The Journal of Systems and Software 150 (2019) 3–21

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

A meta-model for software protections and reverse engineering attacks

Cataldo Basile[a], Daniele Canavese[a], Leonardo Regano[a], Paolo Falcarin[b,*], Bjorn De Sutter[c]

[a] Dipartimento di Automatica e Informatica, Politecnico di Torino, Italy
[b] Department of Computing and Engineering, University of East London, London, E16 2RD, United Kingdom
[c] Computer Systems Lab, Department of Electronics and Information Systems, Ghent University, Belgium

ARTICLE INFO

Article history:
Received 15 April 2018
Revised 9 October 2018
Accepted 21 December 2018
Available online 22 December 2018

Keywords:
Software protection
Security knowledge base
Decision support
Attack modelling
Reverse engineering
Meta-model

ABSTRACT

Software protection techniques are used to protect valuable software assets against man-at-the-end attacks. Those attacks include reverse engineering to steal confidential assets, and tampering to break the software's integrity in unauthorized ways. While their ultimate aims are the original assets, attackers also target the protections along their attack path. To allow both humans and tools to reason about the strength of available protections (and combinations thereof) against potential attacks on concrete applications and their assets, i.e., to assess the true strength of layered protections, all relevant and available knowledge on the relations between the relevant aspects of protections, attacks, applications, and assets need to be collected, structured, and formalized. This paper presents a software protection meta-model that can be instantiated to construct a formal knowledge base that holds precisely that information. The presented meta-model is validated against existing models and taxonomies in the domain of software protection, and by means of prototype tools that we developed to help non-modelling-expert software defenders with populating a knowledge base and with extracting and inferring practically useful information from it. All discussed tools are available as open source, and we evaluate their use as part of a software protection work flow on an open source application and industrial use cases.

1. Introduction

In so-called man-at-the-end (MATE) software attacks, attackers target assets embedded in software. By means of reverse engineering they try to steal confidential information, such as embedded cryptographic keys or intellectual property in the form of algorithms (Falcarin et al., 2011). They also use reverse engineering techniques as a preparatory step towards tampering with the software to break its integrity, e.g., to break license checks. MATE attackers can mount sophisticated attacks, as they can tamper with software and data in their labs, where they have all kinds of software aids, such as debuggers, tracers, emulators, and customized operating systems; and hardware aids such as developer boards with (JTAG-based) hardware debuggers. The latest BSA Global Software Piracy Study[1] states that 39% of software installed on computers worldwide is not licensed, amounting to $52 billion in losses; in particular, 98% of mobile apps lack binary code protection and they can be easily reverse engineered and tampered with[2].

Software protection techniques transform code and inject new code to hamper reverse engineering and tampering. Perfect protection being impossible (Barak et al., 2001), the techniques aim to raise the cost for attackers and the time needed to perform the MATE attacks. When they attack protected software, attackers not only target the original assets in the code, but also the protections themselves. To undo, overcome, bypass, and work around them, they reverse engineer the protections and they tamper with them. From the perspective of the defender, the protections become assets as well.

Recently, some critical steps have been set in modelling the behaviour of MATE attackers (Ceccato et al., 2017; 2018), including how they reason about code under attack, about protections they encounter, about assets they target, and about attack steps they conduct. There also exist formal models such as attack graphs (Sheyner et al., 2002) and Petri Nets to model concrete attack

* Corresponding author.
E-mail addresses: cataldo.basile@polito.it (C. Basile), daniele.canavese@polito.it (D. Canavese), leonardo.regano@polito.it (L. Regano), falcarin@uel.ac.uk (P. Falcarin), bjorn.desutter@ugent.be (B.D. Sutter).

[1] BSA Global Software Piracy Survey: http://globalstudy.bsa.org/2016/.

[2] State of Application Security: https://www.arxan.com/resources/state-of-application-security/.

https://doi.org/10.1016/j.jss.2018.12.025
0164-1212/

paths on concrete assets (Wang et al., 2013b; Chen et al., 2011). We are still lacking overall models, however, that allow the representation of the relevant relations between (i) assets, (ii) the software those assets are embedded in, (iii) deployed protections, (iv) individual attack steps and tools and methods to perform attacks on those protections and on the assets, (v) possible paths of attack that start from scratch and through which attackers can reach their ultimate reverse-engineering end goal, i.e., stealing the original asset. Moreover, these models must support an inferential system that allows reasoning about the data represented with the model. That is, we need to build a Knowledge Base (KB), which is essential to perform a complete risk analysis of software applications and to decide on protections to mitigate reverse-engineering and tampering risks.

In related fields, such as network security, models exist that are consistently used in practice to assess the overall level of protection. Some models, like the Common Vulnerability Scoring System (CVSS) (Mell et al., 2007), Common Weaknesses Enumeration (CWE)[3], and Common Vulnerability Exposure (CVE) (Mann and Christey, 1999), are used to model software weaknesses and vulnerabilities, attacks against them (exploit), as well as estimation of likelihood, expertise needed by hackers, and ease of replication on a large scale (severity). Others depict the landscape where attacks may exploit vulnerabilities, like Common Configuration Enumeration (CCE).[4] These models are used in a more complex ecosystem where defenders can perform their assessment, implement their mitigations, and have a precise snapshot of the system to protect with ad hoc tool and language support, e.g., with the Open Vulnerability and Assessment Language (OVAL) (Wojcik et al., 2003) or Security Content Automation Protocol (SCAP) (Radack and Kuhn, 2011) and Open Checklist Interactive Language (OCIL) (Waltermire et al., 2011). Researchers have also defined KBs, as OWL ontologies, that represent relations between network vulnerabilities and attacks and have been used to assess attacks against corporate assets. Moreover, there are tools that can automate attack replication, like Metasploit. Clearly, the complexity of the network security scenario is limited compared to the target of this paper, i.e., the MATE context, where building the landscape also requires to dig into the semantics and internals of the software to protect.

In the domain of software protection, we also desperately need such models and similar levels of standardization to collect knowledge. The reason is simple: deploying protections is highly complex. All protections come with overhead in different forms (bandwidth, throughput, size, performance, real-time behavior, ...) and with different levels of expected effectiveness. All protections affect the software development life cycle in different ways (debugging capabilities, testing needs, integration issues, updatability, ...). Furthermore, multiple assets with different security requirements often need to be protected within the same application. Finally, combining multiple protections as needed to build a layered defence against all possible attacks, is hampered by both fundamental and practical composability issues. As a result, defenders face the difficult task of selecting the best combinations of protections to protect their software assets.

To help them in making the right decisions, a modelling framework, an inferential system, and a toolbox (i.e., a KB) are necessary that cover all of the relations between software, assets, protections, and MATE attacks. Such a KB has wider value however. It can also help researchers to identify those areas where more research is needed because satisfactory protections are still missing, and they can help software architects to identify which types of assets and related security requirements are safe in certain deployment scenarios, and which are not. Risk assessment and methodologies for evaluating protection strength are other interesting research areas that can benefit from the definition of appropriate meta-models. An additional long term objective would be developing an automatic decision support system for software protection, i.e., an expert system that may help software developers with limited expertise in software protection to select the best way to protect the assets embedded in their applications without the need of a team devoted to this task. A KB represents facts about a particular domain (e.g. software protection and reverse engineering), while an expert system can reason about those facts and use rules and other forms of logic to deduce new facts or spot inconsistencies. An expert system requires structured data, not just tables with numbers and strings, but references to other objects. A KB stores complex structured information and the ideal representation for a KB is an object model or an ontology as a graph linking classes, subclasses and object instances. A meta-model (similarly to a grammar for a programming language) describes the structure of such model, formally defining its syntax and rules. This paper presents a new meta-model in support of a formalized KB on the aforementioned relations. Moreover, it describes an inferential system that builds on the meta-model to perform the risk analysis of software applications.

The research into this meta-model and related tools was carried out in the European FP7 research project ASPIRE.[5] This modelling research was done in conjunction with the design and execution of multiple penetration test experiments and with proof-of-concept research into tool-supported, composable, multi-layered protection of multiple industrial use cases, i.e., on software and assets of real-world complexity. Moreover, ASPIRE researched an evaluation methodology for software protection strength, including a metrics framework, and decision support to automate the selection of the best combinations of software protections. Consequently, a major strength of the presented meta-model and the corresponding tools is that they have been evaluated and to a large extent validated in the context of all those other research activities.

This paper's main contributions are the following: (i) A discussion of the requirements of a MATE software protection KB and a corresponding meta-model in Section 2; (ii) the presentation of the meta-model that can be instantiated to populate a KB that captures the necessary information to reason about and assess MATE software protections in Section 3; (iii) a validation of that meta-model against existing models and taxonomies from the literature in Section 4.1; (iv) a validation of the meta-model by means of tools that demonstrate its practical applicability on concrete use cases in Section 4.2. After those contributions, we discuss related work in Section 5 and draw conclusions and discuss future work in Section 6.

2. Requirements and scope

As already motivated in the introduction, we need a way to collect and represent knowledge regarding MATE attacks and protections, i.e, to model the relations between the already mentioned aspects of assets, applications, protections, and attacks. More precisely, we put forward the following requirements.

Requirement R1: We need a way to capture the application structure. This is important for two reasons. First, to relate attacks and protections to real software parts (e.g., functions and variables) and not entire applications or whole components, allowing a finer granularity and in turn more expressiveness. Second, it is important to have a way to model not only the various application parts,

[3] See http://cwe.mitre.org.

[4] See https://cce.mitre.org/.

[5] Advanced Software Protection: Integration, Research, and Exploitation - https://www.aspire-fp7.eu.

but also their relationships (e.g., variable `x` is contained in function `y` or function `y` calls function `z`) and possibly abstract interpretations involving these parts (e.g., control flow graphs). This is key to build a KB system where to perform meaningful inferences about the security of an application, not in the least because attackers typically dedicate quite some effort reverse engineering the relation between application components and zooming in on the most relevant components in the application under attack (Ceccato et al., 2018).

Requirement R2: We need a way to formalize the concept of assets. *Original assets* are the application parts that have value for the developer, so it is important to represent them in the most precise fashion. In addition, it is important to define which asset security properties the defender should try to protect and, vice-versa, the attacker will try to breach. In this regard, not only the original assets hold value. So do artifacts in the application code that help attackers to execute complete attack strategies, such as hooks where they can attach their tools, and code patterns they can easily identify and that lead them towards the original assets, i.e., that allow them to zoom in the most relevant parts of the software. We call such artifacts *intermediate assets*. Furthermore, we consider *protection assets*, which are artifacts of the deployed protections that either allow the identification of the deployed protections (thus allowing the attacker to pick a strategy to defeat the protection) and protection artifacts that can become the target of individual attack steps to defeat the protections. Unlike original assets, which come with security requirements defined by the application developer, intermediate and protection assets become assets only because they are the target of attack steps. Reasoning about original assets and automatically inferring intermediate and protection assets would be an important task to demand to a KB system.

Requirement R3: We need a way to formally describe attacks and relate them to the various application parts. Even if there are some efforts in the security world to represent attacks (e.g., CWE, CVE), we are still lacking a formal way to represent a complete attack in an unambiguous way in a MATE scenario. This is important, especially since we are interested in building a KB system whose inferences allow performing various kinds of automated security analyses that need to cover all possible attack paths in order to be truly useful.

Requirement R4: We need a way to formally describe the effects of protections when applied to code and data. A protection can be abstracted as a specialized tool that transforms code or injects new code for the purpose of hampering attacks on (original, intermediate, or protection) assets. Several protection tools are available on the market, each one offering support for specific versions of multiple protection techniques, configurable in various ways. Therefore, we are interested in modelling protections both from the perspective of the attacks they prevent or delay, and with respect to the performance overhead and software development life cycle impact they have. Modelling what a protection effectively does is vital for a KB system able to perform automatic inferences for a protection assessment of a software application. Moreover, a KB system to suggest mitigations strongly relies on the ability of the meta-model to represent relationships among protections. On the one hand, a defender may be interested in knowing when protections cannot be applied on the same piece of code, or on the same applications. On the other hand, he could be interested in knowing when applied protections strengthen each other either because one renders defeating another protection more complex or because they work in synergy against the same attack.

Requirement R5: We need a way to formally describe a protected application. Assessing the security of protected applications requires to model with precision where and how the protections are deployed. Since the application of protections can be configured and tuned based on a set of configuration parameters (e.g., opaque predicates of different levels of complexity can be inserted into the code at different frequencies based on an integer parameter) we need a model that is expressive enough to capture all the possible ways protections may be applied on each application part. Note that, while R4 concerns reporting how protections relate to each other and how they change an application, R5 is about describing how protections are applied to application parts. A KB system that can reason on this information, can be used to predict the effect of protections when applied on specific application parts, both in terms of overheads and reached protection strength. Clearly, this is a necessary step to build an expert system that can assist defenders when they have to select the best way to protect the assets in their applications.

