely applied. The expert was expecting that only a few recommendations on SoftVM and Anti-debug would have been in the golden combination. Therefore, two activities have been performed: these two protections have been slightly discouraged by the addition of a second level of weights that alter the effectiveness rating defined into the PIs. The second level of weights will serve to locally diminish or increase the preference to specific protections. That is, while the ADSS now only proposes to avoid some techniques, it is better to have the possibility to discourage or encourage the use of some techniques, or to limit in some way how many parts (and how big these parts must be).
- *data gathered*: the amount of data gathered by the ADSS seems not manageable by hand. This was a positive remark and justified the effort to develop such software protection assistance tool.
- *application parts section*: column names needed to be more explicit. This has been fixed in the ADSS and in the report.
- *scores assigned to the golden combination*: very puzzling values (floating point = real numbers). Suggestion: avoid floats even if they are used internally. Rather use a set of integer values. This input has been considered, however, the mapping to a set of levels has not been performed in time for the second round of the validation.
- *confusion in the golden combination*: the semantic of the order of the protection proposed in the golden combinations was unclear, i.e., it was not clear if the order was significant or not. Colour codes were suggested to visualize the effectiveness.

9.4.2 Experts judgment: second iteration with the internal expert

At the second iteration, we have collected both the second input of the internal expert and the feedback of the external expert. The internal expert reported:

- *ADSS report information*: in the Application Parts and Assets sections, more accurate source code indication should be provided (e.g., beginning and ending lines). This request has been satisfied; indeed, this information was available in the tool but not reported in the report.
- *over-representation of anti-debugging*: the internal expert didn’t know enough details on the anti-debugging to guess the consequences (overhead, effectiveness) of the use of anti-debugging several times in different parts of the application to protect. Therefore, he was expecting a less extensive use of anti-debugging. This is another confirmation that we need to provide another level of weights in the system apart from the effectiveness evaluations in the PI description, to allow users to play and fine tune the ADSS behaviour on their scenarios. The fear of too high performance degradation was real, however, the ADSS was considering it in the computations, based on the PI information.
- *over-representation of Static Remote Attestation:* the golden combination included 9 occurrences of Static RA (compared to only 7 occurrences of Binary Obfuscation). These repetitions can be justified by the fact that Static RA costs are almost constant with the number of code area that are checked for modifications (only the average frequency of attestation of an area needs to be tuned properly). Moreover, there were

several areas to protect from tampering and the Static RA was the only ADSS-supported anti-tampering technique. The expert accepted the explanation as valid. However, the comment was also positive as he noted that the golden combination (L1P), included:
  - Static Remote Attestation (high frequency) @ build_payload.r19
  - Static Remote Attestation (high frequency) @ init_payload.r18
  - Static Remote Attestation (high frequency) @ perform_KDF.r6

  and it was judged as appropriate. He also noted that these functions are located in that part of the code that requires communication with the application server and feared about latency (which does not happen as RA is performed in a separate thread and it does not stop the program's execution).
- *comments on binary obfuscation*: only flattening is specified while for some portions of code opaque predicate should be preferred. This issue was also noted in the first iteration and was not solved because of lacking of metrics able to allow the ADSS to select a specific binary obfuscation technique.
- *improvements made in the proposed golden combination*: quoting "Very meaningful code mobility protections! At least these two:
  - Code Mobility @ derive_storage_key.r14
  - Code Mobility @ OATH_HOTP_generate.r1"
- *very significant improvement regarding SoftVM*: in the golden combination SoftVM is proposed on purpose and sparingly as he would have expected. Quoting "I'm happy".
- *missing technique*: as in the previous report, he was expecting to have Data Obfuscation, which is not supported by the ADSS.
- *balanced golden combination*: taking the most ranked asset (with weight = 2000), quoting "I like the proportion: one SoftVM, one anti-debugging and 9 Binary obfuscation for this exposed code, this is balanced and realistic to me".
- *improved naming and self-explanatory descriptions*: much better label names in Attack & Protections ("Attacker Benefit" and "Expected Maximum Mitigation" are self explicit). In Level 1 Protections the "golden solution" is also named "best", which is good since the initial name was conventionally used but with ASPIRE project internal connotation.

