SEVENTH FRAMEWORK PROGRAMME

Advanced Software Protection:
Integration, Research and Exploitation

D4.02
Preliminary Complexity Metrics

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 / D4.02
WP and tasks contributing:	WP 4 / Task 4.2
Due date:	Oct 2014 – M12
Actual submission date:	31 October 2014
Responsible Organization:	UGent
Editor:	Bjorn De Sutter
Dissemination level:	Public
Revision:	1.0

Abstract:
The novel ASPIRE framework for measuring software protection strenght through the use of software complexity and protection resilience metrics is proposed in Part I of this deliverable. This framework tries to unify existing approaches to cover a wide range of protections and of attacks, incl. both static and dynamic attacks. Furthermore, in Part II, the progress in the ASPIRE security modeling is reported, in the form of updates to the Petri Net modeling approach, and to the UML models used for the ASPIRE Knowledge Base at the center of the ASPIRE Decision Support System.

Keywords:
complexity metrics, potency, stealth, resilience, Petri Nets, UML

Editor
Bjorn De Sutter (UGent)

Contributors (ordered according to beneficiary numbers)
Bjorn De Sutter, Bart Coppens, Stijn Volckaert (UGent)
Christophe Foket, Niels Penneman (UGent)
Cataldo Basile (POLITO)
Paolo Tonella, Mariano Ceccato (FBK)
Paolo Falcarin, Christophe Tartary, Shareeful Islam, Gaofeng Zhang (UEL)

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 fits in the overall ASPIRE project plan in two ways that correspond to the two parts of this deliverable.

In Part I, the initial results of the project research into software complexity metrics in Task T4.2 of the DoW are presented for the first time.

For this research, the goal-question-metrics approach is used. Two times two goals are identified. For combinations of protections, we need to measure and estimate the performance overhead and the increased effort needed in specific attack steps. To pose the relevant questions, the aspects of attack goals, attack subject, attack means, attack object, and attack actions are considered, all of which determine which software features should be measured or estimated, and how they should be measured to get an indication of the effort required per attack step.

In order to cover both static and dynamic attacks, and a wide range of protections, features related to potency, resilience and stealth aspects need to be measured or estimated in a unified framework. To allow this, we propose to approach all protections and attacks from a syntax-semantics perspective: protections try to increase the (apparent) complexity of the program semantics or add semantics, while attacks typically aim for the opposite, i.e., simplifying the representation of the program semantics to facilitate attacks steps. Based on this reasoning, we identify three questions: What code/data fragments are relevant for an attack step? 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? And how easy is it to simplify the syntax, semantics and mapping of the protected code, without oversimplifying it?

To answers the first question, we will rely on code annotations that mark the relevant code. To answer the second: we propose to use a wide range of dynamic and static complexity metrics: static and dynamic code size, static and dynamic control flow complexity, ill-structuredness, code layout variability, static and dynamic data flow complexity, semantic dependencies, and static and dynamic data presence. To answer the third question, we propose not to compute the complexity metrics directly on the relevant syntactic representations of the program semantics, but on simplified versions thereof. The simplification weights are derived from resilience-related and stealth-related features that are based on fundamental aspects of semantic relevance and static and dynamic code variability.

For each of the proposed features, one or more concrete metrics are proposed, totaling 32 different metrics. Finally, an approach is presented to aggregate the many metrics to fewer values that the ASPIRE Decision Support System will be able to use for its selection of the golden protection combination.

In Part II, this deliverable documents the security modeling progress that has been made in Task T4.1 of the project since the first presentation of the ASPIRE Security Model in deliverable D4.01 at M6 of the project, and since the first implementation description of that model in D5.01 at M9 of the project.

First, we have modelled two attacks to defeat relevant protections for the project use-cases: White-Box-Crypto and the SoftVM. This analysis effort has highlighted the need to use high-level Petri Nets for modelling, and we decided to use Petri Nets with Discrete Values (PNDV). These high-level Petri Nets allow adding discrete variables representing the attacker's intermediate knowledge while performing an attack, as well as adding conditions to the transitions. We evaluated different extensible Petri Net editors like CPNtools, PIPE and ePNK, and we chose the last one. ePNK is designed for PNDVs, it supports PNML standard for exporting Petri Net models, and it is Eclipse-based, thus easier to extend for our purposes.

Secondly, the UML Metrics sub-model of the ASPIRE Security Model is instantiated for the metrics presented in Part I.

Contents

1 Introduction 1

I ASPIRE Complexity Metrics 2

2 Goal - Question - Metrics Approach 2

3 Goals and Questions 3
- 3.1 Goals 3
- 3.2 Questions 4
  - 3.2.1 Attack properties that influence attack effort 4
  - 3.2.2 Classifying and unifying the relevant properties 6
  - 3.2.3 Attack fundamentals: syntax and semantics 7

4 Measurable Features 10
- 4.1 Measurable Complexity Features 10
- 4.2 Measurable Resilience and Stealth Features 12

5 Metrics 13
- 5.1 Complexity Metrics 13
  - 5.1.1 Static Code Size 13
  - 5.1.2 Dynamic Code Size 13
  - 5.1.3 Static Control Flow Complexity 13
  - 5.1.4 Ill-Structuredness 14
  - 5.1.5 Dynamic Control Flow Complexity 14
  - 5.1.6 Code Layout Variability 15
  - 5.1.7 Static Data Flow Complexity 15
  - 5.1.8 Dynamic Data Flow Complexity 16
  - 5.1.9 Semantic Dependencies 17
  - 5.1.10 Static Data Presence 18
  - 5.1.11 Dynamic Data Presence 18
- 5.2 Resilience Metrics 18
  - 5.2.1 Static Variability 18
  - 5.2.2 Intra-Execution Variability 18
  - 5.2.3 Semantic Relevance 19
  - 5.2.4 Stealth 19

6 Use of the Metrics 21

7 Related Work 22
- 7.1 Metrics 22
- 7.2 Human Studies 23

8 Conclusions and Future Work 24

II Update to the ASPIRE Security Model 25

9 Updates to the Petri Net Models 25
- 9.1 Extended Models 25
- 9.2 Tool Support 26
- 9.3 Exercises on use cases 28
  - 9.3.1 Attacks on WBC 28
  - 9.3.2 Attacks on a SoftVM 30

10 Updates to the UML models 36

List of Figures

1 Goals of the ASPIRE metrics 4
2 Screenshot of ePNK Eclipse-based tool 28
3 Petri Net for White-Box Cryptography attack 29
4 Petri Net VM attack 30
5 The `Metric` and its subclasses. 36
6 The `MeasurableFeature` and its subclasses. 37
7 The Metrics sub-model of the ASPIRE Security Model. 38

List of Tables

1 Comparison among Petri Net tools with respect to mandatory (M) and desirable (D) requirements. . . . 27
2 Legend 1 of the Petri Net for White-Box Cryptography attack 32
3 Legend 2 of the Petri Net for White-Box Cryptography attack 33
4 Legend 1 of the PN for VM attack 34
5 Legend 2 of the PN for VM attack 35

1 Introduction

Section authors:
Bjorn De Sutter (UGent)

This deliverable fits in the overall ASPIRE project plan in two ways.

First, this is the first occassion at which the initial results of the project research into software complexity metrics in Task T4.2 of the DoW are presented. This presentation constitutes Part I of this deliverable.

Secondly, this deliverable documents the security modeling progress that has been made in Task T4.1 of the project. It hence updates and extends parts of the ASPIRE Security Model as documented in deliverable D4.01 from M6, as well as the first implementation description of that model in D5.01 from M9. These updates and extensions are documented in Part II of this deliverable.

Part I

ASPIRE Complexity Metrics

2 Goal - Question - Metrics Approach

Section authors:
Bjorn De Sutter (UGent), Paolo Tonella (FBK)

In this document, we define a framework for the quantification of software protection, which extends the Goal-Question-Metric approach [11]. Furthermore, we start the instantiation of the framework in the ASPIRE context. The goals and the questions that we consider depend on the protections that the project is elaborating as described in the GA Annex II DoW, in deliverable D1.4, and the protection-specific deliverables of the project. 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 presented in deliverable D1.02. Starting from attacks and protections, we identify 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 will have to be applied iteratively. One reason to revise or extend the metrics over time is when attacks are added to the attack model, or when additional protections are to be evaluated. Another reason is that quite likely, the first instantiation of the framework for use in the ASPIRE Decision Support System (ADSS) will need considerable, iterative tuning before it becomes really useful.

This document therefore focuses on the presentation of the framework and on the initial instantiation results. At this time in the project, however, it is too early to present a complete instantiation or to evaluate the instantiation.

3 Goals and Questions

Section authors:
Bjorn De Sutter (UGent), Mariano Ceccato (FBK), Paolo Tonella (FBK)

3.1 Goals

The technology developed in ASPIRE aims at increasing the cost incurred to engineer a successful MATE attack on software and to lower the potential profit of exploiting such an attack, 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, this quantification needs to be performed in two different scenarios.

The first scenario is when an application has been protected with a combination of protections, and the benefits and costs of that combination of protections have to be quantified. This occurs in the context of the decision problem mentioned in step 1 in the previous section. The decision problem then is to decide whether one set of protections is better than another, and if so, to quantify the difference.

In that scenario, the code of both the unprotected and the protected software versions are available to compute metrics relevant for the attacks under consideration. The benefits and costs of the protections can hence be quantified by computing a vector of metrics on both versions, and by subtracting the two metric vectors.

The second scenario is when only an unprotected application is available, and we need to predict the benefits and costs of a potential set of protections to be applied. This scenario will occur in the ADSS. As described in deliverables D4.01 and D5.01, this ADSS has to select the golden combination of protections, for which it will rely on the ability to quantify the effects of applying potential protections on the Petri Net models of attacks, if possible without actually applying those protections.

In that case, the benefits and costs will be quantified by computing metrics on the unprotected application, and by estimating the impact of the protections on those metrics. In other words, if P is the program to be protected, and P_i with $0 \leq i \leq n$ is a protected version of P that can be generated by applying combination c_i of protections to P, we hope to estimate a metric $M(P, P_i, c_i)$ without actually having to produce the code of P_i itself. Whether or not that is feasible is, at this point in time, an open question. We assume the answer is positive, but more research is needed to validate that this goal is achievable. In particular when the metric M not only depends on a static representation of P, but also on traces T collected by executing P on some representative inputs, estimating the impact of the protections seems challenging. In that case, we want to estimate $M(P, P_i, c_i, T, T_i)$ with T_i the traces that would be obtained by executing P_i on the representative inputs, without generating P_i and hence without collecting the traces T_i. While we foresee that this can work for individual protections, we foresee it will be a major challenge when c_i consists of a combinations of multiple, possibly dependent protections. In the remainder of the project, we hope to overcome this challenge.

