



Advanced Software Protection: Integration, Research and Exploitation

D4.06 **ASPIRE Security Evaluation Methodology**

Project no.:	609734
Funding scheme:	Collaborative project
Start date of the project:	November 1, 2013
Duration:	36 months
Work programme topic:	FP7-ICT-2013-10
Deliverable type:	Report
Deliverable reference number:	ICT-609734 / D4.06
WP and tasks contributing:	WP 4 / Task 4.1, Task 4.2, Task 4.3, Task 4.4, Task 4.5
Due date:	October 2016 – M36
Actual submission date:	To be filled out by coordinator
Responsible Organization:	FBK
Editor:	Mariano Ceccato
Dissemination level:	Public
Revision:	v1.0

Abstract:

This deliverable presents the versions at M36 of the ASPIRE security model, the input/output model of the metrics framework and of the Security Protection Assessment tool, the results of empirical academic studies conducted in the M30-M36 period, the results of the industrial studies, and the result of the public challenge.

Keywords:

ASPIRE knowledge Base, security evaluation, empirical studies



Editor

Mariano Ceccato (FBK)

Contributors (ordered according to beneficiary numbers)

Bjorn De Sutter, Bart Coppens (UGent)

Cataldo Basile, Daniele Canavese, Leonardo Regano, Marco Torchiano (POLITO)

Brecht Wyseur (NAGRA)

Mariano Ceccato, Paolo Tonella (FBK)

Paolo Falcarin, Gaofeng Zhang (UEL)

Michael Zunke, 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 no. 609734. The information in this document is provided as is and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

Executive Summary

This deliverable presents the final status of WP4 activities and the outcomes at M36. Achievements are divided in four parts: the first part presents the final ASPIRE Security Model, the second part presents the final security evaluation including the metrics framework and the security evaluation with ADSS-Light, the third part presents the results of the last replications of academic user studies and the results of industrial user studies, and, eventually, the fourth part presents the results of the public challenge.

In task T4.1, the ASPIRE Security Model v1.2 (ASMv1.2) was already released at M30 and presented in Deliverable D5.11. Anyway, even if at M36 there are no changes in the AKB with respect to what already presented at M30, it is presented also in this deliverable as the final and unique reference document to be used in ASPIRE. ASMv1.2 is composed of a main model and several sub-models for *application*, *assets*, *software protections*, *attacks*, *metrics* and *protection requirements*.

In Task 4.2, the final security evaluation methodology includes the final metric framework and the final Software Protection Assessment tool. The Goal-Question-Metrics approach has been instantiated to define the metric framework. It includes:

- A clear definition of the *goals* that we want to achieve, i.e. to measure the increased effort needed to attack protected code, and the performance overhead caused by protections;
- The *questions* that we want to answer with metrics, they descend directly from goals;
- The abstract *measurable features*, i.e. the properties of the code that are relevant to answer these questions;
- Eventually, the *metrics* that best approximate and quantify these features.

We proposed a catalogue of metrics to measure complexity and resilience of code along many relevant dimensions, either from a static and dynamic point of view, in order to capture properties that are relevant both to attacks that adopt static and dynamic analysis. All these metrics are implemented and available in the ACTC to reason and make informed decisions on protection configurations.

The final security assessment methodology is presented in this deliverable and implemented by the ADSS-Light: this tool contains the PN editor and the Software Protection Assessment component which utilizes the code metrics computed by the ACTC to estimate the security strength of a protection configuration. The ADSS-Light supports the security analyst by inspecting, comparing and deciding among alternative protection configurations. The analyst's preferences and decisions are merged with existing knowledge to support the final decision-making on the configuration of protections. This deliverable then describes the final architecture of the ADSS-Light and its internal workflow. Moreover, detailed instructions are reported on how to install and configure it. The ADSS-Light tool is then applied on the NAGRA case study, by modelling and configuring the attack using as an input the corresponding hacking experiment report (the results of the NAGRA hacking experiment will be presented in detail in Section 9).

In task T4.3, the ASPIRE user studies with academic subjects have been completed. Two remaining replications of the experiment with Client/Server Code Splitting have been done with attack tasks on source code. A replication involved 17 Master students and 2 PhD students in FBK. The last replication involved 86 Master students from Politecnico di Torino. These replications showed that this technique has a dramatic impact on success rate and on the time necessary to complete a successful attack (success rate reduced by a factor of 2).

This quantitative evidence proves that splitting the program between client and server is a powerful deterrent of malicious tampering, because it turns attacks less economically convenient. This is also confirmed by insights on attack strategies. Successful attacks were performed using an iterative and incremental approach, where understanding of protected code proceeds by chunks,

rather than as a monolithic process. After the attackers consider to have acquired enough understanding of a portion of code, trial-and-error tampering is applied to test this knowledge. This process is iterated, until the full attack goal is iterated.

In task T4.4, the ASPIRE user studies with industrial participants have been completed. Three independent studies have been conducted on the three industrial partners.

The first experiment has been conducted at Nagra, to attack the protected version of their case study, the DemoPlayer. First of all, analysts acknowledged that attacks would have been trivial on the unprotected program. Some layer of protection could be defeated, but the attack could not complete in the allowed amount of time. So we can conclude that ASPIRE protections were effective in delaying attacks and in turning them more expensive. We also observed that attacks are quite adaptive, because analysts make assumptions on the code and on protections, they continuously change their attack strategy based on the obstacles encountered. Analysts prefer to identify alternative attack paths than directly defeating protections. This is the way ASPIRE protection was effective in delaying attacks.

The second experiment was run in SafeNet, to attack the license in their case study. Also in this experiment, ASPIRE protections proved to be effective, because analysts had to spend time to defeat protections before accessing the actual assets, objectives of the attack. Based on the detailed results of this experiment, we collected crucial feedback on how to improve some protections:

- Improving control flow obfuscation, to make it harder to be removed by static analysis that resorts to state-of-the-art analyses, such as symbolic execution;
- Make VM-obfuscation more resilient, by using custom and unknown bytecode representation that is not documented, such as diversified bytecode techniques;
- Stronger integration of different protections, for example remote attestation and anti-debugging, to protect each other and not offer the attacker a single attack starting point;
- Extend the boundary of protections not just to assets but as large as possible, in order to avoid that an attacker can easily analyze some parts of the code and reuse them out of context.

The third and last experiment was run in GemAlto, to attack their One Time Password generator case study. Also in this case, ASPIRE protections proven to be useful to delay attacks. In fact, the analysts could not defeat all the protections in the allowed amount of time. From this experiment, we learned some guidelines on how to improve protections and protection configurations. The protection boundaries should be enlarged as much as possible and consider the complete protocol layer, e.g. they should not be limited to the native part of an Android app, but the Java part should be also protected. Protections can be undone by advanced custom attack tools, that might be available to expert hackers. Anyway, these tools require peculiar expertise and knowledge to be adopted successfully.

In Task 4.5, the public challenge has been prepared and successfully run. The public challenge consisted in publishing a set of protected programs in the ASPIRE website, with a bounty for those who could attack them. Different configurations of protections have been chosen, to study both the single protection in isolation and combinations of them. The challenge has been publicized along different channels to maximize the participation. We monitored the evolution of interest in the challenge by tracking the number of visits to the challenge website with using Google Analytics and by observing the rate of creation of new user id to download the challenges.

At the conclusion of the challenge, the hacker who was able to defeat protections has been interviewed. This revealed weak points of protections that represents opportunities for future improvements. The attack made extensive use of IDA-Pro to analyze binary code, with the help of custom analysis scripts built on top of it, in particular to remove control flow obfuscation and obtain a clear program to attack and to remove anti-debugging from the code (after extensive and time consuming understanding). When anti-debugging was defeated, other protections could be

attacked dynamically, by setting specific breakpoints to (i) skip checksum verifications by code guards and (ii) to dump bytecode in VM-obfuscation. Once dumped, bytecode was then subject to offline analysis to eventually solve the challenge.

Contents

1	Introduction	1
I	The final ASPIRE Security Model	3
2	The final ASPIRE Security Model	3
3	The ASPIRE security model	3
3.1	The ASPIRE security model v1.2: main model	3
3.2	Model Extensions: the Sub-Models	7
3.2.1	Application sub-model	7
3.2.2	Assets sub-model	13
3.2.3	SW Protection sub-model	14
3.2.4	Attacks sub-model	18
3.2.5	Metrics sub-model	20
3.2.6	Protection requirements sub-model	21
II	Security Evaluation	23
4	Metrics Approach	23
4.1	Goals of the Metrics	23
4.2	Questions to Consider for Approximating Attack Effort	24
4.2.1	Attack properties that influence attack effort	25
4.2.2	Classifying and unifying the relevant properties	27
4.2.3	Attack fundamentals: syntax and semantics	28
4.3	Measurable Features	32
4.3.1	Measurable Complexity Features	33
4.3.2	Measurable Resilience and Stealth Features	34
4.3.3	Measurable Unavailability Features	35
5	Concrete Metrics	36
5.1	Complexity Metrics	36
5.1.1	Static Code Size	36
5.1.2	Dynamic Code Size	36
5.1.3	Static Control Flow Complexity	36
5.1.4	Ill-Structuredness	37
5.1.5	Dynamic Control Flow Complexity	38
5.1.6	Code Layout Variability	39
5.1.7	Static Data Flow Complexity	39
5.1.8	Dynamic Data Flow Complexity	40
5.1.9	Semantic Dependencies	41
5.1.10	Static Data Presence	42
5.1.11	Dynamic Data Presence	42
5.1.12	Syntactic Stubbornness	43
5.2	Resilience Metrics	43
5.2.1	Static Variability	43
5.2.2	Intra-Execution Variability	43
5.2.3	Semantic Relevance	44
5.2.4	Stealth	45
5.3	Unavailability Metrics	45
5.4	Using the Metrics in Practice	45

5.5	Concrete Metrics Support	46
5.5.1	Metrics integrated in the ACTC	46
5.5.2	Metrics not integrated in the ACTC	47
6	Software Protection Assessment with ADSS-Light	49
6.1	The ADSS-Light Architecture	49
6.1.1	Protection Fitness formulas	49
6.1.2	Normalized Protection Fitness	51
6.2	ADSS-Light Workflow	52
6.3	ADSS-Light: Eclipse Plug-ins	55
6.3.1	Installation and Configuration	56
6.3.2	Creating an ADSS-Light project	57
6.3.3	API of the Software Protection Assessment module	58
6.3.4	ADSS-Light User Interface	59
6.4	Conclusions and Future Extensions	60
III	Experiments	62
7	Client/server code splitting experiment	62
7.1	Experimental Definition	62
7.2	Research questions	62
7.3	Object	63
7.4	Analysis of Runtime Overhead	64
7.5	Metrics	64
7.6	Design	65
7.7	Statistical analysis	65
7.8	Participants Characterization	66
7.9	Analysis of Success Rate	68
7.10	Co-factors of Success Rate	69
7.11	Analysis of Attack Time	70
7.12	Co-factors of Attack Time	71
7.13	Analysis of post-questionnaire	71
7.14	Threats to validity	72
7.15	Lessons Learned	74
8	Common Design of Experiments with Industrial Participants	75
8.1	Experimental Definition	76
8.1.1	Object	76
8.1.2	Data	77
8.1.3	Design	79
8.1.4	Qualitative analysis	79
8.1.5	Threats to validity	79
9	Experiment in NAGRA	80
9.1	Protection Configuration	80
9.2	Asset Information	81
9.3	Experimental Settings	81
9.4	Results	81
9.4.1	Phase 1: White Box Cryptography	81
9.4.2	Phase 2: Anti-debugging	82
9.5	Observations	82

10 Experiment in SafeNet	83
10.1 Protection Configuration	84
10.2 Experimental Settings	84
10.3 Results	84
10.3.1 Phase 1: Internal Evaluation in SafeNet	85
10.3.2 Phase 2: External Evaluation by Professional Hackers	85
10.4 Observations	86
11 Experiment in Gemalto	88
11.1 Protection Configuration	88
11.2 Experimental Settings	89
11.3 Results	89
11.3.1 Phase 1: External Evaluation, DCL protection	89
11.3.2 Phase 2: Internal Evaluation, First Configuration of Protections	90
11.3.3 Phase 3: Internal Evaluation, Second Configuration of Protections	91
11.4 Observations	91
IV Public Challenge	93
12 Public challenge	93
12.1 Promotion	93
12.2 Evolution of interest in the challenge	94
12.3 Exit interview with the hacker	96
12.4 Lessons Learned	98

List of Figures

1	The main model.	5
2	The applications sub-model.	8
3	The ApplicationParts package.	9
4	The Representations package.	11
5	The Compilers&Linkers package.	12
6	The Communications package.	13
7	The ExecutionEnvironments package.	14
8	The asset sub-model.	15
9	The SW protections sub-model.	16
10	The annotation part of the SW protections sub-model.	17
11	The attacks sub-model.	18
12	The attacks, goals and Petri nets.	19
13	The metrics sub-model.	20
14	The protection requirements sub-model.	21
15	ADSS-Light architecture	50
16	Example of Petri Net with two attack paths	51
17	ADSS-Light Workflow	53
18	PN Generation	56
19	ADSS-Light Plug-ins	57
20	ADSS-Light Project Creation	58
21	Screen-shot of the ADSS-Light tool	59
22	Screenshot of SpaceGame	63
23	Demographics of participants in Trento.	67
24	Bar plot of attack success rate in Trento (red=successful attack, yellow=wrong attack).	68
25	Bar plot of attack success rate in Torino (red=successful attack, yellow=wrong attack).	68
26	Boxplot of attack time in Trento (only for successful attacks)	70
27	Boxplot of attack time in Torino (only for successful attacks)	71
28	Post-questions answered by Trento's subjects.	73
29	The number of website sessions per week.	94
30	The number of new users per day.	95
31	Number of sessions per country, limited to countries with at least 10 sessions.	95

List of Tables

1	Summary of the original sources of the final ASPIRE Security Model content.	3
2	Code splitting experiment	62
3	Execution times for split code.	64
4	Design for the code splitting experiment: group G1 is assigned object P1 in its original form, while group G2 is assigned P1 protected with code splitting T1, in a single lab (Lab1)	65
5	Success Rate in Trento.	69
6	Success Rate in Torino.	69
7	General linear model of Success Rate.	69
8	Descriptive statistics of Attack Time in Trento.	70
9	Descriptive statistics of Attack Time in Torino.	71
10	General linear model of Attack Time.	72
11	Nagravision case study	76
12	SafeNet case study	77
13	Gemalto case study	78
14	Size of case study objects (measured by <code>sloccount</code>), divided by file type.	78

1 Introduction

Section authors:

Mariano Ceccato (FBK)

The goal of this deliverable (see GA Annex II DoW part A) is to document the final achievements of ASPIRE Work Package 4 at the end of the project at M36, including the security model, the security evaluation, the experiments with academic and industrial participants and the public challenge.

WP4 created the support needed to allow the ASPIRE Decision Support System (ADSS) to make an informed decision on the best combination of protections for protecting the assets in the application from attacks. The final WP4 outcomes include the ASPIRE security model, the ASPIRE knowledge base, the metrics framework, and the evaluation techniques based on analytic methods (fitness function) and simulation models (Petri nets). Moreover, WP4 includes the evaluation of protection techniques in the field with attackers, with experiments in academic setting and in industrial settings, and with public challenges with bounty. Involving real human beings in attacking protected code is essential to check if (ex-post/ex-ante) estimates on the protection level are correct, to find weak points in protections and to identify promising opportunity for improvement.

Therefore, this deliverable presents:

- The final ASPIRE Security Model (developed in T4.1 “Security Model and Evaluation Methodology”), which allows the formal representation of the ASPIRE concepts, stored by the ASPIRE Knowledge Base in a way to perform the sophisticated reasoning used by the ADSS;
- The final metrics framework (developed in T4.2 “Complexity metrics”), designed to evaluate software protection strength through the use of software complexity and protection resilience metrics;
- The security assessment methodology implemented by the ADSS-Light tool (developed in T4.1 and T4.2), by combining Petri Net attack models and code metrics;
- The remaining two replications of the second round of experiment with academic participants, to evaluate the effectiveness of the ASPIRE protections in measurable and artificial settings (conducted in T4.3 “Experiments with academic subjects”);
- The three experiments with industrial hackers, to assess the effectiveness of combinations of ASPIRE protections in real programs and real settings (conducted in T4.4 “Experiments with industrial tiger teams”);
- The results of the public challenge, to assess the effectiveness of ASPIRE protections publicly, potentially by attackers with any level of expertise and background (conducted in T4.5 “Public challenge”).

This deliverable is organized as follows. Part I presents the final ASPIRE Security model. To help tracking the history of this document, Section 2 presents a detailed mapping where all the components (i.e., sub-models) have been defined in previous deliverables. Then, Section 3 reports the final version of the ASPIRE Security Model (ASMv1.2).

Part II presents the updates to the ASPIRE Security Evaluation. Section 4 presents the theoretical metric framework based on the Goal-Question-Metric methodology, and Section 5 presents the final concrete metrics that have been implemented to measure the effect of protections. Section 6 reports the updates on the ADSS-Light and it shows how to apply the security evaluation

on the NAGRA case study, showing how to integrate in the security decision process all those information that have been collected in the corresponding hacking experiment.

Part III reports the results of the empirical studies conducted to evaluate the effectiveness of the ASPIRE protections. Section 7 presents the remaining academic studies designed to assess the strength of Client/Server Code Splitting, conducted at FBK and at POLITO. Section 8 reports the planning of the industrial studies, which have been conducted by the industrial partners. Results of these experiments are presented and discussed in Section 9 for NAGRA, in Section 10 for SafeNet and in Section 11 for GTO.

Eventually, Part IV presents the public challenge. Section 12 presents how the results of the public challenge and the data collected with the exit interview to who proved to have solves the challenges.

Part I

The final ASPIRE Security Model

2 The final ASPIRE Security Model

Section authors:

Cataldo Basile, Daniele Canavese, Leonardo Regano (POLITO)

This part reports the final version of the ASPIRE Security Model, the ASPIRE Security Model v1.2 (ASMv1.2). This deliverable does not present novel content on the ASPIRE Security Model, which has been previously delivered with D4.01 (ASMv1.0), D4.03 (ASMv1.1), and D4.04 (the updated in ASMv1.2). The purpose of this deliverable part is to have a final and unique reference document about the ASPIRE Security Model. Table 1 summarizes the original documents where the main model and of each submodel have been first presented. Further design principles and explanations can be found in the deliverable D4.01, D4.03, and D4.04.

(Sub)Model	last update	Deliverable
Main model	ASMv1.1	D4.03
Applications	ASMv1.1	D4.03
Assets	ASMv1.2	D4.04
SW Protections	ASMv1.2	D4.04
Attacks	ASMv1.1	D4.03
Metrics	ASMv1.1	D4.03
Protection requirements	ASMv1.1	D4.03

Table 1: Summary of the original sources of the final ASPIRE Security Model content.

We also confirm that there are no changes in the ASPIRE Knowledge base (AKB) since the ASMv1.0. The AKB purpose and organization of the information is unchanged as well and the representation, which is an OWL-DL ontology. Finally, the AKB supports the Enrichment Framework and Enrichment Modules needed to the ADSS to work, which will be reported in the deliverable D5.11.

3 The ASPIRE security model

Section authors:

Cataldo Basile, Daniele Canavese, Leonardo Regano (POLITO)

3.1 The ASPIRE security model v1.2: main model

This section presents a class diagram that represents a formalization of the attack model information represented in D1.02. That attack model included the following high-level concepts and informal relations between them:

- Application;
- Asset;
- Attack;
- Attack path;

- Attacker;
- Software protection;
- Tools.

Figure 1 presents the UML class diagram that shows how these concepts have been formally related. It is worth noting that associations are all many-to-many associations unless differently noted in the text.

First of all, we present the `Application` class. Its instances will be objects abstracting the applications to protect. Next, the class `Asset` describes the assets. A preliminary set of assets has been listed in Section 3 of D1.02. The identified assets in our formal model will be detailed in Section 3.2.2. Presently, the set of assets we identified suffices to describe the types of assets listed in D1.02. Nevertheless, we do not exclude the possibility of updates.

`Application` instances are associated to at least one `Asset` instance by means of the `contains` association. In most cases, we must consider not the whole application but one of its parts. For this purpose, we introduced the `ApplicationPart` class, whose instances may describe logically self-contained components (like server and client stubs, algorithms) or simple pieces of code or similar abstractions (like fragments and slices). `ApplicationPart` instances are connected to the `Application` instances they are part of by means of the one-to-many `hasPart` association. Additionally, `ApplicationPart` instances are associated to `Asset` instances by means of the `partContains` association. More details on `ApplicationPart` class, its subclasses and associations will be presented in Section 3.2.1, where the `Application` sub-model is presented.

`Assets` instances are associated to several security properties a user may be interested in when protecting the asset. This information is conveyed via the `AssetProperty` class instances (named threats in D1.02) that are associated to `Asset` instances by means of the one-to-many `hasProperty` association. Additionally, assets may depend on other assets. The (self) association `requires` is used to represent this dependency.

Attacks are represented as instances of the `Attack` class. Several attacks may threaten an asset, this scenario is represented by means of the `threatens` association. Additionally, attacks may compromise asset security properties, this is represented by means of the `affects` association. Having modelled the attacks, we are able to model attack paths, and naturally, we use the `AttackPath` class for this purpose. To represent that an attack can be mounted following an attack path we use the `hasAttackPath` association that relates `Attack` instances to `AttackPath` instances (one-to-many). Attack paths are decomposed in individual attack steps that are represented as `AttackStep` instances. In theory, each attack step can be an entire attack on its own. We depicted this scenario by means of the inheritance, that is, `Attack` is a subclass of `AttackStep` (bounded via an 'is a' association). This also means that we can describe steps that are not attacks per se (as do not affect any asset property) as they are only needed within an attack path. Each `AttackStep` instance is associated to the tools that can be used to actually mount it. Attack tools are modelled with the `AttackTool` class and `AttackTool` instances are associated to `AttackStep` instances via the `maybePerformedByMeansOf` association. Since attack steps can be shared among different attack paths, we used the `AttackStepInAttackPath` (association) class, which links the `AttackStep` instance via the `refersToAttackPath` association and the `AttackPath` via the `refersToAttackStep` association. The last two associations bind an `AttackStepInAttackPath` instance to exactly one `AttackStep` instance and one `AttackPath` instance. Then, to describe consecutive attack steps we used the `nextStepANDed` and `nextStepORed` associations that serve to also indicate if the consecutive steps must be all executed or at least one must be executed. The first step of the attack path is indicated by setting up the `startsWith` association. It is worth noting that `nextStepANDed`, `nextStepORed`, and `startsWith` are many-to-many associations as many parallel steps can be performed at the same time, that is, these associations allow us to model graphs of attack steps (and Petri nets), not only sequences (see Section 3.2.4).

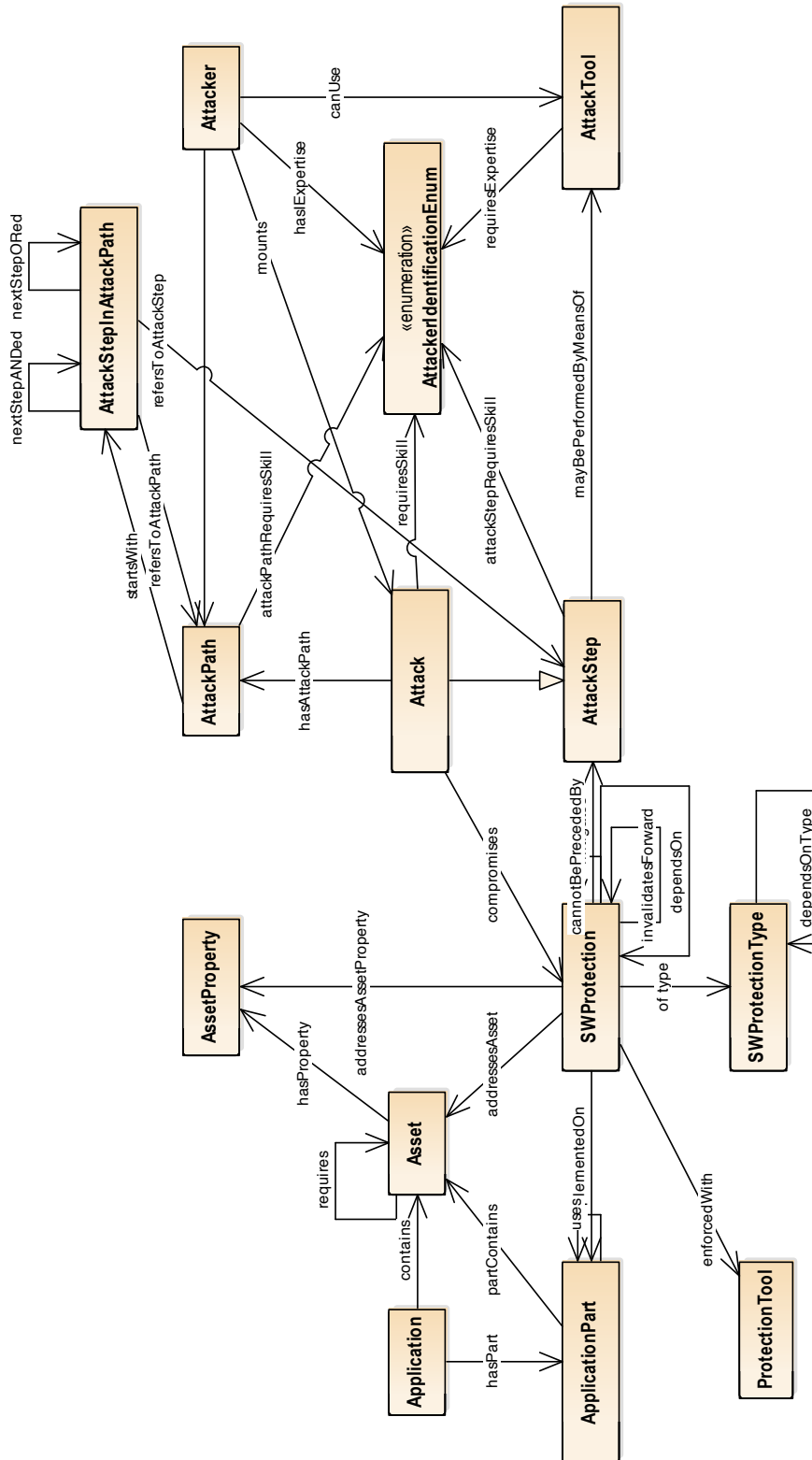


Figure 1: The main model.

Attacks are categorized by the level of exploitation (see Section 4.2 of D1.02). As presented in D1.02, attacks may require different expertise, skill and resources to be mounted, and attacks may have different levels of exploitation. D1.02 introduced the “identification” concept (and distinguished four categories of attackers: gurus, experts, geeks, and amateurs), and the “exploitation” attribute (and distinguished three categories of exploitation: low, medium, high). We model these formally with two enumerations: the `AttackerIdentificationEnum` class with the four identified attacker categories, and the `ExploitationEnum` class with the three identified exploitation levels. We do not expect changes to these enumerations, however it is worth noting how easy it is to extend or modify these classifications.

`AttackStep` instances (and `Attack` instances as well due inheritance) are associated to values of the `AttackerIdentificationEnum` enumeration by means of the `requiresSkill` association and to `ExploitationEnum` values by means of `hasExploitation` association. An attack step is associated to exactly one `AttackerIdentificationEnum` value and exactly one `ExploitationEnum` value. Also `AttackPath` instances and `AttackTool` instances are associated with `AttackerIdentificationEnum` instances via the `attackPathRequiresSkill` and `requiresExpertise` associations.

Software protections are described by means of instances of the `SWProtection` class. To refer to the tools actually deploying protection (which we aim at configuring as output of ASPIRE DSS), `SWProtection` instances are associated to `ProtectionTool` class instances by means of the `enforcedWith` association. To indicate possible dependencies among software protections, the `dependsOn` association is used, while forward incompatibility is represented through the `invalidatesForward` association. That will allow us to model when one protection cannot be enforced after another one because it renders the previous one useless or wrong, like obfuscating code after having inserted guards, or when the first protection invalidates the preconditions to apply the later one. Software protections are categorized in types by associating them to `SWProtectionType` instances via the `ofType` association. A software protection is associated to exactly one `SWProtectionType` instance.

Together with the need of categorizing protections, this class serves to map the ‘lines of defence’ concept, as presented in the DoW (see also Section 3.2.3). Another reason for having this class is that it will allow class-level reasoning about protections, i.e., to answer questions like “which are the categories of protections that can be used in combination with strengthen local integrity protections to increase the protection level?”. Dependencies among `SWProtectionType` instances are represented by means of the `dependsOnType` association. `SWProtection` instances are related to the `Asset` instances (via the `addressesAsset` association) and `AssetProperty` instances (via the `addressesAssetProperty` association) that they may mitigate risks on, and to the `Attack` instances that they aim at invalidating or making more difficult (via the `mitigates` association). By means of the `compromises` association, attacks are associated to the `SWProtection` instances they can invalidate (or render useless or completely remove).

Attacks are mounted by attackers, represented by instances of the `Attacker` class, which are related to the attacks they may be interested in performing by means of the `mounts` association. We just sketch in the main model the relations of the `Attacker` class, which will be presented more in details in Section 3.2.6) where the protection requirements sub-model will be presented. To relate attackers to the attack they can mount, `Attacker` instances are associated via the `hasExpertise` association to the `AttackerIdentificationEnum` values. The attack paths attackers can mount are represented via the `performs` association. Attacks are strictly related to the tools attackers use to actually perform an attack step. `Attacker` instances are thus related to `AttackTool` instances using the `canUse` association. As anticipated before, to be very precise, in the security model, each `AttackStep` instance has been associated to `AttackTools` instances that can be used to mount the attack by means of the `mayBePerformedByMeansOf` association. At present, we don’t expect to model single attackers, we just need the flexibility to express that various attacks can be mounted by several attackers that may have different expertise and differ-

ent tastes on the attack paths to follow to mount an attack (see Section 3.2.6).

3.2 Model Extensions: the Sub-Models

The main model presented above depicts the main concepts and their (high-level) relations. However, a refinement is needed to allow a more fine-grained description of the protection scenarios ASPIRE has to face. The main tool we use to refine the main model is inheritance. It helps us to refine main model concepts and at the same time preserve the associations among them. Refinements to the main model will be presented by means of a set of sub-models that cover six areas:

- *Assets*, which describe what to protect and report categorization already introduced in D1.02.
- *Applications and their execution environments*, which precisely characterize the application to protect. This will allow us to better target the protection protection and to provide information about the execution environment that may determine classes of attacks and protections. For example, attacks against an applications running on Windows may be different from the ones the same application has to face on MacOS, and some protection techniques cannot be available on all the platforms.
- *Protections and protection types*, which define a taxonomy of the different protections.
- *Attacks*, which describe the attacks that can be mounted against applications, including the attacks already identified in D1.02.
- *Metrics*, which provide a very preliminary identification of the classes to use to convey information about metrics and the general types of metrics-related classes.
- *Protection requirements*, which capture our initial ideas on how to specify protection requirements on the target application, and which complete the information presented in D1.03.

The areas and the corresponding list of sub-models is not definitive, as these are the areas we currently identified. The next sections will present the initial definition of these sub-models.

3.2.1 Application sub-model

The sub-model shown in Figure 2 presents an initial definition of information about the target application that the ADSS will require to evaluate the impact of protections and thus decide on the best protection according to user requirements. We highlight five major concepts, which will be developed into separate packages:

- the *parts* and *components* the application is made of, described with the `ApplicationPart` class and detailed in the `ApplicationParts` package;
- the *platform* where the target application or some of its components will be executed, such as client and server stubs. This is described by means of the `ExecutionEnvironment` class and detailed in the `ExecutionEnvironment` package;
- the possible *communications* between two or more of the application components described by means of the `Communication` class and detailed in the `Communications` package;
- the way the source code is *compiled and linked* to obtain the executables, described by means of the `Compiler` and `Linker` classes, and detailed in the `Compilers & Linkers` package;

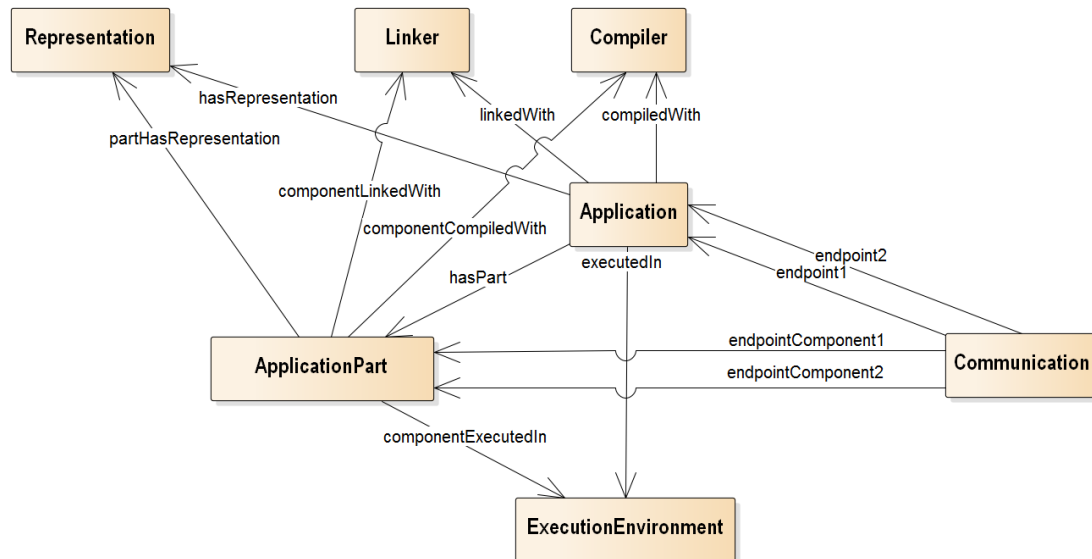


Figure 2: The applications sub-model.

- the types of *abstract representations* of the target application that could be needed to evaluate the impact of protections, select the best protections, and actually enforce them, described by means of the `Representation` class and detailed in the `Representations` package.

These five packages will be independently developed during the next months. This list of packages is also preliminary. Other packages could be added depending on the ADSS design. From Figure 2 it is possible to see that an `Application` instance is connected to:

- its application components and parts, instances of the `ApplicationPart` class, via the `hasPart` association, that has been already illustrated in the main model;
- `ExecutionEnvironment` instances where the application can be run via the `executedIn` association;
- `Representation` instances, that abstractly describe these application parts via the one-to-many association `hasRepresentation` association;
- `Compiler` and `Linker` instances used to compile it, via the associations `compiledWith` and `linkedWith`, which are valid if the single application components are not compiled independently;
- `Communication` instances of which the `Application` is an endpoint, will be identified by navigating the `endpoint1` and `endpoint2` associations in opposite directions.