Requirement R6: Besides the qualitative relations that we need to model between protections and attacks, it is also useful to model the relation quantitatively where possible. This can facilitate more accurate evaluations of the strength of protections given an application, its assets, and potential attacks. In literature, many metrics to measure that strength have already been proposed. For example, software complexity metrics have been proposed to quantify the potency of obfuscations, i.e., the extent with which obfuscations make manual code comprehension tasks and automatic de-obfuscation techniques harder (Collberg et al., 1997; 1998; Schrittwieser et al., 2016). Other authors have proposed combining many different metrics for assessing the strength of a wider range of protections (Anckaert et al., 2007; Tonella et al., 2014; Ceccato, 2016) or have evaluated which metrics are better predictors of obfuscation quality (Ceccato et al., 2015).

Requirement R7: We need a way to help users populate the KB, without requiring them to be modelling experts, i.e., by using visual model editors or textual data that can be easily translated and imported. Similarly we need a way to help users in extending the model by importing new information from different sources into a unique format.

In these requirements, we observe the need to model three forms of information:

- *Generic a priori information* describes features of and relations between aspects that holds invariably for a defender, such as the available attack tools and protection tools and their capabilities, which do not depend on the exact software to be protected;
- *A priori use case information* describes features of and relations between the application, its assets, and their security requirements;
- *A posteriori information* describes features of and relations between a concrete application and its assets, applicable protections, and possible attacks on protected and unprotected versions. This is mostly information that can be inferred from the two other forms of information.

The scope of our model is currently limited to software-only protections. We exclude protections that depend on advanced hardware security features such as Intel's enclaves (Intel, 2014) or TrustZone[6] or Sancus or SOFIA-like cryptography-based enforcement of integrity and confidentiality (Noorman et al., 2013; de Clercq et al., 2016). Furthermore, the attacks we envision are limited to man-at-the-end attacks. Man-in-the-middle attacks, which focus on attacking distributed systems by intercepting communications and by tampering with the communications, are excluded. And so are system penetration attacks. In MATE scenarios, attackers are assumed to have, in their own lab, all the access they

[6] https://www.arm.com/products/silicon-ip-security.

want to the software under attack, they do not need to penetrate systems in order to get that access.

Furthermore, the scope of our work is limited to native software, i.e., code that is distributed to end users and attackers in the form of stripped executable binaries that can be either main executables of applications, or dynamically linked libraries. This excludes threats from insiders such as any developers with access to the source code or intermediate formats. It also excludes hardware descriptions in any source, intermediate, or binary format.

Finally, it is important to point out that the models, inferences, and tools we propose are created first and foremost to aid developers and users of software protections, i.e., the defenders. We approach the link between protections and attacks from the defender's perspective. Individual attackers approach an application and its assets one attack step at a time, and consider alternatives and directions for their next steps after each step, thus executing one sequence of attack steps, i.e., one attack path. Each attacker's path depends on his experience, his access to tools, the exact precision with which their specific tool versions and tool customizations analyse particular deployments of protections, and even sheer luck, such as when they decide to spend limited amount of time on searching for clues through an unordered set of information, and it hence depends on luck whether or not they bump onto the most relevant elements before their time runs out. Defenders cannot reason in terms of individual attack paths and luck, however. They instead have to make worst-case assumptions, including the assumption that multiple attackers may be attempting multiple different attack paths at any point in time, and the assumption that all potentially successful attack steps will actually be successful. In other words, they have to assume that all potentially successful attack paths will be attempted in parallel. Our models reflect this worst-case scenario. For example, they do not contain the notion of failed attack steps. However, not all companies and developers may want or be able, to protect their application against all the possible attacks. They may lack access to the most powerful protection tools because of their cost and expertise needed to use them successfully, or their applications maybe cannot suffer from the performance overhead that invariably comes with stronger protections. Also the application domain matters. While it is reasonable to protect software for critical infrastructure also against sophisticated attacks mounted by very motivated and skilled attackers, several applications (e.g., low-cost games for smartphones) just need to be protected against automated attacks launched by script kiddies. We hence differentiate between different levels of attacker expertise, but in our worst-case analysis, we assume that all the attacks that can be mounted by attackers with a certain expertise are all performed in parallel and successful.

3. Meta-model

We will now introduce our meta-model that, for the sake of readability, is split in four smaller meta-models:

- the *core meta-model* contains the most important classes and relationships, from our perspective;
- the *application meta-model* details the concepts and associations related to a generic application and its code;
- the *protection meta-model* describes the notions that link together the protections and the protected areas of an application;
- the *attack meta-model* finally introduces the attack classes and their relationships with the various application parts.

In the UML class diagrams shown in the rest of the paper, we have adopted a colour code to help the reader in understanding the effort to fill in the meta-model instance:

- Red classes with a double border represent generic a-priori concepts. Instances of these classes are populated by security experts when preparing the KB. It is not expected from defenders to change these instances when they have to protect their applications, unless they are experts in formal models and want to add new features (e.g., inferences, reasoning) that cannot be built with the data in the existing meta-model.
- Blue classes with a double border relate to the a-priori use case information. Instances of these classes are expected to be obtained from the application to protect. These data can be obtained automatically, e.g., name of functions and their relations, i.e., a call graph, can be obtained with static analysis tools, or manually, e.g., to report that a function or a piece of code discovered automatically is an asset and requires the enforcement of specific security requirements.
- The yellow classes with a single border model the a-posteriori knowledge. All a-posteriori data is, by definition, obtained automatically with the inferences performed in the KB system, thus defenders have not to care about their collection.

The reader may have noticed that the effort required by the defender to build the KB for protecting a specific application is limited to part of the a-priori use case information and associations.

3.1. The core meta-model

The core meta-model formalizes the relationships between the main concepts involved in assessing an application's vulnerabilities and protecting its valuable assets. Fig. 1 depicts the main meta-model's UML class diagram. It includes the classes to model the application itself, the assets that must be protected, the available software protections, the attacker, and the potential attacks on the assets' security requirements.

The main class is Application, whose instances abstract the applications or libraries that must be protected. An Application is a composition of one or more ApplicationPart instances, which represent functions, code regions (as defined by the developer, see Section 4.2.2 for more details), and global and local variables. An Asset is an ApplicationPart instance with a set of security requirements, such as confidentiality or integrity, targeted by an attacker and that must be enforced by means of some protection. All the Asset objects must then have at least one hasRequirement association with the SecurityRequirement enumeration, containing all the security requirements an asset can have.

The AttackTarget class represents a possible target of an attacker, who aims at breaking the security requirements of the assets, as explained before. In our meta-model each AttackTarget instance will be associated with one and only one Asset via the threatens association and with one and only one SecurityRequirement element via the affects relationships. If an attacker can target multiple asset requirements, then several AttackTarget on the same asset are instantiated.

Attacks can be typically subdivided in an ordered sequence of steps. For example, if the attacker wants to break the integrity of a function in the application, such as in a license check, he will need to disassemble/decompile the application's binary, find the license check function that forms the asset, and then modify it in order to break its integrity, being its security requirement. We model such basic steps via the class AttackStep. Instances of this class may have one or more hasTarget relationships with instances of the AttackTarget class. Note that some attack steps may not have any target, since they model some preparatory actions needed by the attacker to mount the following attack steps (e.g., attaching a debugger to the application before dynamically changing a function code).

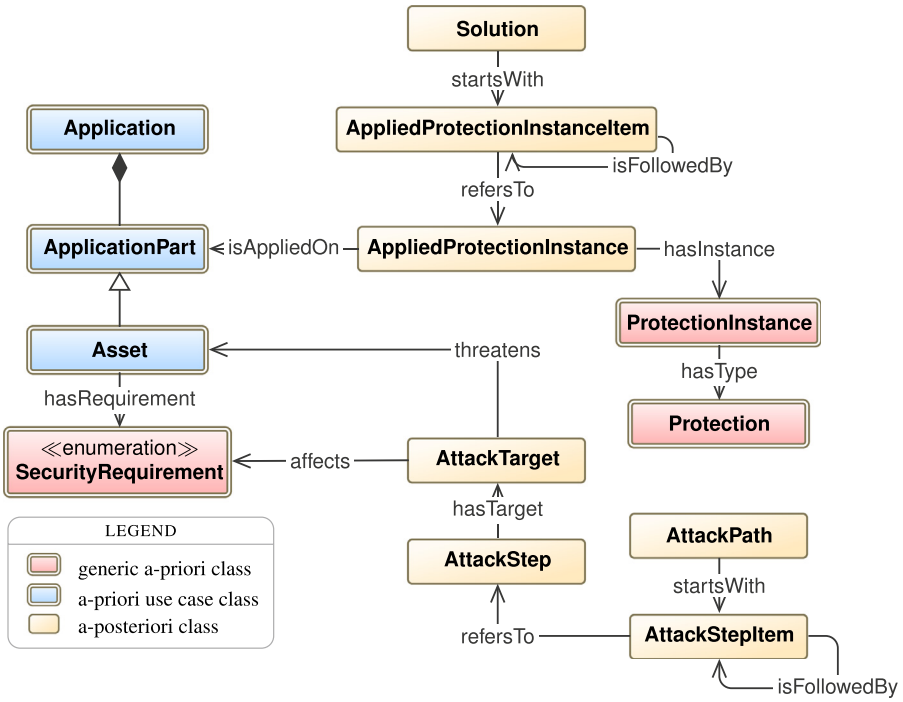


Fig. 1. Core meta-model UML class diagram.

Attacks are modelled via the AttackPath class, whose instances are ordered sequences of attack steps. Note that not only the last attack step will breach the security requirement of an asset. For instance, an attack step that threatens the confidentiality of a particular asset can lead to another step breaching the asset integrity. To formally enforce the attack step order, we introduced the class AttackStepItem, whose instances are associated with a single AttackStep object via the refersTo association and the next step in the attack path through the isFollowedBy association. Each AttackPath instance is related, through the startsWith relationship, to one AttackStepItem instance representing the starting point of the attack path.

Generic protections types are represented via the Protection class. A protection enforced with a specific tool and with a particular configuration is represented in the meta-model as an instance of the ProtectionInstance class. Every ProtectionInstance object has two importation relationships. The first one is represented via the hasType association that binds a ProtectionInstance object with its generic protection, that is a Protection instance. The second one is the isEnforcedWith association, used to relate a ProtectionInstance object with one or more ProtectionTool instances, modelling all the tools needed to actually deploy the protection. For instance, the control flow flattening obfuscation technique is represented as a Protection class instance (Wang et al., 2000).

To slow down an attacker, various protection instances must be applied to the assets in the application. Therefore, we introduced the AppliedProtectionInstance class, representing a protection instance applied to a generic application part. This association is directed towards the application part class, and not the asset concept, since a security expert can choose to protect also non-assets in order to confuse (and hence slow down) the attacker (see Section 4.2 for more information about this subject). Instances of the AppliedProtectionInstance class are bound via the hasInstance and isAppliedOn associations to a ProtectionInstance and ApplicationPart objects, respectively representing the protection instance and the application part where the former is deployed.

The global set of applied protection instances is represented with the Solution class. Different solutions in the same model, i.e., for the same application, are obviously possible. For example, different solutions may be devised to find the best trade-off between the level of security achieved and the introduced overhead. When applying more than one protection to the same asset, the order of application is important, since it could lead to different results in terms of security and even to incoherent cases. Therefore, we enforced an ordering between the applied protection instances in a solution by means of the AppliedProtectionInstanceItem class, representing an applied protection instance inside a solution. Every AppliedProtectionInstanceItem object is linked with a AppliedProtectionInstance object via the refersTo association. Each Solution instance will have an association startsWith with an AppliedProtectionInstanceItem instance to represent the first applied protection instance. The ordering in the solution is then enforced between the AppliedProtectionInstanceItem instances via the isFollowedBy relationship.

3.2. The application meta-model

The meta-model depicted in Fig. 2 defines the fundamental information about the application needed to protect its assets, in order to preserve the security requirements of the latter from the attacks mounted by the attacker, allowing us to satisfy the requirements R1 and R2.

The class used to model the various components of an application is ApplicationPart. Each application part has a name attribute and it is contained into a source file represented with an homonym class, specifying its location in a file system with the path element. All the ApplicationPart instances can be assets, code or data, represented by three distinct sub-classes.

The Datum sub-class represents a generic variable or function parameter. Each datum is characterized by its type (e.g., string, integer variable, cryptographic key or ciphertext), modelled by the

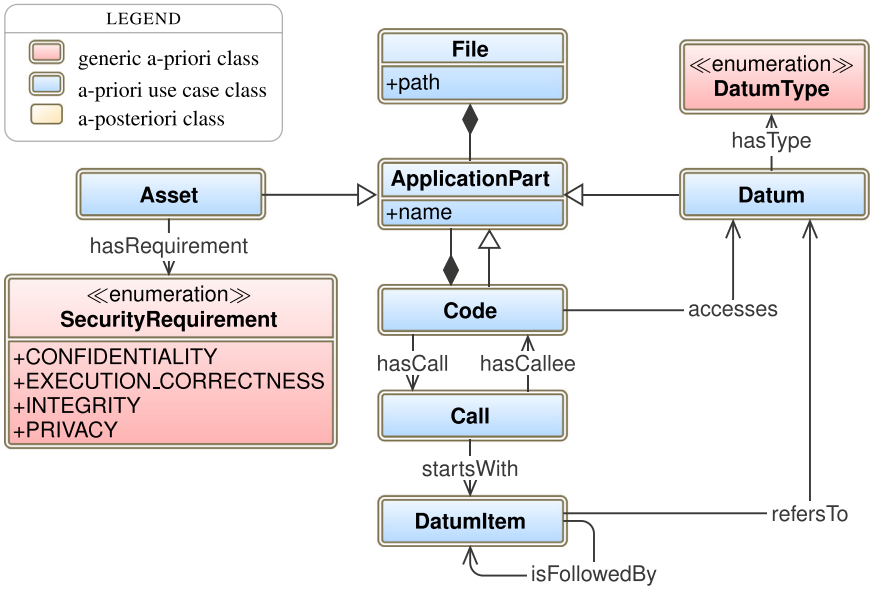


Fig. 2. Application meta-model UML class diagram.