The concrete goals we foresee for the ASPIRE metrics, given an attack and a set of protections to be evaluated, are therefore the four goals of Figure 1.

With respect to goals 1 and 2, we can be very brief: for measuring the performance overhead, we can simply measure execution times, memory footprints, and file sizes. If necessary, these forms of 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

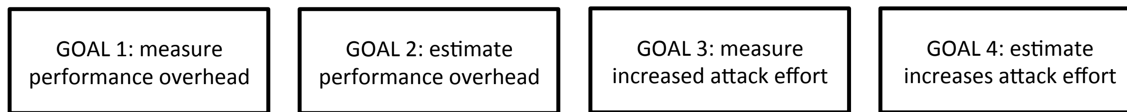

Figure 1: Goals of the ASPIRE metrics

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 and would involve the collection of an amount of empirical data that is beyond feasibility for the ASPIRE project. 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 ASPIRE protection is necessarily limited and insufficient for the training of a predictor.

However, it is definitely within the possibility and the objectives of 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 for an attacker. Such measurable features impacted by the ASPIRE protections give a clear picture of the amount of protection introduced into the code, relatively to the entire code base and with respect to alternative configurations of the same protections. Users of the ASPIRE protections will obtain precise, quantitative indications about the impact of each protection in the code.

3.2 Questions

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 study 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 term 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 attacker to execute one attack step, which roughly corresponds to one specific transition in a Petri Net model (see deliverable D4.01 and Part II of this deliverable) 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 the security modeling in Task T4.1 of the project, and is out of scope for this part of this deliverable.

3.2.1 Attack properties that influence attack effort

We identified five major aspects that contribute to the effort needed to mount an attack: the goal[1], 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 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

[1] In the remainder of this deliverable, the term "goal" will most often denote the objective of an attack step, in line with the use of this term in other deliverables. When the term refers to the goal of the evaluation approach, this will be made clear in the text.

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 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. This will particularly be the case when we will try to estimate the impact of certain protections on metrics without actually generating the protected code.

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. In particular when the impact of protections on total effort needs to be estimated (i.e., predicated as in scenario 2 above), a priori knowledge of the features of added code is needed. Another 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.

3.2.2 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 or estimate 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 goals 3 and 4 of Section 3.1 for the individual, meaningful elements of attack goals $\times$ subjects $\times$ means $\times$ object $\times$ actions. In essence, we have not two goals 3 and 4 for which we have to identify the relevant questions, but we have a whole goal set $\mathcal{G} = \{3,4\}\times$ attack goals $\times$ subjects $\times$ means $\times$ object $\times$ 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 [17, 18]. 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 [8], 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 Section 4.4, we cannot simply neglect dynamic attacks.

In this project, we therefor 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.

3.2.3 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 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 0xfffffffe and its integer value of -2 is

clear. The mapping between a ciphertext 32-bit number 0xfffffffe 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 simple[2] 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.

- Anti-debugging techniques essentially intervene in the semantics of a program. If the original program's semantics can be described as some function $[\![P]\!](i)$, with i the program input, then the protected program is $[\![P']\!](i, s)$, with s the system state affected by the presence of a debugger, and $[\![P]\!](i) = [\![P']\!](i, s)$ iff s does not indicate the presence of debugger.

- 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. If the original program syntax representation can be described as P, and its original semantics can be described as some function $[\![P]\!](i)$, then the protected program with representation P' has semantics $[\![P']\!](i, P)$, with $[\![P]\!](i) = [\![P']\!](i, P')$ iff the checked part of P' is what it should be. In other words, the protected program only provides the original input-output behavior when (part of) its representation being executed is not altered.

- 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.

- Advanced forms of remote attestation in essence add additional inputs to an application, such as nonces sent to the client by the remote attestation server. These inputs are not under control of the attacker. Similarly, additional outputs are added, in the form of the attestations that the client needs to deliver to the server.

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

[2]Here we do not yet consider more concrete meanings of simple, such as maintainable, reliable, testable, etc.

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 Measurable Features

Section authors:
Bjorn De Sutter (UGent)

To answer question Q0, we will assume that code annotations and program analysis allow us to identify the relevant code parts.

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]. This relevance weight will be based on the annotations that answer Q0. When complexity metrics are computed on the graphs and traces, 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.

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 superlinear 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 Debray et al. [22]. 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.

4.1 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 features 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.

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.

One such aspect is still missing, however, because we have not yet been able to completely assess its impact on attack effort and the best way to model or measure it. This is the aspect of missing code, i.e., code of which the attacker has no representation or knowledge. When code is split off from a client-side application and moved onto a secure server, the attacker has no access to that code. To attack the application, he might try to replace the server-side code by his own copy, which he might try to implement after learning the server-side functionality from observer the client-server communication. To cover that aspect and attack

scenario, concepts of learnability as used in the domain of cryptography might prove to be useful, but we need to investigate this further. Another option might be to consider as an additional metric the *fraction of relevant code* that models how much of the code the attacker has available.

Splitting off a code fragment does not only hide that code fragment itself, however. It will often also hamper the attacker in comprehending and attacking the other software fragments that do remain accessible to him. This is the case because, e.g., data dependencies of those fragments' live-in data are obscured. How to model or measure that effect needs further investigation as well. The challenge is that all of the aforementioned features will typically be measured by metrics that increase when a program grows larger, i.e., when functionality is added to the program. When functionality is split off a program as a protection, at least some of the metrics computed on the remaining part of the program should not decrease. To achieve that, we need additional features and metrics. At this point in time, we are still investigating which ones to use.

Finally, we want to point out that the need to correctly model the impact of missing code is not limited to protections such as client-server code splitting. It also arises whenever we need to model the fact that an attacker in some attack step might not have been able yet to collect a complete trace, or to disassemble the whole program. Whether and how we can unify the cases in which an attacker is truly missing code and when he is temporarily missing code, requires further investigation as well.

4.2 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 2014 by Debray et al. [22]. 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 [35, 20].

5 Metrics

Section authors:
Bjorn De Sutter (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 protected code. Instead we will estimate them for the specific set of protections supported by the ASPIRE compiler tool chain, when the ASPIRE Decision Support System is exploring the space of all possible protection combinations, of which it needs select the best, so-called golden combination. This was already discussed and formulated more formally in Section 3.1.

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.

5.1 Complexity Metrics

5.1.1 Static Code Size

Based on Halstead's metric for program length [27], 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) [39]. Rather than using them directly, we will adapt them to incorporate features of Harrison's *nesting depth* [28], and the *cognitive functional weights* [51, 10, 21] 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 [44]. 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 [37]) 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. [37]. 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 iff 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) suppos-

edly 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.[3]

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

[3] As 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, however. So we don't need to measure the change in threading complexity.

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 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 [40]. 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 [43].

Finally, we can measure the *static procedural fan-in/fan-out* (SPFIFO) of individual procedures [30, 42]. 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 they 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 [23, 12]. The link between reuse distance and program comprehension was already investigated in literature [41, 46]. 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.

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 we will 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?

5.2 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, Debray et al. have recently shown that very powerful, automated attacks can be engineered that rely on, in Debray's terms, quasi-constant behavior [22]. 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 3.2, this type of simplification is typically what an attacker

will try to undo or to work around protections. And as Debray has 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.[4]

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.

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 send 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 introduced in Section 4.

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

[4]Note 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).

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.

6 Use of the Metrics

Section authors:
Bjorn De Sutter (UGent), Paolo Tonella (FBK)

In the ASPIRE protection evaluation methodology, the proposed metrics will be used as follows:

- For each Petri Net transition that needs to be evaluated for a certain unprotected program P or protected program version P_i that deploys a certain combination c_i of protections:

  1. Compute the stealth and resilience metrics and derive the simplification weights based on the attack step features, on the a priori known features of the applied protections, and on the traces T (and possibly T_i) that are supposed to be available to the attacker in the considered Petri Net transition, if any.
  2. Compute the complexity metrics on the relevant part of the program, of which the components (instructions, blocks, procedures, ...) are weighted with the simplification weights, i.e., as if the complexity metrics would be computed on a simplified program. Furthermore, the properties of the attack step and of the applied protections determine in which way the weighted metrics computed for the components (graph nodes, sets of nodes representing procedures, ...) of the program are accumulated. This accumulation will typically take the form of simple summing, but we will also study other forms to model learning effects, and non-linear scaling effects of global attack steps. For each of the metrics $M_{i,j}$, with $0 \leq j < m$ and m the number of complexity metrics, this means we compute (or measure) $M_{i,j}(P, P_i, c_i, T, T_i)$. The result is an m-dimensional complexity metric vector $\vec{M}_i$.
  3. Aggregate the m different metrics in the m-dimensional vector $\vec{M}_i$ into a much lower number of metrics (possibly only one) that capture the overall complexity of the attack step, that the ADSS can use in its objective function[5] to optimize, and that the Petri Net simulation tools can accumulate while simulating the nets. For the time being, we foresee the use of aggregation functions of the form
  $$\sum_j w_j \frac{M_{i,j}}{M_{i,j} + \beta_i}.$$
  In these functions, the weights w_j depend on the considered attack step: clearly, the attacker effort needed for a tampering reverse-engineering attack step depends on other features than the effort needed to identify a sensitive value in a trace. The fraction $\frac{M_{i,j}}{M_{i,j}+\beta_i}$ is used to normalize the individual metric values [9]. This requires us to estimate β_i, for which we can take, e.g., the mean value of m_i over all i.

[5]That objective function will also include performance overhead or other forms of potential overhead, which we neglect here.

7 Related Work

Section authors:
Paolo Tonella (FBK)

With sufficient effort and resources, most software protection techniques can be defeated under the man-in-the-end (MATE) attack model. In fact, all the information needed to break a software system is available to the attacker in the executable or bytecode – only the method by which the information is protected in the executable prevents direct analysis. Assessing software protections means to estimate the extra delay an attacker would incur due to a particular protection technique used on a given application.