It is worth noting that the `ApplicationPart` instances are also linked with the `Application` instances in the `ApplicationParts` package. The `ApplicationParts` package presented in Figure 3 describes the parts of an application that are of interest for the protection, for instance because they contain assets, are targeted by a protection, or are the endpoints of some secure communication channel. We identified several concepts and described them by means of inheritance. We consider the following subclasses:

- `File` class is used to model external files and data, such as custom configuration files (represented by means of the `ConfigurationFile` class), or registries/manifest files that might contain relevant data to be taken into account (represented by means of the `Manifest` class);

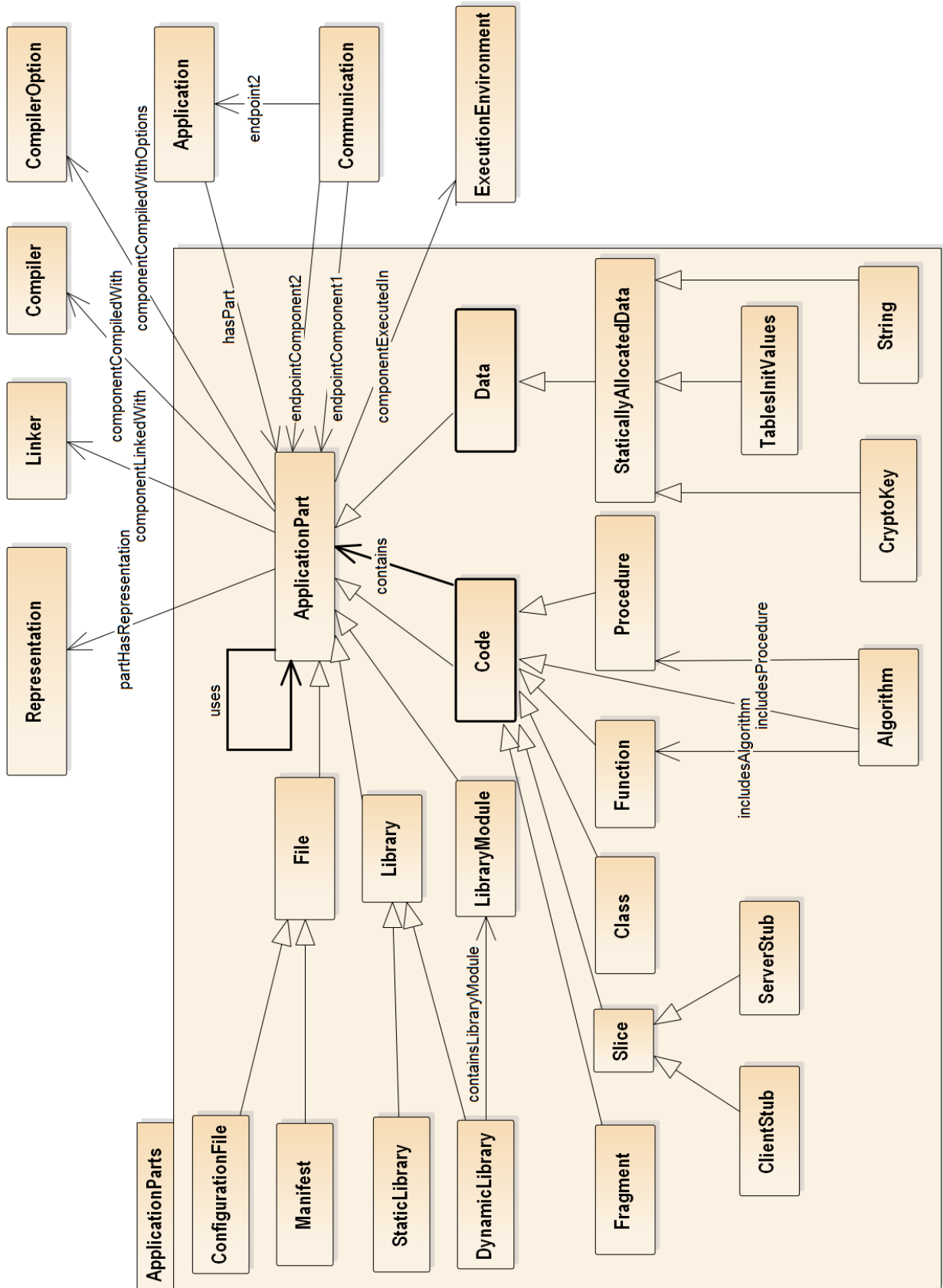


Figure 3: The ApplicationParts package.

- Library class, used to describe static or dynamic libraries, represented as `StaticLibrary` and `DynamicLibrary` class instances. `DynamicLibrary` instances are connected to instances of the `LibraryModule` class by means of the `containsLibraryModule` association;
- Code class instances contain all the kind of executable code (functions, methods, snippets, ...) than can be found inside an application. Note that a `Code` instance can contain other application parts (e.g. a function contains a PIN or a code fragment). This is modeled by the `contains` association. Furthermore, this class has several specialized sub-classes:
 - `Function`, and `Procedure` classes, whose instances explicitly declared functions, procedures, and methods in the target application;
 - `Algorithm` class, whose instances are algorithms, which may be formed of several `Function` and `Procedure` instances (related by means of the `includesAlgorithm` and `includesProcedure` associations);
 - `Slice` and `Fragment` classes, whose instances represent parts of the code not necessarily corresponding to entire functions and procedures. Examples are the barrier slices (investigated in WP2) and the server and client parts generated during code splitting, represented by `ServerStub` and `ClientStub` instances;
 - `Class` is the only element presented in this initial characterization of application components originating from object-oriented programming. We added this mainly as a placeholder to remember us that object-oriented programming constructs and entities need to be considered while extending or adapting the models. Because of resource limitations, however, ASPIRE will only implement techniques for C code for the time being.
- Data class models all the kind of static/dynamic data that an application can contains. Currently it has only one sub-class, `StaticallyAllocatedData`, used to describe (security-sensitive) data embedded in the executable files or libraries, such as cryptographic keys (represented by means of the `CryptoKey` class), initialization values for tables (represented by means of the `TablesInitiValues` class), etc. A notable example of statically allocated data are strings, which attackers look after during certain attack steps, modelled with an ad hoc sub-class `String`.

The `ApplicationPart` instances are connected to:

- themselves via the `uses` association. This relationship is used to model both data exchanges and flow-execution jumps between functions, snippets, ...
- `Application` instances are part of (via the one to main `hasPart` association);
- one or more `ExecutionEnvironment` instances that describe where applications can be run via the `componentExecutedWith` association (as some application is made of several independently executing component, like clients and servers);
- `Communication` instances they are endpoint of (via the one-to-many associations named `endpoint1` and `endpoint2`);
- `Representation` instances, that abstractly describe these application parts (via the one-to-many `componentHasRepresentation` association);
- if they are independently compiled and linked, the `Compiler` and `Linker` instances via the `componentCompiledWith` and `componentLinkedWith` associations.

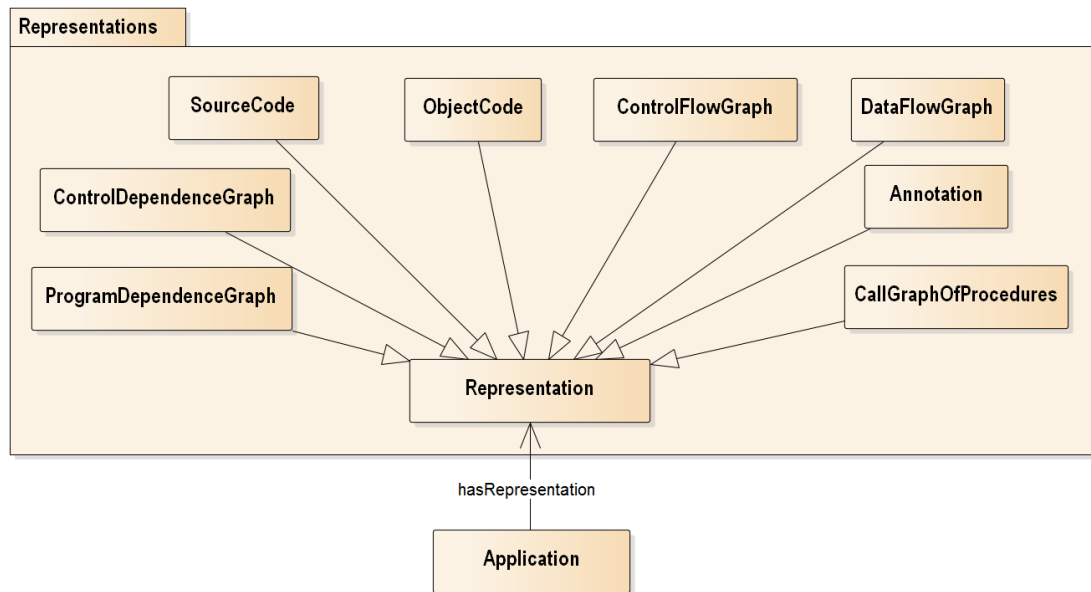


Figure 4: The Representations package.

Note that we preferred the use of the word “component” instead of “part” for the associations that are logically self-consistent like executable portions of the application (like clients and servers) or independently compiled portions of the application.

The Representations package (see Figure 4) includes all the abstract representations that can be used by the ADSS or tool chain components. It includes the abstract `Representation` class which will be sub-classed any time a new abstract representation will be of interest of the ASPIRE project. Currently, we included the following subclasses:

- `SourceCode`, and `ObjectCode`, whose instances are self-explaining;
- `ControlFlowGraph`, a graph representation that models how control can be transferred in the procedure or program, i.e. in which orders instructions can be executed;
- `ControlDependencyGraph`, a graph representation that models to what extent the execution of instructions in a procedure or program depends on the execution of the other instructions in that procedure or program;
- `DataFlowGraph`, a graph that models how values computed in the program are computed out of other values computed (elsewhere);
- `ProgramDependenceGraph`, a graph that combines the data flow graph and control dependency graph presentations to model how the execution of instructions depends on other instructions being executed and on the values being computed by those other instructions;
- `CallGraphOfProcedure`, a graph representation that models which procedures in a program can call with procedures;
- `Annotation`, that will be used to report the annotations added by the developers or by tool chain components to tag pieces of code.

Representation instances are associated not only to `Application` instances, via the one-to-many `hasRepresentation` association, but also to `ApplicationPart` instances, via the one-to-many `componentHasRepresentation` association (as abstract representations are in most cases created for components, like functions and procedures).

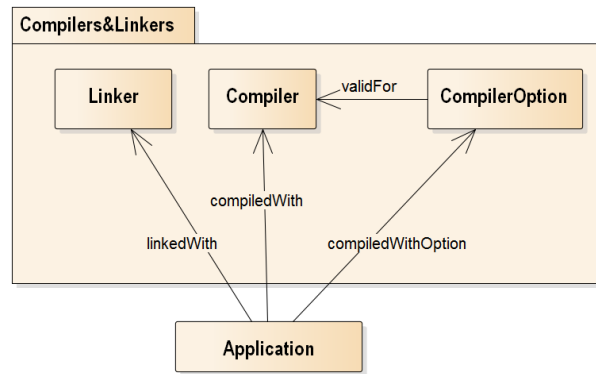


Figure 5: The `Compilers&Linkers` package.

For the `Compilers&Linkers` package depicted in Figure 5, we highlighted two main concepts: compilers, described as instances of the `Compiler` class, and linkers, described as instances of the `Linker` class. `Application` instances are associated to `Compiler` and `Linker` class instances via the `compiledWith` and `linkedWith` one-to-many associations, and to `ApplicationPart` instances via the `compiledWith` and `linkedWith` one-to-many associations.

For the ASPIRE project, it is important to model the options/flags that can be selected during compilation because they can affect the performance and structure of the code and may also be incompatible with the processes performed by some protection tools. For example, Diablo requires the availability of separate object files and a map file of the linked binary or library, all of which can be generated by employing the appropriate options on existing compilers. Furthermore, to select the most appropriate protections to be applied, it can be useful to know, e.g. whether an asset in the form of a code fragment is duplicated because of compiler optimizations such as inlining. For these reasons, we have foreseen the `CompilerOption` class. `Application` and `ApplicationPart` instances are bound to the options used to compile them by means of the `compiledWithOption` and `componentCompiledWithOption` associations. We want also to note that `CompilerOption` instances are independent of the compilers. Each of them models (equivalent) options that can be invoked with different commands on different compilers. So each option is represented as a single `CompilerOption` instance and associated via the `validFor` association to the `Compiler` instances where they are available. The reason for this design approach is that the alternative method, i.e. creating instances of all the options for each compiler and then explicitly requiring their equivalence, is less efficient from the reasoning point of view.

Other code manipulations could also be of interest in this package. In particular, the description of the optimizations that are performed by the compilers can be interesting in order to derive some general information of the object produced and may serve to the decision process. The ASPIRE Consortium is still debating the need to include this information in this package.

Figure 6 displays the structure of the `Communications` package. Its main class is the abstract `Communication` class. We foresee two types of communications, channel-oriented and message-oriented communications. Therefore `Communication` has been sub-classed in the `Channel` and the `MessageExchange` classes. The `Channel` class has been further sub-classed in the `ProtectedChannel` class to describe protected channels. Instances of the `ProtectedChannel` class are bounded to the security properties they guarantee by means of an ad hoc association: `enforcesChannelProtection`. Security properties are represented by means of instances of the `SecurityPropertyEnum` class, an enumeration listing all the possible security properties that can be enforced on communications (e.g. integrity and confidentiality). A `MessageExchange` instance is composed of a set of messages, represented as `CommunicationMessage` instances. To mark protected messages, i.e. messages that have been processed to ensure some security property (for instance, message authentication, integrity, and confidentiality), we use an ad hoc

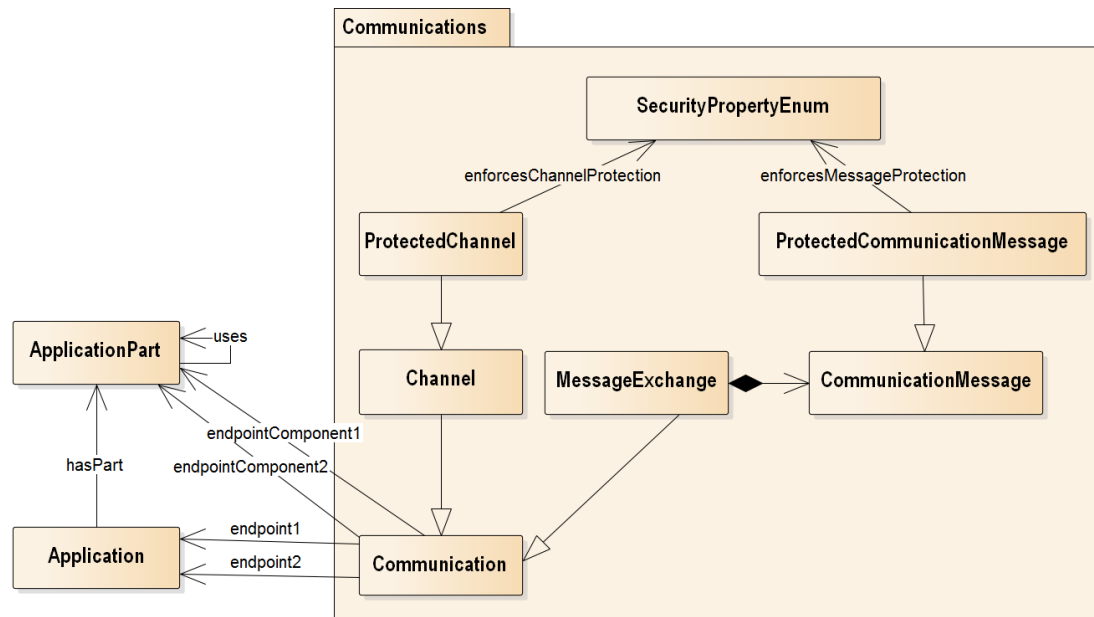


Figure 6: The Communications package.

class: `ProtectedCommunicationMessage`. Its instances are also connected to instances of the `SecurityPropertyEnum` class, via the association named `enforcesMessageProtection`.

Communications are characterized by their endpoints. In order to support non-oriented communications as well, endpoints are described with the one-to-many `endpoint1` and `endpoint2` associations from `Communication` instances to `Application` instances. In case we want to describe oriented communication, the application referenced by the `endpoint1` association can be considered the originator and the application referenced by the `endpoint2` association the consumer. It is worth adding that this enables the description of communications between instances of the `ApplicationPart` class, e.g. when their components are to be executed on different machines, such as the different forms of stubs that will execute on the client and on the server. For that reason, we introduced the one-to-many `endpointComponent1` and `endpointComponent2` associations.

Finally, the `ExecutionEnvironments` package, depicted in Figure 7, refines the abstract class `ExecutionEnvironment` by sub-classing it. Our initial list of the execution environments we consider includes:

- `OS` class, which describes the operating systems where applications may be executed;
- `Processor` and `InstructionSet` classes, which allows us to describe the processing unit for which the application will be compiled;
- `VM`, `Emulator`, `Interpreter` and `InstrumentationToolkit` classes that describe virtual execution environments. Moreover, we will add more information on Virtual machines as they are managed by one-to-many hypervisors, as described using the `Hypervisor` class instances associated with the `managedByHyperVisor` association, and run an operating system, as modeled with the `hasOS` association.

3.2.2 Assets sub-model

Figure 8 reports the categorization of assets presented in D1.02. The different asset categories have been modelled with the sub-classing paradigm. The classes `PublicData`, `TraceableData`,

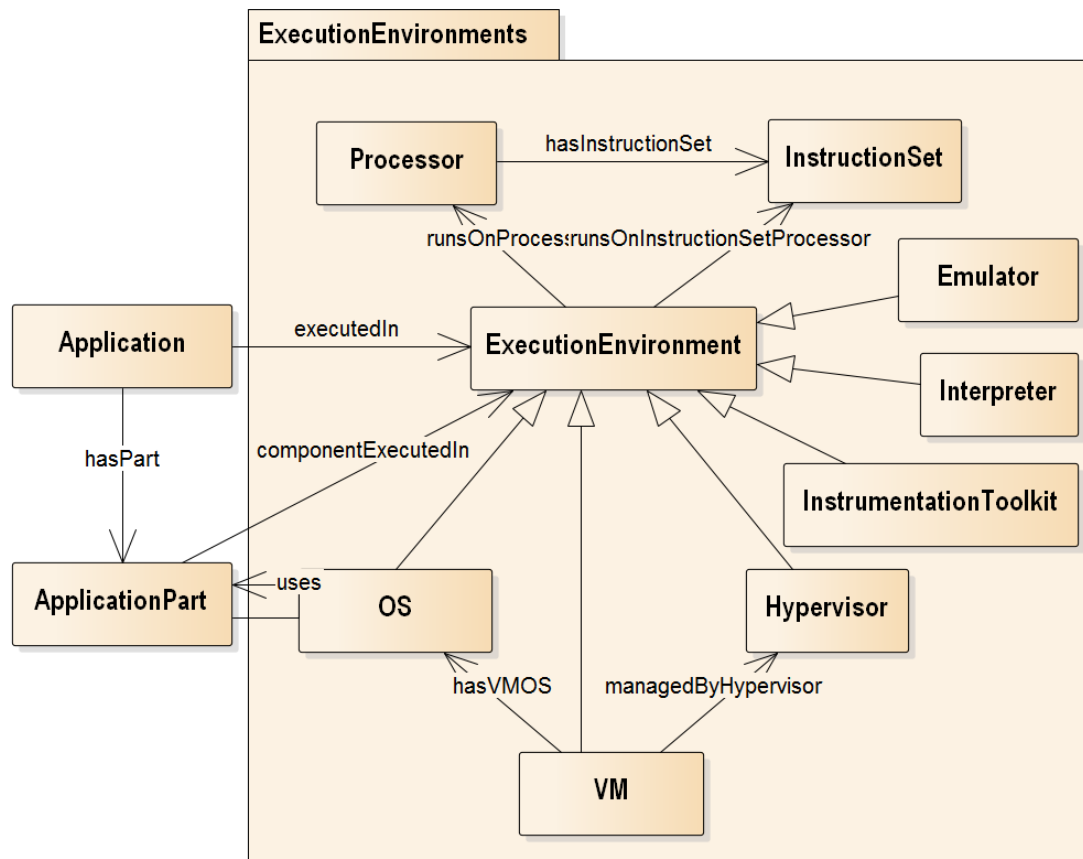


Figure 7: The ExecutionEnvironments package.

TraceableCode, ApplicationExecution, UniqueData, GlobalData, PrivateData, and Code are subclasses of the main Asset class. Moreover, the Code class has been further subclassed in SecurityLibrary, CustomAlgorithm, and PrivateProtocol. There are different asset properties, obtained as subclasses of the AssetProperty class: the Integrity class, used for the assets that must be protected from modifications, the Confidentiality class, for the asset that must remain secret, the Privacy class, for the assets that data whose disclosure affects the privacy of the person owning the application, and the SingleInstance class, used for applications that must be only used on a registered platform (e.g., no clones of the application must be executed).

3.2.3 SW Protection sub-model

Figure 9 shows the categorization of the protection techniques that are considered of interest for the ASPIRE project. First of all, it is possible to see the five lines of defence, i.e. the main, more abstract categories of protections as detailed in the ASPIRE DoW. These techniques are represented by means of the DataHiding, AlgorithmHiding, AntiTampering, RemoteAttestation, and Renewability classes, all subclasses of the SWProtectionType class. Together with the five lines of defence, we also added some more concrete protection types that play a major role in the ASPIRE project. They are represented by ClientSideCodeSplitting, ClientServerCodeSplitting, ClientServerDataSplitting, and ReactiveTechnologies.

All the previously mentioned classes are the first level of categorization. We also specialized some of these techniques, for instance:

- DataHiding has been sub-classed in Source2SourceDataObfuscation and WBC (white-

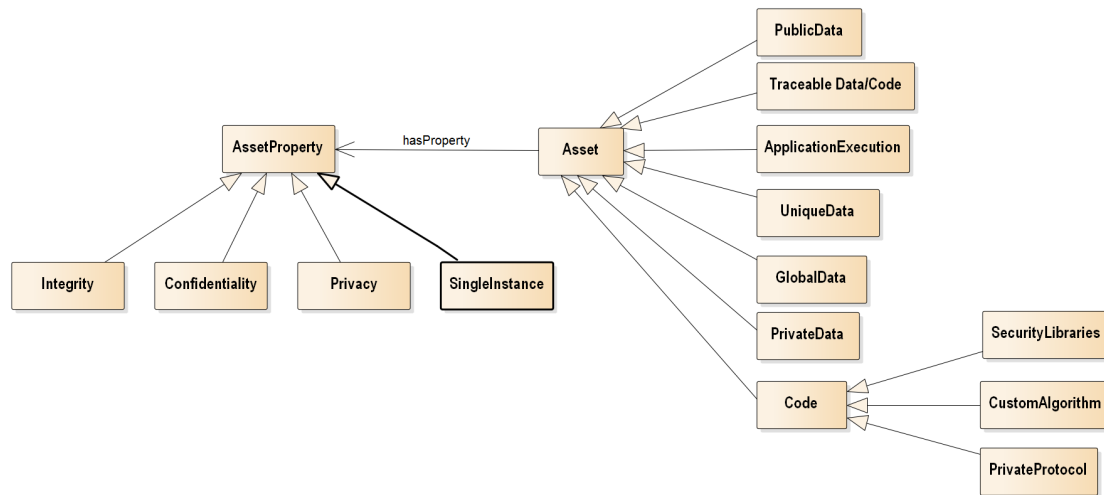


Figure 8: The asset sub-model.

box crypto);

- AlgorithmHiding has been sub-classed in:
 - SourceLevelAlgorithmHiding, further sub-classed in PatternRemoval;
 - ClientSideVM;
 - BinaryCodeObfuscation, further sub-classed in CodeFlattening, BranchFunctions and OpaquePredicates;
- AntiTampering, has been sub-classed in AntiCodeInjection, AntiLibraryCallback, AntiDebug and CodeGuards;
- Renewability has been sub-classed in RenewabilityInSpace and RenewabilityInTime
- ReactiveTechnique has been sub-classed in TimeBombs.

Integrity protection techniques, like remote attestation and anti-tampering techniques, are also categorized as either static and dynamic. However, instead of using the sub-classing paradigm we plan to add a Boolean attribute (`dynamic`) in those classes.

A protection technique exposes one or more protection profiles (see D5.01), implemented as instances of the `SWProtectionProfile` class. The `hasProfile` annotation is used to link the profiles to their related `SWProtection` objects.

Some protection techniques cannot be applied on the same asset in any order, so that a (partial) ordering is needed. The ASM models this by making use of the `cannotBePrecededBy` association, used to express the fact that a protection technique cannot be used after another one.

Protection techniques can be used in different ways by changing their own configuration parameters. However, there are more meaningful ways to use them (which are selected by human beings and made available to the decision process)¹. We introduced the `ProtectionInstantiation` class to represent a set of ways to use protections. The `AppliedSWProtectionInstantiation` association class is then used to associate a `ProtectionInstantiation` class instance (via the `hasInstantiation` association) to the `ApplicationPart` class instance to protect (via the `hasAsset` association) and the annotation to be inserted in the application source code (via the

¹For instance, in some case it is worth considering if 5%, 10%, 15% of the code needs to be obfuscated, it is not really interesting considering if 5%, 5.1%, or 5.5% of the code should be obfuscated.

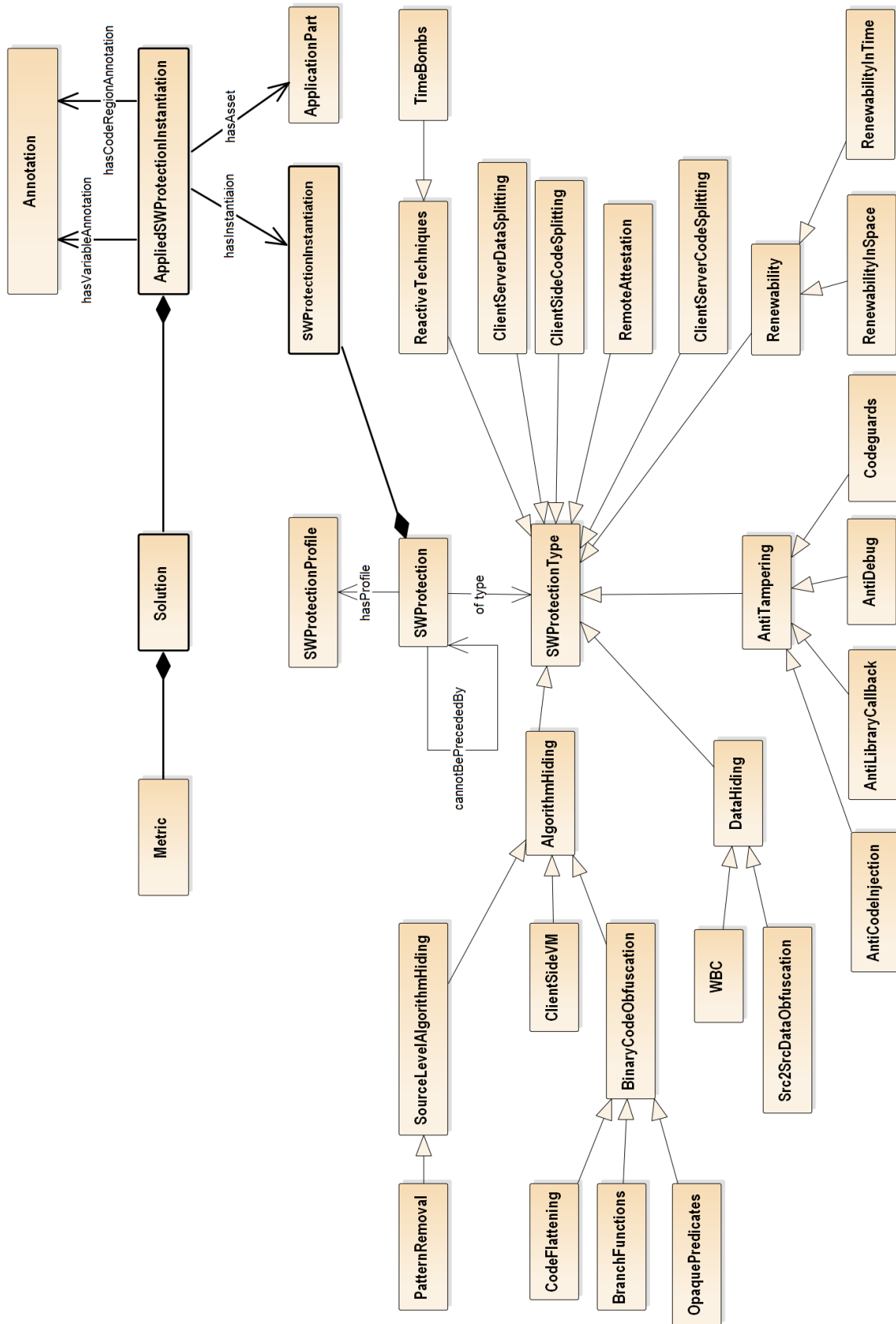


Figure 9: The SW protections sub-model.

hasVariableAnnotation and hasCodeRegionAnnotation associations depending on the asset type.) Finally, the Solution class represent a combination of protections to be use to protect an entire application as an aggregation of AppliedSWProtectionInstantiation association class instances. Moreover, Solution class instances also aggregate all the values of the metrics computed on the application protected with the protection instantiations it aggregates.

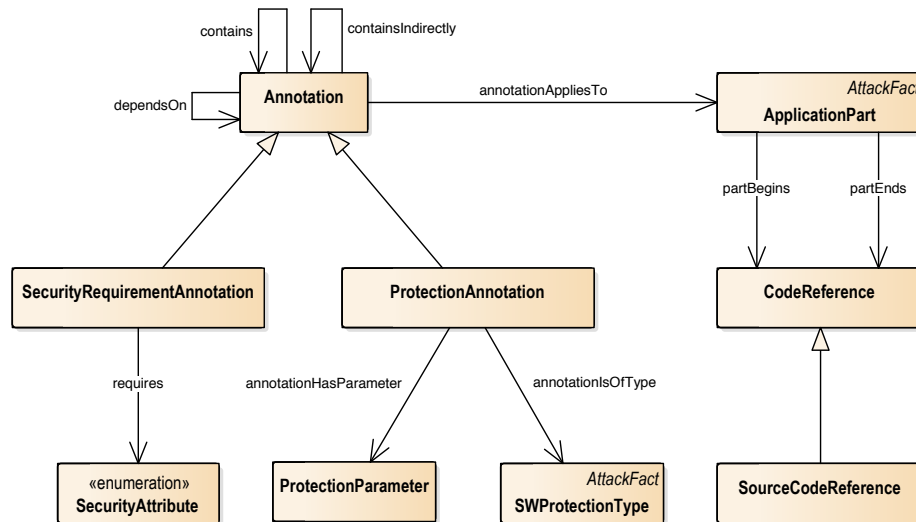


Figure 10: The annotation part of the SW protections sub-model.

The main class of this part of the SW protections sub-model is the Annotation class (see Figure 10). All instances of the Annotation class contain the attribute annotationID to univocally identify them (within the application to protect, of course).

Annotations apply to specific part of the application to protect. These parts are represented with ApplicationPart instances available in the Application. To univocally identify the application parts, we introduced the CodeReference class. Each ApplicationPart instances is associated to two CodeReference instances: one indicates where the part begins and the other one where the application part ends. This scenario is modelled through the partBegins and partEnds associations. Currently, we only need to refer to source code, therefore, file names and the line numbers where the part starts and ends are enough to describe the application part. Therefore, we subclassed the CodeReference into the SourceCodeReference class, which contains the filename and line_number attributes.

Moreover, there are several associations to express dependencies among Annotation instances. These dependencies will be exploited by the ADSS or protection tools able to process them. The contains association is used to describe nested annotations² The containsIndirectly association to represent annotations that are reached when the code within an annotation is executed. For instance, the fact that an Annotation instance a_2 in a function called from within the application part associated to the another Annotation instance a_1 is represented by associating a_1 containsIndirectly a_2 . Finally, to represent dependencies among annotations, e.g., to relate annotations that are part of the same technique, like parts protected with the same guards, or different parts that need to be protected analogously, the dependsOn association has been introduced. The exact semantics of this association depends on the technique that will actually consume/use it.

Annotations are divided in security requirement annotations and protection specific annotations (see deliverable D5.01). Therefore, we introduced two subclasses of the Annotation class, the SecurityRequirementAnnotation and ProtectionAnnotation classes.

²It is worth noting that the ASPIRE consortium has decided that valid annotations can be either nested or disjoint. No cases of partly overlapping annotations is possible. Moreover, they have to start and end in the same file.

`SecurityRequirementAnnotation` instances reflect the security requirements defined in the deliverable D1.04, thus, `SecurityRequirementAnnotation` instances point to instances of the `SecurityAttribute` enumeration class through the `requires` association. This enumeration includes the following values: `confidentiality`, `integrity`, `privacy`, `nonRepudiation`, `executionCorrectness`.

`ProtectionAnnotation` instances refer to the `SWProtectionType` instances they ask to enforce on that application part (via the `annotationIsOfType` association). Moreover, `ProtectionAnnotation` instances link all the protection specific parameters that need to be passed to the protection technique to be enforced according to the software engineer in charge for protecting the application (or the ADSS) by means of the `annotationHasParameter` association. These parameters are characterized as name, value pairs. Therefore, `ProtectionAnnotation` instances are associated to instances of the `ProtectionParameter` class, which contains two attributes, `name` and `value`.

3.2.4 Attacks sub-model

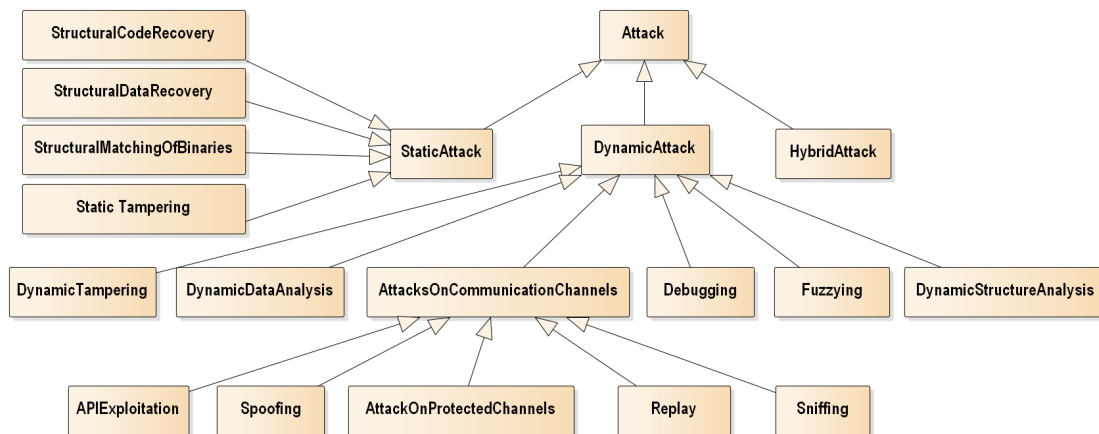


Figure 11: The attacks sub-model.

Figure 11 presents the attack sub-model. The categorization of the attacks comes directly from D1.02, that distinguished static, dynamic, and hybrid attacks. To that extent, we introduced the top-level sub-classes `StaticAttack`, `DynamicAttack`, `HybridAttack`, `PassiveAttack`, and `ActiveAttack`.

`Static attacks` are further sub-classed in `StructuralCodeRecovery`, `StructuralDataRecovery`, `StructuralMatchingOfBinaries`, and `StaticTampering`. `Dynamic attacks` are further sub-classed in `DynamicTampering`, `DynamicDataAnalysis`, `Debugging`, `AttacksOnCommunicationChannels`, `DynamicStructuralAnalysis` and `Fuzzing`. `AttacksOnCommunicationChannels` has been further sub-classed in `APIExploitation`, `Spoofing`, `Sniffing`, `Replay` and `AttacksOnProtectedChannels`.

The attack sub-model also includes the part of the conceptual model that is needed to represent the Petri nets that will be used for the simulation. Figure 12 depicts this part.

There are two ways to support the simulation with Petri nets: (i) static descriptions of known attack paths, and (ii) dynamic discovery of attack paths from descriptions of attack steps. The latter paths will be generated by the enrichment phases, e.g. when constructing Petri nets for specific scenarios. This capability will allow the developers of the AKB and the ADSS to populate it with generic, widely applicable attack steps that can be reused in many attack and protection scenarios. The former, static descriptions will be useful to manually populate the model with known attacks, e.g., because they are very specific combinations of attack steps that make sense only in a

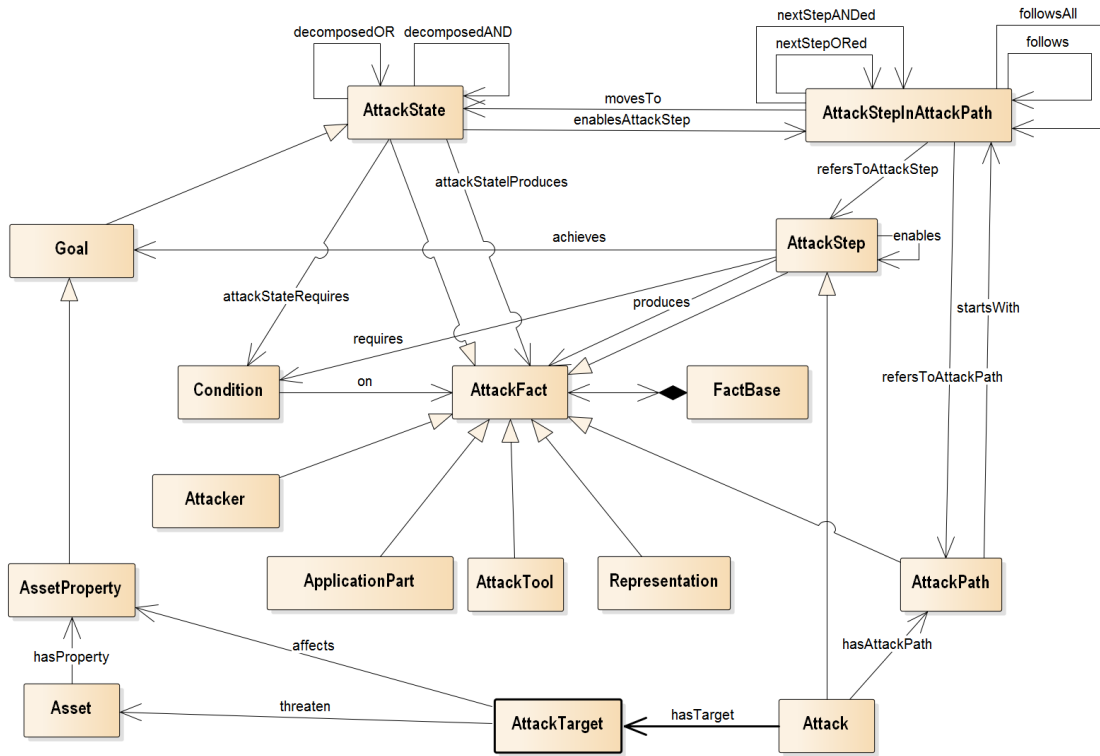


Figure 12: The attacks, goals and Petri nets.

very limited set of scenarios, or, during the project itself, because the automated enrichment and reasoning in the ADSS is not yet mature enough to derive all relevant attack path automatically.