DatumType class and hasType association. Knowing this information is useful for at least two reasons. First, data protections usually are only applicable to specific data types. For instance, in the prototype protection tools developed in the ASPIRE project, XOR masking (Collberg et al., 1997) can only be deployed to protect an integer variable or array of integers. Second, some attacks can only be mounted against some kind of data. For example, differential cryptanalysis (Biham and Shamir, 1993) is only meaningful for some kind of encrypted data.

The Code sub-class is used to model functions, class methods and any generic code region. A code region can be thought of as a container of other application parts (e.g., a function contains variables, but also other smaller code snippets) and this fact is modelled via the containment relationship between the ApplicationPart and Code classes. A piece of code can also access a (local or global) variable, fact represented by the accesses association. In addition, storing also the call graph of the application may prove useful, especially when inferring attacks. Each call to a function is modelled as an instance of the Call class. The caller code is bounded to the call via the hasCall 1-to-1 association, while the call is related to the callee with the hasCallee 1-to-1 association. Each call object contains also to the ordered list of parameters passed to the called function. A parameter in this ordered list is modelled via the DatumItem class, related to the correspondent Datum instance with the refersTo association and the next item via the isFollowedBy relationship. If the called function has at least one parameter, the Call instance will contain a startsWith association with a DatumItem instance modelling the first call parameter. Note that when it is relevant to consider multiple calling sites to the same callee in some caller function, this can be done by considering multiple ApplicationParts in the function, and by associating each of them to the callee with hasCallee.

As introduced in the core meta-model, assets are represented as instances of the Asset concept, a sub-class of ApplicationPart, and are related with their security requirement with the association hasRequirement to items of the SecurityRequirement enumeration. In this context, we limited the list of security requirements to the following values:

- *Confidentiality*, indicating that an asset should not be comprehensible for the attacker (e.g., patented algorithms) or that it should remain hidden completely (e.g., crypto key);
- *Execution correctness*, specifying that a code asset must be called and executed as intended, and should not be bypassed by the attacker (e.g., license checks) or be executable outside the context of the given application (e.g., a white-box crypto algorithm);
- *Integrity*, applicable to an asset that must not be modifiable by the attacker (e.g., a hard-coded PIN number);
- *Privacy*, suitable when the disclosure of an asset could lead to personal data leakage (e.g., credit card numbers).

Note that the meta-model does not restrict the usage of these requirements, but allows the security expert to add additional ones, if needed. Also note that although these requirements can never be met completely (as full protection against MATE attacks is impossible as explained in the introduction), it is useful to express them because the aim of the protections is to delay the attackers that aim for violating the requirements. Thus the expected attacks follow in part from these requirements.

Since filling this meta-model with meaningful instances can be a long process, especially for big applications, we developed some tools to perform this action automatically (see Section 4.2.2).

3.3. The protection meta-model

The protection meta-model, depicted in Fig. 3, contains the classes and relationships related to the protections that can be used to protect the security requirements of the assets against the actions performed by the attacker. This meta-model allows to model not only the protection relationships (requirement R4), but can be also used to precisely describe how an application was protected (requirement R5).

The Protection class is associated with SecurityRequirements values by means of the enforces association. This association characterises the abilities and purposes of applying a given protection. Furthermore, the Protection class has several association loops that are useful to model protection synergies and forbidden precedences. In particular, the shouldBePrecededBy and shouldNotBePrecededBy associations are respectively used to specify that an applied protection instance should or should not be preceded by another applied protection instance of a given kind. This is useful when choosing the best solution since one protection can make another, previously applied protection stronger (e.g., software re-

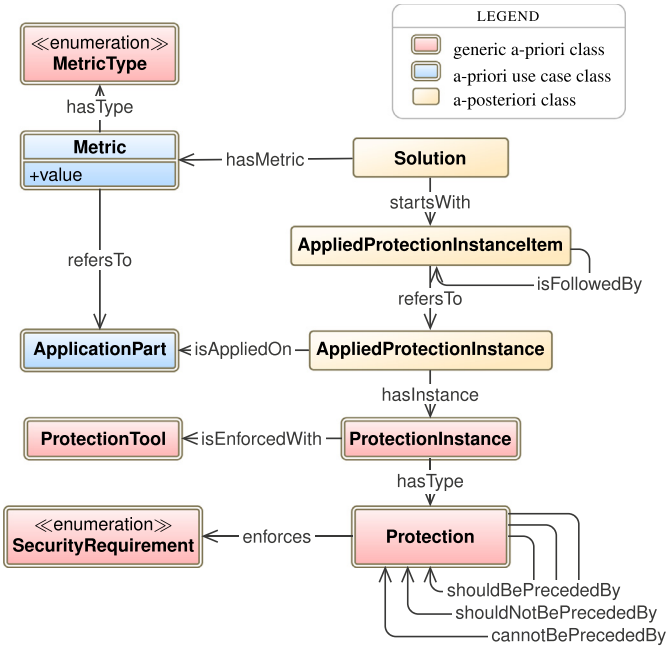


Fig. 3. Protection meta-model UML class diagram.

mote attestation (Viticchié et al., 2018) can be made more robust if coupled with anti-debugging (Abrath et al., 2016)), but applying one protection can also make a later one weaker (e.g., a control flow obfuscation applied first can negatively impact the data flow analysis that checks preconditions for applying a data obfuscation), thus affecting the aggressiveness with which the data obfuscation can be applied. Furthermore, the cannotBePrecededBy relationship is used to model impossible sequences of protections that can lead to incoherent or non-compilable applications (e.g., software remote attestation is usually the last protection to be put, since altering the code after its deployment will trigger an invalid attestation).

The ProtectionTool class contains all the available tools that can be used to deploy a protection on an asset or application part. The supported protection instances are linked to their tool via the isEnforcedWith association.

Finally, the Metric class instances represent the value of a certain complexity metric computed over an application part (usually a code) (Tonella et al., 2014). The value attribute represents the numerical value of the metric, while the kind is modelled via the hasType association towards an enumeration MetricType containing all the available metric categories (e.g., Halstead length, cyclomatic complexity). The refersTo and the hasMetric associations direct towards respectively the relative application part and the current protection solution. Complexity metrics can be useful to quantitatively measure certain security features of an application, as we also discuss in Section 4. Together, these classes allow the meta-model to meet requirement R6.

We developed several tools that enable us to assess the security level of a protected application (see Section 4.2) and further increase the attack effort by strategically protecting some non-asset application parts (see Section 4.2.7).

3.4. The attack meta-model

The attack meta-model, whose UML class diagram is sketched in Fig. 4, contains all the classes and relationships used to represent the attacker, his attacks and their effects on the application and the protections. These classes allow us to model with precision the effects of the attacks on a generic application and its components, thus meeting the requirement R3.

The attackers are modelled via the Attacker class, related with the hasExpertise association to the AttackerExpertise enumeration, representing the various levels of expertise an attacker may have. We envision four levels of increasing expertise (i.e., geek, amateur, professional and guru). Note that this enumeration set is completely customizable and adaptable according to the scenario that needs to be modelled. In addition, the solution itself is related to a specific attacker via the hasAttacker relationship to explicitly indicate that it was generated to counteract a specific attacking profile.

Attack steps usually refers to an application part (not necessarily an asset). This is modelled through the refersTo association and the fact that an attack step can threaten a security requirement of an asset is modelled via the AttackTarget class and its relationships. For instance, if the variable 'x' is an asset whose confidentiality must be enforced, the attack step 'locate the variable x in the function y' refers to the function 'y' and has an attack target for the confidentiality of the asset 'x'.

The requiresExpertise association represents that an AttackStep may need a minimum level of attacker expertise to be mounted, thus representing its base difficulty level. This should represent a best-case scenario from the attacker point-of-view and can be useful to perform additional inferences on the global difficulty of an entire attack path. Analogously, the meta-model includes the requiresExpertise association between AttackTool and AttackerExpertise instances, which allows the classification of tools based on the minimum level of skills the attacker should have to use it.

Each attack step belongs to a specific type such as dynamic tampering or static analysis. This fact is formalized through the AttackStepType class and the hasType association. As stated before, we stress that an attack step does not necessarily need to be a full fledged attack, but it can also be a preparatory step such as 'setup a web server', thus the AttackStepType instances mix together both proper attacks and non attack types. Furthermore, the requiresExpertise association models the fact that an attack step type requires a minimum expertise level to be mounted by an attacker.

An attack step type (e.g., a debugging attack) can be performed by one or more different attack tool types (e.g., a debugger). This fact is represented by the isImplementedBy relationship with the AttackToolType enumeration, in turn related with the AttackTool class, via the hasType association, containing the known attack tools (e.g., IDA Pro).

The hasMitigation property is used to represent that a protection can mitigate an attack step type. For instance, this allows us to express that the opaque predicates obfuscation technique (Collberg et al., 1997) can be used to decrease the feasibility of both static and dynamic analysis attacks. The Mitigation class represents the protection mitigation. It is linked with the softened attack step type through the mitigates association and also allows to specify a non-numeric level of effectiveness by using the hasLevel association and the Level enumeration. Vice-versa, an attack can be used to partially or completely remove a protection. This is modelled via the hasDisruption relationship with one or more Disruption class instances. Analogously to the mitigation case, this class specifies the protection that is affected by an attack via the disrupts association and the effectiveness level of the disruption with the hasLevel relationship.

Risk analysis is an important phase in the software life cycle. We hence created a tool that allows us to automatically discover the attacks that can be mounted against a protected or not protected application (see Section 4.2.5) and another one that per-

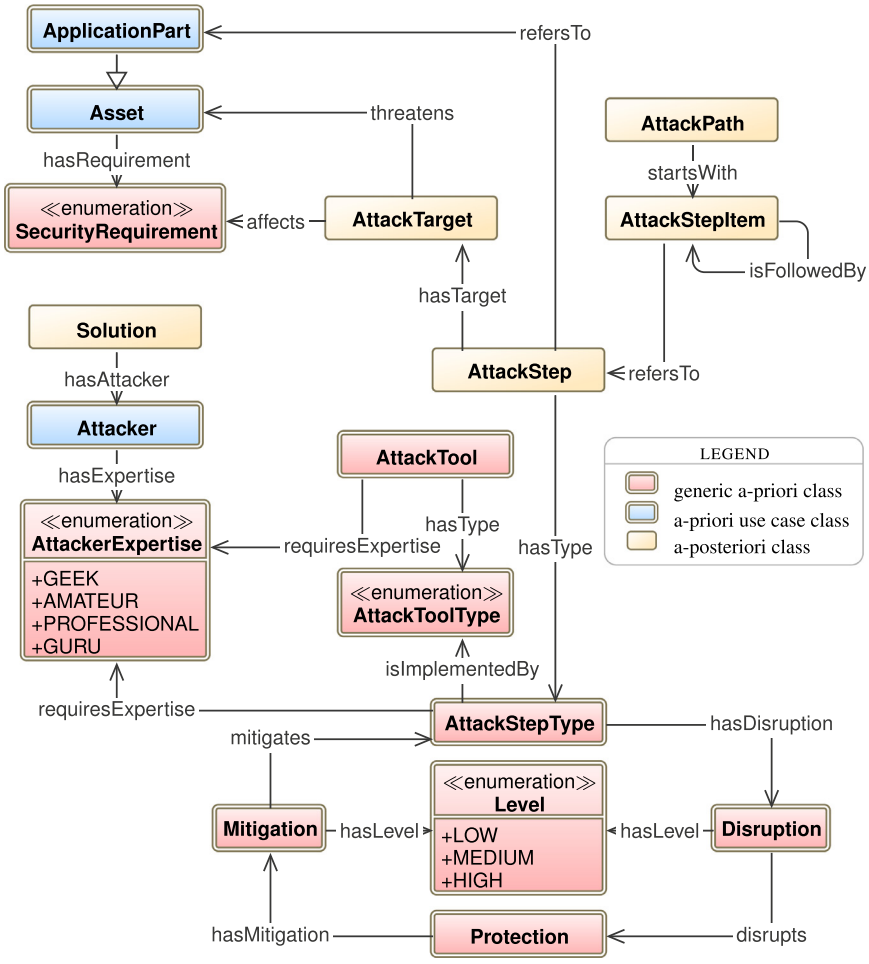


Fig. 4. Attack meta-model UML class diagram.

forms various kind of assessments on attack paths and steps via Petri nets (see Section 4.2.8).