The external expert reported:

- *not self-explanatory information in the report*: the list of the assets may be puzzling as, in the ADSS report, line numbers and asset dependencies are reported in the asset name. This feature has been corrected in the tool but not in the report. Application parts that are not assets are reported with weight=0, this was judged as confusing. Furthermore, label that report the effectiveness of single protections (e.g., the "high" in green) should be put before, not after the suggested application name. Moreover, some column names and attack path/steps data are intuitive but some others would need more expertise/training. Finally, in Level 2 protections, the comment about the order of protections was in his opinion unclear.
  We corrected it and added more information.
- *weights*: the weights assigned to assets were defined as appropriate, especially he appreciated the idea to share weight shared for a family of assets, quoting "acknowledged choice!".
- *assets visualization*: default visualization should be sorting assets by degrading weight order. The feature indeed is available but not the default.
- *order of visualization of the info in the report*: the suggested order would be (1) application parts and their weights, (2) ADSS result summary (i.e., the golden combinations), (3) other details (attack paths) and L2P.
- *overheads*: more information must be shown about the overhead: it would be appreciated indicating the estimated overhead for each golden combination for each category of overhead (code size, data segment, size increase, CPU).

- *good golden combination*: quoting “Like the Best section! Should be pondered with criteria such CPU overhead and/or size overhead”
- *Level 2 Protection replication*: repetitions can be confusing for attackers, as they can interpret them as strange patterns in the code that occurs several times as not to be analyzed in a first step. Indeed, critical protection are usually not replicated everywhere. The external expert also reported that “typically, SoftVM is good to be replicated BUT the same SoftVM protection might be more easy to analyze if it is applied on some not meaningful portion of code. Then this might weaken the protection at the end of the day”. As a reaction, we have extended the AKB in order to associate additional information about the protections: independence or dependence on the “quantity” and “quality” of the code.
- *positive general evaluation of the ADSS*: quoting “Overall evaluation: not bad! "Best" protections part is very good stuff”.

In the end, the external expert expressed the wish to have the tool with the apply button to generate some selected annotations in the code and to test.

9.4.3 Tiger team analysis

The analysis of the tiger team report shows that the ADSS was able to identify the attack actually ported by the hacking team against the master key, that is:

staticallyLocate('build_master_key.r8'(attacker))

where build_master_key.r8 is the region 8 of the function build_master_key function, that is, the master_key variable.

Moreover, the ADSS suggested the use of dynamic techniques to couple with the anti-static analysis technique in order to prevent this attack path, exactly as declared by the tiger team. The list of suggested protections is reported below:

- SoftVM @ build_master_key.r8 [medium]
- Anti-Debugging @ build_master_key.r8 [medium]
- Binary Obfuscation (branch function, medium overhead) @ build_master_key.r8 [high]
- Binary Obfuscation (flatten function, medium overhead) @ build_master_key.r8 [high]
- Binary Obfuscation (flatten function, low overhead) @ build_master_key.r8 [high]
- Binary Obfuscation (branch function, high overhead) @ build_master_key.r8 [high]
- Binary Obfuscation (branch function, low overhead) @ build_master_key.r8 [high]
- Binary Obfuscation (flatten function, high overhead) @ build_master_key.r8 [high]
- Binary Obfuscation (opaque_predicate, low overhead) @ build_master_key.r8 [high]
- Binary Obfuscation (opaque predicate, high overhead) @ build_master_key.r8 [high]
- Binary Obfuscation (opaque predicate, medium overhead) @ build_master_key.r8 [high]

From this analysis, we infer that the ADSS was able to discover the attack path actually used by the tiger team and proposed already valid suggestions on how to counter the attack. Moreover, the golden combination extensively used anti-debugging.