The transitions of a Petri net are instances of the class `AttackStepInAttackPath`, which refers to the `AttackStep` via the `refersToAttackStep`. Places of a Petri Net are instances of the class `AttackState`. The relationships `decomposedAND` and `decomposedOR` are needed to further specify dependencies among attack states, that is, it an attack state can be reached if one or more previous states have been reached.

`Goal` is a generalization of `AttackState`, which means that some of them are goals as they contain relevant information on assets or on the state of protections being undone or worked around, like intermediate achievements. Moreover, some of the goals are the final objectives of an attack, i.e. the targeted asset properties. This is represented by defining the `AssetProperty` as a subclass of `Goals`.

Each arc in a Petri net connects one `AttackState` to an `AttackStepInAttackPath` or vice versa; the relationship `movesTo` represents arcs connecting a transition to a place, while the relationship `enablesAttackStep` represents arcs connecting a place to a transition; in this way the whole static structure of a Petri net can be modelled. Starting from the final goal of an attack the model can be navigated through the relationships `movesTo` and `enablesAttackStep` to dynamically discover all the transitions and places leading to this final attack goal. Together with the `nextStepORed` and `nextStepANDed` associations, we introduced their reciprocal associations to improve querying and reasoning. The relationship `follows` further specifies the fact that one attack step follows another one and can be reached if at least one of the previous attack steps has been completed, and `followsAll` can represent the fact that to perform one attack step more than one previous steps must have been performed.

`AttackState` instances might require some information to be executed, might be performed with an `AttackTool` and will produce some information. Any information used throughout the attack has been named `AttackFact` as the most generic information represented as a fact into a fact base, described by means of the `FactBase` class, composed of `AttackFact` instances.

We can use this information to dynamically discover attack paths. Indeed, `AttackStep` enabling constraints are depicted by (optionally) associating an `AttackState` instance to a `Condition` instance, representing a predicate on some of the facts in the knowledge base, by means of the `attackStateRequires` association between `AttackState` and `Condition` instances, and the on association, between `Condition` and `AttackFact` instances. The description of the information we expect to use has been obtained by subclassing the `AttackFact` class. Our expectation is to use as facts the `ApplicationPart` instances (as it is important to know what to attack), the associated `Representation` instances (as obtaining `AttackTool` he has at his disposal (as we need to know what the attacker is able to do), and the `AttackPaths` the attacker likes. Additionally, having performed and completed some `AttackStep` or reached some `AttackState` (and `Goal`) is also important to evaluate if attack steps and states can be executed (and determine dependencies). The known facts are updated when attack steps and attack states are completed. Indeed, we have foreseen two associations to this purpose, produces from `AttackStep` instances to `AttackFact` instances, and `attackStateProduces`, from `AttackState` instances to `AttackFact` instances. In some cases, it is needed to explicit state that an attack step enables other attack steps (without the need to dynamically derive it). This is done by using the `enables` self-association of the class `AttackStep`.

The fact that `Attack` instances are also `AttackStep` instances (by generalization) shows another advantage of the attack sub-model: it can manage composition of attack paths. This result will be achieved by hierarchical composition of Petri nets, where one transition (an `AttackStep` instance) can actually represent another sub-net included in the main Petri net.

An `Attack` class instance is related to one or more asset since its ‘job’ is to disrupt some assets’ security property. This is modeled by the `AttackTarget` class and the `hasTarget` one-to-many relationship.

3.2.5 Metrics sub-model

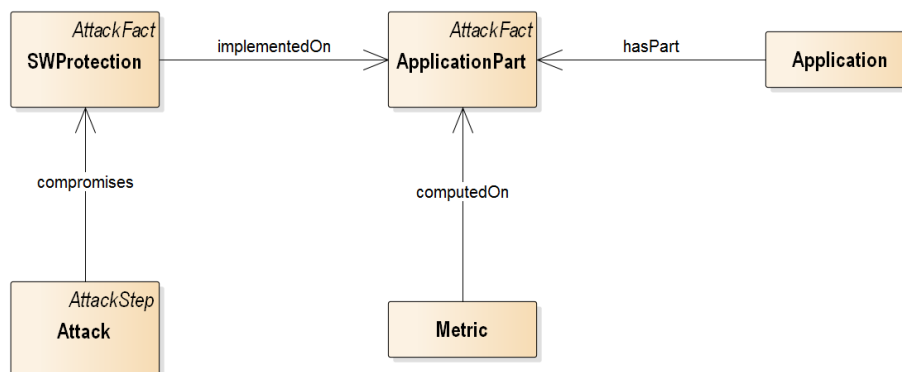


Figure 13: The metrics sub-model.

The metrics sub-model, depicted in Figure 13, considers the relations among metrics, represented by the `Metric`, application parts, and attacks. Metrics are computed on specific portions of the application, therefore we added the association `computedOn` between `Metric` instances and `ApplicationPart` instances. The actual values of metrics will be conveyed by means of attributes whose type (integer or real) will be decided depending on the metrics.

Additionally, the metrics will be used to quantify the impact of a protection, applied on an application part, against a specific attack. For this purpose, we make use of the `evaluatesImpactAgainst` association between `Metric` instances and `Attack` instances. In this case, we expect to have several attributes as a protection might influence the attack probability, the attack time or expertise required, etc.

3.2.6 Protection requirements sub-model

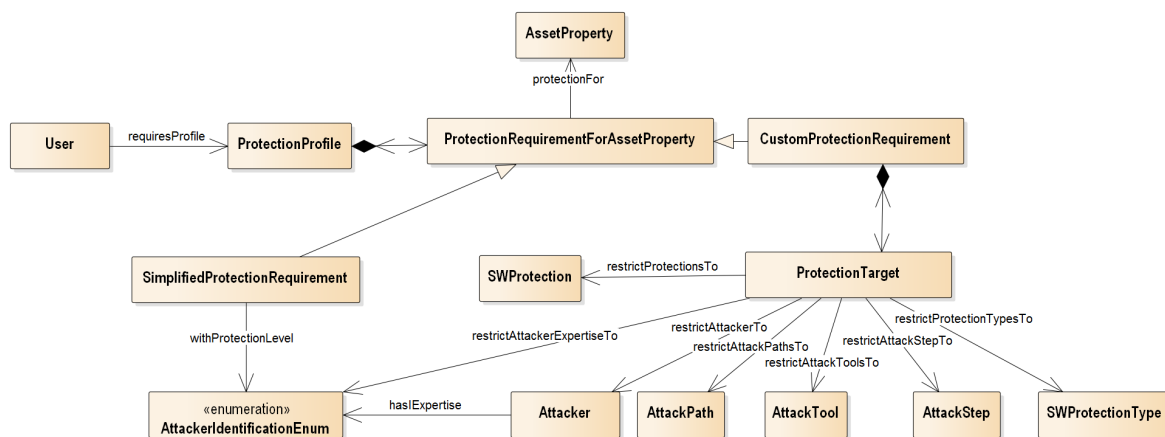


Figure 14: The protection requirements sub-model.

One of the ASPIRE project objectives is to allow an Application Vendor without being an expert on software protection. Therefore, we must allow an ADSS user without an in-depth knowledge on software protection to easily specify his protection requirements. To this purpose, we introduced a set of predefined protection profiles. However, ADSS users can have a strong background on software protection. Therefore, the ADSS must also allow them to fine tune their protection requirements by precisely specifying the attackers they want to face, the attacks they think are relevant, and to restrict the software protection to use. That is, they may want to constrain with precise directives the selection of the best protection for their application. Figure 14 shows the classes, and their associations, used to model this situation.

To describe this scenario, we first introduced a utility concept, the `User` class, which just serves to relate different protection profiles (to different ADSS users). `User` class instances are related via the `requiresProfile` association to the `ProtectionProfile` class instances, the class we introduced to convey information about software protection profiles. Each protection profile is composed of a set of `ProtectionRequirementForAssetProperty` instances that serve to specify the desired protection for a single asset property. The target `AssetProperty` instance is specified by means of the many-to-one `protectionFor` association.

We initially see two types of protection specifications, thus we sub-classed `ProtectionForAssetProperty` in the `SimplifiedProtectionRequirement` and `CustomProtectionRequirement`.

Instances of `SimplifiedProtectionRequirement` can be used to specify the intended hardening level by directly referring to an enumeration that determines the attacker expertise, the `AttackerIdentificationEnum` class. That is, it will be possible for a user to specify that he wants to protect against amateurs, geeks, experts or gurus (see D1.02). The ADSS will first automatically select for the user all the attacks, attack paths and attack tools to protect from, and then investigate the protection techniques to implement.

For a more fine grained specification of the requirements, we introduced the `CustomProtectionRequirement` class, whose instances are composed of a set of `ProtectionTarget` instances. `ProtectionTarget` instances allow users to define a set of “attackers” to protect against. The attackers to protect against can be defined as an `Attacker` instances, which are related with their expertise (i.e. with the `AttackerIdentificationEnum` class), the attack tools at their disposal (i.e. with the `AttackerTool` class), and the attack path they may be interested in performing (i.e. with the `AttackPath` class). Therefore, the `ProtectionTarget` is associated to the `Attacker` instances via the `restrictAttacker` association. Moreover, a `ProtectionTarget` instance is also associated to `AttackStep` instances (via the `restrictAttackStepTo`) to permit a more

fine-grained definition of the weapons available to the attacker. A `CustomProtectionTarget` instance is associated to the `SoftwareProtection` or `SoftwareProtectionTypes` instance the user wants to consider during the selection of protections (via the `restrictProtectionsTo` and `restrictProtectionTypesTo` associations).

Part II

Security Evaluation

4 Metrics Approach

Section authors:

Bjorn De Sutter, Bart Coppens (UGent)

To make this deliverable self-contained, the content of this section was mostly copied from Sections 2–5 of Deliverable D4.02. The copied parts, of which some have been edited lightly, are marked in black text. New text that provide new content compared to previous deliverables is marked in red.

In this section, we present a framework for the quantification of software protection, which extends the Goal-Question-Metric approach [4]. We also present an instantiation of the framework in the ASPIRE context. The goals and the questions that we consider relate to the protections that the project has developed as described in deliverable D1.04 v1.2, and the protection-specific deliverables of work packages WP2 and WP3. Furthermore, the goals and the questions that we consider depend on the attacks that are relevant for such protections. These are all the attacks in the scope of the ASPIRE project, as they were presented in deliverable D1.02 v1.2. Starting from attacks and protections, we identified the relevant features that metrics should capture quantitatively.

The instantiation of our framework consists of the following steps:

Step 1 (goal) Succinct definition of the goals the approach is supposed to achieve, in this case with regards to the quantification of software protections. The used metrics can be used to evaluate to what extent a goal is reached in some instance (which is akin to a decision problem), as well as to actually achieve that goal itself (i.e., as part of the solution to an optimization problem).

Step 2 (questions) Expansion of the goal into a set of questions of which the answers contribute to answering the decision problem and to solving the optimization problem.

Step 3 (measurable features) Identification of a set of measurable features, based on protections and attacks, relevant for answering the questions.

Step 4 (metrics) Derivation of metrics from the measurable features. Metrics are more concretely defined instantiations of the features. Their derivation includes an unambiguous specification of how to actually compute them given the definition of an attack, of applied protections, if any, and of the (un)protected code.

In this context, it is important to note that questions can be relevant to multiple goals, and metrics can be relevant to multiple questions and multiple goals, but they don't need to be.

In practice, steps 3 and 4 are to be applied iteratively, not just within the project, but afterwards as well. An important reason to revise or extend the metrics over time is when attacks are added to the considered attack model, or when additional protections are to be evaluated.

4.1 Goals of the Metrics

The technology developed in ASPIRE aims at increasing the cost incurred to engineer a successful MATE attack on software and at decreasing the potential profit of exploiting such an attack, in order to take away economic motives from attackers. Protections are designed to maximize such

cost increase and profit decrease. Increasing the cost is typically achieved by making the attack steps more complex, hence augmenting the time required to complete each attack step. This attack cost increase is one aspect we want to measure. At the same time, ASPIRE aims at minimal execution overhead associated with its protections. So protection overhead is the other aspect we want to measure. The decrease in attack profit is not something we want to measure automatically by means of metrics. This profit depends too much on non-technical aspects, such as business models, software distribution models, underground value of assets, etc. to be quantified in an automated manner. Similarly, we will not try to measure renewability. We envision that renewability requirements, which clearly depend on the business model in which protections are being deployed, will be passed to the ADSS by its users. We also envision the renewability capabilities of all protections in the ACTC will be documented in the ASPIRE Knowledge Base, such that the ADSS can take them into account without having to really measure those features.

So the general goal of the ASPIRE metrics is to quantify the benefits (increase in attacker effort) and costs (decrease in performance) associated with the adoption of ASPIRE protections.

In ASPIRE, both the increase in attacker effort and the decrease in performance will be measured by computing the metrics on the unprotected software and on protected versions of the software.³

For measuring the performance overhead, we can simply measure execution times, memory footprints, and file sizes. If necessary, the run-time overhead can be measured for multiple use cases, execution contexts, and execution scenarios. For estimating the overhead of potential protections, we can use profile information in the form of so-called basic block and edge execution counts obtained on instrumented versions of the unprotected software. Again, those can be obtained for multiple use cases and execution contexts and scenarios.

By contrast, obtaining precise indications on the exact effort increase for the attackers is very difficult. It requires the collection of an amount of empirical data that is beyond feasibility for the ASPIRE project (if not for any project or company). In fact, predicting the probability distribution of the attack time when a given protection is applied would require the availability of data sufficient to train a prediction model (e.g., a linear predictor). In practice, however, the number of empirical data points that can be collected for each protection is necessarily limited and insufficient for the training of a predictor.

For that reason, the more realistic objective was put forward for ASPIRE to define a set of metrics that provide *approximate* indications about the effort increase that can be expected. Even if such measurements cannot be easily turned into attack time, they provide useful indications about the amount of code elements that are made more complex to analyze, undo, circumvent, ... for an attacker. Such measurable features impacted by the ASPIRE protections give a clear picture of the amount of protection introduced into the software, relatively to the entire software base and with respect to alternative configurations of the same protections. With the metrics, users of the ASPIRE protections can obtain precise, quantitative indications about the protection-wise impact of each protection deployed in the protected software.

4.2 Questions to Consider for Approximating Attack Effort

In order to measure or estimate the attack effort, we first need to consider all the relevant aspects of attacks that influence the attack effort. Because there are so many, we will try to structure, classify and unify them to a certain extent. Next, we will reduce them to their essence and fundamentals, being syntax and semantics of programs and program fragments. Eventually, we will then be able to formulate the right questions in terms of the fundamental aspects of syntax and semantics.

In this section, it is important to realize that with attack effort, we denote the effort needed by the

³Most parts of this section are copied and updated from Deliverable D4.02. The major difference is that text was omitted that foresaw and discussed a second approach that consisted of estimating the impact of protections on concrete metrics rather than measuring the impact.

attacker to execute one attack step, which roughly corresponds to one specific transition in a Petri Net model (see Section 2 of D4.01) of all possible attack paths on some asset in some software. How to combine the efforts of multiple such steps in a whole attack, is the subject of Section 6 of D4.03 and of later sections in this report.

4.2.1 Attack properties that influence attack effort

We identified five major aspects that contribute to the effort needed to mount an attack: the goal, the subject, the means, the object, and the actions of the attack.

Attack Goal First of all, the effort needed to mount an attack depends heavily on the goal of the attack: the type of asset under attack and the threat executed on the asset by the attacker. A categorization of assets and threats was provided in deliverable D1.02 v1.2 Section 3. The subgoal that the attacker wants to achieve with a concrete attack step clearly influences the means that he will use and the actions that he performs. In this regard, it is important to understand that the possible goals and subgoals span a very wide range. This includes, but is by no means limited to, building all kinds of static and dynamic program representations, simplifying all kinds of program representations and performing analyses on them, reducing the search space of attacks, tampering with software either as ultimate goal or as preparation for achieving any of the already mentioned subgoals, etc. In general, the goals hence also determine the object(s) on which the attacker will perform the actions, such as different program representations. Moreover, the goal of the attack or the attack step being performed also influences how the attacker can use his different means, etc., with respect to whether or not sound methods need to be used.

Attack Subject The subject of the attack is the attacker. Clearly, his/her/their experience, capabilities and available resources influence the effort and time that will need to be invested into an attack. Their experience and capabilities determine which means they can deploy, and how fluently they will do so. A coarse-grained identification of capabilities and types of attacks and attack tools was presented in deliverable D1.02 v1.2 Section 4. Moreover, it is important to note that advanced attackers can not only use existing means, but can also extend and customize existing means to develop new ones, e.g., based on plug-ins.

Finally, for many protections learning effects play an important role. Such effects can occur from one software attack to another, but also within one attack, e.g., when an identical fragment of code is injected multiple times into an application, of which only one instance needs to be analyzed.

Attack Means The means are the tools, techniques, methods, heuristics, and all kinds of resources that attackers might use. An overview was provided in deliverable D1.02. It is important to consider the abstraction level at which some mean is considered in an attack step. For example, on the one hand one might consider the tool IDA Pro v6.3 as a very concrete means to perform disassembling. On the other hand, one might consider “recursive descent assembling” as an abstract class of assembler techniques. Considering very concrete means introduces the risk of using metrics that are not relevant for other concrete means (such as IDA Pro v6.4), while the use of very abstract means introduces the risk of using inaccurate metrics.

To some extent, the means used by an attacker depend on the protections that have been applied. This is particularly the case for attack steps that aim for undoing or circumventing specific protections. In addition, it can also be the case for typical code analysis steps, in which the attacker can choose and tune heuristics and used abstractions to trade off precision, soundness, completeness, and complexity (i.e., execution time). If the attacker knows the protections that have been applied, he will use that knowledge for tuning his approach and means.

Attack Object The protected or unprotected software that embeds the asset under attack is the object to which the attacker applies his means. Clearly many properties of this object influence the attack effort. One very important aspect is the representation of the software under attack. An attack step, each tool, and each technique are not applied on an abstract notion of the software, but on a concrete representation or model thereof that the attacker has constructed during preceding attack steps. These representations can be traces, control flow graphs, data flow graphs, program dependency graphs, assembler listings, etc.

It is important to consider the relevant representation when computing metrics and to distinguish between the representation or model (its precision, its completeness, its soundness, ...) that attackers can reconstruct and the ones that defenders can reconstruct. An attacker is able to reconstruct a model or representation based only on his (limited) a-priori knowledge and on information collected during preceding attack steps. The defender, however, can build a representation using information collected from the unprotected (source) code and use knowledge of the configuration with which the protections have been applied. On the one hand, determining the exact representation available to an attacker for a specific attack step is complicated by the fact that one can only estimate the outcome of previously executed attack steps, and that one can only estimate the precise forms (and hence precision) of the underlying code analyses used by the attacker. On the other hand, we can to a certain degree rely on worst-case assumptions: after all, we are interested in blocking attackers that use the best available means of some kind.

As such, we also have to assume that attackers know which tool has been used to produce the protected version, and what protections that tool can deploy. This in turn implies that when computing metrics, we can and in some cases have to take the applied protections and their properties into account.

Furthermore, it can be important to differentiate between code of the original application, and code added during the application of some code transformation that implements a protection. One scenario in which to differentiate between the two classes of code, is when attackers may be able to identify the two parts and exploit that information, e.g., to fine-tune or refocus their attacks, or to speed-up attacks by learning and reusing already obtained results.

As such, it will be important to measure features of code that attackers can use to identify code corresponding to certain protections.

A final aspect of the object to consider, is which parts of the whole object are actually relevant to an attack on a specific asset that is embedded in a limited part of the object. Often, when the attacker can correctly identify which part of the software to attack, or has already correctly identified that part, the other parts of the software will no longer contribute to the effort needed for the attack.

Attack Actions The final aspect to consider, is how the attack step is composed of individual steps. Does the software representation need to be treated as a whole, with some global approach, or does the attack consist of the same local step applied repeatedly for all code fragments at some level of granularity? This is particularly relevant with regards to the learning effect in the subjects conducting the attack.

Like the means, also the actions depend to some extent on the applied protections. Also in this case, the reason is that attackers will tune their actions if they have a priori knowledge about the protections.

From the brief discussion of these five aspects, it is clear that a huge number of factors influence the total effort that an attacker needs to invest in a complete attack. So we seem to be facing a daunting task in identifying the relevant questions and metrics.

4.2.2 Classifying and unifying the relevant properties

However, as stated before, we assume we can target one attack step at a time, and one attack scenario at a time. So whenever we will measure an approximation of the increase in effort needed, we will do so for a given attacker, given his goals, his means, his object, and his actions to perform. We hence have to achieve the goal of measuring the benefits of a protection for the individual, meaningful elements the set of attack $\{\text{goals} \times \text{subjects} \times \text{means} \times \text{object} \times \text{actions}\}$. In essence, we have not one goal for which we have to identify the relevant questions, but we have a whole goal set $\mathcal{G} = \text{attack goals} \times \text{subjects} \times \text{means} \times \text{object} \times \text{actions}$.

Clearly, the set \mathcal{G} is too large to even attempt to cover it by identifying questions (step 2 of our instantiation process), measurable features (step 3) and metrics (step 4) for each of its elements individually.

Instead, we will identify a set of template questions that (hopefully) cover all meaningful combinations in \mathcal{G} , and that can be instantiated for each individual combination. Others have attempted to do this before, and it is generally accepted that the questions need to consider three aspects: potency, resilience, and stealth [7, 8]. Not all three are relevant for all attack steps and all types of protections, but together, these three conceptual aspects cover all relevant features.

Potency Informally, the potency of a protection refers to the amount of obscurity, complexity, and unreadability the protection adds to a program. Numerous metrics have been proposed in the past to measure software complexity: based on analysis of the code, a number of features are measured which represent aspect related to complexity in general, and to more concrete software engineering objectives, such as reliability, testability, maintainability, etc. All of the proposed metrics have shortcomings and limited applicability, however. So instead of using a single software complexity metric, we will have to use multiple, complementary ones, that we will combine into (possibly multidimensional) complexity vectors. This has already been observed in the past [2], but, to the best of our knowledge, has not been concretized for a broad range of protections and attacks.

Resilience The resilience of a protection is the difficulty and effort needed to break the protection with an automated tool, such as a deobfuscator. Two aspects are important here: the programmer effort, i.e., the effort needed to build the tool, and the tool effort, i.e., the execution time and space needed by the tool when applied onto a specific piece of software. With regard to the programmer effort, there is a huge learning effect. Once an attacker has access to a tool that meets all his needs, the programmer effort is reduced to zero, whether he developed the tool himself or not. In case a tool is available, but it needs some tuning, e.g., for the specific forms of protections applied to some application, the programmer effort is reduced significantly, in particular in the case of expert attackers.

Stealth When a protection involves injecting code into the software to be protected, the protection is stealthy if the injected code resembles the original code to the point that an attacker cannot differentiate the two. When a protection transforms code, it is stealthy when the transformed code does not allow the attacker to deduce which protection has been applied. For many protections, a lack of stealth is deadly, because it allows human attackers, and in some cases even tools, to attack the protections, i.e., to undo or bypass them. For other protections, stealth is less important, or not relevant at all.

To the best of our knowledge, no systematic approach has been proposed in the existing literature to instantiate the relevant questions regarding potency, resilience and stealth. Instead only ad hoc approaches have been proposed, some of which are concrete and hence useful for actual quantification, but some of which are also pretty abstract, and hence potentially useful for classification, but not for quantification. Moreover, we know of no approach at all that tries to unify dynamic and static attacks. Typically, literature considers only static attacks. Given the MATE attacks that

we want to protect against, and the dynamic tools and techniques that attackers have available as described in deliverable D1.02 v2.1 Section 4.4, we cannot simply neglect dynamic attacks.

In this project, we therefore propose a more systematic and generic approach, for which we will build on fundamental concepts that

1. link the protections and attacks we want to cover;
2. can be translated into fundamental measurable features that can be computed in a structured, systematic manner;
3. that cover potency, resilience and stealth.

These concepts are syntax, semantics and the mapping between syntax and semantics.

4.2.3 Attack fundamentals: syntax and semantics

In this document, we use the term *syntax* to denote the representation of the operations to be executed (in a static or dynamic program representation), as well as the representation of the operands on which the operations operate. We use the term *semantics* to denote the input-output relationship of a program or of fragments in a program.

The *mapping* between syntax and semantics is the relation between the syntax and the semantics. For example, the instruction set architecture manual describes the mapping between, e.g., 32-bit ARMv7 instruction encodings and the semantics of the instructions in terms of the processor state. Similarly, different IEEE standards specify the meaning of bits in floating-point and two-complement integer number representations used in computers. And programming language specifications describe the mapping between syntactical constructs in a program and their meaning in terms of program state.

There is a clear link between syntax and semantics of a program, because the semantics corresponds to the computer's interpretation of the syntax.

While attacks can operate at the syntactic level whenever the mapping between syntax and semantics is clear, the fundamental goal of attacks in one way or another always relates to the intended semantics of the (original) program:

- In the case of reverse-engineering, the goal is to understand the semantics of a program or a part thereof. This often comes down to finding an appropriate, as simple as possible abstraction for the semantics, i.e., an abstraction that attackers can easily handle, but that does not necessarily need to be directly executable by a processor anymore. When there is a known mapping between syntax and semantics, the reverse engineering can, and most often will, be based on syntactic representations of the program.
- For locating and stealing some program's sensitive data, attackers try to identify and extract the outputs or inputs of certain parts of the program. It is only when the mapping between syntax and semantics is very clear, that attackers can focus their attack on the syntax instead of on the semantics. For example, when the string "Wrong password" is shown on screen during the execution of the program, the attackers might look for an ASCII encoding of that string in the program binary.
- In the case of tampering, the attackers will try to alter the semantics of a program in specific ways. For example, they will try to make it independent of a registration key value or of a PIN code. Or they will make the program produce extra output to leak sensitive data.
- In the case of code lifting, attackers try to extract part of the representation of a program, and try to reuse that representation outside of its original program, while keeping the part's original semantics.

Fundamentally, software protections try to prevent attacks by intervening in the syntax and semantics of software, as well as in the mapping between the two:

- *Cryptography and data hiding protections* aim at hiding the relation between syntax and semantics. The mapping between a plain text, 32-bit two-complement number 0xffffffff and its integer value of -2 is clear. The mapping between a ciphertext 32-bit number 0xffffffff and its real meaning is not known unless one knows the keys to decrypt the ciphertext.
- Instruction set diversification also tries to hide the mapping between syntax and semantics. In a native ARMv7 program, the 32-bit instruction encoding 0xe2411002 represents that the value two is subtracted from the value in register R1, as documented openly for everyone in ARM's manuals. However, in a custom (i.e., randomized) bytecode instruction set interpreter, the semantics of a bytecode 0xe2411002 is not documented at all.
- To some extent, the above two protections can be seen as ways to reduce the a priori knowledge that attackers have about the mapping between syntax and semantics. Other compiler techniques can also be used to reduce this a priori knowledge, such as by bypassing the calling conventions to hide the role of a specific register in passing data between different procedures.
- Some compiler transformations, such as inlining and outlining, have been proposed for use as obfuscation techniques. They repartition the program's syntax in units (i.e., procedures) of which the semantics cannot easily be abstracted to higher levels by the attackers (unlike the original units, which were, by virtue of the software engineering paradigms used by the programmers, easily abstracted).
- Many obfuscation techniques, such as opaque predicates, branch functions, control flow flattening, etc. aim at introducing additional complexity into the syntax of programs, and hence also in the apparent semantics of the program, such that attackers cannot easily obtain the simplest abstraction or representation anymore.

Assuming that

1. programmers try to make their program as simple⁴ as possible;
2. optimizing compilers simplify programs rather than making them more complex;
3. the processor specification is public;

we can consider a compiled program or program fragment the simplest executable representation of its intended semantics. As such, obfuscation techniques try to prevent attackers from reconstructing the original (or equivalent) program or its representations.

- **Classic anti-debugging techniques that try to detect the presence of a debugger, essentially intervene in the semantics of a program: The program gets extra inputs because it queries the environment for symptoms that indicate the presence of an attached debugger. Only if**

⁴In practice, a binary or library is never its "simplest" executable representation of the intended semantics, for the simple reasons that (i) the representation is the result of an engineering process that aims for maintainability, debugability, flexibility, shorter time-to-market, etc., i.e., software engineering and development goals that often conflict with the fuzzy goal of "simplest"; and (ii) that compilers are not yet capable of optimizing all more-complex-than-needed software into the simplest software. For our purpose, however, we will assume (i) that developers and their tools result in "simple enough" software representations, and (ii) that attackers also aim for obtaining the representation developed by the software developers or an equivalent representation in terms of complexity, rather than for the absolute simplest representation. These assumptions do not hold in all MATE attack scenarios, but they do hold for the ones in the scope of the project. Concretely, they do not hold when an attacker is attacking an asset that should, from a security perspective as well as from a functionality perspective, not have been present in the software in the first place. For example, if a software developer leaves the code for expensive features present in a program binary that is distributed as a free trial version, the binary can hardly be considered the simplest representation of the free trial functionality. An attacker trying to enable the included but disabled expensive functionality is then clearly also not interested in reducing the binary to its simplest representation.

the extra inputs reveal that no debugger is present, the program produces the same output as the original, unprotected program for the user-provided input. Whereas in the unprotected program, the presence of a debugger in its execution environment is irrelevant, it becomes relevant in the protected application, thus aiming to reduce the attack vectors available to an attacker.

The last above item points towards an interesting semantic perspective on software protections. A key observation is that what has to be considered as “the relevant semantics” of a program depends on the goal of the user or owner. For benign users, software protections do not influence the observed semantics, i.e., the input-output relation implemented by the software with respect to (i) the inputs of which the user is aware, (ii) the outputs the user observes.

For attackers, however, depending on the attack technique and attack goal, other aspects of the program semantics become relevant. Many protection techniques aim specifically for those aspects of the program semantics.

- More advanced anti-debugging techniques that rely on self-debugging in essence introduce additional complexity into the syntax of programs, by replacing a single-process implementation of the software by a multi-process representation in which two processes (debuggee and debugger) have to collaborate in order to retain the original functionality.

They also alter the semantics of the program, however, at a level in which attackers using debuggers are interested. In self-debugging software consisting of a debuggee process and a debugger process, extra interprocess communication is executed. The semantics of the debuggee process is incomplete without the interprocess communication and without the presence and execution of the attached debugger process. If an attacker tries to detach the debugger process to make room for his own debugger, he breaks the semantics of the protected program.

As we have seen in the public challenge experiment of the project, one attack method is to restore the original semantics.

- Anti-tampering techniques like code guards similarly intervene in the semantics of a program. Instead of letting the semantics depend on external state revealing the presence of a debugger, the semantics in this case become dependent on the representation of the program itself: The protected program only provides the original input-output behavior when (part of) its representation being executed is not altered.

For benign users, the protection does not change any observed semantics. In the face of attackers, however, the protection does aim at breaking the original semantics of the program if they tamper with the code, either permanently (e.g., by editing the executable file), or temporarily (e.g., by setting software breakpoints in a debugger).

- Protections where part of an application is split off and executed on a trusted server instead of on an untrusted client device, make certain parts of the representation and semantics inaccessible for the attacker. Furthermore, additional semantics are added to the client part that is accessible to the attacker, because that part has to produce data for the server-side and takes as input the data that returns from the server. So overall, some semantics are removed from the binary, and other semantics are added.

Please note that to some extent, such a protection technique, like all online protection techniques, also changes the semantics as observed by a benign user of the software. Whereas the original application might be able to work without a network connection to the server providing the online protection services, the protected application does the user to make such a connection available.

- Advanced forms of remote attestation, another online protection, in essence add additional inputs to an application, such as nonces sent to the client by the remote attestation server or

message to trigger reactions in the case tampering has been detected. These inputs are not under control of the attacker (although he might tamper with them). Similarly, additional outputs are added, in the form of the attestations that the client needs to deliver to the server. For online applications (i.e., applications that connect to an application server even without the online protections), the semantics of the application server can also change, e.g., if it stops responding when a tampering has been detected because in incorrect attestation was supplied to the attestation server. Also the inserted reaction mechanisms correspond to new semantics.

Again, all of the additional semantics (except the need to obtain a connection to the server) are transparent to a benign user, but can at the same time make the life of attackers harder.

- Code mobility, another online protection techniques, alters the mapping between representation and semantics. For a benign user, nothing happens except the need to obtain a connection to the mobility server. For an attacker, the representation only becomes available when the program is running.
- Various forms of code and data diversification as developed under the umbrella of renewable protection techniques directly target hiding the relation between the representation and the semantics of a program, by diversifying the representation of multiple instances of the software in space or in time.

When attackers attack a protected program, their tasks related to program comprehension (i.e., reverse engineering) will require more effort as the apparent complexities of the program syntax, of the program semantics, and of the mapping between syntax and semantics increase. With “apparent complexity”, we mean the complexity as observed by the attackers. They will in particular have to invest more effort in the attacks when they fail to undo the increases in complexity introduced by the protections. In some cases, they will even need to undo some of the semantic changes introduced by the protections, such as when they want to use a debugger to trace a program protected by strong anti-debugging mechanisms. In short, in order to make the program observable and to make the observed program as simple as possible, attackers will try to undo syntactic and semantic changes.

When attackers attack a protected program, their tasks related to program tampering typically include removing the additional semantics that were added to make the program tamper-resistant. This is, in other words, not fundamentally different from attack tasks related to program comprehension.

One important aspect is, however, that in the case of online applications, undoing the semantic changes by the online protections might not be possible for even the most advanced adversaries. This is particularly the case with remote attestation, where the attestation server can cause the original application server to abort its services when the client application fails to deliver valid attestations. This does not mean, however, that the attacker cannot try to simplify the added semantics. For example, when no nonce is used in remote attestation to avoid replay attacks, the attacker is given plenty of room to simplify the code that computes the attestations, and to hence remove the added semantics of the attestation computations.

Following this discussion, we are ready to formulate the relevant questions in our goals-questions-metrics approach:

Q0: What code/data fragments are relevant for an attack step?

Q1: How complex is the syntax, semantics, and mapping between syntax and semantics of the protected, relevant code/data fragments compared to the complexity of the corresponding unprotected code/data fragments?

Q2: How easy is it to simplify the syntax, semantics and mapping of the protected code, without oversimplifying it?

4.3 Measurable Features

To answer question Q0, we will assume that code annotations and program analysis allow us to identify the relevant code parts. This is in line with the advice obtained in the ASPIRE project from external experts, that favored us to focus on protection of assets, not on detection of assets and of their relations and the related code and data, which is considered too challenging at the moment.

For answering question Q1, we will mainly rely on a range of existing code complexity metrics that cover all the relevant complexity-related features of static and dynamic code representations. They can be computed on the unprotected and the protected code, or estimated for specific protections to be applied to given code. There is not much new there. The way we will use these metrics is novel, however: rather than simply computing the metrics on standard graphs and traces, we will compute them on weighted graphs and traces.

For example, all elements in graphs and traces will get a *relevance weight* in the interval [0,1]. To some extent, this relevance weight will be based on the annotations that answer Q0: Code that is identified as relevant for a certain asset will get a higher weight, code that is irrelevant for some asset will get a zero weight. When complexity metrics are computed on the graphs and traces of a program, these weights will be taken into account such that irrelevant parts do not contribute to the security evaluation.

Furthermore, we will adapt the aggregation of metrics computed on components to the properties of the applied protections and of the considered attack step, i.e., the location in the Petri Net that models the complete attack.

When evaluating the protection strength against a specific attack step, the attack object (i.e., the representation of syntax and semantics and the mapping between the two as reconstructed by the attacker) essentially determines what to compute the metrics on. The specific attack goals and means mainly determine which measurable features are more or less important. Finally, the attack subject and the attack actions, as well as the nature of a protection (and its implementation) influence how the features measured or computed on individual elements need to be aggregated. For example, when a global approach is needed that attacks multiple instantiations of some protection, on code fragments spread throughout the program, a super-linear aggregation function can be used. On the contrary, when some protection is always applied with the exact same code sequence, a learning effect plays, and only one instance of that sequence should be counted, not all of them. Only when multiple independent elements are added by means of a protection, that all need to be attacked individually, and on which there is no learning effect, will we simply add up the complexities as measured for the individual elements.