The meta-model can indeed represent several simple yet useful inferences. For instance, information about expertise is useful to reduce the complexity of the attack discovery tool. If one wants to protect just against a certain category of attackers, the tool has not to consider all the attack steps and tools, which is an advantage with backward reasoning. As an example, if we consider the AttackerExpertise values as an ordered set (GEEK < AMATEUR < PROFESSION < GURU), an attack path can be considered feasible by an attacker of a given expertise (e.g., AMATEUR) if and only if all the attack steps needed to mount it require at most the same expertise (i.e., AMATEUR or GEEK). As another example, attack step types can be associated to the expertise based on the information about the tools needed to mount them. That is, if an attack step can only be implemented by attack tools that require a minimum expertise, the attack step type cannot be performed by attackers having expertise less than the minimum expertise required by the tools needed to mount it.

4. Validation

Several taxonomies and surveys of software protections and reverse engineering techniques have been presented in literature. In the first part of this section, we discuss to what extent our meta-model covers concepts and relations presented in that literature, thus validating that our models can capture the information considered relevant in literature. In the second part, we discuss a number of tools we developed to populate a KB system using our meta-models and to make practical use of the information in that KB system. With these tools, we validate that the models have practical use.

4.1. Validation against models from the literature

4.1.1. Reverse engineering taxonomy and models

Recently, Ceccato et al. developed models to capture the activities of attackers that target protected software (Ceccato et al., 2017, 2018). On the basis of penetration test reports and public challenge reports produced by professional and amateur hackers, they created a taxonomy of the concepts that were used by the attackers to describe their attack methods and corresponding reasoning processes. This taxonomy is a hierarchy of concepts, in which subconcepts are refinements and concrete instances of higher-level concepts. They also presented four models that capture causal, conditional, temporal and instrumental relations between

(a) The attackers' high-level comprehension activities;
(b) Their attack strategy building activities;
(c) Their attack tool selection, creation and customization activities;
(d) Their selection processes to choose between undoing, overcoming, bypassing or working around protections.

It is interesting to study how our meta-model covers the taxonomy concepts and models from Ceccato's work. With regards to the taxonomy, we observe that all top-level concepts can either be mapped onto the classes of our meta-model, or are irrelevant to it. More in detail, we observe the following regarding these top-level concepts:

Asset. Assets map directly to our Asset class.

Attack strategy. Attackers mention generic attack strategies as justification for how they spend their attack effort. Concrete attack path models in a KB, (through instances of the AttackStep, AttackStepItem and AttackPath classes in our meta-model) represent the result of concrete, executed attack strategies, i.e., the sequence of steps executed as a result of implementing an attack strategy. So while the current meta-model does not directly support modelling attack strategies, it supports concrete instances.

Background knowledge. Attackers rely on their background knowledge for making decisions. Since different attackers have a different background knowledge, they can choose the most disparate attack paths. Remember, however, that a defender reasoning about the protections most often needs to consider the worst-case scenario in which the attackers at the considered level of expertise (as incorporated in our attack meta-model, see Section 3.4) have all the possible knowledge available to the experts at that specific expertise level. The relevance of their combined background knowledge is determined by the attack paths that this knowledge enables. Hence, the relevant information can be incorporated in the meta-model by populating it with all the attack paths that are built on that knowledge. The attack-related classes in our meta-model support this representation, as discussed for the previous concept of attack strategies.

Workaround. In the taxonomy by Ceccato et al., workarounds are a specific class of attacks to defeat protections. They map directly to the AttackStep class.

Analysis / Reverse engineering. These concepts are techniques (e.g., diffing, debugging, profiling, pattern matching) to analyse and reverse engineer different aspects of the software under attack. Those concepts map directly onto the AttackStepType class.

Difficulty. Attackers face all kinds of practical issues in their experimental environment. That is the case because their concrete environments are not perfected a-priori for the attacked software. From a defender's perspective, these are best-case scenarios. On the other hand, in worst-case scenarios, such difficulties do not occur, so there is no strict need to cover them in our meta-model. In case the issues are not mere practical ones, but fundamental limitations (e.g., related to non-scaling analysis and decidability issues) the impact these difficulties have on attacks will be reflected in the absence of certain attack paths in the KB. So, by populating the KB with the relevant attack paths, and excluding the irrelevant ones, this concept can also be covered.

Obstacle. Ceccato et al. consider two kinds of obstacles that attackers face when trying to execute attack strategies. The first are protections, which are clearly covered in our meta-model (by the Protection, ProtectionInstance and AppliedProtectionInstance classes). In addition, also the effects that they have on enabling or disabling certain attack steps can be modelled via the Mitigation and Disruption concepts. The second kind of obstacles are additional limitations to the attack environment (e.g., impossibility to run the protected application, lack of knowledge in the application execution environment), to which the aforementioned discussion for the concept of difficulties applies.

Weakness. Weaknesses are features of application parts that ease attacks on corresponding assets, be it original assets with security requirements or protections that are attacked or intermediate assets that attackers target on their way along a complete attack path. The fact that an application part is weak against some attack step can be modelled in our KB by means of the AttackTarget class and the hasTarget, threatens and affects associations from our meta-model.

Tool. Concrete attack tools map directly to the AttackTool class. Their abstractions (i.e., sets of similar tools with similar capabilities) map onto the AttackToolType class.

Attack step. Ceccato et al. identified a wide range of attack activities at a fine granularity. These steps, such as "locate a variable in a function", have explicit or implicit objects, i.e., targets. For the previous example, these are the function and the variable themselves. These steps map directly to the AttackStep class. In addition, the relationships with the objects is handled by the refersTo association and the AttackTarget class (and its relationships), if the object is an asset.

Analyze attack result. Ceccato et al. noted that while executing the attack paths, the analysis of their results (in particular whether or not attempted attack steps succeeded) is an important aspect for attackers to decide on the next attack steps to try, i.e., to estimate the path of least resistance that they will try to execute. As our meta-model aims to model worst-case scenarios from the defender's perspective, it is not relevant to represent such steps separately. As discussed above, attack paths in our KB model all successful attacks.

Attack failure. Being the outcome of an attempted, but failed attack step, this concept is not relevant in the context of our meta-model.

Software element. Ceccato et al. listed a wide range of software artifacts that are targeted in individual attack steps because they serve either as ultimate targets of the attacks or because they serve as clues while the attacker is still searching for the ultimate assets. These artifacts or elements include both code and data, and static ones (e.g., code fragments, global data, API definitions and invocations) as well as dynamic ones (e.g., variables being assigned values during the program execution, system calls being executed, code patterns in traces). In all cases, they relate directly to the application parts that are covered by the ApplicationPart class in our meta-model.

As for the four inferred models of relationships between concepts in the work of Ceccato et al., we note that these reflect how attack paths are constructed by attackers. The knowledge in those models is not stored directly in the KB, but the constructed paths are, as already discussed above for some of the top-level concepts. Thus, the relevant conclusions to be drawn from Ceccato et al.'s models can be represented in a KB based on our meta-model.

In our prototype tools (see Section 4.2) that populate a KB built upon our meta-model, the inferred models from Ceccato et al. are present but in a strongly simplified form. More specifically, one of the so-called enrichment modules hard-codes some causal and temporal relations between attack steps to infer relevant attack paths starting from a set of attack steps. For simple cases, this module allows us to populate a KB with straightforward attack paths relevant to the use case at hand, i.e., the application at hand with its deployed protections and embedded assets. In future work, we plan to extend our meta-model to cover concepts of attack strategies as well as the relationships in the relational models of Ceccato et al. The relevant information to infer relevant attack paths for a given (protected) application with a set of given constraints from a set of generic, a-priori available attack steps can then all be stored in the meta-model, at which point we foresee that more interesting inference can be performed to populate the KB with attack paths, thus avoiding the need to insert a-posteriori complex attack paths manually with the Petri Net tools (see Section 4.2.8).

From this discussion, we conclude that requirements R3-R4-R5 are met with respect to the concepts and relations considered relevant by Ceccato et al.

4.1.2. Obfuscation taxonomy

Collberg et al. presented the first taxonomy of obfuscation techniques in a seminal paper on software protection against reverse engineering (Collberg et al., 1997). The obfuscation taxonomy includes layout obfuscations, data obfuscations, control obfuscations, and preventive transformations, and discusses several de-obfuscation attacks. The paper also puts forward potency, cost, and resilience as aspects to consider during the evaluation of protections.

The obfuscation techniques discussed in the paper are mostly covered by our protection meta-model: data and control obfuscations operate on code and data; both those forms of application parts are covered by the meta-model. Layout obfuscations are not but they are also outside the scope of our model: they concern source-code aspects such as comments and names of variables, which are mostly irrelevant in stripped binaries. Those binaries, be it main binaries or dynamically linked libraries, are the form in which native software, the focus of our work, is distributed and hence attacked. The sole exception is when identifiers identify external APIs. As discussed in the previous section, API definitions and invocations are covered by the ApplicationPart class. Preventive obfuscations, i.e., obfuscations that do not hide assets but that prevent analysis techniques from providing (very) useful results, are modelled as well, and the Mitigation class and its relations in the attack meta-model enables us to model which protections prevent which attack steps.

To evaluate the potency of obfuscations, Collberg et al. propose to use software complexity metrics that need to be computed on the relevant application parts. Others later extended on this idea, including some of us (Anckaert et al., 2007; Tonella et al., 2014; Ceccato, 2016). Our protection meta-model contains the relevant classes and relations to express the necessary information regarding such metrics and application parts. To evaluate the cost of protections, specific metrics can be used, such as the static number of instructions to measure code size, and the dynamic number of instructions (i.e., the number of executed instructions for some inputs) to approximate performance overhead. To evaluate resilience, Collberg et al. propose a discrete scale with five levels of resilience: trivial, weak, strong, full, and one-way. In the attack model, the Mitigation and Level classes can capture three levels of resilience of protections against attacks. We opted for only three techniques because the level "trivial" is mostly useless when considering only worst-case scenarios, and because one-way is theoretically possible, but in practice not yet achieved in MATE scenarios where attackers have white-box access to the software and assets under attack. When considering resilience, Collberg et al. distinguish between programmer effort, i.e., the effort needed to build or customize tools to perform an attack, and de-obfuscator effort, i.e., the time and resources needed to deploy the thus built tools. In our models, we do not make this distinction explicitly. However, the individual attack steps that our meta-model covers can be both preparatory steps, such as customizing a tool, and actual attack steps, such as deploying a tool. Thus our meta-models are expressive enough to capture all the concepts and relations put forward by Collberg et al.

From this discussion, we conclude that requirements R3-R4-R5-R6 are met with respect to the concepts and relations considered relevant by Collberg et al.

4.1.3. Obfuscations versus program analyses

Much more recently, Schrittwieser et al. surveyed the state of the art in software obfuscation vis-à-vis code analyses (Schrittwieser et al., 2016). The latter are used as attack techniques to directly attack obfuscations, i.e., if obfuscations lack the necessary resilience, and to work around obfuscations, i.e., if obfuscations are not potent with respect to some reverse engineering task. Like the concrete obfuscation techniques surveyed by Ceccato et al. the concrete ones surveyed by Schrittwieser et al. can be modelled with our protection model. Furthermore, the code analysis techniques surveyed by Schrittwieser et al. can be modelled with our attack model.

Schrittwieser et al. provide a taxonomy that partitions concrete attack techniques in categories based on (i) the attack goal, (ii) the generic, abstract technique used to reach that goal, such as "locating code through static analysis", and (iii) whether or not the technique is fully automated or performed with human assistance (or even completely manually). Each of the different combinations they consider can be modelled with multiple instantiations (one for each concrete technique) of the AttackStepType and AttackToolType classes from our attack meta-model. Finally, Schrittwieser et al. analyse the resilience and potency of the obfuscations with respect to different attack classes, and label them in three categories, ranging from "minor increase of costs", over "not unbreakable, but makes analysis more expensive", to "breaks analysis fundamentally". These labels map well onto the three levels of mitigation in our attack meta-model.

From this discussion, we conclude that requirements R3-R4-R5-R6 are met with respect to the concepts and relations considered relevant by Schrittwieser et al.

4.1.4. Integrity protection taxonomy

An interesting taxonomy of software integrity protections has recently been published by Ahmadvand et al. (2019). It covers several concepts that are also represented in our meta-model. However, we noticed two major differences that are related to the goal of the two works. Whereas our approach started from the need of representing the information needed when protecting an application from tampering (to break software integrity requirements or to defeat anti-reverse-engineering protections in MATE scenario with software-only protections), the classification presented by Ahmadvand et al. aims at describing integrity protections. Therefore, our attack meta-model is more precise than their taxonomy. While they define generic attacks, which can be roughly mapped to our attack step types, we also have the possibility to define precise attack steps that refer to the original, intermediate, and protection assets, to group them in paths and associate attack tools on individual attack steps. Moreover, their taxonomy lacks the concept of a deployed protection and of the solutions that are needed when deciding how to protect an application.