Most of the existing assessments of software protection techniques (and in particular for code obfuscation) are based on metrics, estimating the increased code complexity, or the increased difficulty of static code analysis. Only a few works are based on attacks performed by human subjects, but very few of them applied rigorous approaches, such as those available from the area of empirical software engineering.

7.1 Metrics

For an introductory overview of complexity metrics potentially useful in the domain of native code software protection, we refer the reader to the ISSISP 2014 lecture by the ASPIRE coordinator titled "Evaluating the strength of software protections". This presentation can be downloaded from the ASPIRE project website at http://www.aspire-fp7.eu. In addition, we discuss some groundbreaking work here.

The evaluation of the increased strength introduced by obfuscation techniques has been mainly addressed by using code metrics. Even if metrics are based on reasonable assumptions about the expected problems that an attacker would face to defeat the code protections, they just estimate and approximate a specific level of security that the underlying application is supposed to receive.

Collberg et al. proposed the use of complexity measures in code obfuscation tools to help developers choose among different obfuscation transformations [17]. A high-level approach has been proposed by Collberg et al. when they defined the concept of *potency* of an obfuscation as the ratio between the complexity (measured with any metric) of the obfuscated code and the complexity of the original source code, and the concept of *resilience*, i.e., how difficult it is to automatically de-obfuscate the protected code [19].

Karnick et al. defined more precise metrics for potency (combining nesting, control-flow and variable complexities), resilience (as the number of errors generated when decompiling the obfuscated code) and cost (as an increment of memory usage) [34]. Heffner and Collberg used metrics for obfuscation potency and performance degradation as they aimed at finding the optimal sequence of obfuscations to be applied to different parts of the code in order to maximize complexity and reduce performance overhead [29]. With a similar goal, Jakubowski et al. presented a framework for iteratively combining and applying protection primitives to code; they also used code size, cyclomatic number and knot count metrics to evaluate the code complexity [31].

Many authors have chosen just a few specific metrics, under the assumption that these are good indicators of the software complexity and of the task difficulty for attackers trying to break the code. Anckaert et al. attempted to quantify and compare the level of protection of different obfuscation techniques [8]. In particular, they proposed a series of metrics based on *code*, *control flow*, *data* and *data flow*: they computed such metrics on some case study applications (both on clear and obfuscated code), without however performing any validation on the proposed metrics. Linn *et al.* define the *confusion factor* as the percentage of assembly instructions in the binary code that cannot be correctly disassembled by the disassembler, assuming that the difficulty of static code analysis will increase with this metrics, even if it strongly depends on the disassembly tools and algorithms used [37].

Tamada et al. have proposed a mental simulation model to evaluate program obfuscation [46]. The mental model simulates the short term memory of humans as a FIFO queue of limited, fixed size. Then, the authors compute six metrics that account for the difficulty possibly encountered by humans understanding the program, in accordance with the simulation model. They show that the values of such metrics increase – hence making program understanding more difficult – when a number of well-known obfuscation techniques are applied to the program to be protected.

Wang et al. model attacks as Petri nets and compute the cost for a specific attack path from the following parameters associated with the transitions (i.e., the individual attack steps) in the considered path: (1) *Key attribution*: a complexity metrics of the software subjected to the attack step (usually a size metrics); (2)

Importance of the attack step in the attack process: between 1 (low importance; e.g., initialisation step) to 4 (high importance; e.g., core attack step) – the value of this parameter is assigned manually; (3) *Cost of the attack*: either 1, 5, 10, 15, 20 or 25 – the value of this parameter is assigned manually [49]. The cost predictions produced by this approach have not been evaluated empirically and several specific choices (e.g., form of the mathematical function used as predictor and parameter ranges) are not properly justified.

Probably the most comprehensive analysis is represented by the comparison conducted by Ceccato *et al.* [13]. This study considered many different applications from different domains, consisting of more than 4 millions lines of code. Forty four different obfuscations have been applied to the code. The effects of each obfuscation have been quantified separately as the changes occurred to the code according to 10 metrics, including Chidamber and Kemerer's modularity, (cyclomatic) complexity and size (lines of code). In order to provide reliable results, a statistically sound evaluation has been conducted.

Visaggio *et al.* used the *code entropy* and size to detect obfuscated malware code in Javascript [48]. Although their goal and metrics are different from those listed so far, they also analyze the difference between metrics values obtained from obfuscated code and non-obfuscated code. Zeng *et al.* analyzed different obfuscations to discover which ones can break different types of watermarks hidden in the code [52].

Alternatively, tools have been used to assess the *resilience* of code obfuscations. For instance, Sutherland *at al.* relied on a program binary instrumentation tool to measure the fraction of the obfuscating transformations that the attackers can undo automatically [45]. Goto *et al.* proposed the *depth of parse trees* as a measure of source code complexity [26]. Udupa *et al.* measured the resilience of an obfuscation by using the amount of time required to perform the automatic de-obfuscation to evaluate the effectiveness of control flow flattening obfuscation, relying on a combination of static and dynamic analysis [47].

7.2 Human Studies

Empirical software engineering is devoted to the design and execution of controlled experiments to study how developers change their productivity while working with alternative tools/approaches. Empirical software engineering requires several steps for carefully defining the experimental environment, such as stating the experimental hypotheses, defining relevant variables to measure, preparing attack tasks to be executed by the subjects, profiling the subjects, analyzing the data with proper statistics and elaborating the threats to the observation validity.

Despite the benefits of experimental investigation, in the security literature only a few works are based on attacks performed by human subjects on binary code [45] and even fewer works [15, 16] are based on sound empirical approaches, because the latter are expensive to conduct and time consuming. As an example, the comparison of just two obfuscation techniques with the involvement of humans playing the role of attackers took a quite long time to be concluded [14].

Sutherland *et al.* conducted an experimental study on the complexity of binary reverse engineering [45]. The authors of this study asked a group of 10 students (of heterogeneous level of experience) to perform static analysis, dynamic analysis and change tasks on several C (compiled) programs. They found that the subjects' ability was significantly correlated with the success of the reverse engineering tasks they were asked to perform.

A former collaboration amongst participants of the ASPIRE project resulted in some works devoted to present the design, planning and results of a series of experiments devoted to compare the level of protection offered by two of the most common code obfuscation techniques, *identifier renaming* and *opaque predicates* [15, 16, 14]. This study compared, by means of statistical tests and effect size measures, the capability and efficiency of subjects in performing attack tasks on clear and obfuscated code. The study was able to quantify the increased effort necessary to understand and attack an obfuscated program, with respect to the effort necessary for the non-obfuscated one.

8 Conclusions and Future Work

Section authors:
Bjorn De Sutter (UGent)

We have presented the framework in which we will instantiate the concrete complexity metrics that will be used in the ASPIRE protection evaluation methodology. This unifying framework is, to the best of our knowledge, a novel approach, that will require a lot of additional work to be fully instantiated and evaluated. We expect that many iterations and adaptations will still be needed.

First, we will build the tools to compute the metrics, and make sure they cover all revelant combinations of protections and attack steps.

Next, we will instantiate the many parts of the framework in the form of weights and aggregation functions.

Once these tasks are done for a significant set of protections and attack steps, we will start evaluating the framework, which we will do on the ASPIRE project use cases.

In the mean time, we will also consult with the ASPIRE Scientific Advisory Board and with software protection research we meet in our research networks to get their feedback on the novel approach presented here.

Part II

Update to the ASPIRE Security Model

The first version of the ASPIRE Security Model was presented in the deliverable D4.01, and we report in this deliverable about updates in the ASPIRE Security Model.

Section 9 describes the advancements in the use of Petri Nets as Attack Simulation Models. We report about the evaluation and selection of a proper Petri Net editor and the planning for extending it in order to meet the ASPIRE security model requirements. To verify the correctness of our choices and to test the applicability of the selected Petri Net model to real case studies, we have modelled two attacks related to different industrial use cases. We first present attacks against WBC, used in the NAGRA use case, and attacks against a SoftVM, used in the SafeNet use case. We recall here that D4.01 already showed that Petri Nets can be used to model attacks against OTP generators, used in the GEMALTO use case.

Section 10 shows the updates to the Metrics sub-model of the ASPIRE Security Model derived from the metrics framework reported in this deliverable.

9 Updates to the Petri Net Models

9.1 Extended Models

Section authors:
Paolo Falcarin, Christophe Tartary, Shareeful Islam, Gaofeng Zhang (UEL)

Petri Nets (PNs) are often utilized to model the flow of information in concurrent and distributed systems. Specifically, high-level PNs have been developed to achieve more descriptive power; in literature, high-level PNs represent different approaches to extend the basic PN formalism, such as Timed PNs (TPNs) and Colored PNs (CPNs).

While modeling attacks related to the ASPIRE use-cases, we realized the need to extend the PN model to meet the following requirements:

1. Each token represents a different attacker in a team of attackers in collusion to achieve the same goal: this way attacks performed in parallel by two colluding attackers can be simulated.
2. The use of variables can represent the attacker's knowledge, as a set of different types of intermediate information gained by the attackers while performing an attack.
3. The possibility of defining pre-conditions to enable transitions.
4. The possibility of adding a value or a mathematical expression to a transition to represent the effort to perform a particular attack step (transition). Such effort could be estimated using one or more data sources:
   - with a formula where the effort is estimated using code metrics and attacker expertise;
   - elaborating data gathered through empirical experiments;
   - using data provided by security-domain experts.

   We will focus on how to estimate the increase of effort due to the application of a particular configuration of protections, based on the metrics identified in Section 5. We will provide a first estimate by aggregating the different metrics for the different attack paths. Then we will be able to run simulations to refine the estimate of the increased effort.
5. The execution of actions in the transitions that can modify such variables. Such actions could include the estimation of the effort to perform the attack step.

The set of values of such variables represent the current knowledge known to the attacker. The results of the attacker's actions performed in an attack step can possibly update such values until the final goal is achieved.

CPNs use tokens of different colors, thus they can be used to represent different types of data (like code size) flowing through a net that are relevant to compute the efforts needed to perform each attack step. In CPNs a token is like an object that can contain a defined type of data. According to CPN definition, all data are encapsulated in the token, a place can host only one particular type of token and a transition can consume only one type of token. In case of complex sets of data, the PN can soon become complex.