To answer question Q2, we will again use weights. Based on combined analyses of many different available program representations (graphs, traces, ...) and also on the features of an attack step and the applied protections, we will assign *simplification weights* in the interval [0,1] to all elements in the graphs and traces. The weights will model how easily the elements can be simplified by an attacker. To compute these weights, we will identify and quantify the presence of features that existing simplification approaches rely on, such as the powerful hybrid static-dynamic approach recently proposed by Yadegari et al. [32]. For example, when an attacker has traces of a program in which a specific conditional branch is always taken, he can simplify the trace by replacing the conditional branch by an unconditional branch without affecting the apparent semantics of the program. Hence when we can identify this feature in a trace, the weight of the conditional branch in the measured complexity metrics should be lowered. Likewise, if an instruction in a graph or trace produces values that do not contribute to the output of the program, the instruction can be removed. So it should get a simplification weight of zero. The simplification weights will be based on features of the protection as well. For example, for protections that involve the injection of fixed, non-stealthy code patterns into the software to be protected, the non-stealthy character will be modeled by assigning a low simplification weight.

Of course, the simplification weights will depend on the attack step being evaluated, as well as on its location in the overall attack path being evaluated. Concretely, for each attack step, we will consider the simplification that an attacker can be assumed to perform based on the information he has available in that step. For example, if for some attack step the attacker is assumed not to have collected a trace yet, we will also assume that he cannot simplify his program representation in the basis of trace information.

In addition to the complexity metrics and the relevance and simplification weights, we propose to consider *unavailability metrics* and *unavailability weights*. Undoubtedly, when part of the code under attack is not available to an attacker, e.g., because it has been split off from the client-side application and is being executed on a server instead, or because the code is omitted from the binary being analyzed statically and will instead be downloaded at run time, the attacker has a harder time analyzing and modifying that part of the code.

4.3.1 Measurable Complexity Features

The measurable complexity-related features we propose are the following:

- **Static Code Size (SCZ)** This is the size of the disassembled code, possibly structured into graphs like program dependence graphs or control flow graphs. The rationale for including this feature is that larger programs are potentially more difficult to attack.
- **Dynamic Code Size (DCZ)** This is the length of one or more program traces. The rationale for including this feature is that longer running programs are potentially more difficult to attack.
- **Static Control Flow Complexity (SCFC)** This feature measures the complexity of static code representations, i.e., graphs. Reverse engineering of the program structure is more difficult if the control flow of the program is more complicated.
- **Dynamic Control Flow Complexity (DCFC)** This feature measures the variation in control flow as observed during the execution of a program, i.e., the variation in executed paths observed in one or more program traces. The idea is that comprehension of algorithms and code is more complex when more different paths through the static graph representations are realized at run time.
- **Ill-Structuredness (IS)** When a program is structured well into procedures with clear functionality, program comprehension is much faster. With this feature, we measure the negative change in structure quality resulting from protections.
- **Code Layout Variability (CLV)** This feature measures whether there is a one-to-one mapping between code addresses and instructions, or whether that mapping changes within and in between program executions. If instructions do not have unique addresses, or multiple different instructions can occur at the same address throughout the program's execution, it will be much harder to reconstruct a static program representation, and to relate analysis results obtained on that representation.
- **Static Data Flow Complexity (SDFC)** This feature measures the complexity of the data flow throughout static program representations, such as data dependency graphs. The passing of data in programs is more difficult to identify and comprehend when the data dependencies are more complicated, when they rely less on conventions, and when data accesses are less manifest.
- **Dynamic Data Flow Complexity (DDFC)** This feature measures the complexity of the data flow as observed in execution traces. The tracking of computed data values (i.e., instruction

operands) throughout the program execution is more difficult when there is a more complex mapping between the operations operating on the data, and when there is more interaction between those operations.

- **Semantic Dependencies (SD)** In a protected program, it can be more difficult to reach a certain program state during dynamic attacks if reaching that program state depends on the cooperation of a secure server. In that case, the server has to be faked or the attacker needs to make sure that he provides valid inputs to the server and that, depending on the server side control processes, the provided inputs do not reveal that an attack is ongoing. In other words, the ability to collect trace information is influenced by the need to communicate correctly with a secure server. Similarly, it can be more difficult to reach a an observable state when anti-debugging techniques or anti-tampering protections are in place. For that reason, we propose to measure the constraints that are imposed by inserted semantic dependencies, and that make it more difficult for an attacker to reach certain program states.
- **Static Data Presence (SDP)** This feature indicates whether or not sensitive data (such as data that should be protected from leaking, but also data that identifies interesting code fragments that can serve as hooks for attacks) are visible, and more or less easily identified, in the static program representation.
- **Dynamic Data Presence (DDP)** This feature indicates whether or not sensitive data (such as data that should be protected from leaking, but also data that identifies interesting code fragments that can serve as hooks for attacks) are present in memory, in a more or less easily identifiable form, at any point in time.
- **Syntactic Stubbornness (SS)** This feature measures to what extent the correct program execution depends on the syntactic integrity of a code fragment.

While the above list of features should not be considered to be carved in stone, we foresee that they capture the most important complexity aspects that have an impact on attacker effort.

4.3.2 Measurable Resilience and Stealth Features

As measurable resilience-related features, i.e., for computing the simplification weights, we propose the following:

- **Static Variability (SV)** If a protection applied multiple times to different parts of (an) application(s) always results in the same code being inserted or created, or in the same, recognizable patterns, the attacker's learning effort will have much higher impact, and attacks become easier.
- **Intra-Execution Variability (IEV)** If during a program execution, parts of an application or protection always produce the same result they are executed, i.e., they impact the program state independently of their input, the attacker can more easily abstract away their control and data dependencies. In other words, he can then simplify the semantics of individual code fragments without affecting the overall semantics of the program under attack, i.e., the semantics as observed on selected inputs.
- **Semantic Relevance (SR)** If certain operations do not contribute to relevant computations in a program (i.e., to output) or do not in fact depend on the program's input, an attacker can more easily abstract away from their control and data dependencies.
- **Stealth (S)** If injected protection code is easily recognized as non-original application code, this may facilitate targeted attacks.

We should note that some of these features, IEV and SR in particular, are related and can hence not be computed independently. For example, if a trace shows that some conditional branch always goes in the same direction, the attacker might replace it by an unconditional branch without changing the (observed) semantics of the program. This will be accounted by giving the branch a low intra-execution variability. As a result of the replacement, however, (part of) the code computing the predicate of the conditional branch will have become irrelevant, as it no longer contributes to the semantics of the program. Likewise, the code on the never-taken path of the conditional branch might become unreachable and hence also irrelevant. For this reason, advanced analyses are required to model the correct interaction between semantic relevance and intra-execution variability.

The strength of such interactions in automated attacks has been demonstrated in 2015 by Yadegari et al. [32]. We included these resilience-related features based on their work, in which they automatically undo a range of very strong protections by relying on fundamental properties related to general program semantics, rather than on ad hoc semantic approaches as proposed by some others [18, 9].

4.3.3 Measurable Unavailability Features

A first set of unavailability features we propose to measure are the following:

- **Unavailable Static Code Size (USCZ)** This is the size of the code fragment that is no longer present in the static binary because of a protection. If more code is missing, attackers can analyze and alter less code statically.
- **Unavailable Dynamic Code Size (UDCZ)** This is the length of the program trace that is missing because code is not executed in the client app, but on a server instead. This measures the fraction of the program semantics that are out of reach of the attacker.
- **Unavailable Static Data Flow (USDF)** This measures the number of data dependencies (in static program representations) between code present in the static binary and code omitted from the binary. This feature is important because code fragments not available to an attacker are hampering attacks only if the code around the missing fragment actually depends on it.
- **Unavailable Dynamic Data Flow (UDDF)** This measures the number of data dependencies in traces of which either the sinks or the sources are not present in the traces because they are executed on a server instead.
- **Unavailable Data Flow Complexity (UDFC)** This measures the learnability of unavailable code fragments. When a program fragment is missing, but very simple and hence easily learned by the attackers (by observing, e.g., the communication with a server where the original code is executing), it will be easier to undo a protection.

For the static features listed above, we also have to measure *unavailability weights*. Those static features basically measure how much is missing from the static representation of the program. At some point during the execution of the program, the code can become available in memory however. And the longer a fragment resides in memory, the easier it will be for an attacker to obtain a static representation of it, e.g., by means of a run-time memory dump.

5 Concrete Metrics

Section authors:

Bjorn De Sutter, Bart Coppens (UGent)

In this section, we discuss the concrete metrics that we will use to measure the aforementioned features. It is important to note that while these features can be measured on unprotected and protected programs, we will in practice most often not measure them on fully protected code. Instead we will measure them when protections are deployed in isolation or in small combinations.

With regard to dynamic metrics, we want to clarify that they of course depend on the program inputs that are selected to execute and trace the programs. The relevance of those metrics hence depends on the chosen inputs. It is the user's job to select sufficiently representative inputs and to profile the application.

In the specific case of dynamically linked libraries being protected, it is important to trace the invocation of all exported APIs independently, as well as their invocation in a full program trace. The reason they need to be traced independently is that this is the only way to capture the trace-based (i.e., dynamic) metrics as they are relevant for attack steps in which attackers invoke the exported APIs out-of-context.

5.1 Complexity Metrics

5.1.1 Static Code Size

Based on Halstead's metric for program length [15], we will compute the *static program size* (SPS) as the sum of the total number of operators and the total number of operands in a program. In this case, these are the operators and operands appearing in a static program representation, such as a disassembled list of instructions, or the control flow graphs or program dependence graphs constructed from such a list.

5.1.2 Dynamic Code Size

Similarly, we will compute the *dynamic program length* (DPL) as the sum of the total number of operators and the total number of operands executed in a program. In this case, these are the number of occurrences of operators and operands appearing in program traces collected on one or more program inputs.

A natural extension would be to also include coverage based metrics. For example, we could also compute a dynamic program size by counting the number of operators and operands that are executed at least once in a set of program traces. Such an additional metric is not necessary, however, because it is already captured by computing the SPS on a program representation in which instructions that have no SR have been given a simplification weight of zero.

5.1.3 Static Control Flow Complexity

To compute the complexity of the static, graph representations of procedure bodies, we propose to use McCabe's *cyclomatic complexity* (CC) [20]. Rather than using them directly, we will adapt them to incorporate features of Harrison's *nesting depth* [16], and the *cognitive functional weights* [31, 3, 10]. We will also investigate the use of the program dependency graph instead of the program control flow graph to compute the metrics, as proposed by Stetters [28]. These adaptations will result in the metric incorporating more notions of comprehensibility than the original cyclomatic number, which focused more on testability of code.

We also plan to evaluate whether or not the definition and counting of nodes and edges in graph representations of programs needs to take into account the predicated execution of instructions on architectures such as ARMv7. While in some cases predication is simply used to control conditional branches, in which case no special bookkeeping is necessary. In other cases, however, predication is used to control the execution of computations on data. In those cases, a predicated instruction should be considered as an if-then else construct that introduces additional nodes and edges.

As the aforementioned metrics only measure regular program features, i.e., features as found in regular programs, we also have to measure some of the features that are only found in obfuscated programs. In particular, we have to measure the extent to which the control flow is obfuscated by means of indirect, computed control flow transfers (such as complex branch functions [19]) that do not occur in regular programs, or that occur only extremely rarely. This excludes switch statements and other indirect jumps or calls based on fixed, compiler generated instruction patterns, but it includes call-return pairs where the return point is not the instruction following the call instruction.

To do so, we propose to count the number of edges in the whole-program control flow graph that were direct edges or straightforward return edges, but that have become indirect edges or computed return edges in the protected program. We will call this count the *control flow indirection metric* (CFIM).

Additionally, we are considering using the *confusion factor* (CF) metric proposed by Linn et al. [19]. That metric models the fraction of the static code that can be disassembled correctly using state-of-the-art static, recursive-descent disassemblers (like the one from IDA Pro). On the ARM architecture, however, with its fixed word width, it is not yet clear to what extent obfuscation can be used to thwart disassemblers. So the value of this metric is not clear either.

5.1.4 Ill-Structuredness

To measure how well the program structure as observed by an attacker matches the composition of the program semantics into smaller components (i.e, procedures), we propose a novel metric called *procedural ill-structuredness* (PIL).

A basic block “belongs” to a procedure in a control flow graph representation of a whole program if and only if the block is reachable from the procedure entry point through balanced, resolved control flow paths. Balanced means that for every call on the path, there is a corresponding return on the path as well. Resolved means that paths can only include indirect jumps of which the targets are known.

We assume the original program is well-structured by design. In other words, the original program has an ill-structuredness of zero. The ill-structuredness of a protected program then consists of several contributions, which are simply added up.

The first contribution comes from basic blocks “belonging” to more than one procedure (as a result of interprocedural gotos that were added to a program). If we let n_i be the numbers of procedures to which a block i “belongs”, then $\sum_i (n_i - 1)$ is the first contribution to the ill-structuredness metric.

The second contribution comes from inlining, outlining, and the insertion of indirect branches that cannot be resolved statically, with the result that the blocks in a protected program are not partitioned into their original procedures. For this metric, we simply count the number of basic blocks (and their duplicates and replacements) that were an entry point in the original program but are no longer an entry point in the protected version, or that were no procedure entry point in the original program, but are one in the protected version.

Assuming that the code of all procedure bodies in a program will be mixed in the code section, i.e., that the code is no longer grouped per procedure, procedural ill-structuredness will not only be

a theoretical metric. It will also measure a very practical problem for reverse-engineers, because reverse-engineering tools such as IDA Pro have been shown to fail to reconstruct a program's original procedure when deployed on ill-structured code.

5.1.5 Dynamic Control Flow Complexity

Static control flow complexity measures, computed on static program representations like control flow graphs or program dependency graphs, provide a first order approximation of the control flow complexity. For example, in the case of two if-then-else statements following each other, the control flow graph indicates that there are four possible paths: then-then, then-else, else-then, and else-else. However, those graphs do not provide any information regarding how many of those paths will actually be executed at run time. And hence the static control flow complexity metrics do not provide any measurement of the number of different paths that are executed. Such a measurement would be useful, however, as code with few triggered execution paths is much more easily understood than code with many triggered execution paths.

To complement the static control flow complexity metrics, we therefore propose to use the *path coverage* (PC) metric, which will be computed per procedure. If NP_p is the number of possible execution paths throughout a procedure according to its static control flow graph representation, and NP_x is the number of actual paths executed at least once according to the traces, NP_x/NP_p is the dynamic path coverage.

In addition, we add the metric of *path coverage variability* (PCV). As already mentioned, attackers do not always control the whole input to an application. For example, in an online application, or an application protected via online techniques, all inputs coming from a secured server cannot be controlled by an attacker, and might show variation over time, e.g., because the server randomizes its behavior (nonces, delays, remote attestation request diversification, ...). Moreover, many applications feature non-deterministic internal execution, i.e., the observable IO-behavior is deterministic, but internal behavior might be non-deterministic. So when attackers re-execute a program multiple times on (from their perspective) supposedly unchanged inputs, the internal execution of the program might vary considerably. When this is the case, this obviously makes it harder to comprehend the program. For example, it will become much harder to combine information obtained from execution traces collected on different inputs if even the traces on supposedly identical inputs vary.

The path coverage variability metric is computed per procedure. We define NP_{px} as the number of paths through a procedure that, for any user-provided input, might be executed, but does not necessarily have to be executed, i.e., of which the execution depends on the part of the program inputs (or internal non-determinism) not controlled by a user or attacker. Then for each procedure NP_{px}/NP_p is the path coverage variability.

A simplified form of this metric is the *basic block coverage variability* (BBCV). Rather than counting numbers of paths per procedure, this counts, for the whole program the number of basic blocks NBB_{px} of which the fact whether the block is executed or not depends on non-controllable inputs or internal non-determinism. With NBB the total number of basic blocks in a program, basic block coverage variability is defined as NBB_{px}/NBB .

For the above three metrics based on coverage, we will again consider the potential impact on the metrics of predicated execution, as was discussed in Section 5.1.3.

In this current proposal, no multi-threading is considered, and hence non-determinism due to non-deterministic thread scheduling is not yet considered. Since we don't plan to integrate protections that transform the way in which threads of the original program are scheduled, we consider metrics that measure complexities related to multi-threading out of scope of this project.⁵

⁵As for the multi-threaded cryptography protection that we will develop, this protection will inject code with a known number of threads into an application. It will not alter the existing threads in the software to be protected,

5.1.6 Code Layout Variability

In the already discussed metrics, we assumed an abstract program representation where code fragments are linked by edges in graphs or by the order in which they occur in traces. In those representations, concrete addresses are of little to no importance.

In practice, however, code fragments are identified by addresses. In “regular” binary code, the mapping between addresses and instructions is more or less fixed. Within each execution, there is one-to-one mapping between addresses and instructions. In between executions, the only changes in the mapping originate from whole code sections being relocated when address space layout randomization is enabled, each instruction has a fixed address. So even then, most relative code addresses in the program do not vary.

That property simplifies the reverse-engineering of regular binary code a lot, both for humans, and for tools. For example, IDA Pro’s internal operation is completely built on the assumption that each instruction in the program has a unique address and also that instructions never overlap. In other words, each byte in executable code (sections) corresponds to exactly one instruction.

If the property does not hold, this complicates reverse-engineering as well as tampering. To take this complication into account, we propose the *code layout variability* (CLV) metric. This metric is defined as the weighted sum of three terms.

The first term counts the number of instructions in the program that have no a-priori determined location in the program’s address space in memory. This can happen, e.g., because they are loaded as mobile code from a server once an application has started and are assigned a randomized addresses after the download.

The second term counts the number of instructions in the program that have no fixed location during the program’s execution, e.g., because already downloaded mobile code is flushed from the program, and reloaded onto new addresses.

The third term counts the number of instructions in the program that may occur on addresses at which other instructions can also occur during the same program execution. This accounts for mobile code being flushed and overwritten by new mobile code, as well as for self-modifying code.

5.1.7 Static Data Flow Complexity

For measuring the static data flow complexity, we propose to use five concrete metrics.

First, we will count the *number of def-use relationships* (NDUR) in the static program representations. In source code, this can be done for definitions and uses of variables. In binary code, we will do it for register operands and locations on the stack (which correspond roughly to the identifiable variables in binary code). This metric gives an overall impression of the static data flow complexity.

Secondly, in the binary code, we can count the number of *variable-address memory operations* (VAMO). Thus is the number of operations to memory addresses that are not hard-coded in the memory instruction or in the immediately preceding instructions, and that do not address the stack at fixed offsets from the stack pointer. This metric specifically focuses on the number instructions that are harder to analyze statically, because it its unclear from the code alone which state of the program they update.

Thirdly, to compute the *calling convention disruption* (CCD) metric, we will count all instances in the binary code where calling conventions are not respected. In particular, at all procedure entry and exit points, we will count the registers that are used to pass data from caller to callee and back even though those registers are not designated to do so according to the calling conventions. This

however. So we don’t need to measure the change in threading complexity.

metric provides an indication about the a priori knowledge that attackers can use when trying to comprehend interprocedural data flow.

When evaluating or estimating the effectiveness of protections applied to source code, we can add two more metrics.

First, we can measure the *data structure complexity* (DDC), as proposed by Munson and Khoshgoftaar [21]. For example, Munson and Khoshgoftaar attribute a constant complexity to scalar variables, while the complexity of an array increases with the number of its dimensions and with the complexity of the element type, and the complexity of a record increases with the number and complexity of its fields. At first sight, such metrics may seem useless in the context where only the binary programs will be attacked, because binary programs only operate on data in type-less memory locations. Deobfuscating transformations exist, however, that decompile programs to the extent that stack-allocated, statically allocated, or even dynamically allocated heap data are given meaningful semantics. Reps and Balakrishnan, e.g., are able to differentiate between spilled data and procedure parameters in stack-allocated data [25].

Finally, we propose to measure the *static procedural fan-in/fan-out* (SPFIFO) of individual procedures [17, 23]. This is the total number of variables (parameters as well as global variables) that are read resp. written by each procedure. The higher the fan-in and fan-out of a procedure, the more difficult it becomes to abstract the behavior of the procedure. The reason we only propose to compute this on source code, is that these metrics are very hard, if not impossible, to compute exactly on binary code, given the lack of precise points-to information when analyzing binaries.

5.1.8 Dynamic Data Flow Complexity

Whereas some of the static data flow complexity metrics proposed in the previous section give an indication of the potential complexity of the data flow and of the difficulty to analyze data flow statically, they do not provide a good sufficiently precise indication of the actual data flow complexity. For that reason, we propose to complement them with the following dynamic data flow complexity measures, that are to be computed on traces and that consider the complexity of the behavior of instructions that could not be fully analyzed statically.

First, for a metric called *multi-location memory instructions* (MLMI) we propose to count the number of instructions that access more than one memory location during the execution of a program. This can be more than one statically allocated data element, more than one address on the heap, or more than one offset compared to the stack pointer. The count will be weighted with the number of different locations addressed.

Secondly, for a metric called *writable location memory instructions* (WLMI) we will count the number of instructions that read at least once from a non-read only memory location.

Thirdly, we can measure the *memory locations fan-in/fan-out* (MLFIFO) by summing, for each memory location accessed in a program trace, how many different instructions have written to resp. read from the location.

Fourthly, we can compute the *dynamic procedural fan-in/fan-out* (DPFIFO) of procedures by counting, in the traces, how many different memory locations were read/written by each procedure.

Fifthly, we can measure the reuse distances between accesses to memory locations (incl. registers in this case) in the traces [12, 5]. The link between reuse distance and program comprehension was already investigated in literature [22, 29]. The central idea is that human software comprehension is best measured measuring difficulty of “mental execution” of a program, i.e., the execution of a program by a human that interprets the operations in the code and that keeps track of the program state. The difficulty then mostly comes from the limited short-term memory, which means that humans have a hard time remember a lot of program state. For accessing state they cannot store in their shared memory (because a value is not reused fast enough, but other values have been used in between), humans will have to invest more effort (e.g., by looking it up on paper or by

entering a print statement in a command-line debugger).

For the *reuse distance* (RD) metric, we accumulate a cost for each instruction in a trace, with the cost depending on the reuse distance of the accessed memory locations. For example, the cost might be zero for reuse distances less than 4, and some non-zero constant for larger reuse distances.

This metric can be refined, like Nakamura et al. did, by adjusting the costs when the read memory location is a read-only location, or a location to which only one preceding write operation was performed.

Sixthly, we can measure the *instruction operand variation* (IPV). For this metric, we accumulate, for all static instructions, the number of different operand values observed in all occurrences of the instruction in the trace. The rationale is that an instruction's role in a program's semantics is harder to understand if it operates on many different values.

Seventhly, we can measure the *instruction operand type variation* (IOTV). For this metric, we accumulate, for all static instructions, the number of different operand value types observed in all occurrences of the instruction in the trace. The different types are booleans (i.e., one and zero), small integers, addresses in statically allocated sections of the program, addresses on the stack of the program, and addresses in the heap of the program, and large integers that are not a valid address. The reason is again that the role of an instruction in a program is harder to understand if it operates on multiple, seemingly unrelated types of data.

Finally, we propose *heap dynamics* (HD) and *heap complexity* (HC) as a metric. In these metrics, we capture the dynamic nature of the heap, i.e., how much its pointer structures vary over time, and the complexity of the pointer structures on the heap over time.

To measure the dynamic nature of the heap memory, we will simply count the number of relevant changes to the heap memory. So in the metric *heap dynamics*, we simply accumulate, over a program trace

- how many times blocks are allocated, reallocated, and freed from the heap;
- how many times a pointer to the heap is stored on the heap by an executed instruction;
- how many times a pointer value on the heap is overwritten by a non-pointer value;
- and how many times memory blocks containing pointers are freed.

To measure the complexity of the pointer structures of the heap memory, one option would be to consider the heap as a directed graph: Heap blocks are nodes, and pointers from one block to the other are edges. The complexity of this graph can then be computed, e.g., whenever blocks are allocated or freed on the heap. However, recomputing that complexity many times from scratch (as would be needed for lengthy traces) would be very time consuming, and hence not realistic. Another alternative would be to rely on shape analysis, but that will also be too slow.

Instead, we propose to simply count the maximal fan-in and fan-out of individual blocks on the heap, i.e., the maximal number of pointers pointing into or out of each block during the blocks lifetime. These can easily be computed by keeping track of a shadow heap while iterating over a trace. So for the *heap complexity* metric, we accumulate, for all blocks ever allocated to the heap, the maximum fan-in and fan-out during their lifetime.

5.1.9 Semantic Dependencies

Consider a trace of a protected client application that connects to a secure server, and in which the correct execution (or even the continuation of the execution) after receiving input from that server depends on the output previously passed to the server. In essence, this amounts to a remote attestation technique. When the exact dependency is known (i.e., the control logic on the server side is known), the points in the trace can be identified where some valid output needs to be produced. We can call these the attestation points. We can also identify the points where execution

will halt or divert from the correct execution if those outputs have not been not produced, which we will call reaction points. We can the also determine which reaction points related to which attestation points.

On that basis, we propose the *trace reproducibility* (TR) metric, which accumulates, for each instruction in the trace, the number of preceding reaction points weighed by the number of attestation points on which the reaction points depend.

Similar to the remote attestation case, we can define program points at which code guards or anti-debugging checks are executed, and count, for each instruction in the trace, the preceding number of those points in the trace.

Similarly, we can count how many context-switches between the debuggee and the debugger occur before a certain point is reached in the execution of a program protected with the self-debugging protection developed in this project.

5.1.10 Static Data Presence

This feature indicates whether or not sensitive data are visible, and more or less easily identified, in the static program representation. Metrics for this feature hence need to depend on the specific data values that need to be hidden.

The first metric we propose is quite trivial: *static plain data presence* (SPDP) for some values to be hidden is one when the data is present in the binary, and zero when it is not present. For checking its presence, the binary will be scanned for several representations of the data: string representations (in ASCII and unicode), signed and unsigned, little-endian and big-endian integer representations of numbers, etc.

A simple extension of this metric is that the presence of “significant” components of the data, such as substrings, would be checked instead of the whole data. In that case it is up to the user of the metrics to define what “significant” means, and to provide some kind of metric function. This is not a major burden, since he already has to identify the data to check anyway. In practice, this will be done through code source annotations. Alternatively, some default metric functions can be defined,

The second metric we propose is *static ciphered data presence* (SCDP). This is a customizable metric, for which the user has to provide a number of encryption or encoding routines. Rather than measuring the presence of the original data, this metric will then report the presence of an encrypted or encoded version of the data. This metric will also contain a set of standard encoding algorithms, such as the 256 XOR-variants that XOR each byte of the data with the same constant.

5.1.11 Dynamic Data Presence

In line with the static data presence metrics defined in the previous section, we propose two dynamic variants: *dynamic plain data presence* (DPDP) and *dynamic ciphered data presence* (DCDP). Rather than checking for the sensitive data or parts thereof in the binary of the program, we will check them in the memory space of the running program. This can be achieved by means of instrumentation or by emulating a recorded trace. Possible extensions to investigate are:

- In some cases, it might be OK that some values still occur in memory during the execution of a program, as long as attackers are not capable of identifying that the occurring value is the one they are interested in. But how should we measure this? Data dependency chains to data/instructions/events that can serve as hooks to attackers?
- If encoded values occur in the program, how “learnable” is the relation between encoded and decoded values?

5.1.12 Syntactic Stubbornness

This feature indicates whether or not, and to what extent, altering the syntax of a code fragment, i.e., tampering with the code, has an impact on the correct execution of a program. This metric is particularly useful for measuring the impact of anti-tampering techniques such as code guards, either used in an offline fashion or for remote attestation.

We propose three initial metrics for this feature. The first is a static metric: *static syntactic stubborn code size* (SSCS). It is simply the size of the code fragment that is guarded.

The second and third metrics are more dynamic, as they assess the impact on the program's execution in case tampering is detected. Obviously, tamper detection is only useful if, as a result, the program fails to some extent.

Our second metric is *syntactic stubbornness asset impact* (SSAI). It measures how much code related to assets to be protected no longer appears in program traces when any single byte in a guarded region is altered and that tampering is detected. This metric can simply be measured by counting instructions in program traces in which the decision points (verifications) and reaction points of the anti-tamper detection techniques are identified. Those points are available in case the verification code and reaction code has been injected by the protection tool chain. This metric is of interest for attack steps that try to tamper with a certain code fragment in order to enable the execution of other fragments that relate to assets to be attacked in later attack steps.

The third metric is *syntactic stubbornness output impact* (SSOI). It measures the impact of tampering on the correct execution of the program. We propose to simply measure how many output bytes are corrupted or go missing in case any single byte in a guarded region is altered and that tampering is detected. This metrics is of interest for attacks that try to alter the code to avoid security features and workaround protections while still maintaining the user-observable semantics. This metric can simply be measured on a program trace.

5.2 Resilience Metrics

5.2.1 Static Variability

Static variability is a feature that cannot be measured on individual programs. By definition this feature also has to capture the variation in the way a protection is applied to multiple programs to quantify potential learning effects and the potential for attackers to develop targeted tools. We will hence not measure variability.

We will quantify it, however, for each protection, and include this metric in the overall evaluation the strength of (combinations of) protections. At this point in time, it is not yet clear whether one or more metrics are needed for this feature.

5.2.2 Intra-Execution Variability

As already discussed, Yadegari et al. have recently shown that very powerful, automated attacks can be engineered that rely on, in Yadegari's terms, quasi-constant behavior [32]. The central idea is that an instruction in a program that can alter the program state in many different ways according to the processor architecture specification, but that according to a trace always alters the program state in the same way, can be replaced by a simpler instruction without affecting the semantics of the program.

Consider as an example the already mentioned conditional branch instruction. If a specific conditional branch in a program branches in the same direction every time it is executed, the attacker can simplify his program representation and the apparent program semantics, by replacing the instruction with a non-conditional branch. As discussed in Section 4.2, this type of simplification

is typically what an attacker will try to undo or to work around protections. And as Yadegari et al. have shown, this type of simplification, if done right and extensively, can fully automatically break protections that were until now considered pretty strong, including obfuscation by means of custom instruction set interpreters and return-oriented programming. Other forms of quasi-constant behavior are source operands or destination operands that are constant. When a source operand of some instruction proves to have the same value throughout a trace, the instruction can be considered as having an immediate operand instead of a register or memory operand. If an instruction always produces the same value for its destination operands, the instruction can be replaced by a simpler one that does not even have any source operands.⁶

The instruction features of intra-execution variability (of source operands, of destination operands, and with respect to control flow) are the inverse of quasi-constantness features. We prefer the former because more variability relates to stronger protection, as is the case for all already proposed metrics.

These features will initially not be measured on a scale. Instead, the intra-execution variability (IEV) metric are simply boolean values that will be determined by analyzing traces, and that label instructions to indicate whether their features are constant or not throughout a trace.

The labels, in conjunction with features of the attack step against which some protection combination is evaluated, will then be used to compute the simplification weights introduced in Section 4.3.

In the future, we will consider measurements on a scale to quantify the extent to which the variable behavior is context-sensitive, i.e., the extent to which code duplication can be used by an attacker to replace variable behavior by constant behavior. For example, if the behavior of some procedure F is variable overall, but it is constant when F is called by caller A, and constant when F is called by caller B, then this might also be exploited by an attacker to simplify the program representation and semantics.

5.2.3 Semantic Relevance

Instructions in a program trace that do not (directly or indirectly) contribute to the program output, can be removed without affecting the program semantics. Conceptually, they can be replaced by no-ops that have no source operands. These instructions can be determined through data dependency analyses on (simplified) traces (e.g., by means of dynamic slicing techniques).

In this regard, it has to be pointed out that only relevant program outputs should be considered, which may differ from one attack step to another. For example, if some code of a client program is split off from that program and executed on a secure server instead of on the client device, the communication between the client and the server can be considered input/output of the client: If the client at some point sends data to the server, this data is output. If the client then receives an answer, this is input. But if this input is not relevant to the attacker in some attack step, then neither is the data sent to the server.

Like computations that do not contribute to relevant output, computations (in a trace) that do not depend on program inputs can be replaced by instructions that do not have any source operands. These instructions can be determined through (precise) taint analyses on traces.

Moreover, instructions that are never executed in any trace can be considered irrelevant.

As we did for intra-execution variability, we will label instructions that are irrelevant for the program semantics, and those labels will again be used for determining the simplification weights

⁶Note that it does not matter whether the simplified instruction actually exists in the used processor architecture. The relevant effects of the simplification initially only have to be modeled in the more abstract program representations that attackers used. If actual simplified code needs to be generated to be executed (in preparation of another attack step), it suffices, and is always possible to implement the simplified operation as an instruction sequence (that has less or no live-in operands).

introduced in Section 4.3.

In doing so, we will not model specific capabilities of attackers to undo protections, i.e., lacks of resilience against specific forms of attacks. Instead, we will quantify the potential of attackers to undo protections.

5.2.4 Stealth

Apart from invariable code, invariable behavior, and semantic irrelevance, there might be other features that allow attackers to identify code that is not part of the original program, but that instead implements a certain protection, and that can hence potentially be simplified or removed from the program without affecting the program's overall semantics.

For example, in the malware detection world, supervised machine learning techniques have been developed for training malware detectors to recognize features specific to obfuscated code (look for citation). So far, we have not investigated the potential to use similar features in software protection metrics in ASPIRE. We will study them in more detail in the future.

5.3 Unavailability Metrics

The measurable unavailability features proposed in Section 4.3.3 have direct counterparts in the complexity features proposed in Section 4.3.1. Whereas complexity features are computed on present code, unavailability features are computed on missing code, i.e., on code present in the vanilla application, but missing in the protected application. As they are so clearly linked, we refer to the corresponding subsections of Section 5.1 for a description of concrete unavailability metrics.

As for unavailability weights, those can be computed very easily from program traces. It suffices to identify the system and library calls that map code into memory and that free that memory, and to measure the number of executed instructions in between the appearing and disappearing of code in memory.

5.4 Using the Metrics in Practice

In the first iteration of our metrics proposal, in D4.02 Section 6, in year one of the project, we proposed concrete formulas to aggregate the different metrics to model the effort attackers have to invest in individual attack steps. Fundamentally, we proposed to use weighted sums. In Section 6, we discuss how we currently use the metrics in our assessment tools, in ways that are largely in line with our original ideas.

In general, however, many internal discussions revealed that there probably is not one formula that can be used in all attack steps. For some attack steps, absolute metric values, such as the total code size of a fragment present or missing from the binary, seem to be the most interesting (after normalization). For other attack steps, relative values seem a better fit, such as the fraction of the program code that is missing, which can be obtained by dividing one metric by another. In yet other cases, some metrics might require being multiplied.

Furthermore, the tiger experiments conducted late in the project have led to the important insight that the metrics computed and used can differ from one attack step to another. For example, in an attack path that centers around invoking individual exported functions from a protected library, the used dynamic metrics should be computed based on traces collected for invocations of isolated exported functions, not on traces collected for a whole program execution.

In short, we now believe that ideally, the aggregation of metrics into numbers that can be used to compare the estimated effort of attack steps on differently protected program version, is customized for each attack step and the features of the attack step (listed in Section 4.2.1). Within the

project, the resources and time were missing to reach a more concrete proposal. So for the time being, the simpler method discussed in Section 6 is used.

5.5 Concrete Metrics Support

We implemented different metrics in the ASPIRE project. These range from metrics that report on the size of code fragments, to structural complexity metrics. Both static and dynamic metrics were implemented. Most metrics are automatically computed when the ACTC is run on a project. In the case of the dynamic metrics, the ACTC has the ability to run the protected application to collect the required dynamic information. In addition, we implemented some dynamic metrics that currently are not integrated in the ACTC, as these require a slow and large emulator to be run.

5.5.1 Metrics integrated in the ACTC

In Section 10 of Deliverable D5.11, we describe in detail how the ACTC generates and collects both static and dynamic metrics. All metrics are collected both over the entire program, and over every individual code region that has been annotated with ASPIRE protection annotations. The output files contain a line for each code region, which has several fields, one field for each metric. To make manual inspection of these files easier, we have added a header to this file, which contains a short name for the metric recorded in every field.

We have the following static complexity metrics (and their respective short names):

- *Total number of instructions* of a region. In the output files, we refer to it as `nr_ins_static`.
- *Static Program Size (SPS)* as defined earlier in this section. In the output, refer to this metric as `halstead_program_size_static`.
- *Control flow indirection metric (CFIM)*, to count the indirect control flow transfers, as defined earlier in this section. In the output file, this metric is referred to as `nr_indirect_edges_CFIM_static`.
- *Control flow edges*, which counts all control flow edges (including fall-through edges). This is referred to as `nr_edges_static`.
- *McCabe's cyclomatic complexity*, which we refer to as `cyclomatic_complexity_static` in the output files.

Furthermore, we can generate the following dynamic metrics:

- *Total number of instructions executed*, which is labeled as `nr_ins_dynamic_size` in the output.
- *Dynamic program length (DPL)*, which is a dynamic variant of the Static Program Size, where all instructions and the number of operands are weighted by their execution count. We refer to this as `halstead_program_size_dynamic_size`.
- *Coverage-based program size* is a coverage-based version of program size. Rather than weigh everything according to the execution counts, we give every executed instruction a weight of 1. In the output files, this is `halstead_program_size_dynamic_coverage`.
- *Dynamic control flow indirection metric* counts the number of indirect control flow transfers that have been executed. In the output it is `nr_indirect_edges_CFIM_dynamic_size`.