The rest of the considerations in the tiger team report where about the design of the application itself, suggestions on how to protect Java code, and reported about the attack against the anti-debugging protection. Therefore, they were not processed for this validation.

9.4.4 Conclusions

The overall evaluation of the ADSS is positive. They acknowledged that the information shown by the ADSS is useful to have a better idea of the protection process (assets, weights, application parts). They acknowledged that the information automatically extracted by the ADSS is valuable and would require a lot of manual effort by the protection experts. After some iterations and interactions to fine tune the internal ADSS estimations of the protections effectiveness, they were also satisfied by the golden combination (reasonable protections, good stuff) computed by the ADSS. Experts did not object on the correctness of the results automatically deduced by the ADSS and on the correctness and validity of the golden combinations. They were not able to express judgments on the optimality.

They were not satisfied by the limitations of the ADSS used for the validation, as some tamper detection and some data obfuscation techniques were not supported. Moreover, since some metrics (mainly dynamic) useful to make decisions are not extracted, the ADSS was not precise enough when selecting the most appropriate way to apply some protections (flattening vs. opaque predicates for binary obfuscation).

Moreover, they provided useful suggestions to bridge the gap between the ASPIRE terminology and terms used by experts and useful suggestions to improve the usability of the tool and of the reported information.

9.5 Summary of ADSS validation conclusions

After the analysis of the validation data, the ASPIRE consortium concluded that the ADSS has a very high potential even if it is not yet ready to be used to protect real applications. First of all, it can automate operations (like attack path discovery and threat analysis, and identification of suggested protections) that require software experts a huge effort.

After the validation of the experts, attack paths identified by the ADSS seem exhaustive, however, they are not fine grained enough to make the very precise decisions that may be needed in case of application that have to work with limited resources. Moreover, there is also another psychological aspect to consider. Experts looking at the identified attack paths would have preferred having more fine-grained results not only to validate the outcomes but also to build trust in the ADSS. Further research, interactions with experts, and improvements are expected in this field.

The Level 1 Protections as well as the Level 2 Protections have been rated as correct, free from evident incompatibilities or other issues that could lead applicability problems, very meaningful and effective to mitigate the risks against the assets. Limitations have been noted because the ADSS did not support, at the moment of the validation, all the ASPIRE protection techniques.

Unfortunately, experts were not able to state anything about the optimality of the golden combinations and their L2P extensions.

Several concerns have been addressed and remain about the presentation of the results. First, we need to use more general terms and not only the ones that are common in the ASPIRE project. Then, we need to show the users more explanations on the process and we have to output more precise information on the decision process. In particular, the most important part that is missing is the output of the process that estimates the effect of the application of combinations of protections on the application assets, especially in terms of performance degradation introduced by the application of the protection techniques.

Section 10 Conclusions

Section authors:
Bjorn De Sutter (UGent)

To start with the bad news, one protection, namely multi-threaded cryptography, researched in the project was found to be unusable in practice, because its deployment cannot be automated.

Fortunately, the good news is more comprehensive.

In Section 2, the ASPIRE consortium concluded that (i) the technology it develops to protect mobile applications still covers the relevant Man-At-The-End attacks; (ii) the proposed requirements stand; (iii) the ASPIRE reference architecture suffices to deliver the protections as specified in the DoW, and thus provides an excellent vehicle for providing adequate protection at least on the project use cases, and most likely beyond those cases to a broad range of mobile applications.

In Section 3, the ASPIRE consortium concluded that the project results meet almost all requirements. In particular, the most important ones are met.

In Section 4, the ASPIRE consortium concluded that, with the exception of some data obfuscations and multi-threaded cryptography, all protections developed during the project are covered by the validation of the use cases.