A possible usage of CPN in our case would require to define all the intermediate data as attributes of a unique type of token flowing through the network. In this case the example will work in case of a single attacker and a single model. In case of multiple tokens (i.e., multiple attackers), different transitions can apply different modifications to different instances of the token, representing the fact that different attackers gain different knowledge depending on the path they have followed in the PN. However in our case this is not realistic because, in case of colluding attackers, it is more realistic to represent such acquired knowledge as shared data among the different attackers, i.e., as shared variables accessible from all transitions (attack steps). Moreover, in case of multiple PN models merged after being extracted from the ASPIRE Knowledge Base, the overall token in CPN would be the merge of different data types.

PNs with Discrete Variables (PNDVs) are a more recent PN extension that adds modeling convenience and compactness to PNs, while at the same time ensuring that most of the mathematical properties of PNs are still valid [32]. This model is a PN extended with a set of finite global integer variables, used in pre-conditions, that are guards on transitions. This type of extension better matches the requirements of ASPIRE security modeling. In fact, all the information used by the attackers can be decomposed and mapped to a set of global integer variables. For example, when looking for a cryptographic key into a binary file, the attacker usually needs to identify some areas of code worth further investigation. Such intermediate knowledge could be represented with an array of code areas, where each code area is represented by a couple of integer numbers. These numbers represent the initial offset and final offset with respect to the base address of the binary code.

Moreover, the ASPIRE Security Model is used to estimate the effort needed to perform an attack by combining the effort for the different attack steps. As discussed in Section 6, the effort for each individual attack step will be estimated by aggregating a number of metrics that, after aggregation, give an indication of the number of work units required to complete the necessary activities in that step.

In TPNs [50], each transition has a delay, representing the time needed to execute a transition. However we need to decide the type of delay as in TPNs literature, the delay is represented as fixed, stochastic or interval-based. PNs with fixed delays allow for simple analysis methods but it is not a realistic model as in attack modeling the duration of each attack step depends on different parameters (attack goals, means, subject, object, and actions, some of which are captured in the metrics proposed in Part I). Stochastic PNs [38] [25] use a probability distribution for each delay in the PN, but the choice of such a distribution is particularly difficult in security domain where most of the data are often unknown or unreliable. We decide to use interval-based delays. The range of such delays will depend on the aforementioned parameters and will be investigated in the remainder of the project. Therefore *minEffort* and *maxEffort* can be two shared variables continuously updated when running PN model simulations in different scenarios.

9.2 Tool Support

We have defined the list of the ASPIRE security model requirements for selecting a suitable PN tool. The first requirements with 'must' or 'can' are mandatory, while the remaining ones with 'should' and 'might' are desirable features:

1. The tool must be stable, reliable and still part of an active project supported by its developers.
2. It can design a basic PN model with places and transitions.
3. It can design high-level PNs with discrete values, by defining global variables.
4. It must be extensible to perform the evaluations of the ASPIRE security model.
5. It might be possible to add text annotations to places and transitions.
6. It might be possible to define conditions on transitions.
7. It should save models in standard PNML format [6].
8. It should rely on PNML meta-model specification to allow the definition of a formal mapping towards the OWL-based ASPIRE meta-model.

9. It should perform the simulation based on formulas and conditions attached to transitions.
10. It might be Java-based to be easily integrated with the ASPIRE Knowledge Base.
11. It might be based on Eclipse in order to reuse Eclipse plug-in features.

We have evaluated some open-source tools for PN modeling and simulation, alternative to CPN-Tools. For example, YAWL [7] is a powerful modeling tool for high-level business process modeling; initially its semantics relied on Petri Nets, but after many extensions, the current semantics are defined as a labeled transition system that cannot be translated back to a Petri Net. Moreover, the editor cannot be easily extended for our purposes so these issues make YAWL unfit for the scope of our project.

We have used PIPE, a simple Java-based PN editor, to draw simple models of the attacks on the use-cases. They were easily understood by the industrial partners when providing feedback on this modeling activities. PIPE is an open-source project allowing editing of simple low-level PNs [24, 5]. It is therefore not designed to store variables and conditions, which are typical features of high-level PNs. However, as this is an open-source project, the tool might be extended in the future.

We have evaluated CPN Tools [1] trying to use high-level PNs, such as CPNs [33] for adding variables and conditions but the tool's user interface is not very user-friendly as it required more time to learn how to use it (compared to other tools). Moreover the tool sometimes crashed and it is not easily extensible to meet the requirements of the ASPIRE security model: extending this tool with its scripting language could be time-consuming and it would have not been easy to integrate with the future Java APIs of the ASPIRE Knowledge Base. However the need to draw high-level PNs with discrete values led us to consider the Eclipse-based tool ePNK [3, 36], which provides a Java-based extensible open source platform for PN modeling, based on Eclipse Modeling Framework (EMF) [2] and the Graphical Modeling Framework (GMF) [4]. Current ePNK features are:

1. It can design a PN model with discrete values and save it as standardized PNML file.
2. The EMF modeling framework is used to represent the PNML meta-model: the GMF-based editor relies on such a meta-model to easily check that the modeled diagram is compliant to the PNML specification.
3. The PNML meta-model in ePNK is compliant to the PNML standard so that PNML files automatically generated through the EMF framework are consistent with the standard.
4. It can do the simulation based on formulas and conditions attached to transitions. We can use it to perform basic security model evaluations.
5. It is easy to plug-in into Eclipse. We can do further development based on existing programs and packages.
6. Global variables can be defined and used by various transitions in a PN.

In Table 1 the three tools are evaluated with respect to our requirements.

Requirement	Type	Description	ePNK	PIPE	CPN Tools
R1	M	Stable tool	yes	yes	yes
R2	M	PN editor	yes	yes	yes
R3	M	High-Level PN editor	yes	no	yes
R4	M	Extensible editor	yes	yes	yes
R5	M	Text annotations	yes	yes	yes
R6	M	Conditions on transitions	yes	no	yes
R7	D	PNML	yes	no	yes
R8	D	PNML meta-model	yes	no	no
R9	D	Simulation	yes	yes	yes
R10	D	Java	yes	yes	no
R11	D	Eclipse	yes	no	no

Table 1: Comparison among Petri Net tools with respect to mandatory (M) and desirable (D) requirements.

The Eclipse-based tool ePNK (see Figure 2) seems the best candidate as it relies on EMF and GMF, the well-known modeling plug-ins, but there are still some features ePNK cannot provide, so we will have to extend it by developing another Eclipse plug-in to provide support for:

- the connection of metrics formulas to the transitions to be used to estimate the overall effort increment;
- the pre and post conditions in transitions, which are not supported yet, and there is only one type of condition for managing transitions now. We need to extend the related classes to realize this;
- the creation of our simulator to evaluate the attacker effort;
- the connection to the ASPIRE Knowledge Base.

Moreover we can think of adding other information to transitions that might be processed like the probability distribution of a transition's effort. Once this information is added into the model editor, our Eclipse plug-in can be used to simulate and estimate the total extra effort of different attack paths depending on the data also extracted from the ASPIRE Knowledge Base.

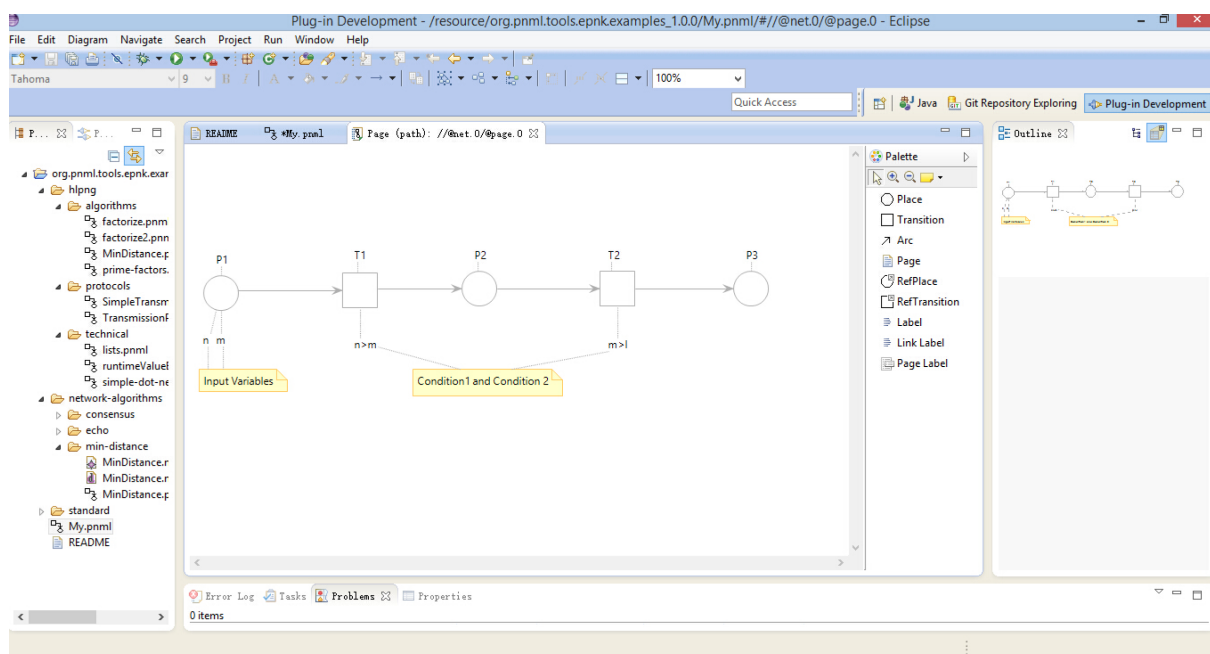

Figure 2: Screenshot of ePNK Eclipse-based tool

9.3 Exercises on use cases

In this section we are reporting the modeling on two other use-cases, which has triggered the need for extending the security model based on PNs towards high-level PNs with variables and conditions (PNDVs).

To build these two attacks with our industrial partners from the perspective of use cases, we have had four rounds of interactions with the experts in the companies in ASPIRE: initial discussion (Apr. 2014), initial modeling (May 2014), model revision (June 2014), modeling completion (Sept. 2014). There were three audio-conferences per use-case discussions to align with our partners, solve unforeseen problems, and reach the final objective.

9.3.1 Attacks on WBC

The attack on White Box Cryptography (WBC) described in the Attack Model (deliverable D1.02 Section 5.2) has been modeled with a high-level PN using variables and conditions, as shown in Figure 3. In this PN model for WBC we have identified three categories of attacks: static analysis, dynamic analysis, and WBC-attacks. For each of these categories we have named the attack steps with labels starting with **Ts** for static analysis attack steps, **Td** for dynamic analysis attack steps, and **Tw** for white-box cryptography attacks.