- *Dynamic control flow edge count* counts how many control flow edges have been followed in the program execution. These are labeled as `nr_edges_dynamic_size`.

In addition, the ACTC generates some metrics about the code fragments that have been made mobile, and code fragments whose integrity has been protected. These metrics are:

- The number of bytes stored in the Area Data Structure for each region of code to be guarded/attested.
- The number of blocks that comprise every region in the Area Data Structure.
- The number of bytes to be attested for every region.
- The number of mobile blocks for every per region to be made mobile.
- The total size of the mobile blocks per region to be made mobile.

5.5.2 Metrics not integrated in the ACTC

The static metrics integrated in the ACTC either report on the static control flow graph, whereas the dynamic metrics integrated in the ACTC are obtained by statically instrumenting the protected binaries to keep track of execution counts of basic blocks. These execution counts are then fed to a post-pass to compute simple dynamic metrics. However, trying to compute more complex dynamic metrics quickly runs in limitations of static instrumentation on the binary level. Furthermore, the metrics should ideally take into account that the execution of the protected programs can be spread over multiple processes, such as in the case of anti-debugging. In that case, the control flow analyses and the data flow analyses should be able to propagate their information over process boundaries.

To solve these issues, we looked into whole-system dynamic analysis frameworks. Dynamic frameworks allow to more easily keep track of data values produced at run time, and to more easily keep track of the actual execution flow of a program. Whole-system frameworks allow us to keep track of the entire operating system and all processes running inside of it. This allows us to track multiple communicating instances of a protected binary and to compute metrics correctly even on programs protected with anti-debugging.

We picked the Panda dynamic analysis framework⁷ [13] as a basis to compute these metrics with. This analysis framework builds on the QEMU whole-system emulator, and allows to record an entire operating system and all the processes running inside in it (including the application that we want to compute metrics on) and replay this recording to perform analyses and computations. It is extensible through a plug-in architecture, which we used to implement the dynamic analyses for these metrics. The only downside is that the version of QEMU Panda currently is based on does not support our anti-debugging technique because it does not correctly implement some hardware debug registers. However, we think that once Panda is updated to a more recent version of QEMU (which contains improved support for these registers), this issue will be solved.

We implemented the following three dynamic metrics on top of Panda:

- *Memory reuse distance*: for each read to a memory location, we count the number of instructions executed since the last write to that memory location. This data can be adapted to fit the *reuse distance (RD)* metric defined earlier.
- *Variation of produced memory data*: for each memory location, we keep track of whether or not the data value in this location ever changes. This metric is related to the *instruction operand variation (IPV)* metric defined earlier, but when only taking memory writes into account.

⁷<https://github.com/moyix/panda>

- *Trace variation*: for each (dynamic) control flow transfer, we count the different number of execution paths through this transfer. We define ‘execution path’ as a sequence of consecutive control flow transfers. The length of the paths that are considered is configurable to the users of this metric. Both the number of control flow transfers before and after the transfer under consideration can be customized. This implements the *dynamic path coverage* metric specified earlier, with a customizable length of the path.

These metrics are all computed across multiple processes. We also had to deal with applications protected with code mobility. With code mobility, code fragments from the application are downloaded from a server. These fragments can be downloaded in any arbitrary order, and they can be loaded at arbitrary memory locations. When these fragments are executed, we need to establish a mapping between the executed code and the regions on which we want to compute metrics. The output produced by the binary tools also includes information about the internal details and location of the data structures that are used by the code mobility protection at run time. This meta information allowed us to add introspection capabilities for mobile code to our Panda plug-in.

To actually collect the metrics, we set up an ARM Android QEMU system image, so that QEMU acts as an actual Android development board. On this emulated board we then run our protected applications. Panda records the execution of the program, and the metrics are computed in a separate analysis step on the execution replay.

To make it easy to compute the metrics on an application, we wrote a script to automate the different steps that are required to compute the metrics:

1. We automatically start the emulator with the correct parameters.
2. We automatically push the protected program to the emulated device.
3. We instruct Panda to start recording the execution once the program is pushed to the emulator. We do so at this point in the sequence, because recording and replaying the booting of the board and pushing it would needlessly slow down the analysis and metrics computation.
4. We actually execute the program on the board.
5. We stop recording once the program has finished its execution.
6. We extract meta-data from Diablo’s output to be able to introspect the mobile code blocks.
7. We perform the actual dynamic analyses.

We performed some qualitative performance measurements on these techniques. While the record phase of Panda has relatively low overhead of less than 100%, the replay phase with our plug-ins enabled has an overhead of 10x. Furthermore, this counts only the record phase, not the booting of the emulated Android board in Panda. These overheads make these techniques less suited to be integrated for the ADSS where a significant amount of metrics might need to be computed.

6 Software Protection Assessment with ADSS-Light

Section authors:

Paolo Falcarin, Gaofeng Zhang (UEL)

This part presents the ADSS-Light, a tool that help the software developer in manually modelling the possible attacks to the application assets with Petri Nets and the users to assess the strength of the protected program by evaluating combinations of binary code metrics. The ADSS-Full tool (released in Deliverable D5.10, and whose architecture is described in Deliverable D5.11) aims at automating as much as possible the search for the golden combination of protections, in order to help non-expert software developers; on the other hand, the ADSS-Light aims at helping expert software developers with a semi-automated approach to assess the security strength of one or more protection instantiations, after some initial configuration steps.

According to the ADSS terminology (see Deliverable D5.11), we recall that a *protection instantiation* consists of a protection technique (e.g., anti-debugging) coupled with its configuration parameters: the same protection technique can offer multiple protection instantiations, depending on the particular combination of the values of its parameters. An *applied protection instantiation* is a pair consisting of a protection instantiation and a part of the application (usually an asset) where such protection should be applied.

The next sections document the architecture of the ADSS-Light tool, with instructions on how to install and configure the tool; then the manual configuration steps consist of associating binary code metrics and code regions to attack steps, customizing the metrics formulas, and finally choosing the type of assessment. The above-mentioned steps are then applied to the NAGRA crypto-libraries (*libnvdrmplugin* and *libnvcryptoplugin*), which are part of one of the ASPIRE industrial use-cases: the application of ADSS-Light to the NAGRA use-case is documented in the D4.06 Annex I.

6.1 The ADSS-Light Architecture

The ADSS-Light is part of the ASPIRE ADSS framework and it is used to assess and compare different protection instantiations, according to different methodologies. The ADSS-Light permits the evaluation of the strength of a protection instantiation against a set of attacks that can be applied to the application's assets. As shown in Figure 15, the ADSS-Light has two main components: the Petri Nets editor and the Software Protection Assessment (SPA) tool. The SPA tool contains the Protection Fitness (PF) function module, and the Petri Net (PN) Simulator. The PN simulator has been already described in Deliverable D4.03, Section 5.3.

The ADSS-Light is utilized to assess protection instantiations through the Protection Fitness function which combines binary code metrics computed by the ACTC. The user can configure the protection fitness function by choosing different evaluation methods and by assigning different formulas to combine such metrics; such assessment of a particular protection instantiation can be done manually or it can be invoked via the SPA API by the ADSS-Full when searching for the golden combination.

6.1.1 Protection Fitness formulas

This section summarizes the concepts defined in the previous Deliverable D4.03 for defining the overall Protection Fitness (PF) function, and then introduces the new Normalized Protection Fitness (NPF) function. The PF function is utilized to numerically assess the additional software protection brought by a particular Protection Instantiation (also known as Protection Configuration in previous deliverables) applied to the program by means of the ASPIRE Compiler Tool Chain

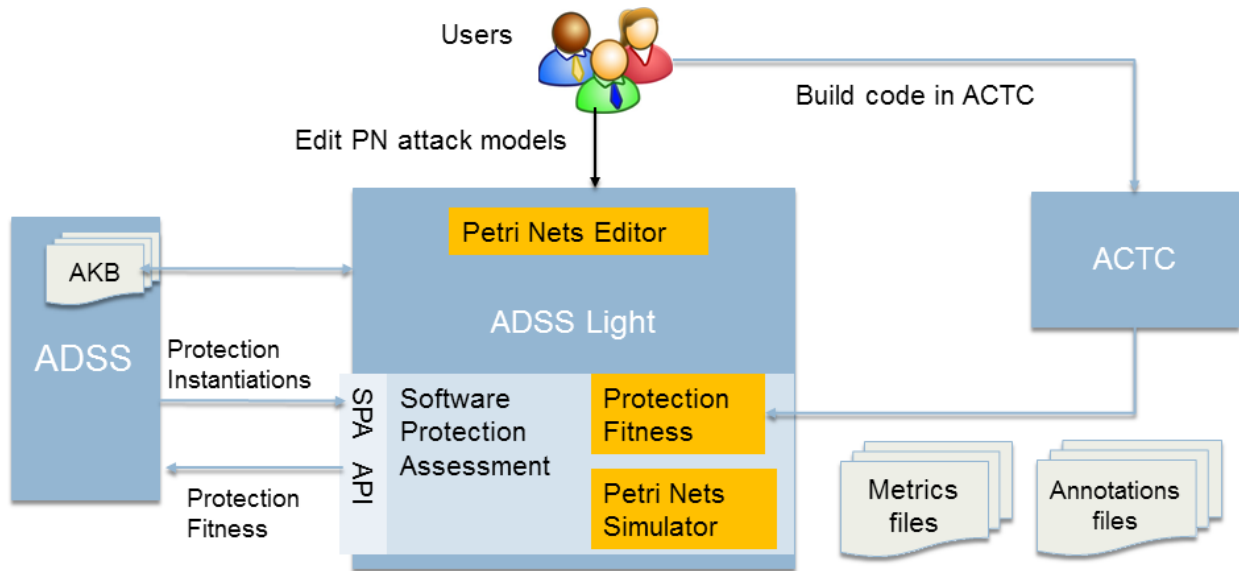


Figure 15: ADSS-Light architecture

(ACTC). In our Petri Net (PN) based attack models, each transition (attack step) can be associated with a different set of binary code metrics, whose values are computed by the Metrics Framework included in the ACTC; the overall PF function of a particular Protection Instantiation is obtained by composing the metric formulas of the attack steps.

The equations 1–5 has been presented in other ASPIRE documents, such as Section 5 in Deliverable D4.03, and they are summarized here to make this section self-contained.

We define all metric formulas (one per attack step) as follows:

$$F = \{f_1, f_2, \dots, f_t\} \quad (1)$$

where t is the number of attack steps, i.e., the number of transitions in the Petri Net model.

The Metric Potency is defined as the ratio between $m_j^a(P_k)$, the metric computed after the protection instantiation P_k is applied to the code, and m_j^b , the same metric applied to the unprotected code; hence, the following formula represents the metric formula for a single attack step i , based on a linear combination of metrics potencies, as defined in Deliverable D4.03:

$$PF_i = f_i(P_k) = w_{i,1} \times \frac{m_1^a(P_k)}{m_1^b} + w_{i,2} \times \frac{m_2^a(P_k)}{m_2^b} + \dots = \sum_{j=1}^p w_{i,j} \times \frac{m_j^a(P_k)}{m_j^b} \quad (2)$$

In this equation, P_k is the Protection Instantiation under assessment, $m_j^a(P_k)$ are the software metric values AFTER protection configuration P_k has been executed, and m_j^b is the software metric value BEFORE protection configuration P_k is executed. We call the resulting value of the calculation of $f_i(P_k)$ as the Protection Fitness PF_i of attack step i . The equations 3 and 4 represents the vectors of metrics values, whose size p is the number of software metrics utilized. We can interpret M_b as the vector of metrics calculated on the 'vanilla' application, related to the particular Protection Instantiation with zero protections applied.

$$M_a(P_k) = \{m_1^a(P_k), m_2^a(P_k), \dots, m_p^a(P_k)\} \quad (3)$$

$$M_b = \{m_1^b, m_2^b, \dots, m_p^b\} \quad (4)$$

The $w_{i,j}$ parameters in the various metric formulas (one per attack step), are stored in the Weight Matrix ($t \times p$) for metrics in each transition:

$$WM = \begin{pmatrix} w_{1,1} & \cdots & w_{1,p} \\ \vdots & \ddots & \vdots \\ w_{t,1} & \cdots & w_{t,p} \end{pmatrix} = (w_1, \dots, w_p) \quad (5)$$

If the value of $w_{i,j}$ is zero, it means that the related software metrics: m_j has no influence on the transition T_i , so it will not appear in the actual PF formula of that attack step.

All weights and metric formulas of transitions from PN based attack models can be decided by security experts, based on their personal knowledge on the specific software attacks. These data can be stored locally or in AKB of the ADSS-Full. If not specified by the user, all the weights of the chosen metrics are set at 0 by default, while the weights of the metrics associated to an attack step are set to 1. For example, the user can associate to an attack step the simple metric formula: $f_3 = SDFC + DDFC$, where there are only two metrics: SDFC and DDFC; this means that all other metrics' weights $w_{i,j}$ are still zero, while the weights corresponding to these two metrics are set to 1. More in general, the user can further customize the metric formulas, by inserting its own customized weights; for example, in case of the metric formula: $f_3 = 0.2 * SDFC + 0.1 * DDFC$, SDFC and DDFC are associated to the attack step and there are two weight values: "0.2" and "0.1", respectively.

In a Petri Net model there are different attack paths that can be followed to achieve the final goal: each attack path is a sequence of attack steps. For example, in the simple Petri Net of Figure 16 there are two attack paths: $AP_1 = T_1, T_2, T_3$ and $AP_2 = T_1, T_4, T_5$.

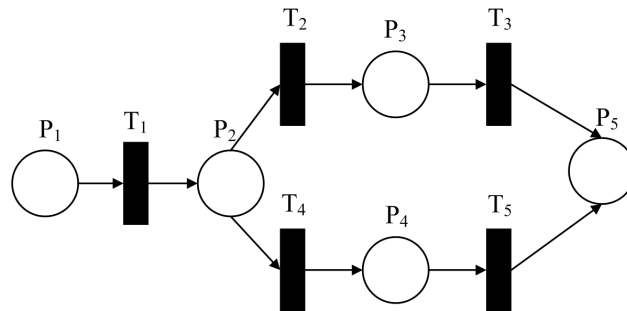


Figure 16: Example of Petri Net with two attack paths

For each Attack Path AP_i in the Petri Net model we can define its own Δ Effort (ΔE) as the sum of the Protection Fitness values PF_i calculated on the n Attack Steps which are included in the attack path AP_i , using the metrics values related to the protection instantiation P_k .

$$\Delta E(AP_i, P_k) = \sum_{j=1}^n PF_j(P_k) \quad (6)$$

The overall Protection Fitness is then defined as the minimum Δ Effort among the ones calculated for all the attack paths:

$$PF(P_k) = \min_i (\Delta E(AP_i, P_k)) \quad (7)$$

6.1.2 Normalized Protection Fitness

In the ADSS-Light is now possible to choose which type of Protection Fitness to use: the default one based on metrics potency (summarized in the previous section) or the new Normalized Pro-

tection Fitness (NPF) function in equation 8, which has been added to guarantee that in the attack steps' metric formulas each addend of the sum can be normalized in the range [0..1]. In this way all metrics involved in a metric formula can vary between 0 and 1. After trying different Protection Instantiations with the ACTC, and calculating the corresponding metric values, the ADSS-Light can keep track of the maximum value for each metric and use it to normalize the metric formulas in the following computations of the Protection Fitness. We define $MAX(m_j)$, as the largest value of the software metric m_j among the ones calculated after trying different Protection Instantiations.

$$NPF_i(P_k) = \sum_{j=1}^p w_{i,j} \times \frac{m_j^a(P_k)}{MAX(m_j)} \quad (8)$$

Similarly to the default Protection Fitness, for each Attack Path AP_i in the Petri Net model we can calculate its own Δ Effort as the sum of the Normalized Protection Fitness values PF_i calculated on the n Attack Steps which are included in the attack path AP_i , using the metrics values related to the protection instantiation P_k .

The overall Protection Fitness $NPF(P_k)$ is then defined as the minimum Δ Effort among the ones calculated for all the i attack paths:

$$NPF(P_k) = \min_i \left(\sum_{j=1}^n NPF_i(P_k) \right) \quad (9)$$

When ranking the different protection instantiations to find the one with the best protection fitness, the ADSS-Light automatically recalculates the Protection Fitness of all the Protection Instantiations previously evaluated by the user, because the maximum values of the metrics has likely changed over time. We are going to apply both the PF and NPF function on the NAGRA use case in the following sections.

6.2 ADSS-Light Workflow

This section describes the main workflow of the ADSS-Light, as a sequence of features of the tool that have to be used to create an attack model with its Petri Net editor, and then to estimate the additional security of the protection instantiation, by invoking the Protection Fitness (PF) function. In the ADSS-Full workflow, the ADSS-Full populates the knowledge base with all the possible *applied protection instantiations* and pass them to the PF function by calling the SPA API (see Section 6.3.3) to estimate their security strength while searching for the golden combination. The ADSS-Light shares many data structures with the ADSS-Full but it can also work as an independent tool for manually choosing and evaluating some protection instantiations, as depicted in the workflow in Figure 17.

The ADSS-Light work-flow consists of five phases:

1. **Petri Net Attack Model Editing** — users can edit a Petri Net attack model with our extended Petri Net editor, as introduced in D4.03, Section 5.1: the Petri Net is composed of Attack Steps fetched from the AKB and the user can add new custom attack steps. The Petri net attack model can be edited with this tool and stored locally in a PNML file; then it can be added to the ASPIRE knowledge base (AKB) to enrich the knowledge base and then reloaded from the AKB.
2. **Metrics-Attack Steps Association** — users can associate metrics and attack steps, as introduced in D4.03, Section 5.1. To model these associations, users can edit the "labels" in PN models to define the metric formulas to combine such metrics for each attack step.

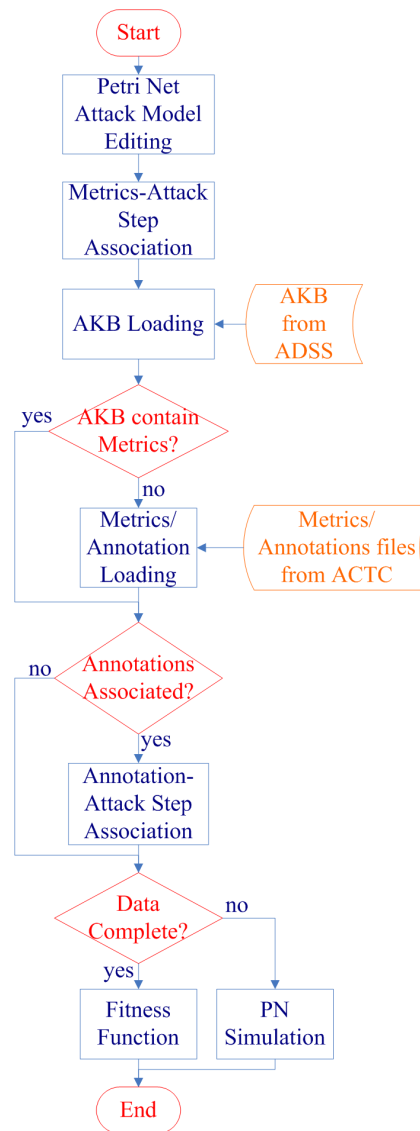


Figure 17: ADSS-Light Workflow

3. **AKB loading** — users need to specify the AKB file and load it to obtain the protection instantiations pre-calculated by the ADSS-Full.
4. **Metrics/Annotation Loading** — users need to specify the metrics files and annotation file, and load them to obtain metrics data and annotation data; it is an optional phase, if the AKB from ADSS-Full already includes full metrics and annotation data (inserted in a previous session), this phase can be skipped.

If not done yet, in this phase the ADSS-Light will trigger the build of the code to apply the protection instantiation. Code annotations are extracted from the source code files and stored in a separate *annotations.json* file: these annotations are later used by the binary transformations. This phase will then take as an input the metrics files and *annotations.json* files (whose formats have been defined in D5.11).

5. **Annotations-Attack Steps Association** — users can associate annotations to attack steps: this is an additional configuration task to get a more fine-grained assessment by combining metrics calculated on specific code regions:
 - if an attack step has no code regions assigned, then the corresponding metrics are calculated on all code regions;

- if more than one code region is assigned to an attack step, then each single metric value associated to such attack step is calculated as the mean value of the metric values calculated on the code regions associated to this attack step;
- if no code region is specified then the metric will be calculated on all the code regions annotated.

When code regions are associated to the attack steps, then the single metric $m_i^a(P_k)$ in an attack step is calculated as the mean value of the metric values $m_i^a(P_k, C_j)$ calculated on the n different code regions C_j associated to that attack step:

$$m_i^a(P_k) = \frac{\sum_{j=1}^n m_i^a(P_k, C_j)}{n} \quad (10)$$

When calculating the Protection Fitness (PF), these mean values of the metrics will be used for $m_i^a(P_k)$ in equation 2; similarly, when calculating the Normalized Protection Fitness (NPF), they will be used in equation 8. These detailed metric values are read from the metrics files produced by the ACTC after building the program with the P_k protection instantiation.

It is important to remind that the ACTC protects only the annotated code regions and ignores the rest of the code. This phase is optional, thus when this phase is not selected, the tool performs the security assessment using the metrics values calculated on the whole program; more precisely, it is actually performed on all the code regions identified by all the annotations defined in the program. When this phase is selected, then only the subset of metrics values related to the associated annotations are utilized in the computation of the Protection Fitness.

- 6. Protection Fitness Function** — users can run this function to obtain estimate the security of the protection instantiation. As stated in the previous section, ADSS-Light offers two different assessment methods to calculate the Protection Fitness: the default Protection Fitness (based on Metrics Potency), or the new Normalized Protection Fitness (NPF).
- 7. PN Simulation** — when some attack steps have no metrics associated, then the user can run the Petri Net Simulator (introduced in D4.03, Section 5.3), to obtain the simulation results for protection assessment. By default, the PN simulator uses random variables for each attack step with no associated metrics; such random variables represent an approximation of the expected value for the Protection Fitness PF_i of attack step i . The user can manually insert a minimum and maximum value for each of these random variables, so that the ADSS-Light can randomly choose a number in this user-defined [min,max] range with a uniform probability distribution (i.e., each value in the range can be picked with the same probability by the PN simulator).

In the workflow, phases 1, 2, 6, and 7 have been previously introduced in D4.03, while phases 3, 4 and 5 are the manual configuration steps, which read and parse the metrics and annotations files generated by the ACTC build process, and combine such data in the software protection assessment processes in the ADSS-Light.

The inputs required by the ADSS-Light are:

- **Metrics files:** These files are generated by ACTC, they include the software metrics values for vanilla and protected applications. These metric data are essential for the assessment processes in the ADSS-Light. The metrics file format has been introduced in D4.04, and the ACTC metrics framework currently computes 7 code metrics values.
- **Attacks and Protections data from AKB:** these data come from AKB and ADSS-Full. The ADSS-Light can load directly the attack paths and protection solutions from AKB to carry out the assessment.

In the ADSS-Light results, there are two parts: the protection fitness value (returned by the PF function) of one protection instantiation and the detailed logs.

- The protection fitness of a protection instantiation on attack paths or one specific attack path is a rational number that determines the effectiveness of protection solutions: the higher, the better. Based on this number, the ADSS-Light can compare different protection solutions to identify the best one.
- The detailed logs on the assessment process include all formulas and data for assessment on each attack step. These are saved as text files in the Eclipse project where the ADSS-Light is executed.

The Petri Net editor in the ADSS-Light can be used in two ways:

1. to define the Petri Net attack models and store them locally in PNML, the XML-based standard Petri Net Markup Language used to represent Petri net models;
2. to populate the AKB with attack models, by converting the PNML models into OWL, the XML-based standard Web Ontology Language utilized to store the AKB, by using the AKB Java API provided by the ADSS-Full.

The Software Protection Assessment component in the ADSS-Light can be used in two ways:

1. receiving from the ADSS-Full only the set of Protection Instantiations: in this case the software protection assessment is performed by the ADSS-Light on the attack paths derived from the local Petri Net model;
2. receiving from the ADSS-Full the set of Protection Instantiations and the list of attack paths pre-computed by the ADSS-Full and stored in the AKB.

In the second case the user does not need to manually edit the Petri Net based attack model but it can generate it from the attack paths provided by the ADSS-Full by using the PN-generation feature of the ADSS-Light. As showed in Figure 18, users can select the specific “ApplicationParts” in the combo menu, which are introduced in more detail in ADSS-Full part. After this selection, some attack paths in AKB can be selected to generate the PN based attack model.

Hence, the SPA component included in the ADSS-Light can be executed for protection assessment with all data inputs from ADSS and ACTC, by calling the Protection Fitness function.

6.3 ADSS-Light: Eclipse Plug-ins

The following sections document the Plug-ins of the ADSS-Light utilized at the implementation level. As introduced in these previous deliverables (D4.02, D4.03 and D4.04), the SPA component of the ADSS-Light tool is an assessment tool consisting of multiple building blocks distributed in different plug-ins.

There are four main Eclipse Plug-ins in ADSS-Light, whose dependencies are shown in Figure 19.

- “org.uel.aspire.spa” : this Plug-in provides the main GUI of the ADSS-Light. This Plug-in controls all steps in the workflow (Figure 17).
- “org.uel.aspire.wp4.assessment” : this is the core Plug-in including all data structures and algorithms for protection assessments, like the extended PN editor, Protection Fitness function and PN simulator. With respect to the workflow (Figure 17), this Plug-in control the phases: “Metrics-Attack Step Association”, “Metrics/Annotation Loading”, “Annotation-Attack Step Association”, “Fitness Function”, and “PN Simulation”.

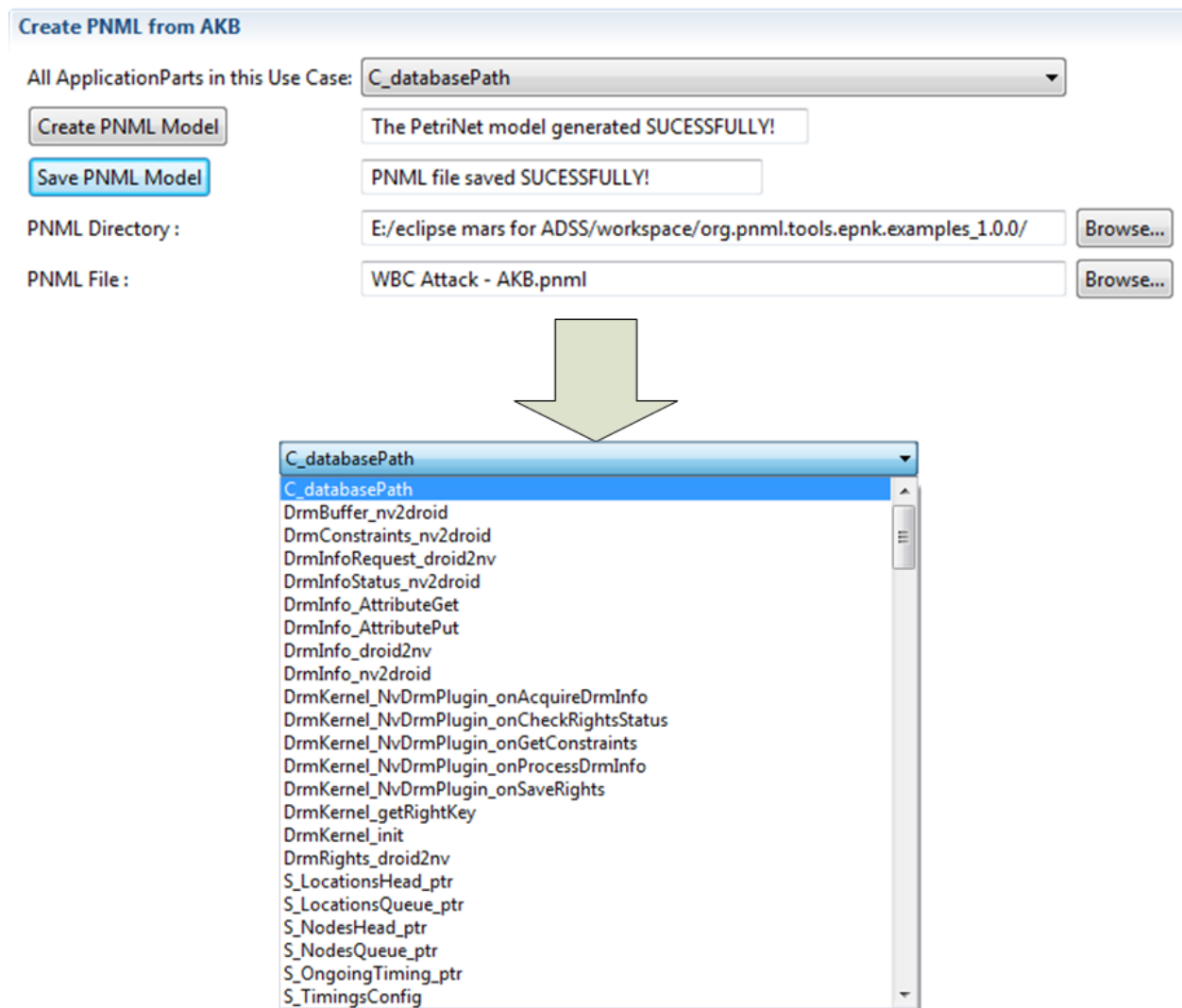


Figure 18: PN Generation

- "org.uel.aspire.wp5.assessment" : this is the Plug-in which focuses on the integration between ADSS-Light and ADSS-Full, such as data conversion, SPA API and some supplementary data structures.
- "eu.aspire.fp7.adss.light.akb.ui" : this Plug-in contains classes to access the AKB file and other helper functions for data conversion.

As introduced in D4.02, ePNK [1] is a Eclipse Plug-in for Petri Net modelling. The extended PN editor of our tool is built on it. Hence, in the view of dependencies, all four Plug-ins in our tool depend on ePNK.

Shortly, these four Eclipse Plug-ins work together to execute protection assessment independently, and can also work with ADSS-Full for finding out the golden combination of protections. The following subsections describe how to install and configure the ADSS-Light and the API of the SPA component that can be utilized by the ADSS-Full.

6.3.1 Installation and Configuration

This section describes how to install and configure the ADSS-Light tool. Due to the cooperation with ADSS-Full, the installation of ADSS-Light relies on the installation of ADSS-Full. Before the

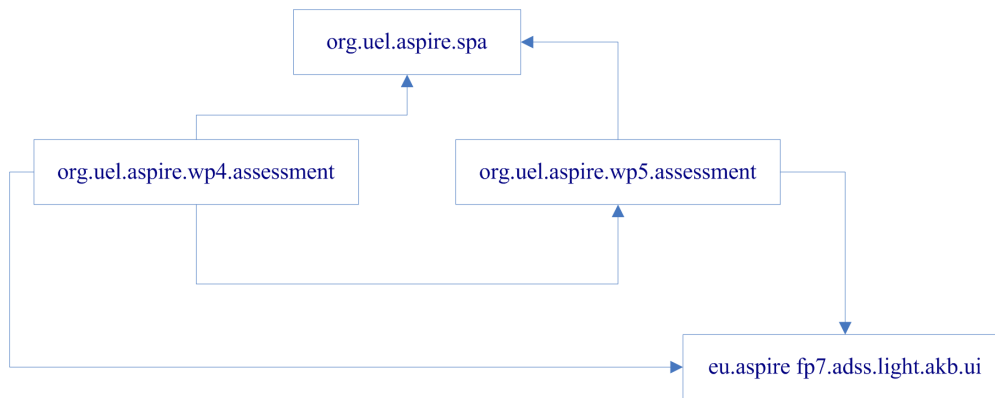


Figure 19: ADSS-Light Plug-ins

installation of ADSS-Light, the installation process of ADSS-full need to be processed as introduced in D5.11, Section 15.2.2. For example, a working instantiation Eclipse for RCP and RAP Developers installation is needed. The version of Eclipse for which the instruction are given is Mars.2 (4.5.2). Newer versions of Eclipse should be compatible. Besides, some other Plug-ins needed to be installed inside Eclipse (Install New Software in the Help menu). More details can be found in D5.11, Section 15.2.

- from <http://download.eclipse.org/releases/mars> install EMF - Eclipse Modeling Framework Xcore SDK;
- from <http://download.eclipse.org/tools/cdt/releases/8.8.1> install C/C++ Development Tools;
- from <http://www2.compute.dtu.dk/~ekki/projects/ePNK/indigo/update/> install:
 1. ePNK Core;
 2. ePNK Basic Extensions;
 3. ePNK HLPNGs;
- from <http://download.eclipse.org/nattable/releases/1.4.0/repository/> install:
 1. NatTable Core;
 2. NatTable GlazedLists Extension Feature.
- from <http://download.eclipse.org/modeling/tmf/xtext/updates/composite/releases/> install:
 1. Xtext Complete SDK;
 2. Xtext Redistributable.

After this previous installations, ADSS-Light can be installed as a standard Eclipse Plug-in.

6.3.2 Creating an ADSS-Light project

In this section, we introduce how to create an ADSS-Light project.

A new ADSS-Light project can be created using the Eclipse *New Project* wizard (from the *File* menu, *New, Project...*). As indicated in Figure 20, users can specify the name of this project.

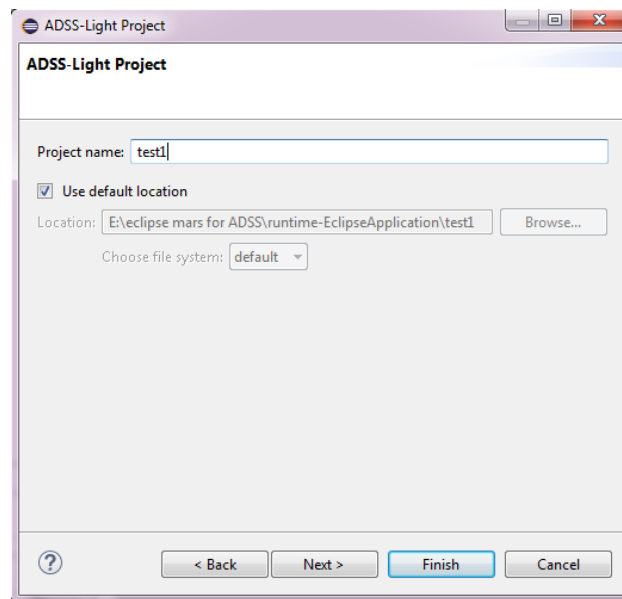


Figure 20: ADSS-Light Project Creation

After the Eclipse wizard, a ADSS-Light project is created. In this project, the most important content is an empty PNML file. As introduced in previous D4.02, D4.03 and D4.04, this PNML file is the Petri Net attack model which will be used in the following protection assessment. The protection assessments can be run on this local PNML file or on the attack paths provided as input by the ADSS-Full.

6.3.3 API of the Software Protection Assessment module

This section describes the Java API that can be invoked by the ADSS-Full to assess the strength of one or more protection instantiations; such assessment can be used to compare protection instantiations while searching for the golden combination.