On the other hand, they expanded the high-level classification of protections with intermediate concepts that group protections in a way that is interesting for categorisation purposes, but is unnecessary for our goals. Moreover, their classification includes information about the life cycle, which describes information about management and production stages of the application to protect. It will be certainly interesting to study how that information can be integrated in our model, as life cycle information can be useful when protecting libraries or when protecting applications without having the possibility to access source code. Moreover, the proposed taxonomy includes high-level concepts like overhead (which we have explicitly avoided as too coarse grained by resorting to a broader concept of metrics), and trust anchor (which defines hardware security mechanisms that we have excluded by hypothesis). Moreover, in their work, authors explicitly defined the granularity of representation of the assets, which we can avoid as our application meta-model conveys precise information on the application parts that allows us to infer the granularity.

4.2. Validation with practical tools

The meta-model presented in Section 3.2 has been used in conjunction with various tools to support the process of analysis and protection of an application.

These tools have been developed and used in the context of the ASPIRE project. Here they are introduced to demonstrate that our meta-model is able to convey information useful to perform real software protection tasks.

4.2.1. Integration with Eclipse EMF

The meta-model has been implemented using the Eclipse Modeling Framework (EMF)[7], allowing its manipulation and navigation directly in Java applications. EMF is a well supported standard in the Eclipse world and several tools (e.g., Eclipse Epsilon[8]) are available to perform various modelling tasks, such as validation and model-to-model transformation. The code of our meta-model is publicly available at https://github.com/SPDSS/adss/tree/master/eu.aspire_fp7.adss.akb.

This implementation contributes to the coverage of the R7 requirements, as it concerns the usability of our meta-model.

4.2.2. Automatic analysis of the application to protect

We developed a tool based on the Eclipse C Development Toolkit (CDT)[9] that is able to parse a set of C/C++ source files, identifies the functions, their parameters, reconstruct the call graph, locate the local and global variables, and to translate such information into appropriate instances of our EMF-based meta-model, in particular the application meta-model. The fact that we were able to correctly and properly represent all the information extracted by CDT about an application that we deemed important for attack evaluation and protection purposes validates the effectiveness of our meta-model in meeting the requirement R1.

Moreover, the CDT tool is also used to parse annotations, directly applied to the code by software developers or analysts, that indicate which parts of the applications are assets and which are their security requirements. The ASPIRE tools support annotations in the form of pragmas to identify and annotate code regions of interest and attributes to identify and annotate variables of interest (Basile et al., 2016). Using this approach, we were able to validate the capability of our meta-model to represent what constitutes an asset, thus fulfilling requirement R2.

4.2.3. Text to OWL conversion

We developed a text2OWL tool[10] for developers who are not familiar with the OWL formalism and tools. It was developed to create or update a valid OWL ontology out of a text file containing a taxonomy of reverse engineering attacks. The input text file consists of two parts: the first one contains the taxonomy of concepts, while the second part consists of additional rules between such concepts. In the first part the taxonomy is made of a set of trees of concepts whose hierarchy is defined by the indentation, as in the following excerpt of the textual taxonomy:

```
Analysis-reverse engineering
=Static analysis
==Diffing
==Control flow graph reconstruction
=Dynamic analysis
==Dependency analysis
```

The number of = characters indicates the sub-concepts' nesting depth. This tool generates the same class hierarchy in OWL via an axiom for each tree edge as a triplet of 'concept, relationship, concept' (e.g., 'Diffing isSubConceptOf StaticAnalysis').

A list of similar triplets forms the second part of the text file, but with different types of relationships (e.g., 'Analysis-Reverse Engineering usedTo IdentifySensitiveAssets'), that actually transform the taxonomy in a thesaurus (a graph of concepts, not bound by a tree structure like a taxonomy). Furthermore, the tool also checks for inconsistencies among the concepts defined in the rules and the taxonomy (e.g., concepts in the rules that do not appear in the taxonomy). This tool has been specifically developed for, and tested on, the taxonomy and models of Ceccato et al. that were discussed extensively in Section 4.1.1. This tool also helps us to meet the usability requirements of R7.

4.2.4. Integration with OWL ontologies

Given the huge amount of information required to perform security analysis of software applications in MATE scenarios, supporting the KB enrichment with automatic inferences was one of our primary goals. Ontologies are an important tool that we have positively evaluated to perform basic inferences and checks. For this purpose, we developed an API to translate the EMF meta-model in an ontology[11], written in the Web Ontology Language 2 (OWL2) and vice-versa (from OWL2 to EMF) to feed the meta-model with the inferred data. In addition, this API allows the manipulation of the ontology (e.g., create/remove classes/individuals or write SWRL[12] rules), uses a reasoner (we support both the Hermit[13] and Pellet[14] reasoners) and performs advanced queries using the SPARQL-DL language[15]. This allows executing advanced searches, coherence checks (e.g., test if a Solution instance does not contain any forbidden precedence between its applied protection instances) and various logical inferences (e.g., infer all protections that mitigate a particular attack step with a given level of efficacy).

With the help of such tool, and eventually manually filling out the missing information, it is possible to generate instances of the application and protection meta-models constituting a strong KB, to be used with more advanced inference and analysis tools. This tool fulfills all the R1–R6 requirements, as it concerns the generation of a-posteriori information and hence covers all the meta-models.

4.2.5. Deriving attack paths against an application

We have developed a tool, written in Java, which infers various types of attack paths on application assets by using Prolog-based reasoning (Basile et al., 2015; Regano et al., 2016). We have used the meta-model to instantiate a KB with various types of attack steps that include dynamic and static tampering attacks as well as network attacks, such as sniffing and spoofing the client-server communications.

The tool manages a fact base that is initialized with the information, taken from the KB, about the assets and their security requirements. Moreover, the tool imports from the KB the attack steps, which have been annotated (manually by us at the tool design time) with pre-conditions and post-conditions. Pre-conditions are predicates built on the facts in the fact base. Examples of facts are:

[7] See https://www.eclipse.org/modeling/emf/.
[8] See https://www.eclipse.org/epsilon/.
[9] https://www.eclipse.org/cdt/.
[10] Online at https://github.com/uel-aspire-fp7/text2owl
[11] Its source code is available at https://github.com/daniele-canavese/ontologies.
[12] See https://www.w3.org/Submission/SWRL/.
[13] See http://www.hermit-reasoner.com/.
[14] See https://github.com/stardog-union/pellet.
[15] http://www.derivo.de/en/resources/sparql-dl-api/.

- The asset *a* is a code region inside function *f*, which is used to infer relations of attack steps related to static and dynamic analysis;
- Traces collected for function *f*, which indicates that the application has been executed in a previous attack step, and which enables all attack steps that involve dynamic analysis;
- The value of the variable *x* is known, which may be the target of an attack (e.g., knowing the license key) or enable cryptographic operations (together with the fact *x* is a symmetric/asymmetric private key).

When a pre-condition is true, the attack step can be executed and adds new facts in the fact base. The tool uses Prolog to infer, with backward reasoning, if there is a sequence of attack steps, i.e., an attack path, that compromises the security requirements of the assets. All the discovered attack paths are then added in the KB. In the end, this tool is able to fill in an instance of the attack meta-model in a completely automatic fashion. The effort of annotating attack steps is only needed once and it needs an update only in the rare event of new attack steps added to the KB. With this tool we have been able to validate the satisfaction of requirement R3, as the meta-model was able to properly store all the inferred attack paths and steps on industrial ASPIRE use cases (as will be discussed in more detail in Section 4.2.12).

4.2.6. Protections and their potency estimation

The protections that counter the attack paths can be found with various inference rules. We implemented them as custom enrichment modules that integrate ontology reasoning with our EMF implementation of the meta-model. Once these protections are found, they must be applied in the right order on each (original, intermediate and protection) asset, thus producing a Solution instance. Waiting for an effective automatic decision support system that finds such, these solutions are manually devised. In order to assist the security expert to estimate the effectiveness of such solutions, the concept of potency introduced by Collberg et al. (1997) can be used. The potency is essentially a value stating how good the security of a protected asset is based on the value of selected software metrics. In his work, Collberg proposed the use of seven static and dynamic metrics. Since metrics need to be measured on the protected asset, evaluating the potency of a protection over a specific asset means that the protection needs to be actually applied, the program possibly rebuilt and some complexity metrics needs to be extracted by an ad-hoc tool. This process can be time consuming, especially if the application is big and/or if the number of candidate solutions to choose from is high. To avoid the actual application of protections, we developed an estimator that uses a set of neural-networks trained to predict, with a high degree of accuracy, the variations of the metric values used to compute the potency. Therefore, with this tool a defender is able to accurately estimate the potency of a solution starting only from the unprotected assets' complexity metrics without rebuilding the application each time (Canavese et al., 2017).

By using the protection meta-model to store the information about (single and combination of) protections applied to an asset and the various application part metrics, we validated the satisfaction of the requirements R4 and R6.

4.2.7. Hiding protected assets

Protected assets have recognizable fingerprints that can be identified and exploited by attackers. For instance, obfuscation techniques may flatten the control flow or increase the number of if statements (opaque predicates) to render code understanding more difficult. However, static analysis and inspection allow an attacker to identify these protected parts with respect to unprotected areas. Therefore, after having protected the assets, a security expert might decide to fool the attacker by applying the same protection on other application parts that are not real assets with the purpose of delaying the attacker activities, who will have to evaluate more candidate assets fingerprints. We named this protection step *assets hiding*. We developed a tool (Regano et al., 2017) that automatically generates a mixed-integer linear problem for the IBM ILOG CPLEX[16] solver to select the best applications parts where to apply these decoy protections in order to maximize the attacker confusion and delay, by leveraging the information in the application and in the protection meta-models.

Also in this case, the protection meta-model served his purpose, as it allowed us to model both the protected assets and the other protected application parts, thus validating the requirement R5.

4.2.8. Petri net modelling of attacks

Petri Nets (PNs) (Peterson, 1977) are often used to model the flow of information in concurrent and distributed systems. We chose a Petri net editor to model reverse engineering attacks visually, thus helping to meet requirement R7.

Petri nets are bipartite graphs, with two types of nodes: places and transitions, visualized as circles and rectangles respectively. In our interpretation, places represent sub-goals reached during an attack and transitions correspond to attack steps being executed. The final place in the model represents the final goal of the attacker, i.e., accessing or compromising the security-sensitive asset. By correctly connecting the places and transitions in a single PN, one can easily model one or more sub-goals that need to be reached before the next attack step can be executed, which attack steps can be performed concurrently or sequentially, and which alternative attack paths lead to the same goal. In a Petri net model there are different attack paths that can be followed to achieve the final goal. Each attack path is a temporal sequence of attack steps, visited by a token (a black dot within a place in the PN model) traversing the net from the initial state to the end state through one of the possible attack paths. Each token represents a different attacker in a team of attackers in collusion to achieve the same goal. In this way attacks performed in parallel by two colluding attackers can be represented.

PNs with Discrete Variables (PNDVs) are a more recent PN extension with a set of finite global integer variables, used in preconditions, that are guards on transitions (Kindler, 2011). In our experience with all the ASPIRE use cases, we noted that the information used by the attackers can be decomposed and mapped to a set of 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 of further investigation. Such intermediate knowledge can be represented with a code region array, where each code region is represented by a couple of integer numbers, representing the initial and final offset with respect to the base address of the binary code.

To design the attack models we used ePNK, an Eclipse-based tool[17] which provides a Java-based extensible open source platform for PN modelling, based on EMF and Graphical Modeling Framework (GMF).[18] The current ePNK plug-ins allow designing a PN model with discrete values and save it as standard PNML[19] file, as the GMF-based editor is built on top of an EMF meta-model of PNML. We used this tool to model attacks on two software protection techniques. The first, as shown in Fig. 5, aims to extract the cryptographic key from a White-Box Crypto (WBC) (Wyseur, 2008)

[16] https://www.ibm.com/analytics/data-science/prescriptive-analytics/cplex-optimizer.

[17] See http://www.imm.dtu.dk/ekki/projects/ePNK/.

[18] See http://www.eclipse.org/gmf-tooling/.

[19] Petri Net Mark-up Language (PNML) standard ISO/IEC 15909, on-line at http://www.pnml.org/.

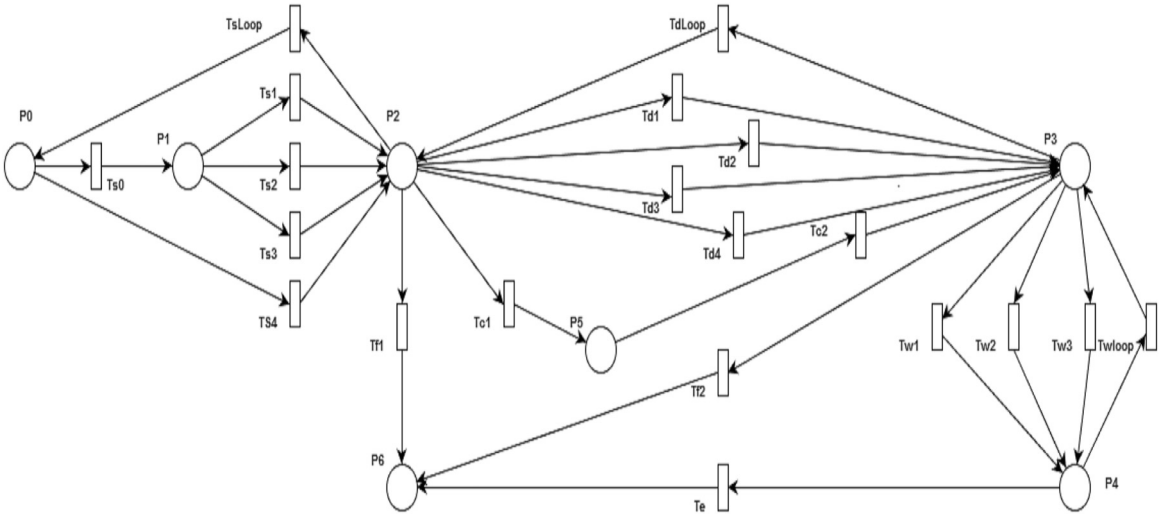


Fig. 5. Petri net for the attack on white-box cryptography.