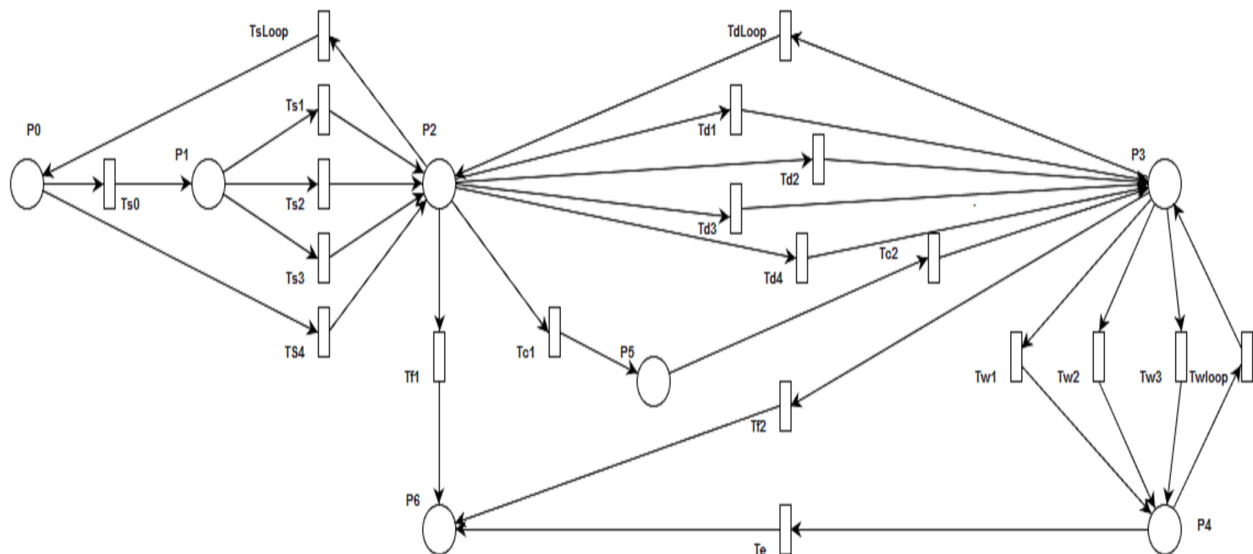

Figure 3: Petri Net for White-Box Cryptography attack

Starting from the initial state P0, Ts0 represents the task of using static analysis tools to identify the cryptographic primitive used in the binary code. We hence need a variable to store the name of the primitive used.

The variable *crypto* can be defined to store one value from the enumeration {AES, RSA, unknown} and it is initialized with the value *unknown*.

In case the attacker has detected AES-related binary code (i.e., *crypto* = 'AES'), he will run a more precise static analysis with AESKeyFinder (represented by **Ts1**), or in case RSA-related code is found (i.e, *crypto* = 'RSA'), he will run a RSAKeyFinder (represented by **Ts2**); otherwise the IdaPro findCrypt2 plug-in can be used (**Ts3**). The attack step Ts1 is then represented by a guarded transition (if *crypto* = 'AES') and so is Ts2 (if *crypto* = 'RSA'). The attack step **Ts4** represents the usage of other static analysis tools on binary code.

In place P2 the attacker has run one of the former attacks and the possible outputs can be different as the success of such attacks might depend on code complexity, attacker expertise and other factors. The possible outputs available at P2 are the secret key hidden in the code, or a set of the code locations where to further search the binary code with other tools, or none of the above. We can then define the Boolean variable *keyFound* that will be set to *true* as soon as the key is found. We can define the type *CodeArea* as a pair of integer numbers, to represent with two integer indexes the beginning and the end of an area of binary code (interpreted as an array of bytes starting from index 0). Therefore we can now define the variable *codeLocation* as an array of CodeArea types, to represent different (and in general non-adjacent) areas in the binary code. Such *codeLocation* arrays can be used to store the different areas of the binary code identified, e.g., by the static analysis tools as primary suspects of hiding or processing the key.

In case the attacker immediately finds the key, he can skip to the final place **P6** by firing the transition **Tf1** (guarded by the condition *if keyFound = TRUE*). If only code locations are found, at this point the attacker has a choice of trying static attacks again (firing the dummy transition **Tsloop**) or of changing his strategy by trying to use dynamic attacks. Such a choice can be taken randomly during the model simulation or it might be customized (for example depending on the attacker's expertise) to give higher priority to one transition than to another. In case the crypto primitive has been identified and its code location has been found, the attacker can also decide to skip the dynamic analysis and instead immediately try the cryptographic attacks by firing the guarded transitions **Tc1** and **Tc2** and moving the token to place **P3**.

In case the attacker wants to try dynamic analysis techniques, he has several options. If the code can be instrumented, the attacker can try to extract the execution traces (**Td1**). If the code can be executed, he can analyze the memory pages (**Td2**). If the code can be instrumented/rebuilt, he can try to tamper with the binary code and change its execution (**Td3**), otherwise he can intercept the system calls made by the binary to discover the code locations of such system calls (**Td4**). After running one dynamic analysis the token representing the attacker will be in place **P3**. In case the attacker has then found the key, he can skip to the final place **P6** by firing the transition **Tf2** (guarded by the condition *if keyFound = TRUE*). Otherwise, these attacks can be iterated (by firing the loop transition **Tdloop**) until the key is found or the attacker decides

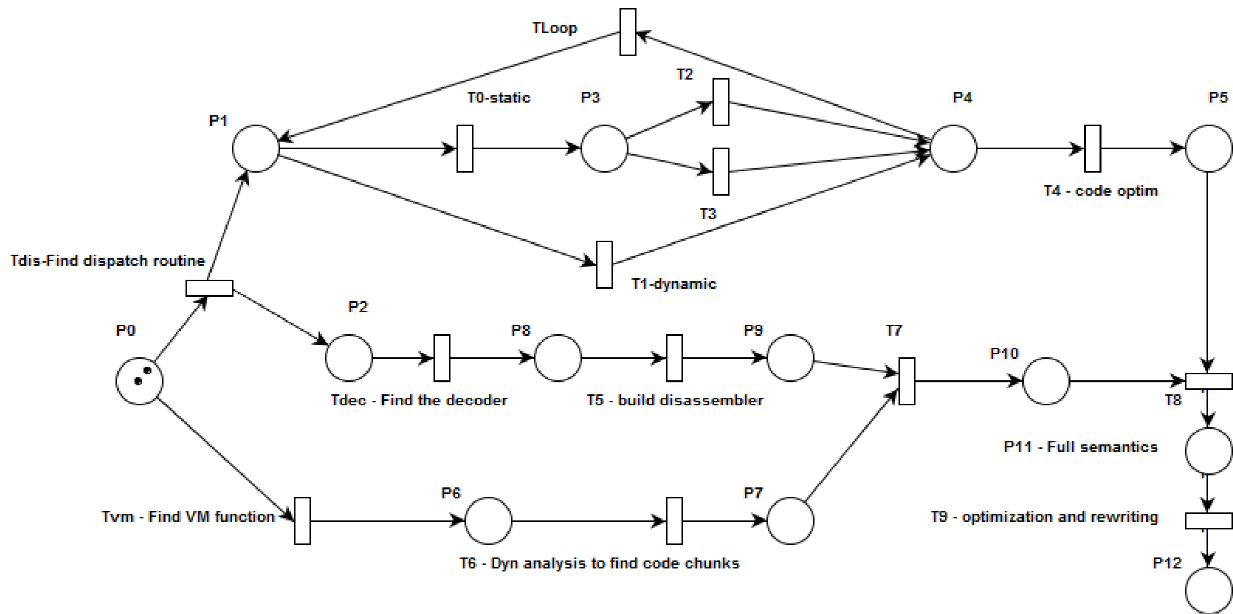

Figure 4: Petri Net VM attack

to change strategy and try the crypto attacks.

Table 2 depicts more details on each attack step for the aforementioned static and dynamic attacks. Table 3 does the same for the remaining attack steps.

Once the attacker reaches the state P3, he starts to analyze the mathematical properties of the cryptographic implementation: attacks Tw1 (differential cryptanalysis) or Tw2 (linear cryptanalysis) or Tw3 (algebraic attacks) are available depending on different pre-conditions. These attacks can be iterated (by firing the loop transition **Twloop**) until the number of key fragments identified in this process can be considered small enough to proceed with **Te** to apply heuristics or a brute-force attack to attack the missing key fragments. If this attack is successful and the key is found, the final goal P6 is reached.

9.3.2 Attacks on a SoftVM

The SoftVM is an interpreter that fetches bytecode from memory. For each bytecode, the SoftVM executes the corresponding native code stored in the respective Instruction Handler (IH), and then loads the next bytecode. The bytecode is not stored in a single file or data structure but it is split in different code chunks spread throughout the native code. The VM implementation is split in a set of IHs which might be obfuscated and then encrypted and then spread through the native binary code using a binary rewriting tool (e.g., Diablo). Each code chunk can contain one or more bytecode instructions. The VM contains different code portions that are interesting to the attacker:

- the VM function called by the native code to transfer control to the VM;
- a decoder, which translates the bytecode into native code;
- a dispatch routine that given a particular bytecode invokes the IH.
- the different Instruction Handlers (IH).

Figure 4 shows a PN representation of attacks on these assets. **P0** is the starting state with two tokens to represent the fact that the attacker has two parallel attack steps to perform, represented by the initial transitions **Tdis** (finding the dispatch routine in the binary code) and **Tvm** (finding the VM function in the binary code). Once the dispatch routine is located in the binary code, two other parallel paths start from places **P1** and **P2**: one to search for the IHs, the other starting by finding the decoder routine (**Tdec**). The first path to search for IHs can use static analysis tools (**T0**) to analyze the binary to identify the interface between native binary code and VM-related IHs spread around into the binary. Alternatively the attacker

can use dynamic analysis tools (**T1**) for the same goal to identify the IHs in the binary. The attacker can iterate again and again (through transition **Tloop**) using different static/dynamic analysis tools to better identify these code and possibly data locations. In **P4** the attacker has a set of code and/or data locations in the binary that correspond to the IHs, or at least to the attacker's best estimate using the available tools. If the attacker has found the decryption function using dynamic analysis (**T1**), then he also observed the arguments of the calls to the function and therefore also knows the addresses of the result buffers with the plain IH-code and he moves to P4. On the other hand when the attacker uses static analysis he can find out if IHs are encrypted: if the IHs are not encrypted the attacker can skip to P4 through the transition T3. If IHs are encrypted, the attacker has identified the decryption function using static analysis (**T0**): as he does not know the addresses of the plain text buffers (in fact the buffers might not even exist yet), the attacker can try to decrypt the encrypted data using statically identified decryption keys (**T2**).