This API has been designed and implemented to be used by the ADSS-Full and it is located in the package `org.uel.aspire.wp5.assessment.APIs.SPA`;

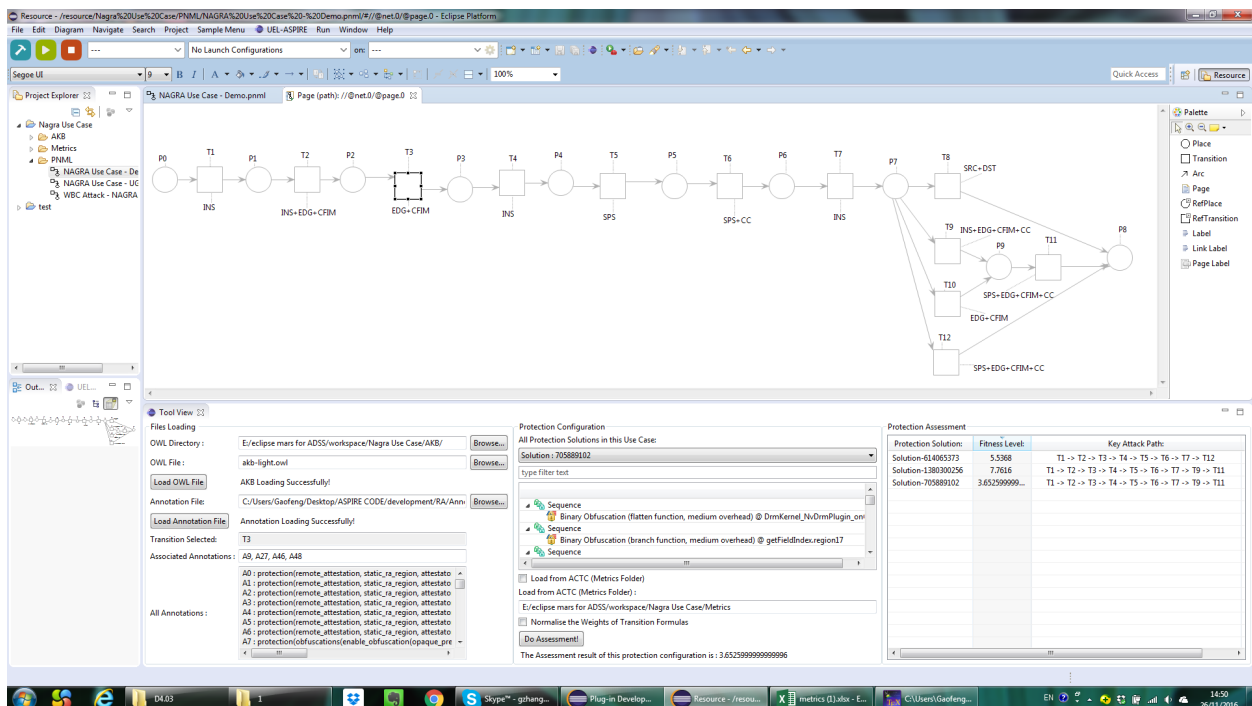
- **public boolean initModel(ArrayList(AttackPath) aps, boolean metrics)** – *this method sets a specific attack model which will be calculated for protection assessment;*
- **public boolean initCodeBase(String codebasePath, String timestamp)** – *this method initializes the datacodebase for the toolchain metrics, and the timestamp which represents version of the code;*
- **public boolean setMetricFile(String filefullpath)** – *this method tells ADSS-Light that the metrics data will come from the existing files not the runtime ACTC;*
- **public boolean initMeasureList(ArrayList(String) measurelist)** – *this method initializes the metrics names;*
- **public boolean initMetrics(ArrayList(Metric) metrics)** – *this method initializes the metrics data;*
- **public boolean initProtectionSolution(Solution ps)** – *this method initializes the protection solution;*

- **public boolean checkModel()** – this method checks the data completeness: if the return value is "true", customers can use the fitness function to obtain assessment results; if the return value is "false", customers need to use the simulator to obtain assessment results;
- **public void initPNSimulator(long timeout, double difference)** – this method initiates the simulation. Parameters: the timeout and the difference are used for terminate the simulation;
- **public double runPNSimulator()** – this method runs simulation, and returns the result of the simulation;
- **public double runProtectionFitness()** – this method runs the fitness function, and returns the result of fitness function;
- **public boolean enableAttackModel()** – this method checks attack model being set or not;
- **public boolean enableProtectionSolution()** – this method checks protection solution being set or not;
- **public boolean enableCodeBase()** – this method checks codebase being set or not.

Shortly, this API provides a direct way to call compute the Protection Fitness without using the ADSS-Light's GUI; the ADSS-Full can then invoke this API repeatedly when searching for the golden combination.

6.3.4 ADSS-Light User Interface

In Figure 21, a screen-shot of the running ADSS-Light is shown. In the left-bottom part of the "ToolView", we can find out one Text editable area: "Associated Annotations". On the basis of loaded Annotations json file in the use case, users can edit the associated annotations for each transition in PN attack models.



The screenshot displays the ADSS-Light tool interface. The main window shows a state transition diagram with nodes P0 through P8 and transitions labeled with attack models like INS, SPS, and EDG+CFM. The left sidebar contains a Project Explorer with a tree view of the project structure. The bottom section is divided into several panels:

- Files Loading:** Shows the OWL Directory and OWL File (akb-light.owl).
- Annotation File:** Shows the path to the annotation file (C:/Users/Gaofeng/Desktop/ASPIRE_CODE/development/RA/Ann/).
- Transition Selected:** Shows the selected transition (T3).
- Associated Annotations:** A list of annotations associated with the selected transition, including A0 through A7.
- All Annotations:** A list of all annotations in the system.
- Protection Configuration:** Shows the current protection solution (705889102) and a list of protection solutions with their fitness levels.
- Protection Assessment:** A table showing the results of the protection assessment, including the fitness level and the key attack path.

Protection Solution	Fitness Level	Key Attack Path
Solution-614065373	5.3368	T1 -> T2 -> T3 -> T4 -> T5 -> T6 -> T7 -> T12
Solution-1360390256	7.7616	T1 -> T2 -> T3 -> T4 -> T5 -> T6 -> T7 -> T9 -> T11
Solution-705889102	3.652599999...	T1 -> T2 -> T3 -> T4 -> T5 -> T6 -> T7 -> T9 -> T11

Figure 21: Screen-shot of the ADSS-Light tool

For example, in this screen-shot, a user picked transition "T3" in the graphic view of PN attack models. Then, user can edit associated annotations in this Text editable area: "Associated Annotations", with a comma-separated list, such as "A0,A2,A20". All potential annotations in this use case are listed in the text area "ALL Annotations" which are fetched from the previous loaded *annotations.json* file.

Just above the annotation part in the "ToolView", the ADSS-Light can load the AKB file; it contains the protection instantiations to be assessed for the ADSS-Full.

Hence, in the central part of the "ToolView", users can check and select one protection instantiation to be assessed. When this is selected, the user also can browse to find the metrics files' locations. If AKB and ADSS-Full already run ACTC to obtain metrics data and store them into AKB, users do need to tick the button "Load from ACTC (metrics Folder)", and obtain these data from AKB directly. Otherwise, this button needs to be ticked to let ADSS-Light load metrics data from ACTC.

In the central-bottom part of the "ToolView", the Normalized Protection Fitness formulas can be chosen instead of the default ones. In the right part of the "ToolView", there is an ordered list for assessed protection instantiations, ranked by the protection fitness value: the minimum paths for each solutions are also indicated as a sequence of attack steps.

Besides, our tool has been utilised in NAGRA use-case. A PN based attack model has been built, and related metrics/annotations associations have been configured. Then, the PF and NPF functions have been used to assess and rank some protection instantiations: the details can be found in D4.06 Confidential Annex I, Section 2.

6.4 Conclusions and Future Extensions

The integration of ADSS-Light and ADSS-Full, by UEL and POLITO, required time and effort due to data structure consistency and Eclipse Plug-in dependencies. The data structures initially used in the ADSS-Light have been adapted or extended to be aligned to the ADSS-Full data structures, in order to implement the SPA API to support the software protection assessment for the ADSS-Full. Both ADSS-Light (including SPA) and ADSS-Full have been build as Eclipse Plug-in, and both of them require other Plug-ins, such as ePNK (SPA), and Ontology (ADSS-Full). The dependency among these plug-ins, along with the different Eclipse versions supported by them, has caused some additional efforts in integrating ADSS-Light and ADSS-Full. Possible future extensions of the ADSS-Light are listed below:

1. The Petri Net editor can store attack models locally and the user can decide to store them into the AKB; this is currently limited to a single user, as we did not have time to investigate the management of conflicts in a scenario where different users can work in parallel to insert and edit new attack models into the AKB. Moreover, a possible improvement could be replacing the currently developed ad-hoc translator from PNML to OWL into a standard-compliant translator between these two standard XML-based languages.
2. We plan to look into new methods for determining the weights in the metrics formulas, which are currently edited by the security expert (or all set to 1 by default). To provide a better guess in defining the values of such weights, we are considering adapting AHP (Analytic Hierarchy Process), a well-known methodology in risk management for organizing and analysing complex decisions. Users of the AHP first decompose their decision problem into a hierarchy of more easily comprehended sub-problems, each of which can be analysed independently: then it can be used to combine experts' opinions together. In the ADSS-Light users need to edit the PN attack models, and then associate metrics to attack steps, and associate code regions to attack steps: it is likely that different experts have different opinions on these settings. For example, with AHP, users can use their expert opinion to

rank such associations; then AHP combines such rankings together in order to determine a relative ranking of alternatives with numerical weights that can be used. We plan to use such ranking to derive the weights in the metric formulas as a result of different experts' opinions.

3. In the NAGRA use-case example, the ADSS-Light utilizes only static metrics as NAGRA did not provide the corresponding unit-tests (necessary to run the code) to obtain the dynamic metrics. The ADSS-Light allows to load and then associate dynamic metrics values, and we plan to extend the experiment on other use cases where dynamic metrics could be computed.

Part III

Experiments

7 Client/server code splitting experiment

Section authors:

Mariano Ceccato, Paolo Tonella (FBK)

To make this deliverable self-contained, some content of this section was copied and slightly adapted from Section 6 of Deliverable D4.04. The copied sentences are in blue.

7.1 Experimental Definition

The content of this section is partially copied from the Section 6.1 of Deliverable D4.04

Goal	Analyze the ability of code splitting to prevent malicious attacks and measure the performance penalty to be paid
Treatments	T0 = original code; T1 = small/medium code portion split
RQ1	How effective is code splitting for preventing code tampering as compared to the clear code?
RQ2	How effective is code splitting for increasing the attack time as compared to the attack time for the clear code?
RQ3	What strategies and tools are used to complete successful attacks?
Subjects	Students from FBK and POLITO
Objects	P1: space game
Tasks	Make spacecraft move faster by doubling the effect of a move
Metrics	Success rate; time to mount a successful attack
Design	Lab1: P1-T0; P1-T1

Table 2: Code splitting experiment

Table 2 provides a schematic overview of the code splitting experiment. The *goal* of this experiment is to analyse the degree of protection offered by the ASPIRE code splitting technique. Splitting is expected to bring increased protection but also to cause increasing performance degradation. Large portions of code executed on the server may involve a significant server execution overhead, especially when serving many clients at the same time. Moreover, splitting a complex portion of code may require frequent synchronisations between the code remaining on the client and the code moved to the trusted server. Because of the implementation of this protection, the more dependencies between client and server, the more synchronizations are required (see Deliverable D3.1 for technical details about this protection). In order to evaluate the impact of the split code size on both the level of protection and the performance penalty to be paid, we consider two treatments: the baseline T0, which is the original code, and T1, associated with a *small* or *medium* code portion being moved from the client to the trusted server. This experiment was conducted at FBK and at POLITO.

7.2 Research questions

The content of this section is partially copied from the Section 6.2 of Deliverable D4.04

The experiment aims at answering the following three research questions:

- **RQ1:** How effective is code splitting for preventing code tampering as compared to the clear code?
- **RQ2:** How effective is code splitting for increasing the attack time as compared to the attack time for the clear code?
- **RQ3:** What strategies and tools are used to complete successful attacks?

The first research question is about the effectiveness of code splitting as a code protection technique that limits the likelihood for attackers to be able to complete a successful attack. The second question is about the cost for porting a successful attack in terms of human time. The evidence collected during the experiment will be used to assess the average increased attack time provided by code splitting. We will measure also the performance degradation associated with code splitting, so as to be able to pair the increased attack effort with the increased cost of protection (performance penalty). The last research question is intended to investigate, more qualitatively, how attackers work to complete their task.

7.3 Object

The content of this section is partially copied from the Section 6.3 of Deliverable D4.04

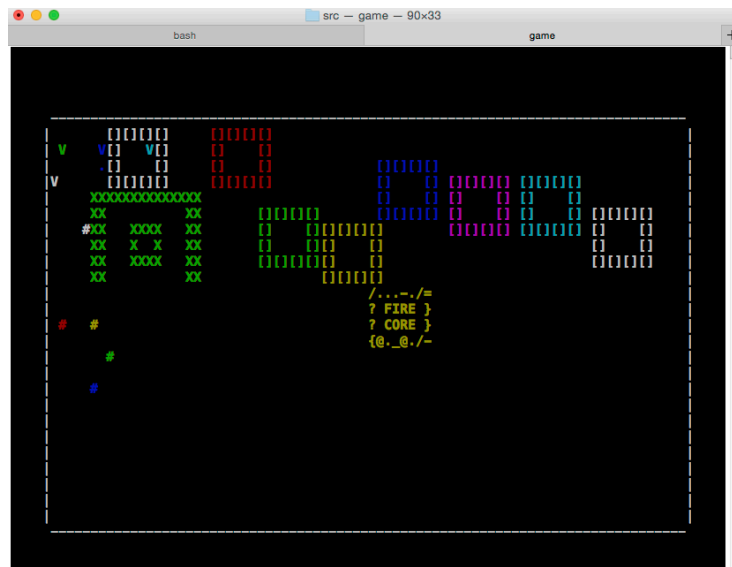


Figure 22: Screenshot of SpaceGame

The object of this experiment is an open source C program, SpaceGame, obtained from SourceForge. SpaceGame is a demonstrator for the framework GAME (Geometrical Ascii Multigame Environment), a C language framework for creating geometrical games using `ncurses` text screens. While GAME was originally created for Unix platforms, it can be ported to more systems, because it uses standard ANSI C. The framework and the demonstrator amount to 1,873 SLoC (measured by `sloccount`), including header files. Players can move their spacecraft on the screen by means of the numeric keyboard or by pressing the keys for characters 'h', 'j', 'k', 'l' ('s' stops the game, while 'q' quits it). Figure 22 shows a screenshot of SpaceGame.

The attack task to be executed by the experiment subjects on SpaceGame aims at gaining an unfair advantage over other competing players:

Attack task: Modify the source code of SpaceGame so as to move twice as fast as allowed by the game rules. Specifically, each key press must translate into a 2-character length move, instead of a 1-character move.

While in the clear code the required modification consists just of changing the unitary increment or decrement of the position into a double increment/decrement, when the code handling player movements is moved from client to server, the modification to be done becomes increasingly more difficult, depending on the split code size. It might for instance involve a double function call that replaces a single call, or a modification of some client-server messages. Maybe, an attacker could also observe the input/output data exchanged with the server, *learn* the behaviour of the split part and reimplement locally the missing features in a fake-server, that could be later tampered with to complete the attack.

7.4 Analysis of Runtime Overhead

The content of this section is partially copied from the Section 6.4 of Deliverable D4.04

The execution time (ET) measures the time for a complete execution of SpaceGame under a pre-defined interaction scenario. In order to obtain meaningful and comparable execution times, a program driver is used to stimulate the program without requiring any user intervention. The driver executes the program in batch mode and it sends a predefined key sequence to SpaceGame, so as to simulate the interaction of the user with the game. The length of the chosen key sequence was 120 (which is the maximum value for the execution in batch mode), which means that an execution of the driver simulates the user pressing the game keys 120 times. The key-press rate is not relevant for a batch execution, because key-press events are stored in a file and the game consumes exactly one key-press event for each step of the game. To accommodate for random fluctuations in the execution time measurements, ET is measured multiple (100) times.

The performance overhead (PO) is the relative increase of the average execution time between split code and original code:

$$PO = \frac{\overline{ET(P')} - \overline{ET(P)}}{\overline{ET(P)}} \quad (11)$$

where P , P' indicate the original and protected program, respectively.

Version	Average time [sec]	SD (σ)	PO
Original	0.020	0	-
Split-Medium	2.106	0.048	104

Table 3: Execution times for split code.

Table 3 reports the average execution times (expressed in seconds), the standard deviation and the performance overhead for two different versions of SpaceGame (original code and medium split). We can observe that the performance overhead for the split version is significant. Protected code takes 104 times longer than the original code. This is mainly due to the interaction between client and trusted server, because applying this protection to SpaceGame means to turn a client-only program into a client-server architecture. Anyway, this overhead was measured in a batch execution, and the performance overhead does not impact negatively the user experience.

7.5 Metrics

The content of this section is partially copied from the Section 6.5 of Deliverable D4.04

The metrics collected to answer research questions RQ1 and RQ2 are:

AT: Attack time

SR: Success rate

Subjects are asked to mark down the start and end time when starting and after finishing the attack task, so one key metrics collected during the experiment is the attack time (AT). Subjects are also asked to send the attacked code to the experimenters, who manually verify if the attack was successful or not. Metrics AT is meaningful only for successful attacks. The proportion of successful attacks provides a second metrics, which complements AT, called success rate (SR). SR measures the proportion of subjects that successfully completed the attack task either on the original code or on code protected by code splitting.

Based on the metrics chosen to quantify the effectiveness of the code splitting protection, we can formulate null and alternative hypotheses associated with research questions RQ1, RQ2:

- $H_{0_{SR}}$: There is no difference in the average SR between participants working on code protected with code splitting and participants working on clear code;
- $H_{0_{AT}}$: There is no difference in the average AT between participants working on code protected with code splitting and participants working on clear code.

In addition to the metrics AT, SR, we ask subjects to answer a pre-questionnaire and a post-questionnaire. The pre-questionnaire collects information about the abilities and experience of the involved subjects. This is very important to analyse the effect of ability and experience in the successful completion of the attack tasks either on original or on split code. The post-questionnaire collects information about clarity and difficulty of the task, availability of sufficient time to complete it, and on the tools used and the activities carried out to complete the task.

7.6 Design

The content of this section is partially copied from the Section 6.6 of Deliverable D4.04

Lab	P1-T0	P1-T1
Lab1	G1	G2

Table 4: Design for the code splitting experiment: group G1 is assigned object P1 in its original form, while group G2 is assigned P1 protected with code splitting T1, in a single lab (Lab1)

Table 4 shows the design of the single experimental session (Lab1). Subjects are divided into two groups, G1 and G2. Object P1 (SpaceGame) is provided as clear code (T0) or split code code (T1).

7.7 Statistical analysis

The content of this section is partially copied from the Section 5.6 of Deliverable D4.04

The difference between the output variable (SR and AT) obtained under different treatments (clear code vs. protected code) is tested using non-parametric statistical tests, assuming significance at a 95% confidence level ($\alpha=0.05$). So, we reject the null-hypotheses when $p\text{-value}<0.05$. All the data processing is performed using the R statistical package [24].

To analyse whether data obfuscation reduces the success rate of attack tasks, we used a test on categorical data, because the tasks can be either correct (completed successfully) or incorrect (completed unsuccessfully). In particular, we used Fisher's exact test [11], which is more accurate than the χ^2 test for small sample sizes, another possible alternative to test the presence of differences in categorical data. The same analysis was used in previous empirical works [26].

To be conservative (because of the small sample size and non-normality of the data), a non-parametric test has been used to test the hypotheses related to differences in the attack time. In particular, we perform the one-tailed Mann-Whitney U test on all samples [27]. Such a test allows

us to check whether differences exhibited by participants under different treatments (clear and protected code) over the two labs are significant.

While these tests allow for checking the presence of significant differences, they do not provide any information about the magnitude of such a difference. This is particularly relevant in our study, since we are interested in investigating to what extent the use of obfuscation reduces the likelihood of completing an attack and increases the time needed for an attack. As such, two kinds of effect size measures have been used, the *odds ratio* for the categorical variable SR and the Cliff's delta effect size [14] for the AT variable. The effect size is computed using the `effsize` package [30].

The odds ratio is a measure of effect size that can be used for dichotomous categorical data. An odds indicates how likely it is that an event will occur as opposed to it not occurring [27]. The odds ratio is defined as the ratio of the odds of an event occurring in one group (e.g., experimental group) to the odds of it occurring in another group (e.g., control group), or to a sample-based estimate of that ratio. If the probabilities of the event in each of the groups are indicated as p (experimental group) and q (control group), then the odds ratio is defined as:

$$OR = \frac{p/(1-p)}{q/(1-q)}$$

An odds ratio of 1 indicates that the condition or event under study is equally likely in both groups. An odds ratio greater than 1 indicates that the condition or event is more likely in the first group. An odds ratio less than 1 indicates that the condition or event is less likely in the first group.

For independent samples, Cliff's delta provides an indication of the extent to which two (ordered) data sets overlap, i.e., it is based on the same principles of the Mann-Whitney test. Cliff's Delta ranges in the interval $[-1, 1]$. It is equal to $+1$ when all values of one group are higher than the values of the other group and -1 when the opposite is true. Two overlapping distributions would have a Cliff's Delta equal to zero. An effect size d is considered small when $0.148 \leq d < 0.33$, medium when $0.33 \leq d < 0.474$ and large when $d \geq 0.474$ [6].

The analysis of co-factors is performed using a General Linear Model (GLM). It consists of fitting a linear model of the *dependent* output variables (success rate or attack time) as a function of the *independent* input variables (all factors, including the treatment, i.e., the presence of obfuscation). A general linear model allows to test the statistical significance of the influence of all factors on the attack success rate and attack time. In case of relevant factors, interpretations are formulated by visualising the associated interaction plots.

The analysis is conducted on the main co-factors (shown in Table 2) and reports the individual coefficients and their significance level. The coefficients are used to understand magnitude and direction of the effects.

7.8 Participants Characterization

Figure 23 shows some statistics about the participants involved in the replication conducted at University of Trento. These data were collected by means of the pre-experiment questionnaire. Participants have a quite homogeneous background, because they are 17 master students and 2 PhD students. Half of them have no experience as professional programmers, while the other half have some experience either as part-time or as full-time professional programmer. Most of them have at least 1 year of programming experience in C. Some of them have limited experience in C and in using IDE to program in C, while others have more than three years of experience in C. Moreover, most of the participants can use a C debugger to set break-points, do stepwise execution, inspect the call stack and inspect program variables.

Participants involved in the replication at Politecnico di Torino are 86 Master students. Partici-

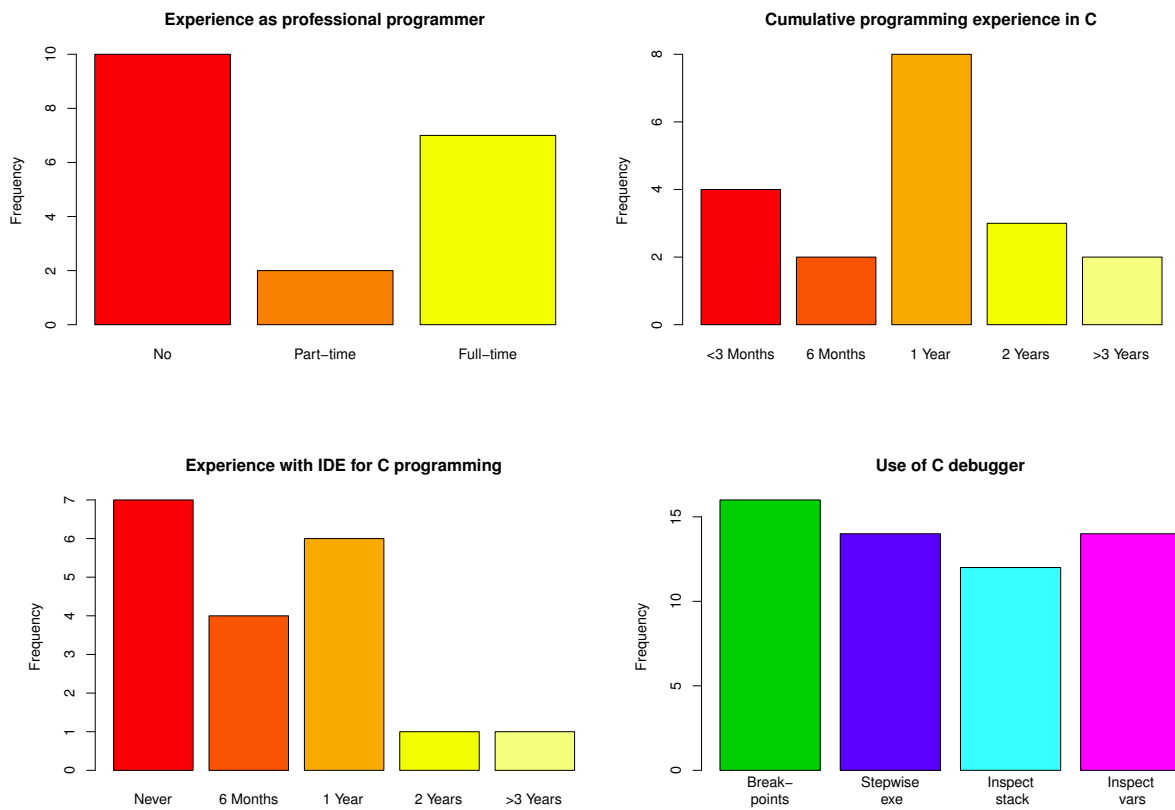


Figure 23: Demographics of participants in Trento.

pants have a homogeneous background, because they are all Master students at the Politecnico di Torino. Some of them have some experience either as part-time or as full-time professional programmers. All the participants have at least 1 year of experience, however, the great majority of them has at least 2 years of programming experience in C. All the participants were able to use a C debugger to set break-points, do stepwise execution, inspect the call stack and inspect program variables. Only two students preferred to compile and debug with a command line approach (gcc + gdb), all the other ones had good experience with at least one of the supported IDE (Visual Studio, CodeBlocks, XCode, Eclipse).

7.9 Analysis of Success Rate

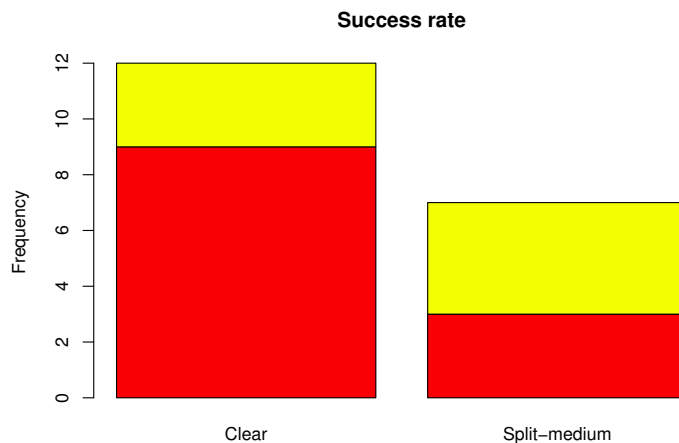


Figure 24: Bar plot of attack success rate in Trento (red=successful attack, yellow=wrong attack).

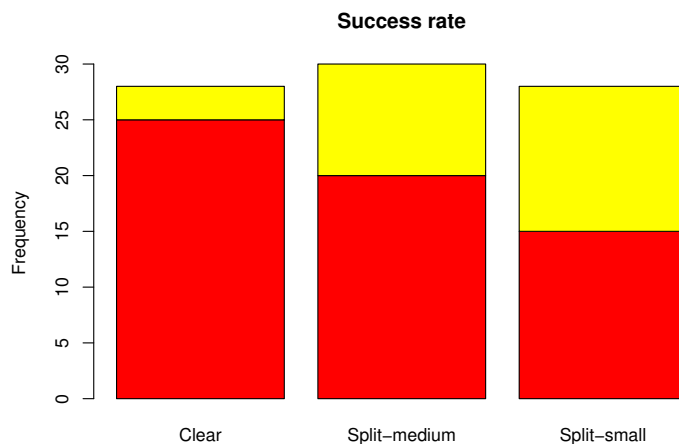


Figure 25: Bar plot of attack success rate in Torino (red=successful attack, yellow=wrong attack).

Table 5 shows the distribution of correct and wrong attacks in Trento, performed on the clear code and on the code protected with code splitting, as well as the success rate SR. Figure 24 shows the bar plots of these data. From the table and the plot, we can observe many successful attacks (red region) on clear code and quite few on protected code. Correspondingly, there are more wrong attacks on protected code than on clear code.

	Clear	Split-medium
Correct	9	3
Wrong	3	4
SR	0.75	0.42

Table 5: Success Rate in Trento.

	Clear	Split-medium	Split-small
Correct	25	20	15
Wrong	3	10	13
SR	0.93	0.67	0.54

Table 6: Success Rate in Torino.

We computed the effect size (as the odds ratio) and the Fisher’s test, to check if the trend observed on the graph is statistical significant. The odds ratio is 3.69 and the Fisher’s test reports a p -value = 0.33. Even if a trend seems to be quite clear in the graph, the p -value is not < 0.05 , so we can not reject the null-hypothesis on success rate and we can not claim statistical significance, probably because of the small sample size.

Table 6 shows the distribution of correct and wrong attacks in Torino, in this case we have the clear code, and the code protected with two different configurations of splitting, they are *small* and *medium*. Figure 25 shows the corresponding bar plots. In this case, we can observe a decrease in the number of successful attacks on protected code, the decrease seems more marked when the code is protected with the *small* variant of splitting.

The odds ratio for the *split-medium* configuration is 0.24, while for the *split-small* configuration the odds ratio is 0.14. The fisher test reports a p -value = 0.01, so we the trend observed in the graph is statistically significant and we can reject the null-hypothesis on success rate. We can formulate the following alternative hypothesis: *the adoption of client/server code splitting reduces the likelihood of a successful attack. The likelihood of a successful attack is 0.24 or 0.13 depending on the adopted splitting configuration.*

7.10 Co-factors of Success Rate

	Estimate	Std. Error	t value	Pr(> t)
(Intercept)	0.2396	0.3476	0.69	0.5028
Treatment	-0.2539	0.2436	-1.04	0.3163
Pre1 (Programmer experience)	0.0738	0.1584	0.47	0.6489
Pre2 (C experience)	0.1783	0.1097	1.63	0.1281
Pre3 (IDE experience)	-0.0923	0.1379	-0.67	0.5147
Position (MsC Vs PhD)	0.4448	0.3731	1.19	0.2545

Table 7: General linear model of Success Rate.

Then, we report the analysis of co-factors that could have influenced the success rate. Table 7 reports the analysis of co-factors obtained by applying the general linear model method. P-values are reported in the last column; any significant coefficient would be reported in boldface. Similarly to what reported previously for the main factor (presence of protection), co-factors have no statistically significant influence on attack success rate.

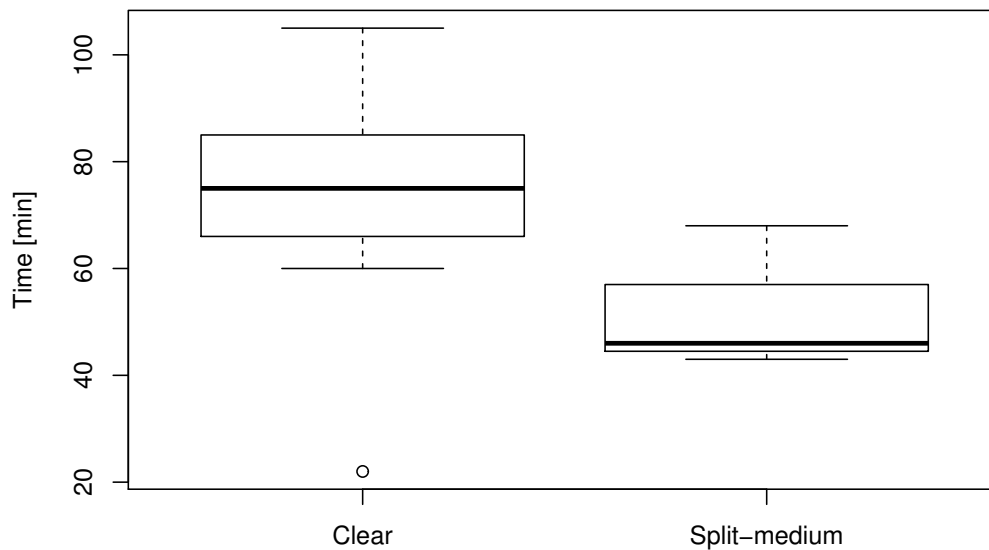


Figure 26: Boxplot of attack time in Trento (only for successful attacks)

Treatment	N	Mean	Median	SD
Clear	9	74.33	75	24.42
Split-medium	3	52.33	46	13.65

Table 8: Descriptive statistics of Attack Time in Trento.

7.11 Analysis of Attack Time

Now we analyse the time required to elaborate and complete a successful attack. Here we only consider *successful* attack tasks and we drop tasks that were not successfully completed. Thus, we have less data points. Table 8 reports the descriptive statistics of Attack Time (number of data points, mean, median and standard deviation) either on clear code and on code protected with code splitting for the replication in Trento. Figure 26 shows the boxplot of the attack time required in Trento to deliver correct results when the attacked code is clear or protected. Apparently, less time is taken to deliver a correct attack on protected code, than on protected code. This result is quite counter-intuitive, because protected code should require *at least* as much time as clear code. However, to draw conclusions we have to consider the result of the statistical test. We applied Mann-Whitney U test, and we obtained p -value = 0.1384, which does not allow us to reject the null hypothesis. Thus, the difference observed on the graph is not statistically relevant, even if the Cliff's delta reports a large effect size ($d=1.11$).

Then we the time for successful attack tasks in the replication in Torino. Table 9 reports the descriptive statistics of Attack Time on clear code and on code protected with code two configurations of splitting. Figure 27 shows the boxplot of the attack time taken in Torino to deliver correct results when participants attacked clear code or code protected with one of the two splitting configurations.

Apparently, less time is taken to deliver a correct attack on code protected with the *split-medium* configuration, than on protected code. However, more time is required to attack the code that is protected with the *split-small* configuration. While the second observation is intuitive, the first ob-

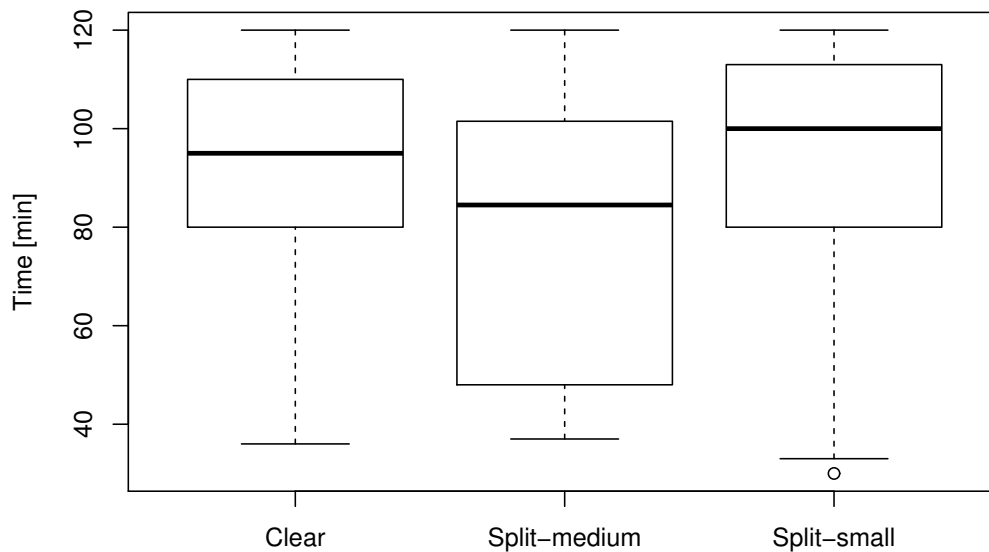


Figure 27: Boxplot of attack time in Torino (only for successful attacks)

Treatment	N	Mean	Median	SD
Clear	25	90.56	95.0	25.69
Split-medium	20	80.65	84.5	28.33
Split-small	15	92.13	100.0	29.38

Table 9: Descriptive statistics of Attack Time in Torino.

servation is counter-intuitive, similarly to the first replication in Trento, so we look at the statistical test to interpret these data.

Since in this case we have three treatments (clear and two configurations of splitting), we can not apply the Mann-Whitney U test, as we did for the former replication. Instead, we apply general linear model. We obtain a p -value of 0.24 and 0.86, respectively for *split-medium* and *split-medium*. So, the difference observed on the graph is not statistically relevant, and we can not reject the null hypothesis.

7.12 Co-factors of Attack Time

We report the analysis of co-factors that could have influenced the Attack Time. Table 10 shows the result of the of general linear model of Attack Time. P -values are reported in the last column, but no case is statistically significant (no case has p -value < 0.05). Thus, no co-factor have statistical significant influence on Attack Time.

7.13 Analysis of post-questionnaire

Figure 28 shows the distribution of questions to the feedback post-questionnaire. For many participants the task was clear (*Post1*) and the time to complete the task was enough (*Post2*) and the task was easy to perform (*Post3*).

	Estimate	Std. Error	t value	Pr(> t)
(Intercept)	54.4481	28.3031	1.92	0.1027
Treatment	-33.9888	20.3364	-1.67	0.1457
Pre1 (Programmer experience)	-4.7205	9.3027	-0.51	0.6300
Pre2 (C experience)	9.8126	7.7507	1.27	0.2524
Pre3 (IDE experience)	-1.3439	8.4309	-0.16	0.8786
Position (MsC Vs PhD)	24.4414	22.2559	1.10	0.3142

Table 10: General linear model of Attack Time.

To work on the attack task, participants mostly resorted to the *editor*, to the *compiler* and to *internet search* (*Post4*). Eventually, most of the time was devoted to *reading and understanding* the code, and less time was devoted to *changing and executing* the code (*Post5*).

Questions *Post6*, *Post7* and *Post8* are open questions, meant to let the participants free to formulate their answers and describe their attack.

The sequence of activities (*Post6*) performed by the participants who solved the task on clear code was quite simple. They adopted an iterative approach, consisting of making some assumptions on the code and changing the code to verify them (participants with IDs 107, 109, 114 and 19). This process possibly started from elements of the graphical user interface (participants 108, 109 and 116). The attack strategy for the participants who were successful on the protected code was more complex, because not all the code was available at client side. The successful strategy consisted of starting code understanding from those parts related to the interaction with the remote server (participants 101 and 106). Those participants who did not complete the task either did not adopt an iterative approach (participants 110 and 112) or, when an iterative approach was used, it was not starting at crucial points, such as user interface and network communication (participants 103, 111).