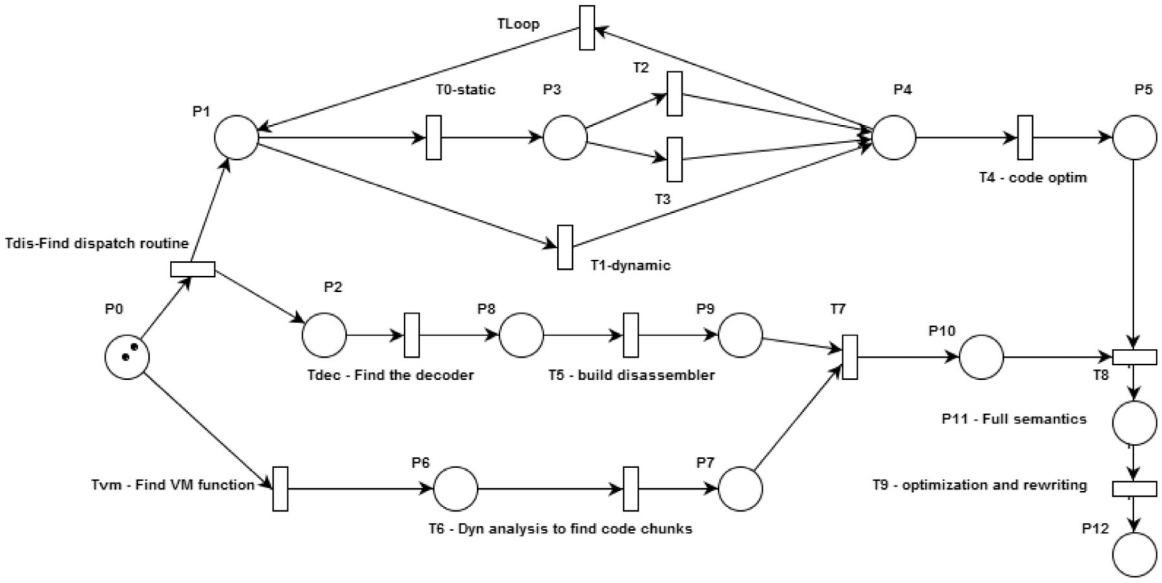


Fig. 6. Petri net for attacks on a SoftVM.

library. The second aims at de-obfuscating the code of an application protected by the use of custom, randomized instruction sets that are interpreted by a software virtual machine (SoftVM) (Ghosh et al., 2010), as shown in Fig. 6. These models have been designed after four rounds of interactions with the security experts from the ASPIRE project's industrial partners responsible for the development of these protections.[20]

These modelling exercises have helped the security analysts in identifying and visually defining the different attack steps and related attack tools. Moreover, the Petri net editor has been used to populate the OWL KB with new attack step types and new instances of attack steps. Each new attack step in the Petri net can be mapped to a new or existing AttackStepType and to a new AttackStepItem object in OWL, according to the syntax defined in the attack meta-model of Fig. 4. Similarly the attack paths represented by the PN are mapped to many AttackPath objects in the KB, while the temporal sequence between attack steps in a PN model is mapped into a set of OWL axioms instance of the isFollowedBy relationships between two AttackStepItem objects. Finally the Petri net models have been used by other tools for more advanced analysis and simulation (Zhang et al., 2016).

The Petri net modelling helped us in validating the R3 requirement of the meta-model by showing that it can represent even complex attacks on industrial use cases.

In the remainder of this section, we report more details about the modelling of the attacks on the two protections to demonstrate the level of precision that our attack meta-model can reach. First, the PN attack model on white box cryptography contains preconditions to transitions. For example, in case the attacker has detected AES-related binary code (represented by Ts0), 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 IDA Pro-findCrypt2 plug-in can be used (Ts3). After an initial phase with the static analysis tool, the attacker is in P2 and can choose among many following attack steps representing different dynamic analysis techniques (Td1 to Td4): each attack step can be executed depending on the results and the type of the static analysis attack

[20] WP4 deliverables on the ASPIRE website present a full description of the attacks on the two use cases.

step previously executed. The pre-condition can help defining that, for example, Ts1 isFollowedBy Td1 is a possible sequence of AttackStepItem while other combinations of attack steps are not actually feasible.

On the other hand, the design of a PN attack model on virtualization obfuscation as in Fig. 6 has shown that an attack can be performed by a team of attackers working in parallel. In the virtualization obfuscation, the SoftVM contains 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 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. Each code chunk can contain one or more bytecode instructions. The VM contains different code portions that are interesting to the attacker: (i) the VM function called by the native code to transfer control to the VM; (ii) a decoder, which translates the bytecode into native code; (iii) a dispatch routine that given a particular bytecode invokes the IH, and (iv) the different IHs.

Petri nets are particularly useful to model parallel processes, and in this example we can see how three attackers can work together to achieve the common goal. Attacker1 can start looking for the VM function with dynamic analysis (attack step Tvm), and then search for the bytecode chunks within the binary code (T6). The other two attackers can find the dispatch routine of the VM (Tdis) and then split the work: Attacker2 can focus on the decoder function and building a custom disassembler for the bytecode (Tdec followed by T5), while Attacker3 can search for the IHs using various static and dynamic analysis tools (To, T2, T3, T4).

In order to rebuild the de-obfuscated code (attack step T7), Attacker2 and Attacker3 must synchronize to understand the bytecode semantics by running the code chunks (found in attack step T6) through the custom disassembler (built in attack step T5). Once Attacker1 will find the IHs he will have to synchronize with the others to combine the bytecode semantics and IHs manually to understand the full semantics of the de-obfuscated code. This case study with parallel attacks can be represented in the KB system with a set of axioms representing the different sequences of *AttackStepItem* linked by the relationship isFollowedBy; in fact, (T5, T6, T7) or (T6, T5, T7) are two valid attack paths representing the fact that T7 can start only when both T5 and T6 have been performed in any order.

4.2.9. Validation on software protection tool chain

In the ASPIRE project, a tool chain for composable native software protections was developed (Basile et al., 2016), which integrates a wide range of protections, and of which almost all components are available as open source at https://github.com/aspire-fp7/. This tool chain is called the ASPIRE Compiler Tool Chain (ACTC). It uses compiler techniques to deploy software protections on applications. Those protections all implement different parts of a layered software protection architecture (Wyseur et al., 2016; De Sutter et al., 2016a). The ACTC's protections aim at defending against reverse-engineering, tampering, and cloning. They include code and data obfuscations Collberg et al. (1997), white-box cryptography (as also discussed in Section 4.2.8) (Wyseur, 2008), code mobility (Cabutto et al., 2015), code diversity, code guards, code renewability, remote attestation and migration of sensitive code to secure servers (Viticchié et al., 2016), use of custom instruction sets interpreted by virtual machines (Ghosh et al., 2010) (as also discussed in Section 4.2.8), anti-debugging by means of self-debuggers (Abrath et al., 2016), and more.

```
int func(int x)
{
   int i=0;
   //start of the asset
   _Pragma("ASPIRE begin requirement(integrity)");
   x++;
   i--;
   _Pragma("ASPIRE end");
   //end of asset
   return i-x;
}
```

Fig. 7. Example of a code annotation used to define assets.

During the project, we validated that the presented meta-model can capture the necessary aspects of all of those protections, of the tool chain that allows the composition of those protections to varying degrees, and of the attacks we surveyed in the project and collected in the so-called ASPIRE attack model. This includes, e.g., the two attack models discussed in Section 4.2.8. We cannot discuss the full attack model in detail, as it was a confidential document. It consists of a survey of the different types of assets and their security requirements; the different types of attackers that we might face; the concrete methods, tools, and techniques that are available to the attackers and the different types of attack activities that can be performed with them to reach specific intermediate or final attack goals; as well as the possible ways in which the attackers combine different attack activities to reach their final goal. We do confirm, however, that all attacks considered as relevant in the scope of the ASPIRE project by both its academic and its industrial partners, are covered by our meta-model.

From this discussion, and from the final validation report (De Sutter et al., 2016b) of the ASPIRE project, we conclude that requirements R1–R6 are met with respect to the concepts and relations considered relevant in the scope of the ASPIRE project.

4.2.10. Software protection work flow

The tools presented in Section 4.2.2 and Sections 4.2.5 to 4.2.8 have been integrated with the ACTC as introduced above, to assist software developers in (semi-automatically) protecting their applications with the ACTC. The meta-model allowed us to integrate inferences as needed for providing decision support for using the tools in the ACTC into a KB system. The integrated tools and the ACTC thus form a tool-supported work flow for semi-automated software protection.

As a first step, the work flow calls the tool of Section 4.2.2 to an instance of the application meta-model by parsing and analysing the structure of the C/C++ application to be protected. The user only needs to link the application parts he considers as assets to security requirements. This is done manually, via pragma annotations. An example of an asset, in this example a part of the C code that requires integrity, is provided in Fig. 7. For a complete specification of the supported annotations, we refer the reader to the ASPIRE Framework Report (Basile et al., 2016) and the ASPIRE Open Source Manual (Coppens et al., 2016).

The structure of the target application is described by means of classes from the application meta-model of Section 3.2. Variables and functions are translated into instances of the ApplicationPart, assets as Asset instances, while security requirement annotations are translated in hasRequirement relationships between the Asset instances and values of the SecurityRequirement enumeration.

The structure of the application stored in the KB is then analysed by the automatic attack discovery tool described in Section 4.2.5. The identified attacks against the application's assets are then translated by using the a-posteriori classes defined in the attack meta-model described in Section 3.4. For each instance of the hasRequirement relationship, i.e., for each security requirement of each asset, the tool generates an instance of

the AttackTarget class, with the threatens and affects relationships set accordingly. Then, for each AttackTarget instance, the tool tries to generate any possible AttackPath containing at least one AttackStep having a hasTarget relationship with the AttackTarget instance. Attack paths are generated by following a set of Prolog rules, contained in an external KB system, as described in Regano et al. (2016). Identified attacks may also be manually visualized and refined by the software developer with the Petri net tool described in Section 4.2.8. Manual attack paths may be added to the attack meta-model; they will hence be compatible with the later tools in the work flow.

After inferring the possible attacks against the application with the custom enrichment modules of our EMF meta-model implementation described in Section 4.2.6, the work flow identifies the protections that can be applied on the target software in order to block the attacks found in the precedent step of the work flow. The tool automatically generates an instance of the ProtectionInstance class for each AttackPath instance for each Protection having a Mitigation for the AttackStepType of at least one AttackStep instance in the target AttackPath. The candidate ProtectionInstance instances can be manually combined by the application developer into an instance of the Solution class.

The tool described in Section 4.2.6 produces an estimation of the software metrics on the ApplicationPart instances after being protected with the AppliedProtectionInstance instances in the Solution (linked with the isAppliedOn relationship). The estimated metrics serve to compute an estimated potency of the solution without actually applying any protection on the application. With this approach, the defender can quickly compare several solutions in terms of effectiveness and overhead, without spending time to actually apply the solutions and measure and compute the metrics on the target software. Clearly, the usefulness of the estimation relies on its precision. In the ASPIRE project, in which we used profile information collected on the unprotected software to drive the estimation, we found it sufficiently precise for selecting protections. These data are saved in the protection meta-model by means of hasMetric relationship between each Solution and Metric class instances, for each pair of ApplicationPart and MetricType. Moreover, the original metrics of the unprotected application can be modelled using a dummySolution instance that links no protections and it is not related to any AppliedProtectionInstance.

Next, assets are hidden in other code with the tool of Section 4.2.7. It refines a Solution by adding decoy protections as AppliedProtectionInstance class instances, both on already protected assets and other ApplicationPart instances not marked as assets. In the latter case, there is no need to link the ApplicationPart to security requirements.

As a final step, a tool is executed to annotate the source code of the application with data that can be processed by the ACTC tool chain to automatically apply the protections. The tool, starting from the solution selected by the defender, navigates the associations in the meta-model to identify the code to be protected (i.e., files and line numbers) and determines, for each code region to protect, the low-level parameters that configure the deployment of each protection. All the data that will drive the tool chain for that deployment is injected into the source code in the form of low-level annotations, named protection annotations. Like the aforementioned security requirement annotations, these are pragmas and attributes. With the protection annotations, however, the developer configures the ACTC to deploy concrete protections on the assets, i.e., on the annotated code fragments. We again refer to the ASPIRE Framework Report (Basile et al., 2016) for a complete spec of those annotations.