He can repeat the process for each IH (going through Tloop). When all the IHs are available then the attacker can continue from place P4: now the IHs are possibly obfuscated so a tool can be used to optimize the binary code of these IHs (**T4**): at **P5** all the IHs are known to the attacker. The aforementioned steps are detailed in Table 4. The following ones are described in Table 5.

The other path from P1b starts with Tdec to find the decoder routine that translates the bytecode (given as input to the decoder) into native code. Once the decoder is found and its input bytecodes have been observed, the attacker can build a custom disassembler (**T5**). When the attacker has found the VM function (**Tvm**) called by the native code, he can use dynamic analysis (T6) to find the code chunks sent to the VM to be interpreted. Once the attacker has built a custom disassembler (**T5**) and has found the code chunks, he can manually understand (**T7**) the full semantics of the bytecode program. Then the attacker has to combine (**T8**) the information about the bytecode behavior with the semantics of the IHs. Once the full semantics is known the attacker can use a binary rewriting tool to optimize and rewrite the VM code (**T9**) and transform it back to binary code equivalent to the original code transformed into VM code.

Label	Description	Pre-condition	Input	Output	Goal
Ts0	Determine the crypto primitive used		Binary code	AES or RSA or unknown, The knowledge of which crypto primitive is implemented, and how	P1
Ts1	Static analysis with AESKeyFinder	AES	Binary code	Key or key location	P2
Ts2	Static analysis with RSAKeyFinder	RSA	Binary code	Key or key location	P2
Ts3	Static analysis with IdaPro findCrypt2 plug-in		Binary code	Key or key location	P2
Ts4	Analyze the binary code (static structural code and data recovery)			Code fragments possibly containing the Key location	P2
Tsloop	Go back and try different static analysis	Key not found: choice to loop back or deploy dynamic attacks			P0
Td1	Execution traces	Binary code can be instrumented	Binary code	Execution traces	P3
Td2	Analyze the memory pages during execution	Binary code can be run in debug mode	Binary code	Location of target memory fragments	P3
Td3	Tamper with binary and its execution	Binary code can be instrumented and rebuilt	Binary code	Tampered binary code, Possible Key location	P3
Td4	Intercept the system calls		Binary code	Code location of System calls and data passed to O.S.	P3
Tdloop	Go back and try different dynamic analysis	Key or its location in code not found or attacker choice			P2
Tc1	Fake transition	Code containing the cryptographic algorithm located + Cryptographic primitive identified	N/A	Code lifting possible	P5
Tc2	Fake transition	Code containing the cryptographic algorithm located+ Cryptographic primitive identified + It is known how to efficiently crypto-analyze the primitive	N/A	Cryptanalysis of the primitive enabled	P3

Table 2: Legend 1 of the Petri Net for White-Box Cryptography attack

Label	Description	Pre-condition	Input	Output	Goal
Tw1	Differential cryptanalysis	Code containing the cryptographic algorithm located + Cryptographic primitive identified	Code containing the cryptographic primitive + Pairs of input/output (empty for the first attack) + Candidate values for the key	Updated set of candidate values for the key (ideally, one) + Pairs of input/output which could be reused in another attack if needed	P4
Tw2	Linear crypto-analysis	Code containing the stream or block cipher located + Cryptographic primitive identified with AESKeyFinder	Code containing the stream or block cipher + Pairs of input/output (empty for the first attack) + Candidate values for the key	Updated set of candidate values for the key (ideally, one) + Pairs of input/output which could be reused in another attack if needed	P2
Tw3	Algebraic attacks	Code containing the cryptographic algorithm located + Cryptographic primitive identified	Code containing the cryptographic primitive + Pairs of input/output (empty for the first attack) + Candidate values for the key	Updated set of candidate values for the key (ideally, one) + Pairs of input/output which could be reused in another attack if needed	P2
Te	Heuristics of Brute force on remaining	Identified information about the key	Key fragments	Attack successful	P6
Tf1	Return the key and terminate attack	Key recovered at P2	Key	Attack successful	P2
Tf2	Return the key and terminate attack	Key recovered at P3	Key	Attack successful	P6

Table 3: Legend 2 of the Petri Net for White-Box Cryptography attack

T	Description	Input	Tool/Technique	Action/Process	Output	Goal
Tdis	Find dispatch routine	Binary code	Dynamic analysis	Extract execution traces to find routine invoked more frequently	Dispatch routine location	P1,P2
T0	Find IHs statically	Binary code	Static analysis tools	Process of identifying the location of IHs using static analysis	IH locations	P3
T1	Find IHs dynamically	Binary code	Dynamic analysis tools	Process of identifying the location of IHs using dynamic analysis: traces the actions performed by the VM calls	IH locations	P4
T2	Decrypt IHs	Binary code and encrypted IHs	static analysis tools (findcrypt,...)	Perform decryption routine in the binary	IH code	P4
T3	Dummy transition			If IHs are not encrypted then skip to P4	Decrypted IHs	P4
T4	IH optimization	IHs obfuscated	Binary rewriter	Compiler's optimization and manual optimization	Code location, Binary code	P5

Table 4: Legend 1 of the PN for VM attack

T	Description	Input	Tool/Technique	Action/Process	Output	Goal
Tdec	Find the decoder routine	Binary code	Dynamic analysis tools	Search for decoder routine in binary	Decoder function location	P8
T5	Build custom disassembler	Decoder location		Build custom disassembler	Custom disassembler created	P9
Tvm	Find VM function location	Binary code	Dynamic analysis	Dynamic analysis to find the VM function	VM function location	P6
T6	Find bytecode chunks	Binary code	Dynamic analysis	Dynamic analysis to find code chunks	Bytecode chunks location identified	P7
T7	Understand bytecode semantics	Code chunks, disassembler	manual	Build the bytecode control flow by running the code chunks through the disassembler	Understanding of bytecode semantics	P10
T8	Combine bytecode semantics and IHs	Code chunks,His locations	manual	And full semantics by combining bytecode with corresponding IHs	Understanding of full semantics	P11
T9	Rebuild and link	Binary code + optimized code chunks	Binary rewriter	Optimize binary and link	Original native code restored	P12

Table 5: Legend 2 of the PN for VM attack

10 Updates to the UML models

Section authors:
Cataldo Basile (POLITO)

The metrics framework proposed in Part I of this deliverable is an advancement of the project knowledge in the field of metrics. Metrics will be used for evaluating the potency, resilience, and stealth of applications after the application of a combination of protections. That use would not require an update to the ASPIRE Security Model as it is an operation performed after the deployment of protections. However, metrics will be also used to estimate the impact of combinations of protections during the Evaluation phase of the ADSS. Therefore, the ASPIRE Security Model needs to be updated to describe this new information. In particular, changes are all focused in the Metrics sub-model. Note that this update to the ASPIRE Security Model corresponds to step 5 of the "Roadmap for the Knowledge Base Definition", presented in Section 3.2 of the deliverable D4.1.

There are three major updates to the Metrics sub-model:

- We updated the Metrics sub-model to represent the metrics framework presented in Section 3.1;
- We modelled the measurable features listed and commented in Section 4;
- We modelled the metrics listed and commented in Section 5.

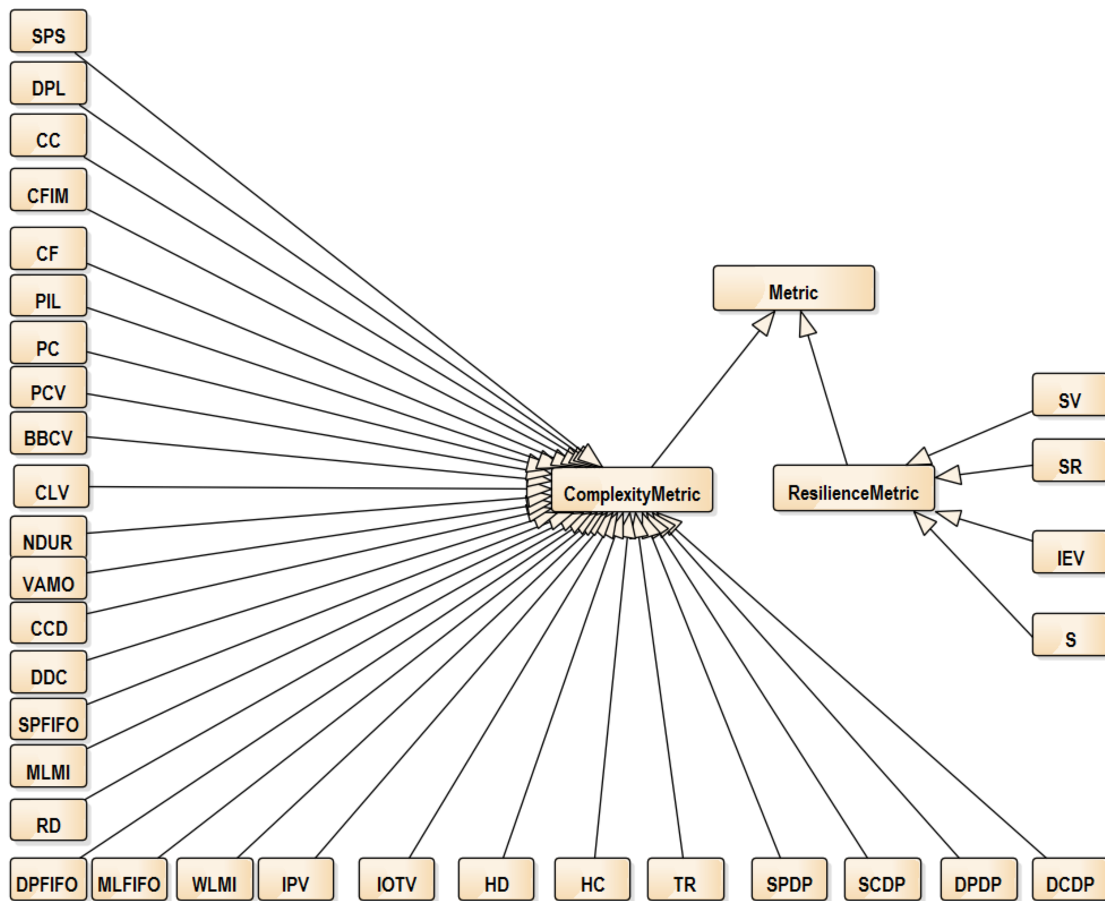

Figure 5: The `Metric` and its subclasses.