For the participants who successfully attacked clear code, the most difficult task (*Post7*) was understanding the code to attack (participants 107, 108, 116, 119). Only in one case the most difficult task was the formulation/validation of assumptions (participant 114). On the other hand, for the participants who successfully attacked the protected code, the hardest task was figuring out why the client was crashing after some changes (participant 101) and understanding what operations were missing from the client and had been moved to the server side (participant 106). Participants who did not complete the task reported either understanding the business logics of the code to attack (participant 103) or locating the features of interest (participant 115) as the hardest tasks.

Considering where most of time was spent (*Post8*), most of the participants agree that the most time consuming task was reading and understanding the code. Only one of the participants, who successfully attacked protected code, reported that the most time consuming task was understanding the server logics.

7.14 Threats to validity

The content of this section is partially copied from the Section 6.12 of Deliverable D4.04

The main threats to the validity of this experiment belong to the conclusion, internal, construct and external validity threat categories.

Conclusion validity threats concern the relationship between treatment and outcome. We could not use statistical significance to draw our conclusions. Inability to reject the null hypothesis exposes us to type II errors (incorrectly accepting a false null hypothesis). This was expected and already acknowledged in the beginning of this section: a relatively limited number of participants (they were 19) have been involved in this replication. Moreover, among them, only a fraction (only 12 participants) had the right knowledge and skill to complete the attack task. We will mitigate this

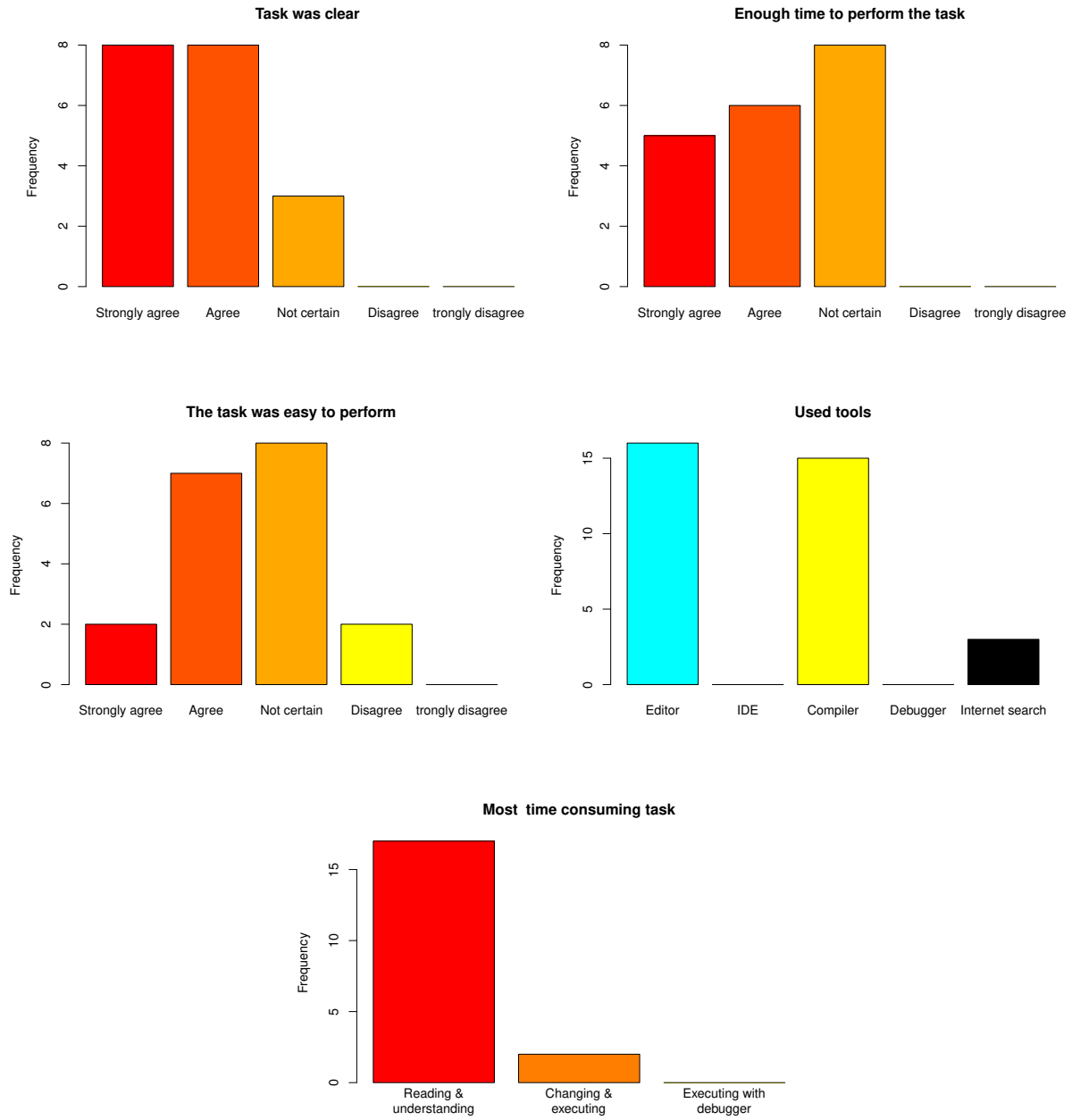


Figure 28: Post-questions answered by Trento’s subjects.

threat by replicating this experiment, so as to increase the number of participants. In fact, the probability of a type II error can be reduced by increasing the sample size. Subjects were provided with the source code, not the binary code, because students are not proficient enough in binary and assembly code analysis at FBK.

Internal validity threats concern external factors that may affect the independent variable. Subjects were not aware of the experimental hypotheses. Subjects were not rewarded for the participation in the experiment and they were not evaluated on their performance in doing the experiment.

Construct validity threats concern the relationship between theory and observation. They are mainly due to how we measure the effectiveness of code splitting and the related performance overhead. We manually assessed the successful completion of each task in order to measure SR. During the labs, the experimenters made sure that times are accurately marked in the time sheets upon start and completion of the attack task. Performance measurements were repeated 100 times so as to remove the effect of the possible random fluctuations of the measured execution time under slightly different conditions.

External validity concerns the generalisation of the findings. In our experiment we considered one program, SpaceGame. Results may not generalise to different programs. However, the chosen program has been written by third parties, completely unrelated with ASPIRE, so as to ensure that it is not crafted to provide optimal application conditions for the ASPIRE protection. Moreover, the chosen program is publicly available as an open source project in SourceForge. Hence, it can be regarded as a representative for this category of software. Further replications of the study on additional objects will corroborate the external validity of our findings.

7.15 Lessons Learned

Based on the quantitative and qualitative data collected during the experiment, we can make some general observations and distill some important lessons. Specifically, from the objective, quantitative data, the following observations can be made:

- Usage of the code-splitting protection introduced an execution time degradation that can be estimated approximately as a factor $\times 100$. This confirms that code splitting can have a non negligible impact on the execution time of the application being protected. However, for the specific application considered in this experiment, there was no perceivable impact on the user experience, because the interaction between the user and the game was equally responsive to the user commands before and after protection. This indicates that even a slow down as high as $\times 100$ might be acceptable for an interactive application in which the intervals between user commands dominate the real time demands. If such intervals are still larger than the execution time of the slowed down application, the final user will not perceive any appreciable negative effect of the protection on the game experience.
- The quantitative data that we collected indicate a reduction of odds of a successful attack approximately by a factor $\times 2$. This confirms the effectiveness of the code splitting protection, which removes relevant sources of information for the attackers by moving them to a server, where they are not accessible. As a consequence, static and dynamic analysis of the application has no access to all code fragments that have been moved to the server and only the messages exchanged with the server can be observed.
- **In summary:** *Despite the substantial performance cost, code splitting did not affect the user experience, while at the same time halving the chances of successful attacks.*

The following observations can be derived from the qualitative data collected through the post questionnaires:

- Editor and compiler are the most used tools to attack the code. This means that attack attempts involved substantial program understanding and code tampering activities. We can conjecture that the typical attack pattern involving editor and compiler consists of: (1) using the editor to extract pieces of knowledge from the code and to verify hypothesis about the behaviour of the program by code reading; (2) using the compiler to be able to execute code changes that test the attacker's conjectures about the code behaviour or that implement the attack strategy being elaborated.
- Successful attacks adopt an iterative approach, where assumptions are first formulated, then checked by changing and running the changed code. We have collected interesting evidence about the most successful attack process adopted by attackers, pointing to an *iterative and incremental process*. This means that a monolithic process, consisting of thorough understanding of the code and its protections, followed by a thorough attack implementation, is unlikely to work. On the contrary, when attacks are successful, understanding of the program and its protections proceeds in chunks. After a portion of protected code is even just partially understood, code tampering is applied to test the knowledge acquired so far and implement the attack strategy by trial-and-error.
- Successful attacks start from analyzing code from crucial points, such as the interactions with the server and the GUI. This has an important implication about the skills of successful attackers: they must be able to recognize quickly the code portions that are critical for their task. This finding is consistent with the need of attackers to optimize the usage of the scarce attack time at their disposal, so as to quickly make decisions that eventually lead to the discovery of the minimum resistance attack path. Implementing such adaptive strategy requires advanced program understanding skills and substantial knowledge about protections and static/dynamic program analysis tools. Not all students involved in our experiment had such skills, which explains the relatively low success rate observed in the experimental sessions. Successful attackers were able to quickly identify the code fragments implementing the protections to be defeated to complete the attack task.
- When attacking the code splitting protection, the most difficult task was reconstructing what is done at server side (split code) by observing the messages exchanged and the interactions between client and server. In cryptography, this is called the *learnability* of the protection. In the case of SpaceGame, learning the server behaviour from black-box observation of the messages exchanged was difficult but not infeasible. Indeed, some successful attackers were able to do that. However, this task requires non trivial skills and competences. The fact that learning the server behaviours was deemed as the critical and most difficult attack task confirms our hypotheses on the nature and strength of the code-splitting protection: it leaves attacker with just black-box information about a critical behaviour of the application and reconstructing such behaviour from black-box observations (i.e., from observation of the exchanged messages) is an extremely challenging and difficult task.

8 Common Design of Experiments with Industrial Participants

Section authors:

Mariano Ceccato, Paolo Tonella (FBK), Bjorn De Sutter, Bart Coppens (UGENT)

To make this deliverable self-contained, the content of this section was copied and slightly adapted from the first part of Section 9 of Deliverable D4.04. The copied sentences are in blue.

Goal	Analyze the ability of ASPIRE to prevent DRM attacks
Treatments	T1 = protection configuration 1; T2 = protection configuration 2; T3 = protection configuration 3
RQ1	To what extent do ASPIRE protections delay attacks against DRM?
RQ2	What ASPIRE protections are most effective in delaying attacks against DRM?
Subjects	Hackers from the NAGRA tiger team
Objects	DemoPlayer (binary code)
Tasks	Violate specific DRM protection
Data	Report about the different reverse engineering and attack activities (e.g., data de-obfuscation; identifier renaming; control flow reconstruction; code understanding; decompilation) carried out by attackers
Design	Long running (30+ days) case study

Table 11: Nagravision case study

8.1 Experimental Definition

Tables 11, 12, 13 provide a schematic overview of the industrial case studies. The *goal* of these case studies is to evaluate the degree of protection offered by the ASPIRE techniques as a whole, considering those operating on the source code as well as those operating on the binary code. The entire ASPIRE tool chain is applied to the industrial case studies, so as to ensure maximum protection. The subjects involved in these case studies are professional pentesters employed by the industrial partners of ASPIRE.

The case study aims at answering the following research questions:

- **RQ1** To what extent do ASPIRE protections delay attacks?
- **RQ2** What ASPIRE protections are most effective in delaying attacks?

These research questions deal with the effectiveness of the ASPIRE protections, when these are applied to the industrial case studies. We want to assess the capability of the ASPIRE protections to resist a massive attack mounted by professional hackers during a long time period. Moreover, we want to assess the relative importance of different defence lines implemented by the various components in the ASPIRE tool chain, by analysing the activities carried out by the professional hackers to defeat each specific ASPIRE protection.

8.1.1 Object

The objects of this experiment are programs provided by the industrial ASPIRE partners:

DemoPlayer: Media player provided by Nagravision and requiring DRM protection

Diamante: License manager provided by SafeNet

OTP: One time password authentication server and client

Table 14 shows some size data about the involved objects (reported SLoC include any library that must be compiled with the application code). The tasks that hackers are asked to perform on these programs are respectively:

Goal	Analyze the ability of ASPIRE to prevent attacks against license protections
Treatments	T1 = no information about assets; T2 = detailed information about assets
RQ1	To what extent do ASPIRE protections delay attacks against license management?
RQ2	What ASPIRE protections are most effective in delaying attacks against license management?
Subjects	Hackers from the SFNT tiger team
Objects	Diamante (binary code)
Tasks	Forge valid license
Data	Report about the different reverse engineering and attack activities (e.g., data de-obfuscation; identifier renaming; control flow reconstruction; code understanding; decompilation) carried out by attackers
Design	Long running (30+ days) case study

Table 12: SafeNet case study

- **Nagravision:** Violate a specific DRM protection of DemoPlayer
- **SafeNet:** Forge a valid license key that is accepted by Diamante
- **Gemalto:** Authenticate on OTP without having any valid credential

8.1.2 Data

Professional hackers are asked information about their expertise and experience. They are required to answer the following questions:

1. What is your programming experience in C? What C programming environment do you use?
2. What is your programming experience in assembly? What assembly programming environment do you use?
3. What is your experience in disassembly? What tools do you use for disassembling binary code?
4. What is your experience in analysing compiled binary code?
5. What is your experience in tampering/altering binary code?
6. What are the static analysis tools that you use when tampering/altering binary code (e.g., Ida-pro)?
7. What are the dynamic analysis tools that you use when tampering/altering binary code (e.g., Olly-dbg)?

After completing the attack, professional hackers are asked to complete a *Final Attack Report*. The attack report should cover the following points:

1. **Type of activities carried out during the attack:** detailed indications about the type of activities carried out to perform the attack and the proportion of time devoted to each activity.

Goal	Analyze the ability of ASPIRE to prevent attacks against secure authentication
Treatments	T1 = no information about assets; T2 = detailed information about assets
RQ1	To what extent do ASPIRE protections delay attacks against secure authentication?
RQ2	What ASPIRE protections are most effective in delaying attacks against secure authentication?
Subjects	Hackers from the GTO tiger team
Objects	OTP (binary code)
Tasks	Authenticate with no valid credentials available
Data	Report about the different reverse engineering and attack activities (e.g., data de-obfuscation; identifier renaming; control flow reconstruction; code understanding; decompilation) carried out by attackers
Design	Long running (30+ days) case study

Table 13: Gemalto case study

Object	C SLoC	H SLoC	Java SLoC	Cpp SLoC	Total
DemoPlayer	2,595	644	1,859	1,389	6,487
Diamante	53,065	6,748	819	-	58,283
OTP	284,319	44,152	7,892	2,694	338,103

Table 14: Size of case study objects (measured by `sloccount`), divided by file type.

For instance, hackers may want to indicate the following activities: (1) data de-obfuscation; (2) identifier renaming; (3) control flow reconstruction; (4) code understanding; (5) decompilation; (6) execution inspection (e.g., via debugger); (7) execution modification (e.g., via debugger scripts). Hackers should provide such classification for each working day, not just for the whole attack session.

- Encountered obstacles:** detailed description of the obstacles encountered during the attack attempts. In particular, hackers should report any software protection that they think was put into place to prevent the attack and that actually represented a major obstacle for their work.
- Attack strategy:** description of the attack strategy and how it was adjusted whenever it proved ineffective. Hackers should describe the initial attempts and the decisions (if any) to change the strategy and to try alternative approaches.
- Return of the attack effort:** quantification of the attack effort, if possible economically, so as to provide an estimate of the kind of remuneration that would justify the amount of work done to carry out the attack.
- Level of expertise required:** which of the successful actions required a lot of expertise, which could be done by script kiddies?
- Identification vs. exploitation:** for attacks succeeded once in the lab, attackers should describe what work would be required to exploit them in the real world (i.e., on a large scale, on software running on standard devices instead of on lab infrastructure, with other keys, etc.).

8.1.3 Design

The design of this experiment is a long running case study, with loose control on the involved subjects and mostly qualitative data collected during the execution of the experiment.

The plan originally included two replications of the experiment on each industrial partner. Since an exact replication would bring limited value to the project, we decided to replicate each experiment under different conditions, in order to collect more information about protections and attacks. The different conditions that we investigate are:

- **Protection configuration:** The same case study protected with different combinations of ASPIRE protections.
- **Information:** The same case study with the same combination of ASPIRE protections, but with more or less information provided to the participants. The information varies in terms of what are the sensitive assets protected with ASPIRE protections, what protections are deployed and, possibly, how these protections work.

Moreover, instead of conducting the two replications in separate moments in time, we decided to merge the two replications and conduct them one after the other, as two consecutive phases of the same experiment, to give hackers a larger continuous amount of time to work on their task. This allows a more flexible allocation of time to replications. In fact, if the first (harder) phase requires more time than expected, and if we consider it useful to the project, with this setup we can decide to let the first phase last longer and to consume a fraction of the time originally allocated to the subsequent phase, that is hence shortened in time. On the contrary, if the industrial hackers are able to complete the attack before the end of the first phase, they can anticipate the start of the subsequent phase, before the official beginning, thus limiting the waste of time and delivering interesting results sooner.

8.1.4 Qualitative analysis

Qualitative analysis of the reports collected from hackers will be the basis to answer RQ2. Evidence about the activities performed and the obstacles encountered will be mapped to the ASPIRE protections that were most effective in blocking the attacks mounted by professional hackers.

For what concerns RQ1, the answer may be boolean, i.e., the attack was or was not successful. However, in case of a non-successful attack, there might still be some degree of exploitation that was achieved, such as leakage of sensitive information, denial of service, or any other attack that was not the direct goal of the case study task. For this reason RQ1 is formulated in terms of the *extent* to which the attack is prevented. Again, qualitative data analysis will be employed to answer this question.

8.1.5 Threats to validity

The main threats to the validity of the case study belong to the conclusion, internal, construct and external validity threat categories.

Conclusion validity threats concern the relationship between treatment and outcome. We assume that unsuccessful attacks can be attributed to the ASPIRE protections and that the obstacles encountered during successful attacks can be also attributed to the ASPIRE protections, while we do not know what would have happened without the ASPIRE protections. To mitigate this threat, we will collect extensive feedback from the professional hackers, in order to be able to support our conjectures with objective evidence collected in the field.

Internal validity threats concern external factors that may affect the independent variable. While subjects are professional hackers who are used to the kind of attack tasks requested to them during the study, there might be factors out of our control that affect their performance. Being a long running case study, the degree of control that we can have on the activities performed daily by the professional hackers is limited. We reduce this threat to validity by introducing a structured and systematic data collection procedure and by scheduling regular conference calls with industrial participants.

Construct validity threats concern the relationship between theory and observation. They are mainly due to how we measure the effectiveness of the ASPIRE protections. We manually assessed the successful completion of the attack tasks to decide on the answer to RQ1. For RQ2, we performed a qualitative analysis of the report to obtain evidence in support to our conjectures. To minimise the risk of committing errors in the design of the experiment, the design has been discussed and revised by those project P.I. who are more expert in empirical studies with human participants. Moreover, to minimise the risk of committing errors in the setup of the material required to run the experiments, the industrial partner NAGRA and the protection owners have been involved in the preparation.

External validity concerns the generalisation of the findings. Being based on a case study, results may not generalise to different cases. On the other hand, the considered object is an industrial application, which makes it a quite meaningful case, and the protected asset (i.e., DRM) is also quite meaningful and representative, so we expect some degree of generalisability to similar applications and to similar industrial contexts.

9 Experiment in NAGRA

Section authors:

Mariano Ceccato, Paolo Tonella (FBK), Bjorn De Sutter, Bart Coppens (UGENT), Brecht Wyseur (NAGRA)

9.1 Protection Configuration

In this experiment, two configurations were considered.

In the first configuration, all techniques except anti-debugging and remote attestation were deployed. This configuration combines all protections that mitigate static attacks steps and tampering with the code, but allows the attackers to collect execution traces and to use live debugging techniques in support of dynamic reverse-engineering attacks. In other words, this configuration evaluates the effectiveness of combined protections to mitigate attack steps on those attack paths on which attackers have already succeeded in attaching debuggers or running the software in emulators or with instrumentation tools. The concrete list of protections is the following:

- White box cryptography.
- Data obfuscation (convert static data to procedure);
- SoftVM (client side splitting);
- Call stack check;
- Code guards;
- Binary code obfuscation (with flatten function and opaque predicate);
- Code mobility;

In the second configuration, the anti-debugging and remote attestation techniques were enabled as well, to steer the penetration testers towards attack paths consisting of static attacks on un-tampered code and/or to force them to focus on breaking the specific anti-debugging and/or remote-attestation techniques developed in the ASPIRE project.

9.2 Asset Information

The second dimension is intended to investigate the role of additional information provided to attackers (about protected assets and about protections) to make attackers complete the attack.

In Phase 2, we provided the professional hacker with information about the ASPIRE anti-debugging technique, to understand if any way to circumvent the protection could be elaborated based on detailed information about the employed anti-debugging technique.

9.3 Experimental Settings

The experiment was conducted by hackers in the forensic lab of Kudelski Security (. This lab has over 10 years of experience in forensics related to the video industry such as forensics of pirate set-top boxes. This lab has been assisting in evidence collection to providing expert witness in a court of law. In addition, this lab was also used for security evaluation of NAGRA's products prior to release. Since 2014, this lab is now also expanding beyond the sector of Digital Television and is offering services to third parties, such as security evaluation for other companies or evidence collection for police forces.

The lab is embedded in Kudelski Security, which is a business unit of the Kudelski Security that is separated from the digital TV business unit.

The main analyst in this experiment has been working for NAGRA since 2008, and has expertise in assembly, C, C++ and higher languages like Java and Python. In his current position he does reverse engineering and digital forensics for NAGRA, for external customers of Kudelski Security, and for police.

9.4 Results

The analyst has produced a detailed report as deliverable, which comprises the different steps that he has conceived in the analysis. In this section, we briefly elaborate on some main observations from this analysis. For more details and other observations we refer to the report, available in D4.06 confidential Annex I, Section 3.

9.4.1 Phase 1: White Box Cryptography

The task given to the tiger team was to extract a cryptographic key from the protected binary. In this first analysis, the binary did not comprise the anti-debugging technique.

From the beginning, the tiger team made the assumption that code obfuscation and anti-tampering techniques would be present in the protected use-case. Therefore, the attacker made immediately the assumption that analysis of traces would be the path of least resistance. As a result, the tiger team did not notice any integrity protection techniques that were present.

Static analysis of the traces enable the tiger team to identify the location of the white-box implementation. A custom script to perform pattern analysis was implemented to do so. The tiger team did not have time to complete the analysis of the traces in order to extract the secret keys.

9.4.2 Phase 2: Anti-debugging

Section authors:
Bjorn De Sutter

During the second phase of the experiment, the tiger team focused on breaking UGent's anti-debugging protection. Concretely, the tiger team tried to find a way to trace or live-debug the software despite the presence of the anti-debugging protection.

To assess the stealthiness of the protection, the tiger team was at first not introduced to the self-debugging approach of UGent's anti-debugging protection. While the team did discover "debugging"-like functionality in the application (notably ptrace activity), discussions during conf calls revealed that it was not trivial for the attackers to correctly identify the overall self-debugging approach.

During a subsequent conf call, the tiger team was then given a presentation about the overall scheme. After that presentation, it was relatively simple for the tiger team to observe different steps of the scheme's implementation, but building a complete picture still proved impossible, in particular because tracing or live-debugging of the application process (the debuggee) did not succeed, and because the alternative of static analysis was hampered by the code obfuscations (such as flattening).

The attackers attempted to use multiple tools for collecting traces, but failed. An overview of the precise attempts and of considerations on third-party tools can be found in D4.06 confidential Annex I, Section 1.

Finally, we want to point out that at no point (except for the already mentioned control flow obfuscation), the tiger team mentioned any of the other deployed protections as making their work on the anti-debugging protection harder. For example, all individual ptrace calls were still easy to observe with IDA Pro, incl. the preparation of the call arguments, from which the precise semantics of the individual calls can be derived.

9.5 Observations

Based on the data collected in the Attack Report, we can answer the research questions of the case study as follows:

RQ1 (attack delay): *The ASPIRE protections have prevented successful completion of the tried attack paths within the penetration testing time frame. No kind of exploitation was possible for the attacker within that time frame. By contrast, exploitation would have been trivial in an unprotected application. Hence the extent of protection can be regarded as effective in this case study.*

RQ2 (effectiveness of protections): *The (assumed) presence of obfuscation techniques and anti-tampering techniques lead the attackers to an attack path of least resistance dominated by tracing and live-debugging. Without anti-debugging protection, and after considerable engineering to tools to retrieve useful traces (which already required much more work than would have been needed with static analysis on an unprotected application), the attackers were able to identify the WBC code in the traces, but they could not extract the keys from that code within the time frame of the penetration test. When the software was protected with anti-debugging techniques, the attackers were not able to collect traces. Thus their dynamic attack path towards the WBC code identification was blocked. Overcoming the anti-debugging protection was determined to require a significant, but doable, amount of effort, for which resources were lacking in this experiment.*

Based on the results collected from the interviews with hackers and from the hackers' final report, we have distilled the following observations and lessons learned:

- *The attack strategy of hackers is highly adaptive.* Hackers change strategy based on the experienced or conjectured defence lines. In order to optimize the usage of the attack time at their disposal, hackers continuously adapt their attack strategy. They look for the attack path with minimum resistance, trying to circumvent rather than defeat the protections. The adaptive choice of the path to follow is based on obstacles actually observed and deeply analyzed during their attack attempts, but sometimes it was also based on hypothesis about defences that they assume are in place. The aggressive usage of attack time by means of an adaptive attack strategy demands for quick decision making, sometimes based on incomplete, partial or just conjectured information. As a consequence, adding multiple levels of protections is important. In fact, the presence of multiple lines of defence makes it more difficult for the adaptive attack strategy to converge to the path of minimum resistance.
- *The mere presence of protections affects the attack strategy, regardless of their effectiveness.* In this case study, hackers decided to perform dynamic attacks and gave up with static reverse engineering just because they realised that some form of source code obfuscation had been applied, without even trying to defeat the static protections. This is a consequence of the way hackers optimize the use of their attack time, looking for the path of minimum resistance. To save their scarce attack time, hypotheses about the protections that are in place are not necessarily verified in depth and this occasionally leads to wrong or inaccurate assumptions about the protections and their behaviour. The practical implication of this observation is that even defences with moderate strength can be useful to prevent the hackers from certain attack paths, leaving them with a smaller set of alternatives at their disposal.
- *Previous hackers' knowledge about identified protections guide the analysis attempts.* In this case study, statistical analysis was employed to identify bits that are likely to store pseudo-random, hence likely encrypted, values, based on the hacker's knowledge of AES, a technique that was supposed to have been used in the protected program. This clearly indicates a major role of the hackers' knowledge and previous experience in shaping the attack strategy and in selecting the attack tools. A lesson that can be learned from this observation is that involvement of professional hackers with remarkable experience is extremely important for the validation of protections, because attackers with such profile have the knowledge necessary to quickly identify the most promising approach to attack the application and circumvent its protections, have the experience with the tools necessary to implement the approach, and have the competence to quickly analyze the outcome of the attack attempts in order to steer and adjust the attack strategy.
- *Software protection is and remains an arms race.* Whereas current versions of attacker tools might be inadequate to circumvent certain protections, it is not infeasible to extend the tools to make them useful again.
- *Security by obscurity remains useful.* It was clear that attackers were delayed more when they were facing protections of which they had not been introduced to the inner workings and overall design of the protection. Of course, in that light a novel protection scheme can remain novel only once. Having a mix of possible implementations and overall designs, as well as deploying them in different ways such that the stealth is improved and attackers cannot readily identify which versions have been used, therefore remains important.

10 Experiment in SafeNet

Section authors:

Mariano Ceccato, Paolo Tonella (FBK), Bjorn De Sutter, Bart Coppens (UGENT), Michael Zunke,

Werner Dondl (SafeNet)

10.1 Protection Configuration

In this experiment, only one configuration was considered. The replication with different conditions consists in asking different groups of hackers to attack the case study, i.e. an internal expert from SafeNet and an external group of professional hackers.

The protected code delivered to industrial hackers is an Android app (ARM code) that includes all techniques that were compatible and in the scope of this case study. The concrete list of protections is the following:

- Remote attestation;
- White box cryptography;
- Data obfuscation (convert static data to procedure);
- Binary code obfuscations (control flow flattening, branch functions, opaque predicates);
- Call stack checks;
- SoftVM (client side splitting);
- Code guards;
- Client/server code splitting.

10.2 Experimental Settings

The experiment was conducted by two groups of analysts with complementary background: an industrial hacker working in SafeNet and an external industrial hacker team.

The internal hacker is a passionate reverse engineer expert and security engineer with 10 years of experience also in obfuscation and deobfuscation, LLVM, reversing and all the needed tools on Windows, Linux and mobile environments.

The external hacker team consists of experts on reverse engineering of Windows and Linux applications, kernel and user mode system development in C/C++ under Windows and Linux, Code Obfuscation, Microsoft .NET common language runtime and IL.

10.3 Results

The analysts have produced a confidential report as final deliverable, which comprises the different activities that have been performed, what obstacles have been encountered and what attack tools have been used. The report also includes considerations on the level of expertise required to perform each specific task and if it is possible and easy to automate the attack in a script. In this section, we briefly elaborate the main observations based on the analysis of the report. For more details and other observations, we refer to the report, available in D4.06 confidential Annex II, Section 1.

The attack can be split in a first static analysis part and a subsequent mainly dynamic analysis part.

10.3.1 Phase 1: Internal Evaluation in SafeNet

The first evaluation conducted internally in SafeNet was mainly based on static analysis. The attack consisted in the subsequent attack steps.

- **Analysis and reconstruction of direct control flow:** The analysts realised that control flow obfuscation was used in which direct transfers had been replaced with indirect transfers. This hampered their tools (Hexrays IDA Pro) in reconstructing the control flow graphs of the functions in the software. So they started to tackle the problem of reconstructing the direct control flow, such that tools like IDA Pro could reconstruct the original functions in the software. They took advantage of IDA Pro, on top of which they developed custom python scripts to replace the indirections (by means of branch functions) by direct transfers. That way, they successfully recovered the original control flow.
- **Analysis and removal of opaque predicates:** The next task was devoted to removing opaque predicates, which were correctly identified in the code and which complicated the original control flow. To analyse and get rid of opaque predicates, the analysts again wrote a python script on top of IDA Pro, in which they also used the Z3 solver.

This task also required a quite high level of expertise and automation is not easy. Even if still feasible (with the currently used opaque predicate patterns), the complete automation of this attack would require the use of complex research tools to perform symbolic execution.

- **Analysis of VM-protected code (bytecode):** The attackers observed that a virtual machine was embedded in the application, and by analysing the referenced strings, they recognized the use of the LLVM interpreter. In this version, once that was recognized, the attackers used existing LLVM tools to disassemble the bytecode that was embedded in the binary. The LLVM disassembler was able to decompile it and LLVM compiler was able to compile it back into ARM code.

This attack task required high level of expertise and it would be quite easy to automate it.

10.3.2 Phase 2: External Evaluation by Professional Hackers

The evaluation conducted by the external group of hackers included both static analysis and dynamic analysis. It is composed of the subsequent detailed attacks.

- **Disassembly:** To ease the dynamic analysis, the attackers first tried to perform dynamic attacks on the code that was deobfuscated statically. This failed, because the deobfuscation had replaced position-independent (obfuscated) code with position-dependent code. In the context of a dynamically linker library, like the use case under attack, position-dependent code cannot function correctly. So while the static deobfuscation attacks had proven to be good enough to reconstruct static images (control flow graphs) of the software under attack, it did not suffice to reconstruct a simplified, executable library.
- **Debugging in the emulator:** Debugging the code in an emulator with IDA Debugger caused the program to crash when opcodes were executed, i.e., opcodes not supported by the emulator. These opcodes were in fact part of the original, unprotected programs. More precisely, they were part of the functions that execute instructions in ARMv7 instruction set extensions to check for the presence of those extensions (in order to exploit that presence to obtain better performance). To overcome this limitation, the analysts patched those functions, in particular `_armv7_tick` and `_armv7_neon_probe`. This was a task that required a low level of expertise.
- **Debugging on a rooted device:** Then the debugging was moved to an actual Android device with root privileges. Here the next obstacle was represented by the used anti-debugging

protection that attached a debug process to the software, thus preventing the attackers from attaching their own debugger process. The attackers again used IDA Pro to analyse the code and to spot all uses of `ptrace` (in the anti-debugger protection). To overcome the protection (partially), the attackers patched the `.init_array` section and the import section. As a result, the software as a whole did not function anymore, but then at least the attackers could execute single functions in the software in isolation and observe their execution with a debugger.

To perform this task, a middle level of knowledge about Linux internals was required, in particular about how a process forks, about the process representation in the file system (i.e., `/proc` folder) and about how `ptrace` works.

- **Write custom loader:** In order to execute and observe the library functions in isolation and independently of the Dalvik app in which the library was embedded, the attackers extracted the library from the app, and wrote their own main application that loaded and invoked the extracted library functionality. A main obstacle for doing so was represented by the peculiar naming and call convention in the JNI (Java Native Interface) that in essence formed a wrapper about the functionality under attack. Still it was possible to patch the code in order to remove that wrapper layer, and to load the patched library into a main application written by the attackers to invoke the library functions in their own application. The analyst studied APIs (i.e., functions injected by Diablo as part of the binary-level protections) by calling them out of context, with new arguments and by checking the results.

This task required specific knowledge about C/C++ and about cross compilation.

- **System call Tracing:** To reverse engineer the code and understand its functionality at least partially, the attackers collected system call traces on the functions invoked by their custom main application. To collect those traces, they used the `strace` tool, which allowed them to gather information about, e.g., the opened and used files and sockets. This requires a quite low level of expertise.
- **String analysis:** To further comprehend the executed functionality, the attackers analysed the referenced strings. The search for potentially interesting strings and modules revealed the presence of known modules. These are `LibTomMath`, `LibTomCrypt`, `LibWebSocket`, `OpenSSL` and `LLVM`. This task required just low expertise.
- **Detection of crypto function:** Based on the obtained insights through executed system calls (including the files that were accessed) and referenced strings, the last step aimed at detecting relevant functions related to cryptography and the files storing the licenses under attack. A combination of static analysis and dynamic analysis was effective to achieve this objective. The experiment ended at this stage, with the attackers conjecturing that, with more time, they might be able to reverse-engineer the encryption algorithms and possibly extract the keys or reverse-engineer the use of data in the license files. However, they did not actually try that part of the attack.

A high level of expertise was required for this combined analysis.

10.4 Observations

Based on the data collected in the Attack Report, we can answer the research questions of the case study as follows:

RQ1 (attack delay): *The ASPIRE protections demonstrated to be effective in delaying attacks. In fact, before starting with the actual sensitive assets, attackers directly addressed the deployed protections with the objective to undo them. The attackers also deployed existing and customised analysis tools to this aim, to get a clear program to attack.*

RQ2 (effectiveness of protections): *Obfuscation was only moderately effective in delaying the attack to the remaining protection, since it could be resolved quite easily. This was expected because just off-the-shelf, basic obfuscations have been used in this study. Novel and more advanced protections, such as anti-debugging, were stronger and required much more time and effort to be resolved or circumvented.*

Based on the results collected from the interviews with hackers and from the hackers' final report, we have distilled the following observations and lessons learned:

- *The deployed control flow obfuscations need to be improved.* Even if control flow obfuscation and opaque predicates were effective in delaying the attack, because the hacker had to spend some time to undo them, they could be defeated with limited effort. In order to achieve a higher level of protection and to delay attacks more significantly, it is important to push the state of the art, and deliver stronger obfuscation protections. For example, opaque predicates can be formulated in such a way that it becomes extremely difficult to attack and solve them by means of symbolic execution. In particular, the slices of individual opaque predicates should be made dependent on each other.
- *VM-obfuscation should use custom bytecode.* Attackers could break VM-based obfuscation because the implementation used in the experiment was based on LLVM bitcode, an existing well-known and well documented representation. As such, tools are available to identify, read and manipulate this language. A stronger protection should use a custom representation of the VM bytecode, developed internally and not disclosed to the public. In this way, an attacker should at least recover the semantics of bytecode before attacking it. In the final stages of the project, SFNT actually contributed diversified bytecode techniques, but those came too late for inclusion in the tiger experiment.
- *Protections should protect each other.* Anti-debugging has been defeated by tampering with the library initialisation routine. It would be important to exclude this attack scenario, by protecting this feature with anti-tampering. Moreover, anti-tampering routines should be protected so that it should not be possible to attach a debugger to it. Since the protections, as deployed in this experiment, did not mutually protect each other, the attacker could identify a starting point to attack protections, to then proceed incrementally with the others, one by one. A mutual protection between complementary approaches is required to limit the possibility that an attacker finds a starting point.
- *On-line protections should be applied to crucial assets.* Some protections were not detected by the attackers. In particular, on-line protections (remote attestation, client-server code splitting) did not represent a major obstacle to the attack, because hackers simply did not notice their presence while attacking critical assets. This means that on-line protections were applied to assets that were not crucial, or at least that on-line protections were not applied to *all* the crucial assets. As a results, attackers focused on the assets what were protected only with off-line protections.
- *Anti-debugging protections should be deployed at a fine granularity.* Whereas the anti-debugging protections deployed in this experiment prevented the attackers from tracing and debugging the whole application, they were still able to execute some parts in isolation. The fundamental issue here is that anti-debugging was deployed to block traced or debugger-controlled execution of the early parts of the whole application execution. When the attackers skipped that part to study the assets in isolation, they were no longer blocked by the protection. To improve the strength of the protection of the library, it should be deployed independently of any knowledge about the expected or known control flow in a "normal" application: attackers are not bound by those expectations or that knowledge.