4.2.11. Validation of work flow on open source application

We have executed our work flow on an open source application, Sumatra[21], a C console application used to compare DNA sequences. More information on the meta-model instance and the meta-model parts that have been instantiated during the phases of the presented work flow is available as support material to this paper.

To simulate a risk analysis and mitigation task of a software application, even if Sumatra is open-source and free, we treated it like it was commercial software, whose comparison algorithms must be safeguarded against reverse engineering to protect intellectual property. We have thus manually identified the assets, 25 functions related to the DNA comparison, performed in four consecutive phases, which we have associated to the confidentiality security requirement.

We have identified 162 attack paths able to compromise the security requirements associated to the assets. Then, we have identified nine types of protections that may help in stopping/delaying the identified attacks. These protections can be applied in different ways to the assets by changing their configuration and application parameters, our tool flow identified 299 different protection instantiation instances that can be considered during the protection phase (e.g., for trading off performance and potency). Based on this information about attacks and useful protections, we have defined one solution that, according to our experience, properly protects the assets. This solution includes 27 protections instances, at least one for each asset. For an asset that has been considered more sensitive, the solution foresees the application of a combination of three protections. Finally, we have refined this solution by adding 45 protections to additional application parts to help hiding the original assets.

The instantiation of the meta-model and the associated knowledge base is available as an ontology file[22], written in the Web Ontology Language 2 (OWL2). In a supplementary document associated with this paper (Basile et al., 2019), we present a detailed analysis of how the work flow performed on the Sumatra application, and how the meta-models were instantiated for this application.

4.2.12. Validation of work flow on industrial use cases

As part of the ASPIRE project, the ACTC was validated on industrial use cases. The three industrial project partners, Nagravision, SafeNet and Gemalto, are world market leaders in their digital security fields. They developed the uses cases, and in particular the client-side Android apps of which the security-sensitive parts were implemented in native dynamically linked libraries that were protected by means of the ACTC. DemoPlayer is a media player provided by Nagravision. It incorporates DRM (Digital Right Management) functions that need to be protected. LicenseManager is a software license manager provided by SafeNet. OTP is a one time password authentication server and client provided by Gemalto. Table 2 shows their lines of code (measured by `sloccount` Wheeler (2001)). All security-sensitive code is implemented in the C code part, which is the code protected with the ACTC.

Security experts from the industrial partners determined the assets in the C code, as well as their security requirements. A pseudonomynous list of them can be found in Section 5 of the ASPIRE Validation Report (De Sutter et al., 2016b). The security experts, together with the developers of the ACTC, then also determined which configurations of protections have to be deployed on each asset to achieve sufficient protection against attacks on the assets. Table 1 lists the deployed protections on the use cases. Note

[21] https://git.metabarcoding.org/obitools/sumatra/wikis/home.

[22] https://github.com/uel-aspire-fp7/text2owl/EMSE2018.owl

Table 1
Protections applied to each industrial use case.

Industrial UC	Data Obf.	Anti Debug	Remote Attestation	Code Mobility	Client-Server Splitting	SoftVM Obf.	WBC	Binary Obf.	Diversified Crypto Libs
DemoPlayer	×	×		×		×	×		
LicenseManager	×		×		×	×	×	×	
OTP		×						×	×

Table 2
Size of industrial use case applications in SLoC per file type, before the ACTC is deployed.

Application	C	H	Java	C++	Total
DemoPlayer	2595	644	1859	1389	6487
LicenseManager	53,065	6748	819	-	58,283
OTP	284,319	44,152	7892	2694	338,103

the use of the SoftVM obfuscation and WBC for which we discussed Petri net attack models in Section 4.2.8. To generate the protected use cases, their thus annotated source code was sent through the ACTC.

At this point, it is useful to remark that the penetration testing experiments with professional, hired hackers mentioned in Section 4.1.1 as the basis for the models developed by Ceccato et al. (2017, 2018) were performed precisely on these protected use cases. When we validated that our meta-model covers all protection and attack concepts taken into account by Ceccato et al. as discussed in Section 4.1.1, this therefore already implied the validation of the meta-model with respect to all attack activities performed on the protected industrial use cases by the professional penetration testers. For those penetration tests, the necessary protection annotations were injected into the use cases' C code, and the thus annotated use cases were compiled and protected by the ACTC.

Access to the industrial use cases, to the security requirements of their assets, to experts' opinions on how to best protect the assets with the ACTC, and to reports of actual penetration test experiments performed on the protected use cases provided an ideal basis for validating the meta-model and the work flow engineered around it.

We hence validated the work flow presented in Section 4.2.10 on the use cases. For this validation, we started from use case source code annotated with the security requirements annotations, not with the protection annotations. Also in this validation effort, we involved security experts from the industrial partners. In particular, we asked them to assess the practical usefulness of the work flow.

The security experts were satisfied by the level of detail of the information obtained by our tools about the applications to protect. This implicitly validates the meta-model that allow to represent these data.

The security experts were surprised by the number of attack paths our tool was able to identify and appreciated the possibility to add new attack paths manually. Again, the information represented by the meta-model was defined sufficient and appropriate. However, they found the attack steps we instantiated for our analysis had been defined too coarse grained. As the meta-model supports more fine-grained attack steps (we simply did populate the KB a-priori knowledge with such steps), this is not a fundamental issue.

Furthermore, the experts were satisfied by the protections identified by the tool to mitigate the risks of each attack path. To a large degree, these identified protections overlapped with the ones they had proposed manually. They also appreciated the possibility to precisely link each protection to the attack step it affected. Moreover, the possibility to indicate combination of protections and an optional order of application was an important characteristic, in their opinion, for the adoption of the work flow.

Even if they were a bit reluctant on considering the potency score we computed for each combination of protections as trustworthy, they were convinced that the possibility to visualize metrics and protection scores for each asset to protect was a useful feature.

We can conclude from the feedback received that the meta-model and the corresponding work flow can be considered positively validated.

5. Related work

In this section we provide some additional insights on the current state-of-the-art on the use of meta-models, ontologies and Petri nets in cyber-security, complementary to the related work already discussed in the introduction.

Meta-models. Various meta-models and modelling languages have been proposed to represent threats in enterprise networks. Sommestad et al. (2013) presented the Cyber Security Modeling Language (CySeMoL), which can be used to model computer systems in enterprise networks. In addition, the authors presented a way to infer threats against such systems using an inference engine on the models developed with CySeMol, evaluating also the success probability of the inferred attacks. Based on this work, Välja et al. (2015) proposed an improved security analysis, that considers attacks by attackers external to the enterprise network mounted and by legitimate users inside the network.

Kritikos and Massonet (2016) presented a meta-model to assess the security of cloud applications, alongside a domain specific language, namely CAMEL (Cloud Application Modelling & Execution Language). It permits the description of the design and the security requirements of cloud applications and allows the validation of the model against a set of constraint expressed using OCL (see https://www.omg.org/spec/OCL/).

In the field of access control systems, Mouelhiv et al. (2008) proposed a meta-model to represent access control policies, with a particular focus on mutation analysis, a testing technique for security policies based on the voluntary injection of flaws (mutation) in policies, in order to evaluate the efficiency of the security tests. Mutation operators are included in the meta-model to represent the aforementioned testing process.

Model-Driven Reverse Engineering approaches usually aim at extracting models from code (Raibulet et al., 2017); our work is the first proposed meta-model including software protections and reverse engineering attacks.

Ontologies. A significant deal of work has been done by the scientific community in defining ontologies for cyber-security purposes. Herzog et al. (2007) presented an ontology in OWL to model vulnerability and threats on assets in network domains, with the relative countermeasures. The authors presented, alongside the ontology itself, a set of possible inferences that can be done on it, e.g., finding all the appropriate countermeasures for

a specific threat. They also show how to query the ontology using the SPARQL language. Ekelhart et al. (2006) developed another security ontology, built to simulate attacks against assets in corporate networks, in order to support a cost-based analysis of these threats. It is an extension of a previous work by Landwehr et al. (1994), where the authors created an ontology as a centralized KB of flaws for computer systems designers and security analysts. Costa et al. (2016) proposed a security ontology focused on the modelling of insider threats, e.g., potential malicious activities by legitimate users inside an organization. They also described a database of real life incident reports, named MERIT, built by the authors to validate the ontology against real life use cases.

Petri nets. Petri-nets are a super-set of state-transition diagrams, and their usefulness for attack modelling was pointed by McDermott as an alternative to attack trees (McDermott, 2000), as Petri nets are better at representing the actions of simultaneous attackers collaborating on the same attack. Traditionally attack trees have been the most common type of model for representing known attacks (Dewri et al., 2007) as a hierarchy of sub-goals leading to the final goal. Attack trees have been extended to attack graphs where nodes might have associated values or logical and/or conditions (Sheyner et al., 2002). Other proposals of attack graphs have emerged with different semantics and visual representation to document attack paths (Gupta and Winstead, 2007), analyse risks (Sheyner and Wing, 2003) or generating attack graphs from a PROLOG KB (Ou et al., 2006). Roy et al. (2012) proposed Attack countermeasure trees (ACT) to extend attack trees to take into account both attacks and protections. Attack trees and attack graphs lack a common standard for representing and exchanging models and the fact that they are subset of Petri Nets models made us choose the latter modelling for visual editing of attacks and exporting in standard PNML format.

Recently they have also been used to combine hierarchical Petri nets to model specific cyber-physical attacks on smart grids (Chen et al., 2011), while Wang et al. (2013a) focused on Petri net based attack modelling for software security where the attack step difficulty is ranked within five categories (from automated to fully manual). Xu and Nygard (2006) also models attacks with aspect-oriented Petri nets to superimpose protections as sub-nets to be interconnected with the attack model. Dalton et al. (2006) suggested generalized stochastic Petri nets for attack modelling; stochastic Petri nets are a type of timed Petri nets where transitions fire after random times. Coloured Petri nets (CPN) are used to design coloured Petri nets, where tokens represent different data types (colours) (Jensen, 1987); a similar open-source project is PIPE Petri Net editor and simulator (Dingle et al., 2009), however both tools cannot export the model to standard formats, making more complicated the conversion of their models towards standard formats like OWL.

6. Conclusions

This paper has presented a meta-model developed to describe the knowledge needed to perform risk analysis in the context of software protection against MATE attack scenarios that involve reverse engineering and tampering attacks. We discussed how the meta-model meets a set of concrete requirements, we discussed how existing models and taxonomies in the domain of software protection are covered, and we presented a range of tools that demonstrate the practical usefulness. Moreover, we provided a detailed use case analysis in the form of an instance of the meta-model filled in with the data from the risk analysis and mitigation of an open source software application. doi:10.1016/j.jss.2018.12.025.

Developing an automatic decision support system is the long term goal of our research, which we have started addressing with the ASPIRE project. There are several open issues to solve before such a system can be used in the real world. The most relevant one is the weak correlation between measurable characteristics of the software (protected and unprotected) with the empirical assessment of the effort needed to perform successful attacks.

One important result of the research in this field would be instantiating the meta-model with an as much as possible complete representation of the generic a priori information, to be shared with the software protection community. However, this goal will certainly face major issues. For political aspects (related to the adherence to a security by obscurity principle) companies do not share their data about protection assessment (e.g., weak points, attack paths against their protections).

We also foresee that the model may be extended in the future, e.g., to cover different software distribution formats, such as (more symbolic) bytecodes.

Acknowledgements

This research is supported by the European Union Seventh Framework Programme (FP7/2007-2013), project ASPIRE (Advanced Software Protection: Integration, Research, and Exploitation), under grant agreement no. 609734. The research by Bjorn De Sutter was also funded by the Fund for Scientific Research - Flanders (FWO), as part of the project 3G0E2318.

Supplementary material

Supplementary material associated with this article can be found, in the online version, at doi:10.1016/j.jss.2018.12.025.

References

Abrath, B., Coppens, B., Volckaert, S., Wijnant, J., De Sutter, B., 2016. Tightly-coupled self-debugging software protection. In: Proc. 6th Workshop on Software Security, Protection, and Reverse Engineering (SSPREW), pp. 7:1–7:10.

Ahmadvand, M., Pretschner, A., Kelbert, F., 2019. A taxonomy of software integrity protection techniques. Adv. Comput 112.

Anckaert, B., Madou, M., De Sutter, B., De Bus, B., De Bosschere, K., Preneel, B., 2007. Program obfuscation: a quantitative approach. In: Proc. ACM workshop Quality of protection, pp. 15–20.

Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S., Yang, K., 2001. On the (im) possibility of obfuscating programs. In: Advances in cryptology CRYPTO 2001. Springer, pp. 1–18.

Basile, C., Canavese, D., D'Annoville, J., De Sutter, B., Valenza, F., 2015. Automatic discovery of software attacks via backward reasoning. In: Software Protection (SPRO), 2015 IEEE/ACM 1st International Workshop on, pp. 52–58.

Basile, C., Cavanese, D., Regano, L., Falcarin, P., De Sutter, B.,. A meta-model for software protections and reverse engineering attacks: an instance of the meta-model. J. Syst. Softw. Under submission. doi:10.1016/j.jss.2018.12.025.