Figure 5 presents the `Metrics` and its subclasses. Metrics are described by means of `Metric` class instances. The `Metric` class has been subclassed according to the two categories of metrics: metrics related to complexity are described by the abstract `ComplexityMetric` class, and metrics related to resilience are described by the abstract `ResilienceMetric` class. These two abstract classes have been further subclassed to represent all the ASPIRE metrics. Please refer to Section 5 and the Abbreviation List for the meaning of all the acronyms used to name the subclasses.

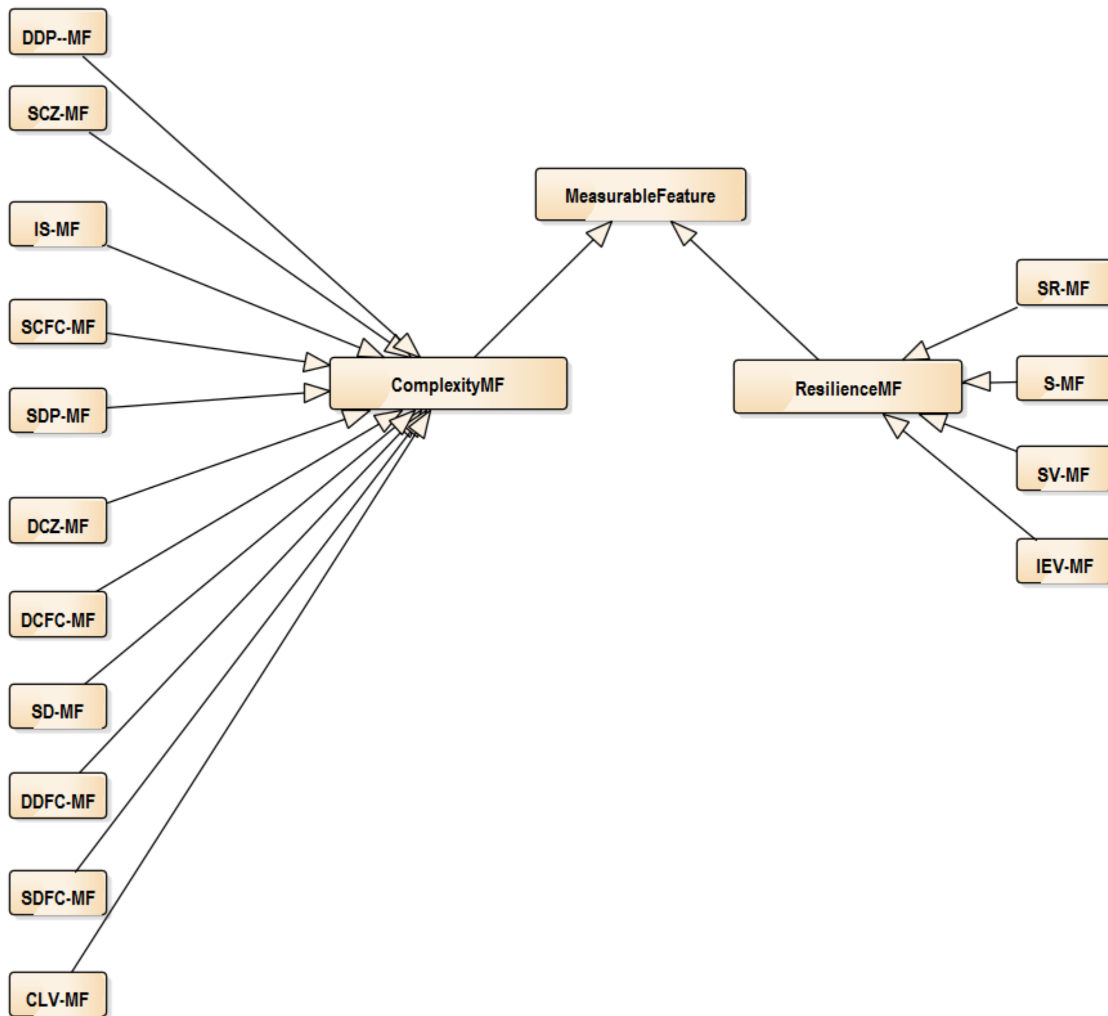

Figure 6: The `MeasurableFeature` and its subclasses.

Figure 6 presents the `MeasurableFeatures` and its subclasses. The measurable features are described by means of `MeasurableFeature` class instances. The `MeasurableFeatures` class has been subclassed according to the two categories of measurable features: measurable features related to complexity are described by the abstract `ComplexityMF` class, and measurable features related to resilience are described by the abstract `ResilienceMF` class. These two abstract classes have been further subclassed to represent all the ASPIRE measurable features. Please refer to Section 4 and the Abbreviation List for the meaning of all the measurable features acronyms.

Figure 7 shows the Metrics sub-model. The four ASPIRE goals shown in Figure 1 have been modelled by means of the `ASPIREMetricsGoal` enumeration class. Also the three conceptual aspects, i.e., potency, resilience, and stealth, have been modelled by means of the `ConceptualAspect` enumeration class.

The elements of the set $\mathcal{G}$ have been modelled as instances of the `GoalSetItem` class. Instances of this class are linked to instances of:

- the `ASPIREMetricsGoal` class by means of the `itemHasMetricGoal` association (**METRIC GOAL**) ;
- the `Goal` class by means of the `itemHasAttackGoal` association (**ATTACK GOAL**);
- the `Attacker` class by means of the `itemHasAttackGoal` association (**SUBJECT**);
- the `AttackTool` class by means of the `itemHasMeans` (**MEANS**);
- the `ApplicationPart` class by means of the `itemHasObject` association (**OBJECT**);
- the `AttackPath` class by means of the `itemHasActions` association (**ACTIONS**).

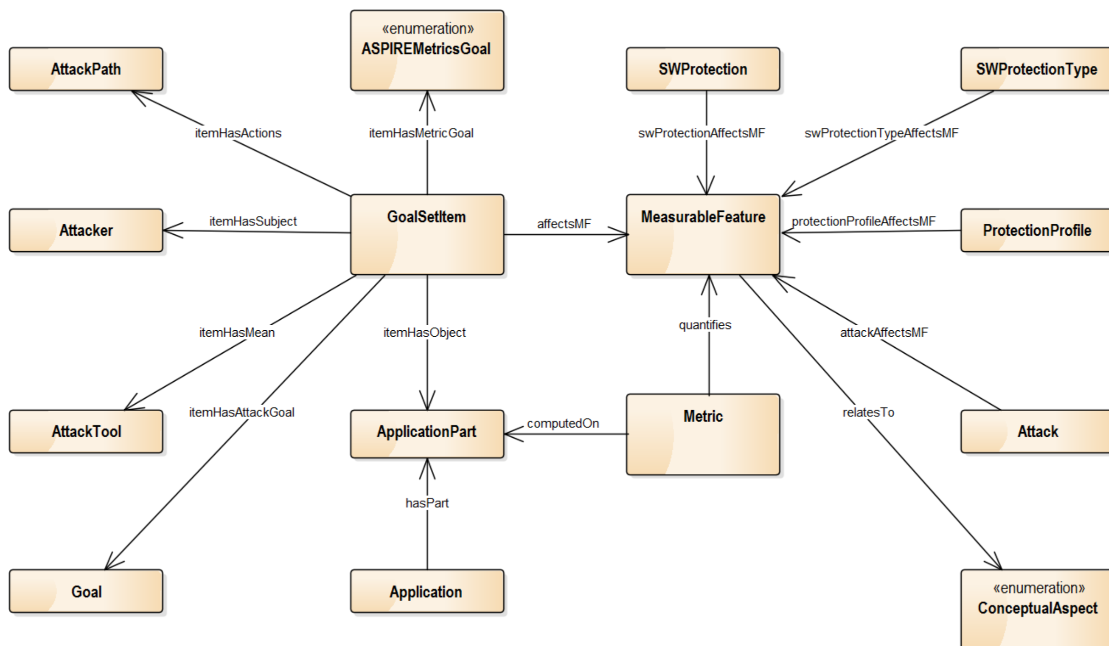

Figure 7: The Metrics sub-model of the ASPIRE Security Model.

All the `itemHas*` associations from the `GoalSetItem` have one-to-many cardinality.
Measurable features are associated with metrics that concretely serve to measure them by means of the `quantifies` association between the `Metric` and `MeasurableFeature` class instances.
To record into the a priori knowledge of the AKB the relations between measurable features and other elements of the Security Model that can affect them, we also introduced the following associations:

- `affectsMF` records that an instance of the `GoalSetItem` class can affect an instance of the `MeasurableFeature` class;
- `attackAffectsMF` records that an instance of the `Attack` class can affect an instance of the `MeasurableFeature` class;
- `protectionTypeAffectsMF` records that an instance of the `ProtectionType` class can affect an instance of the `MeasurableFeature` class;
- `swProtectionTypeAffectsMF` records that an instance of the `SWProtectionType` class can affect an instance of the `MeasurableFeature` class;
- `protectionProfileAffectsMF` records that an instance of the `ProtectionProfile` class can affect an instance of the `MeasurableFeature` class.

Finally, to describe that a measurable feature is related to one of the conceptual aspects, we introduced the `relatesTo` association between instances of the classes `MeasurableFeature` and `ConceptualAspect`.
The associations from and to the `MeasurableFeature` class have many-to-many cardinality.