- *It is absolutely necessary to avoid that human-readable identifiers remain present in the linked-in protection components.* Many protections contain error handling code or debugging features that, even if they are disabled, still leave traces in the code, such as identifiable strings. Those prove to be excellent leads for attackers.
- *Attackers with different backgrounds deploy widely different attack strategies.* In the NAGRA tiger experiment, the attackers tried to avoid delay by control flow obfuscations by focusing on dynamic techniques. Part of the reason to choose dynamic techniques as the assumed path of least resistance was their extensive experience with dynamic techniques. In the SFNT experiment, at least some experts were more experienced with static deobfuscation techniques, and hence deployed those first. Also, in the NAGRA case, full traces were studied by the experts, while some of the SFNT experts decided to focus on selected forms of information obtained from partial traces, such as the syscall traces collected with strace.

11 Experiment in Gemalto

Section authors:

Mariano Ceccato, Paolo Tonella (FBK), Bjorn De Sutter, Bart Coppens (UGENT), Jerome d'Annoville (Gemalto)

11.1 Protection Configuration

In this experiment, three configurations of protections have been considered.

The first configuration of protections (used in Phase 1) was explicitly intended to assess Diversified Cryptography Libraries (DCL for short) conceived and developed by GTO. The DCL protection was the only protection applied to the code.

In the second configuration (used in Phase 2), all techniques that were compatible and in the scope of this case study, except anti-debugging were deployed. The concrete list of protections is the following:

- Diversified Cryptography Libraries;
- Binary Obfuscation (control flow flattening, branch functions, opaque predicates);
- Code Guards;
- Call Stack Check.

In the third configuration (used in Phase 3), anti-debugging was enabled as well, to steer the penetration testers towards attacks specific to anti-debugging. The complete list of protection techniques applied in Phase 3 are:

- Diversified Cryptography Libraries;
- Binary Obfuscation (control flow flattening, branch functions, opaque predicates);
- Code Guards;
- Call Stack Check;
- Anti-debugging;

11.2 Experimental Settings

The experiment was conducted by two groups of analysts: an external independent team and a team internal to Gemalto. A validation from an external and independent team was needed because Gemalto is interested in commercial exploitation, and customers appreciate and expect external evaluation.

It is a common practice in Gemalto to subcontract the security evaluation of some future products to external companies. These specialized companies challenge the security by using thorough testing to discover security weaknesses. Motivation can be either to detect threats during the development process as it has been done for DCL or to be able to let customers to evaluate if the potential vulnerabilities are acceptable regarding their constraints.

People from the company that have done the DCL security evaluation are security analysts and have more than fifteen years of experience with performing penetration tests, code reviews, reverse code engineering. They also have experience analyzing code from several architectures and are very qualified on vulnerability research, exploitation/post-exploitation techniques and implementation attacks on cryptographic systems. The company can also provide classical security evaluation based on side channel issues such as extraction of sensitive data using Differential Power Analysis or related attacks and bypassing security protections by means of fault injection.

The internal hacker team consists of a main contributor and someone else that helped for the configuration and the binary code analysis. The main contributor is working since more than 10 years in security and has been working for Gemalto since 2 years with a dedicated expertise on mobile security. In his background he contributed to Android malware detection when he was working for a Mobile Network Operator. He is comfortable with binary codes targeted for iOS, Android, Linux X86, Linux Mips and is familiar with popular tools used by the hacker' community.

11.3 Results

The analysts have produced a confidential report as final deliverable, which describes the weaknesses that they found in the protected code and the attacks that they could complete. In this section, we briefly elaborate the main observations based on the analysis of the report. For more details and other observations, we refer to the report, available in D4.06 confidential Annex II, Section 2.

The report is split into two: the part on the results obtained by the external team and the part results obtained by the internal team.

11.3.1 Phase 1: External Evaluation, DCL protection

Note that a specific application was given to the team for this evaluation. Code has been reduced to the minimum to keep the analysis focused on the DCL protection in order to maximize the DCL protection exposure. It is a simple Android application to demonstrate how an application-specific secret key provisioned on the device is secure at rest and also at run-time during the usage of the key without ever revealing the plain text value. The code of the DCL protection itself is not produced with the ACTC and does not take advantage of ASPIRE protections yet.

External analysts reported the following weaknesses of the protected code:

- **Shared libraries:** This vulnerability consists of missing protection of the shared library to prevent static reverse engineering. In fact, a global array of function pointers has been used to identify where the API is stored and when it is used. Through a static analysis, such APIs were tracked.

- **Encryption:** The internal encryption mechanism of the code was easy to bypass. In fact, the code was encrypted in the library and decrypted before being executed. By intercepting the clear code in memory before execution, the encryption has been broken.
- **Weak protection of library:** No anti-analysis techniques have been implemented in the library (root, debug), so it could be attacked.
- **AES Key extraction:** AES encryption Key (device key) should be strongly protected, instead of being present just as "plaintext" in the code. The AES key has been found through reverse engineering and dynamic injection.

For what concerns the extraction of the AES device Key, it should be noted that the Master Key was instead not identified neither extracted by the analysts.

As a result of the external evaluation the following recommendations have been formulated.

- Evaluators recommend enforcing anti-tampering and anti-hooking at Java level, because the library fully relies on Java for persistent storage.
- Evaluators strongly recommend protecting credentials by avoiding it being handled in clear in the device's physical memory.
- Evaluators strongly recommend to enforce anti-tampering on the whole solution, so as to prevent illegal manipulation of the system calls when the library uses the Android native OS functionalities.
- Evaluators recommend enforcing the protection of the AES key at runtime with a robust implementation that uses advanced cryptographic techniques, which are not prone to hooking attacks when combined with static analysis.

11.3.2 Phase 2: Internal Evaluation, First Configuration of Protections

The internal evaluation has been performed mainly in two steps, a first step of static analysis and then a combination of static and dynamic analysis.

Static Analysis: The first step was intended to identify how the application manages its persistent data. The adopted strategy consisted in decompressing the packaged Android app and reverse engineering the Java code.

The file *StorageManager.smali* was analysed in order to find sensitive data and information. In this file, data are stored using the Java *preferences* format, i.e. as key-value string pairs.

An XML file in the application's home directory contained an *optinfo* string, but it was encrypted. An assumption was made that the encryption key (or part of the encryption key) was embedded in the string. From that point, a dynamic analysis became necessary to continue the attack.

In this part of the attack, ApkTool was mainly used.

Combination of Static and Dynamic Analysis: During this second step, the following weaknesses have been detected:

- **Native function names not obfuscated:** The first goal was to find those functions where sensitive data are manipulated. Once the relevant functions have been found, a static analysis combined with dynamic analysis were used to retrieve sensitive data. In fact, function names of native functions (i.e., JNI interface) were not obfuscated.
- **PIN code can be brute forced:** It was possible to find the PIN code by brute force attack. In fact, various PIN have been tried and the right value has been found.

By using `gdb` and by checking the parameters used by `memcpy` in a specific function, some valuable assets were retrieved. Among the relevant assets that could be read in clear, there are: `counter_value`, `device_key` and `storage_key`.

To achieve this objective there was no need to break the authentication control, because the PIN value was obtained previously by brute force.

The tools used to accomplish this task are ApkTool, GDB, GdbServer and IDA-Pro. However, the analysts mainly used GDB and GdbServer, because they exploited a custom kernel module developed internally (on past projects) that was particularly useful and effective for reverse engineering and information collection.

This kernel module is proprietary and is not publicly available. Anyway, even if it were, analysts think that in order to use it to mount successful attacks an attacker would need very specific and advanced skills, such as knowledge on Linux kernel messages and on kernel memory and structures.

11.3.3 Phase 3: Internal Evaluation, Second Configuration of Protections

In the third phase of this experiment, anti-debugging was added and the attack was focused on breaking it.

The analysts realised that anti-debugging was based on calls to `ptrace` and they considered it totally effective in preventing the possibility to attach a debugger (e.g., GDB). Hence, analysts developed the following alternative attack strategies:

- Use an emulator such as QEMU to hook `ptrace` system calls. On these calls fake values are set on virtual CPU registers, in order to deceive the debugged application.
- A custom kernel module is run to hook the `ptrace` syscall.

However, because of time constraints, analysts were not able to complete this attack. They acknowledged that anti-debug is quite sophisticated.

At the end of the interview, we asked a general question to analysts about the possibility to script and automate the attack to this program (against all protections). Analysts conjectured that it might be possible to script all the attacks and package them in a single attack tool. The attack tool would perform static tampering to the library and it would involve a *cloning* attack to defeat the anti-tampering protection. The tampered library would be executed, but anti-tampering checks would anyway read the memory from the real protected library. However, in case there is no anti-tampering protection, this cloning step would not even be required and the running library could be directly tampered. By applying static tampering, the automatic attack tool would not need to defeat anti-debugging. All in all, no special skill would be required to use this tool and attack a program, as long as the attack is run on a device where the address space randomisation feature is disabled.

11.4 Observations

Based on the data collected in the Attack Report, we can answer the research questions of the case study as follows:

RQ1 (attack delay): *The ASPIRE protections have proven to be effective in delaying attacks. In particular, even if analysts had an idea on what strategy to deploy to defeat anti-debugging in Phase 3, they could not complete it before the end of the experiment.*

RQ2 (effectiveness of protections): *The most effective protection was anti-debugging, because it managed to delay analysts in completing the successful attack. In fact, analysts reported anti-debugging as the main obstacle they encountered during their attacks.*

Based on the results collected from the interviews with hackers and from the hackers' final report, we have distilled the following observations and lessons learned:

- *The boundary of protections should be enlarged.* The protections are not strong enough if they are only applied directly on the critical assets themselves, because the rest of the code may also leak relevant information. In particular, analysts reported that they started from the Java part of the Android app to understand how to call the native part. The (JNI) interface between the Java and the native code, in fact, has to respect documented naming conventions that depends on the Java names. Whole-software-stack obfuscation and other protections are needed to prevent such attacks.

This observation was in fact already suggested by members of the Industrial Advisory Board during the first meeting with the Board, and acknowledged by the project PIs at the time.

- *Brute force should not be possible on protected code.* To guess an asset (the PIN) attackers did not need to defeat protections, because they could brute force it. Protections can not be used to fix design faults in code that intrinsically leak information. Attackers will, in fact, try first of all to attack code through its weak points (e.g., side channels) before spending a lot of effort to defeat protections, which are supposed to represent the hardest tasks.
- *Advanced attack tools are effective, but they require peculiar and sophisticated expertise and knowledge.* To complete the attack, analysts used quite sophisticated and advanced attack tools, some of which are available only to industrial hackers because they were self-built. However, even if these advanced custom tools were publicly available, they could not be used so easily. In fact, according to the analysts, these advanced tools require specific and advanced skills with respect to the Linux kernel.

Part IV

Public Challenge

12 Public challenge

Section authors:

Bart Coppens (UGent)

The details of how the public challenge was designed is described in detail in Deliverable D4.05. The design of the website for the public challenge was also described in Deliverable D4.05.

After we released the challenge to the public on August 4th, we started promoting the public challenge. After the initial peak of activity that correlated with this promotion, we noticed that the interest had waned. To that end, we did some additional posts on social media in the first half of September. Details of this promotion are described in Section 12.1.

We had originally set the deadline of the challenge on September 30th. We had hoped to use this deadline to put pressure on the hackers to have something ready by the end of September so that we could analyse the results during the remainder of the project. At the end of September, however, only 5 out of 8 challenges had been solved. To try and maximize the results from the challenge, we announced an extension of the deadline. Hackers could still participate and get a bounty for solving any of the remaining 3 challenges. However, the paid bounty is only €175, rather than the €200 per solved challenge hackers could earn during the original period from the beginning of August until the end of September.

All 5 solved challenges have been solved by the same person, a hacker from Croatia. After conducting an exit interview with him, we have transferred him his €1000 in prize money.

The remainder of this section is structured as follows: first, we describe the promotion of the challenge in Section 12.1. Next, we describe how the interest in the challenge evolved over time in Section 12.2. Finally, in Section 12.3, we describe what we learned from the exit interview.

12.1 Promotion

We advertised the public challenge through various publication channels. In particular, we announced the challenge on both Twitter and Reddit. On Twitter, this was not only tweeted by our *aspirefp7* account, but also by people related in some ways to the project. These tweets were also retweeted, increasing the target audience significantly.

When we found that interest in the challenge was waning, we made additional postings that were intended to re-ignite interest in the challenge. This is the full list of social media postings:

- https://www.reddit.com/r/ReverseEngineering/comments/4wp5r9/aspire_public_challenge_prize_money/
- https://www.reddit.com/r/ReverseEngineering/comments/52jnht/still_some_challenges_left_in_the_aspire_public/
- <https://twitter.com/aspirefp7/status/774139109715550209>
- <https://twitter.com/aspirefp7/status/788820634574946304>
- <https://twitter.com/bwyseur/status/763034058477543424>
- <https://twitter.com/bwyseur/status/773134327483867136>

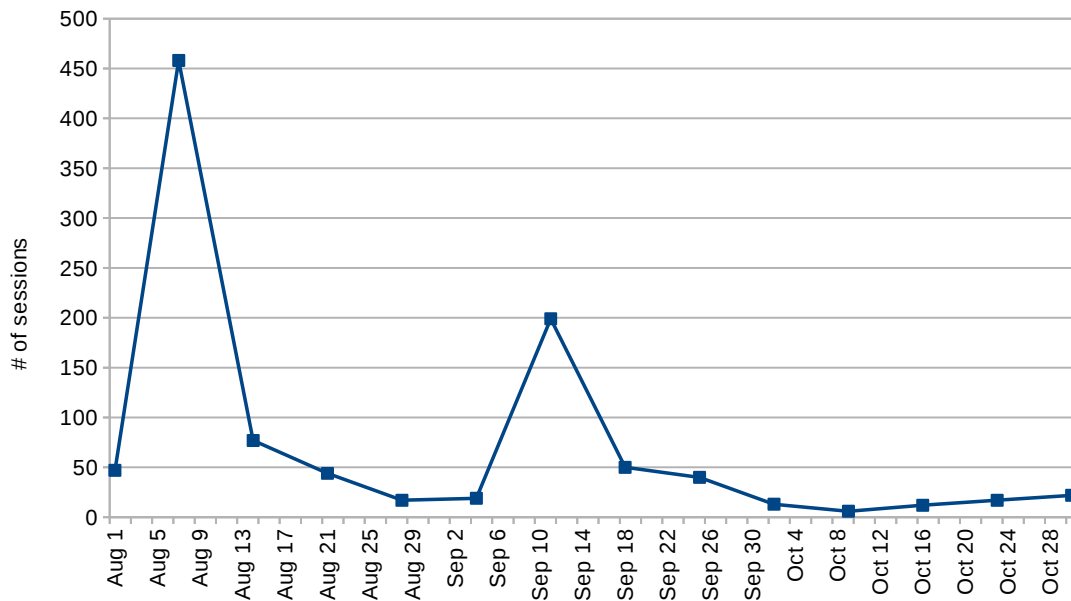


Figure 29: The number of website sessions per week.

- <https://twitter.com/veorq/status/763023899714084865>
- <https://twitter.com/sjzaib/status/762577644260995072>

Furthermore, we also advertised through word of mouth: the challenge was advertised to a circle of hackers known to Brecht Wyseur at NAGRA, it was circulated amongst students at FBK, it was mentioned at different talks given by PIs, etc.

12.2 Evolution of interest in the challenge

To get insights in how well our challenge was attracting users, we deployed Google Analytics on our web site, locate dat <https://bounty.aspire-fp7.eu/>. Figure 29 shows the number of *sessions* per week. A sessions is a period of time in which an individual user is active on the website, and can thus constitute multiple page views. The average number of pages per session is 2.25.

It is interesting to see how well website activity correlates with the registrations to our website. Figure 30 shows the number of new user registrations for each day of our period of activity. In both graphs, we see two big spikes of activity. These correlate with our promotions on social media. Note that the interest had completely dropped near the original deadline, but that after the announcement of the deadline extension there was a small increase in activity (both with respect to the number of sessions, as with new users). However, this final resurgence is not of the magnitude of the original peaks.

It is also interesting to know how diverse our audience was in terms of country of origin. As our registration only asked for a username and email address, we do not have a detailed breakdown of the nationality of our registered users. However, we do have this information for *all* website visitors through Google Analytics. Figure 12.2 shows a breakdown of sessions in terms of the originating country (limited to countries from which at least 10 sessions were observed.) First of all, we note that our audience is indeed quite global, and is not limited to the countries of ASPIRE partners, nor is it limited to European countries. Perhaps the most surprising aspect is that the number one country in terms of number of sessions is Croatia. This can be explained by the fact that our single successful attacker lives in Croatia.

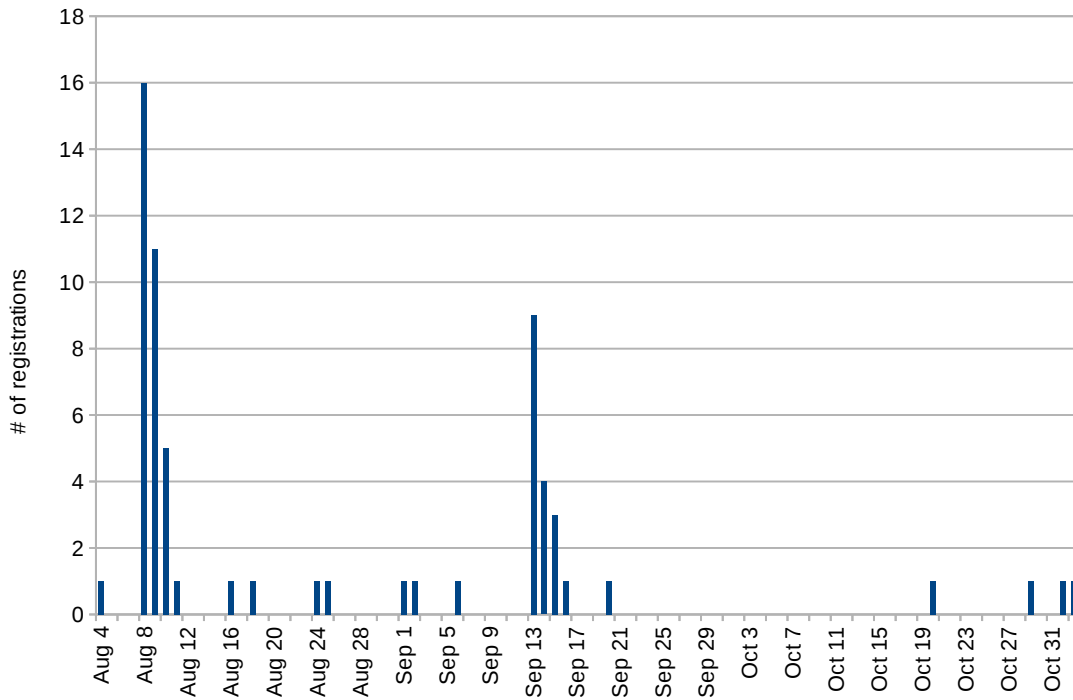


Figure 30: The number of new users per day.

Croatia	225
United States	197
United Kingdom	95
Italy	85
Germany	45
France	42
Belgium	38
Netherlands	24
Switzerland	22
Madagascar	20
Israel	17
Australia	16
Japan	15
China	14
Canada	12
Norway	10
Russia	10

Figure 31: Number of sessions per country, limited to countries with at least 10 sessions.

12.3 Exit interview with the hacker

In the end, only a single hacker has successfully attacked the challenges. However, he was able to break 5 of them, resulting in a total bounty of €1000. We conducted an exit interview through e-mail. In this section, we describe what we learned through this interview with regard to how he attacked our challenges. What we learned from this interview is served as input for Section 7 of Deliverable D1.06 on the validation of the protection techniques.

Most important to note is that the hacker did not succeed in successfully running the protected online applications. He tried running the binaries on two different devices (a MediaPad 7 Lite, running Android 4.0.3 with kernel 3.0.8, and a Huawei ALE-L21 phone, running Android 6 with kernel 3.10.86). The programs did not start on the phone as most recent Android devices require stand-alone command-line binaries to be Position-Independent Executables, which we had not provided; however, we only discovered this during the exit interview after the competition had finished. His MediaPad tabled *was* able to start the binaries, but they could somehow not connect to the protection servers and exited.

The hacker is currently a student, and his interest in reverse engineering is purely as a hobby. This challenge was his first major encounter with both the ARM architecture and Linux, as he usually reverse engineers x86(-64) binaries for Windows. He reported that some of the tools he used did sometimes seem to have problems with analysing the ARM code.

The hacker was able to solve all 5 offline challenges. He used a mix of both static and dynamic techniques, both for analysis and tampering. The main tool for the static techniques was IDA Pro (v6.8), where he made extensive use of the Python scripting support. He also used a decompiler. For the dynamic techniques, he used gdb. While he found it hard to estimate exactly how much effort he spent attacking each challenge or technique, he spent most of his time on the IDA Pro scripts he created to deobfuscate the code.

The attacks succeeded in the following order

- *Challenge 5*, which was protected with White-Box Crypto
- *Challenge 7*, in which we started from source code that had an ‘ugly’ structure, and was then furthermore protected with binary control flow obfuscations and anti-debugging.
- *Challenge 2*, which was protected with Residue Number Coding.
- *Challenge 3*, which is the same as Challenge 2, but adds anti-debugging.
- *Challenge 4*, where a main part of the checks is moved into the SoftVM.

Furthermore, all of these challenges were protected with code guards and binary obfuscations. All of these Challenges are described in more detail in Deliverable D4.05.

Next, we describe the attacker’s strategy (if any) for breaking protection techniques:

- *Binary Control Flow Obfuscations*: To deal with the binary obfuscations, the attacker wrote extensive Python scripts for IDA Pro that automatically removed opaque predicates and indirect jumps.

These scripts start from the program’s entry point and the list of initialization routines, and recursively traverse all code from there to remove as many obfuscations as possible.

This involved significant effort in first manually identifying the opaque predicates and understanding the control flow indirections, and then in automating the process of removing those.

We learned that the binary control flow obfuscations are able to delay attacks to some extent. Attackers are forced to spend effort in removing the obfuscations if they want statically analyse the program.

That the attacker wrote scripts to automate the removal of the obfuscations with pattern matching does suggest that we should try to insert our obfuscations in such a way that they are harder to detect and remove using only static techniques.

- **White-Box Crypto:** The attacker rather quickly identified that the main protection in Challenge 5 was WBC. He identified this through the way the program transformed the user's input. He did not bother to do any kind of cryptanalysis to recover the (secret) AES key, as it was not an asset that was required to solve the challenge. He brute-forced the correct answer, which was possible because the challenge processed 16 byte blocks independently of each other.

In this challenge, the asset was not the AES key itself. It was data which had been encrypted with that AES key. The strength of WBC is in protecting AES keys, not in protecting data encrypted/decrypted with such keys. Thus, the challenge was broken because the wrong kind of asset was protected with WBC, and WBC in itself remains unbroken. In future, we should pay more attention to this fact when choosing where and how to apply WBC.

- **Anti-Debugging:** The first time the attacker encountered anti-debugging was in Challenge 3. (He had solved Challenge 7 with static analysis.) He had the advantage that he did so right after solving Challenge 2, which is the same as Challenge 3 except that Challenge 3 has anti-debugging added. So the structure of the challenge itself was already familiar to him.

The attacker immediately recognized the technique as self-debugging. By studying the code. He noticed that our anti-debugging technique uses the BKPT instruction to switch from the debuggee to the debugger, which then uses ptrace to change the registers, memory state, and changes the control flow.

As there were not many of these BKPT instructions (and associated control flow transfers), the attacker was able to analyse exactly how these transfers worked. As the debugger and debuggee processes share the same binary executable with our anti-debugging technique, the targets of the debuggee-debugger control transfers are actually already present in the process image of the application that the attacker tries to attack. This means that he could just patch all the BKPT instructions to redirect the control flow to the correct locations inside the same process. After this, he was able to remove the anti-debugger initialization routine from the binary, as it was no longer needed to correctly redirect the control and data flow across both processes.

We learned that the attacker was delayed by this technique, but that it was easy for him to deduce the exact working of the anti-debugging technique because the challenge was too similar to a challenge without anti-debugging. Furthermore, even though he had to spend effort to work around it, this was still relatively manageable because of the limited code size of the challenge.

- **Data Obfuscations:** The attacker used dynamic analysis with a debugger to observe the decoded values when they were being compared with his input.

We learned that the obfuscations delay the attacker, but that they are also vulnerable to attack once an attacker can perform dynamic analyses.

- **Code Guards:** The attacker quickly discovered the presence of code guards in all our challenges. He treated the checksum functions as blackbox functions: they computed something but he did not really care about what or how (even though he eventually correctly identified the checksum function).

He circumvented the code guards as follows: he found where the checksum computation was performed and where the checksum is verified, and then put a breakpoint on the location of the verifier. Using the debugger he could then simply step over the verification to ignore the code guards.

Code guards were effective in delaying the attacker. However, they were relatively easy to circumvent because of multiple reasons: 1) we only used a single, simple hash function, and 2) the guards themselves were not verified.

- *SoftVM*: The attacker was able to identify the SoftVM's implementation as an LLVM-based virtualization protection through the fact that LLVM-related strings were present in the binary. By cross-referencing the functions present in the binary with the public source code of LLVM, he discovered that the function that executes the LLVM IR (`llvm::Interpreter::run`) has the ability to dump all IR instructions to the console when a debugging flag is enabled. He was again able to identify this function in the binary by studying the strings that this function refers to. He then verified that the function present in the binary had the same behavior as the source code he was studying, including the ability to dump the IR. Then he ran the program under a debugger, put a breakpoint on that location, and changed the control flow so that the LLVM function dumped the IR. Using the dumped IR, the attacker was able to determine its functionality and solve the challenge.

We learned that the SoftVM protection delayed the attacker, but that it was still relatively easy to reverse thanks to the easily identifiable off-the-shelf components used in the implementation, and to the debugging aids that were present in those components.

12.4 Lessons Learned

The previous section already listed the concrete conclusions and lessons learned regarding the protections that the one successful attacker was able to overcome.

Note that the attacker did not notice the call stack checks, as his attack never needed to execute code out of the context of the protected application. As the online techniques were not successfully attacked, we can learn no lessons about those.

One generic lesson we can learn from this experience is that it can be problematic to protect small applications with few protections. Since the protected applications are small and few protections are combined, attackers can easily identify the path of least resistance. Furthermore, as the program is small, this path of least resistance will be relatively short. This means that they will be able to quickly analyse and overcome the protections. Still, our protections *did* succeed in delaying the attacker, which is the goal of the ASPIRE protections.

List of abbreviations

ACTC	ASPIRE Compiler Tool Chain
ADSS	ASPIRE Decision Support System
AES	Advanced Encryption Standard
AHP	Analytical Hierarchical Process
AKB	Aspire Knowledge Base
API	Application Programming Interface
ASM	ASPIRE Security Model
ASPIRE	Advanced Software Protection: Integration, Research, and Exploitation
AT	Attack Time
BBCV	Basic Block Coverage Variability
BKPT	Breakpoint
CCD	Calling Convention Disruption
CC	Cyclomatic Complexity
CD	Control Dependency
CFIM	Control Flow Indirection Metric
CF	Confusion Factor
CFG	Control Flow Graph
CG	Call Graph
CLV	Code Layout Variability
DCDP	Dynamic Ciphred Data Presence
DCFC	Dynamic Control Flow Complexity
DCL	Diversified Cryptography Libraries
DCZ	Dynamic Code Size
DDFC	Dynamic Data Flow Complexity
DDP	Static Data Presence
DD	Data Dependency
DoW	Description of Work
DL	Description Logic
DPDP	Dynamic Plain Data Presence
DPFIFO	Dynamic Procedural Fan-In/Fan-Out
DPL	Dynamic Program Length
DRM	Digital Rights Management
DSC	Data Structure Complexity
ET	Execution Time
GDB	GNU Debugger
GUI	Graphical User Interface
HC	Heap Complexity
HD	Heap Dynamics

IDA Pro Interactive Disassembler Professional
IDE Integrated Development Environment
IEV Intra-Execution Variability
IOTV Instruction Operand Type Variation
IOV Instruction Operand Variation
IR Intermediate Representation
IS Ill-structuredness
JNI Java Native Interface
LLVM LLVM is not an acronym, it is the name of a compiler project
MATE Man in the End
MCFIFO Memory Locations Fan-In/Fan-Out
MF Measurable Feature
MLMI Multi-Location Memory Instructions
NPF Normalized Protection Fitness
OR Odds Ratio
OTP One time password
OWL Web Ontology Language
PCV Path Coverage Variability
PC Path Coverage
PDG Program Dependency Graph
PF Protection Fitness
PI Principal Investigator
PIL Procedural Ill-structuredness
PIN Personal Identification Number
PNML Petri Net Markup Language
PO Performance Overhead
RD Reuse Distance
RQ Research Question
S Stealth
SD Standard Deviation
SCDP Static Ciphred Data Presence
SCGC Staic Control Flow Complexity
SCZ Static Code Size
SDFC Static Data Flow Complexity
SDP Static Data Presence
SD Semantic Dependencies
SLoC Source Lines of Code
SoftVM Software Virtual Machine
SPA Software Protection Assessment
SPDP Static Plain Data Presence

SPFIFO Static Procedural Fan-In/Fan-Out
SPZ Static Program Size
SR Semantic Relevance
SR Success Rate
SSAI Syntactic Stubbornness Asset Impact
SSOI Syntactic Stubbornness Output Impact
SSCS Static Syntactic Stubborn Code Size
SV Static Variability
TR Trace Reproducibility
VAMO Variable-Address Memory Operations
UDCZ Unavailable Dynamic Code Size
UDDF Unavailable Dynamic Data Flow
UDFC Unavailable Data Flow Complexity
UML Unified Modeling Language
USCZ Unavailable Static Code Size
USDF Unavailable Static Data Flow
WBC White-Box Cryptography
VM Virtual Machine
WLMI Writable Location Memory Instructions
WP Work Package
XML eXtensible Markup Language
XOR Exclusive OR

References

- [1] The EPNK Petri Net tool. <http://www2.imm.dtu.dk/~ekki/projects/ePNK/>. Accessed: 2014-10-15.
- [2] Bertrand Anckaert, Matias Madou, Bjorn De Sutter, Bruno De Bus, Koen De Bosschere, and Bart Preneel. Program obfuscation: a quantitative approach. In *Proc. ACM Workshop on Quality of protection*, pages 15–20, 2007.
- [3] B. Auprasert and Y. Limpiyakorn. Underlying cognitive complexity measure computation with combinatorial rules. *Proceedings of World Academy of Science: Engineering & Technology*, 47:400–504, 2008.
- [4] Victor R. Basili. Software modeling and measurement: the Goal/Question/Metric paradigm. Technical Report CS-TR-2956, Department of Computer Science, University of Maryland, 1992.
- [5] B. T. Bennett and V. J. Kruskal. Lru stack processing. *IBM J. Res. Dev.*, 19(4):353–357, July 1975.
- [6] J. Cohen. *Statistical power analysis for the behavioral sciences (2nd ed.)*. Lawrence Earlbaum Associates, Hillsdale, NJ, 1988.
- [7] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical Report 148, Dept. of Computer Science, The Univ. of Auckland, 1997.
- [8] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '98*, pages 184–196, New York, NY, USA, 1998. ACM.
- [9] Kevin Coogan, Gen Lu, and Saumya Debray. Deobfuscation of virtualization-obfuscated software: A semantics-based approach. In *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS '11*, pages 275–284, New York, NY, USA, 2011. ACM.
- [10] J.S. Davis. Chunks: A basis for complexity measurement. *Information Processing & Management*, 20(1–2):119–127, 1984.
- [11] Jay L. Devore. *Probability and Statistics for Engineering and the Sciences*. Duxbury Press; 7 edition, 2007.
- [12] Chen Ding and Yutao Zhong. Predicting whole-program locality through reuse distance analysis. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, PLDI '03*, pages 245–257, New York, NY, USA, 2003. ACM.
- [13] Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. Repeatable reverse engineering with panda. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop, PPREW '15*, 2015.
- [14] Robert J. Grissom and John J. Kim. *Effect sizes for research: A broad practical approach*. Lawrence Earlbaum Associates, 2nd edition edition, 2005.
- [15] Maurice H. Halstead. *Elements of Software Science (Operating and Programming Systems Series)*. Elsevier Science Inc., New York, NY, USA, 1977.
- [16] Warren A. Harrison and Kenneth I. Magel. A complexity measure based on nesting level. *SIGPLAN Not.*, 16(3):63–74, March 1981.

- [17] S. Henry and D. Kafura. Software structure metrics based on information flow. *Software Engineering, IEEE Transactions on*, SE-7(5):510–518, Sept 1981.
- [18] Johannes Kinder. Towards static analysis of virtualization-obfuscated binaries. In *Proceedings of the 2012 19th Working Conference on Reverse Engineering, WCRE '12*, pages 61–70, Washington, DC, USA, 2012. IEEE Computer Society.
- [19] Cullen Linn and Saumya Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proc. ACM Conf. Computer and Communications Security*, pages 290–299, 2003.
- [20] Thomas J. McCabe. A complexity measure. In *Proceedings of the 2Nd International Conference on Software Engineering, ICSE '76*, pages 407–, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.
- [21] John C. Munson and Taghi M. Khoshgoftaar. Measurement of data structure complexity. *J. Syst. Softw.*, 20(3):217–225, March 1993.
- [22] Masahide Nakamura, Akito Monden, Tomoaki Itoh, Ken-ichi Matsumoto, Yuichiro Kanzaki, and Hirotsugu Satoh. Queue-based cost evaluation of mental simulation process in program comprehension. In *Proceedings of the 9th International Symposium on Software Metrics, METRICS '03*, pages 351–, Washington, DC, USA, 2003. IEEE Computer Society.
- [23] Enrique I. Oviedo. Software engineering metrics i. chapter Control Flow, Data Flow and Program Complexity, pages 52–65. McGraw-Hill, Inc., New York, NY, USA, 1993.
- [24] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2015.
- [25] Thomas Reps and Gogul Balakrishnan. Improved memory-access analysis for x86 executables. In *Proceedings of the Joint European Conferences on Theory and Practice of Software 17th International Conference on Compiler Construction, CC'08/ETAPS'08*, pages 16–35, Berlin, Heidelberg, 2008. Springer-Verlag.
- [26] F. Ricca, M. Torchiano, M. Di Penta, M. Ceccato, and P. Tonella. Using acceptance tests as a support for clarifying requirements: a series of experiments. *Information & Software Technology*, 51:270–283, 2009.
- [27] D.J. Sheskin. *Handbook of Parametric and Nonparametric Statistical Procedures (4th Ed.)*. Chapman & All, 2007.
- [28] Franz Stetter. A measure of program complexity. *Comput. Lang.*, 9(3-4):203–208, December 1984.
- [29] H. Tamada, K. Fukuda, and T. Yoshioka. Program incomprehensibility evaluation for obfuscation methods with queue-based mental simulation model. In *Proc. ACIS Int'l Conf. Software Engineering, Artificial Intelligence, Networking and Parallel Distributed Computing*, pages 498–503, Aug 2012.
- [30] Marco Torchiano. *effsize: Efficient Effect Size Computation*, 2015. R package version 0.5.5.
- [31] Yingxu Wang and Jingqiu Shao. Measurement of the cognitive functional complexity of software. In *Proc. IEEE Int'l Conf. Cognitive Informatics*, pages 67–74, 2003.
- [32] Babak Yadegari, Brian Johannismeyer, Ben Whitely, and Saumya Debray. A generic approach to automatic deobfuscation of executable code. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy, SP '15*, pages 674–691, Washington, DC, USA, 2015. IEEE Computer Society.