Basile, C., et al., 2016. ASPIRE framework report. Deliverable. ASPIRE EU FP7 Project.

Biham, E., Shamir, A., 1993. Differential cryptanalysis of the data encryption standard. Springer-Verlag, London, UK.

Cabutto, A., Falcarin, P., Abrath, B., Coppens, B., De Sutter, B., 2015. Software protection with code mobility. In: Proc. Second ACM Workshop on Moving Target Defense, pp. 95–103.

Canavese, D., Regano, L., Basile, C., Viticchié, A., 2017. Estimating software obfuscation potency with artificial neural networks. In: Security and Trust Management. Springer International Publishing, pp. 193–202.

Ceccato, M., 2016. ASPIRE security evaluation methodology. Deliverable. ASPIRE EU FP7 Project.

Ceccato, M., Capiluppi, A., Falcarin, P., Boldyreff, C., 2015. A large study on the effect of code obfuscation on the quality of java code. Empirical Softw. Eng. 20 (6), 1486–1524.

Ceccato, M., Tonella, P., Basile, C., Coppens, B., De Sutter, B., Falcarin, P., Torchiano, M., 2017. How professional hackers understand protected code while performing attack tasks. In: Proc. 25th Int'l Conf. on Program Comprehension, pp. 154–164.

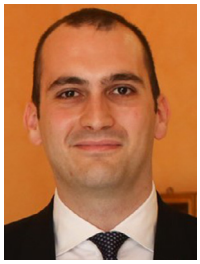
Ceccato, M., Tonella, P., Basile, C., Falcarin, P., Torchiano, M., Coppens, B., De Sutter, B., 2018. Understanding the behaviour of hackers while performing attack tasks in a professional setting and in a public challenge. Empirical Software Engineering.

Chen, T.M., Sánchez-Aarnoutse, J.C., Buford, J.F., 2011. Petri net modeling of cyber–physical attacks on smart grid. IEEE Trans. Smart Grid 2 (4), 741–749.

de Clercq, R., De Keulenaer, R., Coppens, B., Yang, B., Maene, P., de Bosschere, K., Preneel, B., de Sutter, B., Verbauwhede, I., 2016. SOFIA: software and control flow integrity architecture. In: Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 1172–1177.
Collberg, C., Thomborson, C., Low, D., 1997. A taxonomy of obfuscating transformations. Technical Report. Department of Computer Science, The University of Auckland, New Zealand.
Collberg, C., Thomborson, C., Low, D., 1998. Manufacturing cheap, resilient, and stealthy opaque constructs. In: Proc. 25th Symp. Principles of Programming Languages, pp. 184–196.
Coppens, B., et al., 2016. ASPIRE Open Source Manual. Deliverable. ASPIRE EU FP7 Project.
Costa, D., Albrethsen, M., Collins, M., Perl, S., Silowash, G., Spooner, D., 2016. An insider threat indicator ontology. Technical Report. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.
Dalton, G., Mills, R.F., Colombi, J.M., Raines, R.A., et al., 2006. Analyzing attack trees using generalized stochastic Petri nets. In: Information Assurance Workshop, pp. 116–123.
De Sutter, B., Falcarin, P., Wyseur, B., Basile, C., Ceccato, M., D'Annoville, J., Zunke, M., 2016. A reference architecture for software protection. In: 13th Working IEEE/IFIP Conf. on Software Architecture (WICSA), pp. 291–294.
De Sutter, B., et al., 2016. ASPIRE Validation Report. Deliverable. ASPIRE EU FP7 project.
Dewri, R., Poolsappasit, N., Ray, I., Whitley, D., 2007. Optimal security hardening using multi-objective optimization on attack tree models of networks. In: Procs. ACM Conf. Computer and Communications Security, pp. 204–213.
Dingle, N.J., Knottenbelt, W.J., Suto, T., 2009. PIPE2: A tool for the performance evaluation of generalised stochastic petri nets. ACM SIGMETRICS Perform. Eval. Rev. 36 (4), 34–39.
Ekelhart, A., Fenz, S., Klemen, M.D., Weippl, E.R., 2006. Security ontology: simulating threats to corporate assets. In: Bagchi, A., Atluri, V. (Eds.), Information Systems Security. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 249–259.
Falcarin, P., Collberg, C.S., Atallah, M.J., Jakubowski, M.H., 2011. Guest editors' introduction: software protection. IEEE Software 28 (2), 24–27.
Ghosh, S., Hiser, J.D., Davidson, J.W., 2010. A secure and robust approach to software tamper resistance. In: Information Hiding, pp. 33–47.
Gupta, S., Winstead, J., 2007. Using attack graphs to design systems. IEEE Secur. Privacy 5 (4), 80–83.
Herzog, A., Shahmehri, N., Duma, C., 2007. An ontology of information security. Int. J. Inf. Secur. Privacy (IJISP) 1 (4), 1–23.
Intel, I., Software guard extensions programming reference, revision 2, 2014.
Jensen, K., 1987. Coloured Petri nets. Advances in Petri Nets. Springer.
Kindler, E., 2011. The ePNK: an extensible Petri net tool for PNML. In: Applications and Theory of Petri Nets. Springer Berlin Heidelberg, pp. 318–327.
Kritikos, K., Massonet, P., 2016. An integrated meta-model for cloud application security modelling. Procedia Comput. Sci. 97, 84–93. 2nd International Conference on Cloud Forward: From Distributed to Complete Computing
Landwehr, C.E., Bull, A.R., McDermott, J.P., Choi, W.S., 1994. A taxonomy of computer program security flaws. ACM Comput. Surv. 26 (3), 211–254.
Mann, D.E., Christey, S.M., 1999. Towards a common enumeration of vulnerabilities. 2nd Workshop on Research with Security Vulnerability Databases, Purdue University, West Lafayette, Indiana.
McDermott, J.P., 2000. Attack net penetration testing. In: Proc.2000 Workshop on New Security Paradigms, pp. 15–21.
Mell, P., Scarfone, K., Romanosky, S., 2007. A complete guide to the common vulnerability scoring system version 2.0.
Mouelhiv, T., Fleurey, F., Baudry, B., 2008. A generic metamodel for security policies mutation. In: 2008 IEEE International Conference on Software Testing Verification and Validation Workshop, pp. 278–286.
Noorman, J., Agten, P., Daniels, W., Strackx, R., Van Herrewege, A., Huygens, C., Preneel, B., Verbauwhede, I., Piessens, F., 2013. Sancus: low-cost trustworthy extensible networked devices with a zero-software trusted computing base.. In: USENIX Security Symposium, pp. 479–494.
Ou, X., Boyer, W.F., McQueen, M.A., 2006. A scalable approach to attack graph generation. In: Proc. 13th ACM Conf. on Computer and Communications Security, pp. 336–345.
Peterson, J.L., 1977. Petri nets. ACM Comput. Surv. 9 (3), 223–252.
Radack, S., Kuhn, R., 2011. Managing security: the security content automation protocol. IT Prof 13 (1), 9–11.
Raibulet, C., Fontana, F.A., Zanoni, M., 2017. Model-driven reverse engineering approaches: a systematic literature review. IEEE Access 5, 14516–14542.
Regano, L., Canavese, D., Basile, C., Lioy, A., 2017. Towards optimally hiding protected assets in software applications. In: 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS), pp. 374–385.
Regano, L., Canavese, D., Basile, C., Viticchié, A., Lioy, A., 2016. Towards automatic risk analysis and mitigation of software applications. In: Information Security Theory and Practice. Springer International Publishing, pp. 120–135.
Roy, A., Kim, D.S., Trivedi, K.S., 2012. Attack countermeasure trees (ACT): towards unifying the constructs of attack and defense trees. Secur. Commun. Netw. 5 (8), 929–943.
Schrittwieser, S., Katzenbeisser, S., Kinder, J., Merzdovnik, G., Weippl, E., 2016. Protecting software through obfuscation: can it keep pace with progress in code analysis? ACM Comput. Surv. 49 (1), 4:1–4:37.
Sheyner, O., Haines, J.W., Jha, S., Lippmann, R., Wing, J.M., 2002. Automated generation and analysis of attack graphs. In: IEEE Symp. Security and Privacy, pp. 273–284.
Sheyner, O., Wing, J., 2003. Tools for generating and analyzing attack graphs. In: International Symposium on Formal Methods for Components and Objects. Springer, pp. 344–371.
Sommestad, T., Ekstedt, M., Holm, H., 2013. The cyber security modeling language: a tool for assessing the vulnerability of enterprise system architectures. IEEE Syst. J. 7 (3), 363–373.
Tonella, P., Ceccato, M., De Sutter, B., Coppens, B., 2014. POSTER: a measurement framework to quantify software protections. In: Proc. ACM SIGSAC Conf. Computer and Communications Security, pp. 1505–1507.
Viticchié, A., Basile, C., Avancini, A., Ceccato, M., Abrath, B., Coppens, B., 2016. Reactive attestation: automatic detection and reaction to software tampering attacks. In: Proc. 2016 ACM Workshop on Software PROtection, pp. 73–84.
Viticchié, A., Basile, C., Lioy, A., 2018. Remotely assessing integrity of software applications by monitoring invariants: present limitations and future directions. In: Risks and Security of Internet and Systems. Springer, pp. 66–82.
Välja, M., Korman, M., Shahzad, K., Johnson, P., 2015. Integrated metamodel for security analysis. In: 48th Hawaii Int'l Conf. on System Sciences, pp. 5192–5200.
Waltermire, D., Scarfone, K., Casipe, M., 2011. The Open Checklist Interactive Language (OCIL) Version 2.0.
Wang, C., Hill, J., Knight, J., Davidson, J., 2000. Software tamper resistance: obstructing static analysis of programs. Technical Report. Technical Report CS-2000-12, University of Virginia, 12 2000.
Wang, H., Fang, D., Dong, H., Lei, Y., Gong, X., Gu, Y., 2013. Software attack modeling and its application. In: 10th IEEE Int. Conf. High Performance Computing and Communications, pp. 1152–1158.
Wang, H., Fang, D., Wang, N., Tang, Z., Chen, F., Gu, Y., 2013. Method to evaluate software protection based on attack modeling. In: Proc. 10th IEEE Int. Conf. High Performance Computing and Communications, pp. 837–844.
Wheeler, D. A., 2001. More than a gigabuck: Estimating GNU/Linux's size.
Wojcik, M., Bergeron, T., Wittbold, T., Roberge, R., 2003. Introduction to OVAL: a new language to determine the presence of software vulnerabilities. Available online at http://oval.mitre.org.
Wyseur, B., 2008. White-box cryptography. KU Leuven.
Wyseur, B., De Sutter, B., et al., 2016. ASPIRE Reference Architecture. Deliverable. ASPIRE.
Xu, D., Nygard, K.E., 2006. Threat-driven modeling and verification of secure software using aspect-oriented petri nets. IEEE Trans. Softw. Eng. 32 (4), 265–278.
Zhang, G., Falcarin, P., Gómez-Martínez, E., Tartary, C., Islam, S., De Sutter, B., D'Annoville, J., 2016. Attack simulation based software protection assessment method for protection optimisation. In: Proc. Int'l Conf. Cyber Security and Protection of Digital Services, pp. 1–8.

Cataldo Basile received a M.Sc. (summa cum laude) in 2001 and a Ph.D. in computer engineering in 2005 from the Politecnico di Torino, where is currently a research associate. His research is concerned with software security, software attestation, policy-based security management in networked environments, policy refinement, and general models for detection, resolution and reconciliation of security policy conflicts.

Daniele Canavese received a M.Sc. degree in 2010 and a Ph.D. in computer engineering in 2016 from Politecnico di Torino, where he is currently a research assistant. His research interests are concerned with security management via inferential frameworks, software protection systems, public key cryptography and models for network analysis.



Leonardo Regano is a Ph.D. student and research assistant at Politecnico di Torino, where he received the M.Sc. degree in computer engineering in 2015. His current research interests focus on software security, applications of artificial intelligence and machine learning to cybersecurity, analysis of security policies, and assessment of software protection techniques.

Paolo Falcarin is a Reader in Computer Science at the University of East London, leading the Software Systems Engineering research group. He was awarded a Ph.D. degree in software engineering from Politecnico di Torino (Italy) in 2004. He was the General Chair of SPRO 2015 and guest editor of a special issue on software protection of IEEE Software. His research interests span over software engineering, distributed systems, software protection and security. He co-authored over 70 peer-reviewed papers in international conferences and journals and led the online protections activities in the ASPIRE project.

Bjorn De Sutter is professor at Ghent University in the Computer Systems Lab. He obtained his M.Sc. and Ph.D. degrees in computer science from Ghent University's Faculty of Engineering in 1997 and 2002. His research focuses on the use of compiler techniques to aid programmers with non-functional aspects of their software, such as performance, code size, reliability, and software protection. As for the latter, he developed techniques to mitigate reverse engineering, software tampering, code reuse attacks, fault injection attacks, and timing side channel attacks. He co-authored over 80 peer-reviewed papers in international conferences and journals. He coordinated the ASPIRE project.