List of abbreviations

ADSS ASPIRE Decision Support System

ACTC ASPIRE Compiler Tool Chain

AT Attack

BBCV Basic Block Coverage Variability

CCD Calling Convention Disruption

CC Cyclomatic Complexity

CD Control Dependency

CFIM Control Flow Indirection Metric

CF Confusion Factor

CG Call Graph

CLV Code Layout Variability

CPN Coloured Petri Net

DCDP Dynamic Ciphered Data Presence

DCD Delta Control Dependencies

DCFC Dynamic Control Flow Complexity

DCG Delta Call Graph

DCZ Dynamic Code Size

DDD Delta Data Dependencies

DDFC Dynamic Data Flow Complexity

DDP Static Data Presence

DD Data Dependency

DET Delta Execution Time

DMS Delta Memory Size

DPDP Dynamic Plain Data Presence

DPFIFO Dynamic Procedural Fan-In/Fan-Out

DPL Dynamic Program Length

DPT Delta Pointers

DSC Data Structure Complexity

DST Delta Statements

FIFO First In First Out

GQM Goal, Questions, Metrics

HC Heap Complexity

HD Heap Dynamics

IEV Intra-Execution Variability

IOTV Instruction Operand Type Variation

IOV Instruction Operand Variation

IS Ill-structuredness

MATE Man in the End

MCFIFO Memory Locations Fan-In/Fan-Out

MF Measurable Feature

MLMI Multi-Location Memory Instructions

NDUR Number of Def-Use Relationships

PCV Path Coverage Variability

PC Path Coverage

PDG Program Dependency Graph

PF Performance Feature

PIL Procedural Ill-structuredness

PNDV Petri Net with Discrete Variables

PN Petri Net

PT Protection

RD Reuse Distance

SCDP Static Ciphered Data Presence

SCGC Staic Control Flow Complexity

SCZ Static Code Size

SDFC Static Data Flow Complexity

SDP Dynamic Data Presence

SD Semantic Dependencies

SPDP Static Plain Data Presence

SPFIFO Static Procedural Fan-In/Fan-Out

SPZ Static Program Size

SR Semantic Relevance

SV Static Variability

S Stealth

TR Trace Reproducibility

VAMO Variable-Address Memory Operations

VM Virtual Machine

WBC White Box Cryptography

WLMI Writable Location Memory Instructions

References

[1] CPNtools 4.0. http://cpntools.org/. Accessed: 2014-10-15.

[2] EMF eclipse modeling framework. http://www.eclipse.org/emf/. Accessed: 2014-10-15.

[3] The EPNK Petri Net tool. http://www2.imm.dtu.dk/~ekki/projects/ePNK/. Accessed: 2014-10-15.

[4] GMF graphical modeling framework. http://www.eclipse.org/gmf/. Accessed: 2014-10-15.

[5] PIPE2 platform independent Petri Net editor 2. http://pipe2.sourceforge.net/. Accessed: 2014-10-15.

[6] PNML petri net markup language. http://www.pnml.org/. Accessed: 2014-10-15.

[7] YAWL yet another workflow language. http://www.yawlfoundation.org/. Accessed: 2014-10-15.

[8] Bertrand Anckaert, Matias Madou, Bjorn De Sutter, Bruno De Bus, Koen De Bosschere, and Bart Preneel. Program obfuscation: a quantitative approach. In *QoP '07: Proc. of the 2007 ACM Workshop on Quality of protection*, pages 15–20, New York, NY, USA, 2007. ACM.

[9] Andrea Arcuri. It really does matter how you normalize the branch distance in search-based software testing. *Softw. Test., Verif. Reliab.*, 23(2):119–147, 2013.

[10] B. Auprasert and Y. Limpiyakorn. Underlying cognitive complexity measure computation with combinatorial rules. *Proceedings of World Academy of Science: Engineering & Technolog*, 47:400–504, 2008.

[11] 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.

[12] B. T. Bennett and V. J. Kruskal. Lru stack processing. *IBM J. Res. Dev.*, 19(4):353–357, July 1975.

[13] Mariano Ceccato, Andrea Capiluppi, Paolo Falcarin, and Cornelia Boldyreff. A large study on the effect of code obfuscation on the quality of java code. *Empirical Software Engineering*, page (to appear), 2014.

[14] Mariano Ceccato, Massimiliano Di Penta, Paolo Falcarin, Filippo Ricca, Marco Torchiano, and Paolo Tonella. A family of experiments to assess the effectiveness and efficiency of source code obfuscation techniques. *Empirical Software Engineering*, 19:1040–1074, 2014.

[15] Mariano Ceccato, Massimiliano Di Penta, Jasvir Nagra, Paolo Falcarin, Filippo Ricca, Marco Torchiano, and Paolo Tonella. Towards experimental evaluation of code obfuscation techniques. In *Proceedings of the 4th ACM workshop on Quality of protection*, QoP '08, pages 39–46, New York, NY, USA, 2008. ACM.

[16] Mariano Ceccato, Massimiliano Di Penta, Jasvir Nagra, Paolo Falcarin, Filippo Ricca, Marco Torchiano, and Paolo Tonella. The effectiveness of source code obfuscation: An experimental assessment. In *IEEE 17th International Conference on Program Comprehension (ICPC)*, pages 178–187, may 2009.

[17] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical Report 148, Dept. of Computer Science, The Univ. of Auckland, 1997.

[18] 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.

[19] Christian S. Collberg and Clark Thomborson. Watermarking, tamper-proofing, and obfuscation: tools for software protection. *IEEE Trans. Softw. Eng.*, 28:735–746, August 2002.

[20] 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.

[21] J.S. Davis. Chunks: A basis for complexity measurement. *Information Processing & Management*, 20(1–2):119–127, 1984.

[22] Saumya Debray et al. Presentation at Dagstuhl Seminar 14241 - Challenges in Analysing Executables: Scalability, Self-Modifying Code and Synergy., June 2014.

[23] 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.

[24] Nicholas J Dingle, William J Knottenbelt, and Tamas Suto. PIPE2: a tool for the performance evaluation of generalised stochastic Petri Nets. *ACM SIGMETRICS Performance Evaluation Review*, 36(4):34–39, 2009.

[25] Gerard Florin and Stéphane Natkin. Evaluation based upon stochastic petri nets of the maximum throughput of a full duplex protocol. In *Application and Theory of Petri Nets*, pages 280–288. Springer, 1982.

[26] Hideaki Goto, Masahiro Mambo, Kenjiro Matsumura, and Hiroki Shizuya. An approach to the objective and quantitative evaluation of tamper-resistant software. In *Third Int. Workshop on Information Security (ISW2000)*, pages 82–96. Springer, 2000.

[27] Maurice H. Halstead. *Elements of Software Science (Operating and Programming Systems Series)*. Elsevier Science Inc., New York, NY, USA, 1977.

[28] Warren A. Harrison and Kenneth I. Magel. A complexity measure based on nesting level. *SIGPLAN Not.*, 16(3):63–74, March 1981.

[29] Kelly Heffner and Christian Collberg. The obfuscation executive. In *Information Security*, pages 428–440. Springer, 2004.

[30] S. Henry and D. Kafura. Software structure metrics based on information flow. *Software Engineering, IEEE Transactions on*, SE-7(5):510–518, Sept 1981.

[31] Mariusz H Jakubowski, Chit Wei Saw, and Ramarathnam Venkatesan. Iterated transformations and quantitative metrics for software protection. In *SECRYPT*, pages 359–368, 2009.

[32] Jonas Finnemann Jensen, Thomas Nielsen, and Lars Kærlund Østergaard. Petri nets with discrete variables. *Dept. Comput. Sci., Aalborg Univ., Aalborg, Denmark, Tech. Rep. Thesis Rep*, 2012.

[33] Kurt Jensen. *Coloured petri nets*. Springer, 1987.

[34] Matthew Karnick, Jeffrey MacBride, Sean McGinnis, Ying Tang, and Ravi Ramachandran. A qualitative analysis of java obfuscation. In *Proceedings of 10th IASTED International Conference on Software Engineering and Applications, Dallas TX, USA*, 2006.

[35] 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.

[36] Ekkart Kindler. The epnk: an extensible petri net tool for pnml. In *Applications and Theory of Petri Nets*, pages 318–327. Springer Berlin Heidelberg, 2011.

[37] Cullen Linn and Saumya Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM conference on Computer and communications security*, CCS '03, pages 290–299, New York, NY, USA, 2003. ACM.

[38] M Ajmone Marsan. Stochastic petri nets: an elementary introduction. In *Advances in Petri Nets 1989*, pages 1–29. Springer, 1990.

[39] 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.

[40] John C. Munson and Taghi M. Khoshgoftaar. Measurement of data structure complexity. *J. Syst. Softw.*, 20(3):217–225, March 1993.

[41] 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.

[42] 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.

[43] 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.

[44] Franz Stetter. A measure of program complexity. *Comput. Lang.*, 9(3-4):203–208, December 1984.

[45] Iain Sutherland, George E. Kalb, Andrew Blyth, and Gaius Mulley. An empirical examination of the reverse engineering process for binary files. *Computers & Security*, 25(3):221–228, 2006.

[46] H. Tamada, K. Fukuda, and T. Yoshioka. Program incomprehensibility evaluation for obfuscation methods with queue-based mental simulation model. In *Software Engineering, Artificial Intelligence, Networking and Parallel Distributed Computing (SNPD), 2012 13th ACIS International Conference on*, pages 498–503, Aug 2012.

[47] Sharath K. Udupa, Saumya K. Debray, and Matias Madou. Deobfuscation: Reverse engineering obfuscated code. In *Proceedings of the 12th Working Conference on Reverse Engineering*, pages 45–54, Washington, DC, USA, 2005. IEEE Computer Society.

[48] Corrado Aaron Visaggio, Giuseppe Antonio Pagin, and Gerardo Canfora. An empirical study of metric-based methods to detect obfuscated code. *International Journal of Security & Its Applications*, 7(2), 2013.

[49] Huaijun Wang, Dingyi Fang, Ni Wang, Zhanyong Tang, Feng Chen, and Yuanxiang Gu. Method to evaluate software protection based on attack modeling. In *IEEE 10th International Conference on High Performance Computing and Communications 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC)*, pages 837–844, Nov 2013.

[50] Jiacun Wang. *Timed Petri nets: Theory and application*, volume 39. Kluwer Academic Publishers Dordrecht, 1998.

[51] Yingxu Wang and Jingqiu Shao. Measurement of the cognitive functional complexity of software. In *Proceedings of the 2Nd IEEE International Conference on Cognitive Informatics*, ICCI '03, pages 67–, Washington, DC, USA, 2003. IEEE Computer Society.

[52] Ying Zeng, Fenlin Liu, Xiangyang Luo, and Chunfang Yang. Software watermarking through obfuscated interpretation: Implementation nad analysis. *Journal of Multimedia*, 6(4):329–339, 2011.