In Section 5, the ASPIRE consortium concluded the following:

- The envisioned, useful protections for the NAGRA use case are available in the ASPIRE protection framework, with the exception of interprocedural data protections. Control flow tagging was delivered too late for deployment and validation on the use case, however.
- The envisioned, useful protections for the SFNT use case are available in the ASPIRE protection framework, with the exception of interprocedural data protections.
- The envisioned, useful protections for the GTO use case are available in the ASPIRE framework.

In Section 6, the ASPIRE consortium concluded that all the protections supported in the ACTC and listed as potentially useful for the use cases can be deployed on those use case, given a slight redesign.

In Section 7, the ASPIRE consortium concluded that although many of developed and integrated protections still offer a large potential for improvement, those protections

- effectively delay attacks and increase the effort that attackers need to invest in identifying attack vectors;
- make it harder to exploit identified attacks at a large scale;
- and hence effectively reduce the profitability of engineered attacks.

It also concluded that it can conjecture that the developed and integrated renewability techniques deliver improved protection, as they make the scaling up of attacks harder, as they can delay attack vector identification, and as they can help in raising the costs faced by attackers.

To a large degree, the project has hence achieved its goals to demonstrate that software-based protection techniques can deliver true protection.

In Section 8, the ASPIRE consortium concluded that the overhead they introduce is limited and can be controlled by deploying the protections cautiously. In particular, on the project use cases, the protections could be deployed as foreseen by their developers and the project's security experts, with an acceptable overhead.

In Section 9, the ASPIRE consortium concluded that the ADSS has a very high potential even if it is not yet ready to be used to protect real applications. On one hand, it can automate complex operations (discover attack paths, suggest protections), it gathers a huge amount of data useful for making decisions, and it propose effective protections. On the other hand, in some cases, the level of details of the output is not yet enough to be used in practice (attack paths description are too coarse grained), it was not possible to prove that the golden combinations are actually optimal, and there are still concerns about the presentation of the results, especially outside the ASPIRE project.

Section 11 List of Abbreviations

3G	Third Generation
3SAT	3-Boolean Satisfiability Problem
ACTC	ASPIRE Compiler Tool Chain
ADSS	ASPIRE Decision Support System
AES	Advanced Encryption Standard
ADS	Attestation Data Structures
AKB	ASPIRE Knowledge Base
API	Application Programmer Interface
ARM	Not an acronym, name of a company and its processor architecture
ASR	Assurance Security Requirement
ASPIRE	Advanced Software Protection: Integration, Research and Exploitation
CF	Control Flow
CFG	Control Flow Graph
CPU	Central Processing Unit
DB	Database
DCL	Diversified Cryptography Library
DDR	Double Data Rate
DoW	Description of Work
DRM	Digital Rights Management
FSR	Functional Security Requirement
GHz	Gigahertz
GB	Gigabytes
GMRT	Global Mobile Redirection Table
HSDPA	High Speed Downlink Packet Access
IDA	Interactive Disassembler
kB	kilo bytes
L2P	Level 2 Protections
LAN	Local Access Network
LLVM	Low-Level Virtual Machine (but now simply the project name)
LOC	Lines of Code
MHz	Megahertz
MTD	Multi-Threaded Crypto
NSR	Non-functional Security Requirement

OTP	One-Time Password
PIN	Personal Identification Number
PKI	Public Key Infrastructure
RA	Remote Attestation
REC	Recommendation
REQ	Requirement
RQ	Research Question
RAM	Random Access Memory
SPRO	Software Protection
SoftVM	Software Virtual Machine
TCAS	Traffic Collision Avoidance System
TCP	Transmission Control Protocol
WBC	White-box Cryptography
WPx	Work Package x
XOR	Exclusive Or

References

[Cop13a] 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

[Cop13b] 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

[Wys13] Brecht Wyseur, “White-Box Cryptography”, International Summer School on Information Security and Protection 2013, ISSISP 2013, Xian, China, August 2013.