Automatic selection of protections to mitigate risks against software applications

Computers & Security 168 (2026) 104959

Contents lists available at ScienceDirect

Computers & Security

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

Full length article

Automatic selection of protections to mitigate risks against software applications

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

[a] Istituto di Matematica Applicata e Tecnologie Informatiche "E. Magenes" (IMATI), Consiglio Nazionale delle Ricerche, Via de Marini, 6, 16149, Genova, Italy
[b] Dipartimento di Ingegneria Elettrica e Elettronica, Università di Cagliari, Via Marengo 3, 09123, Cagliari, Italy
[c] Dipartimento di Automatica e Informatica, Politecnico di Torino, Corso Duca degli Abruzzi, 24, 10129, Torino, Italy
[d] Computer Systems Lab, Ghent University, Technologiepark-Zwijnaarde 126, 9052, Gent, Belgium

ARTICLE INFO

Keywords:
Software protection
Man-at-the-End attacks
Software risk mitigation
Software potency and resilience

ABSTRACT

This paper introduces a novel approach for the automated selection of software protections to mitigate Machine-At-The-End risks against critical assets within software applications. We formalize the key elements involved in protection decision-making — including code artifacts, assets, security requirements, attacks, and software protections — and frame the protection process through a model inspired by game theory. In this model, a defender strategically applies protections to various code artifacts of a target application, anticipating repeated attack attempts by adversaries against the confidentiality and integrity of the application's assets. The selection of the optimal defense maximizes resistance to attacks while ensuring the application remains usable by constraining the overhead introduced by protections. The game is solved through a heuristic based on a mini-max depth-first exploration strategy, augmented with dynamic programming optimizations for improved efficiency. Central to our formulation is the introduction of the Software Protection Index, an original contribution that extends existing notions of potency and resilience by evaluating protection effectiveness against attack paths using software metrics and expert assessments. We validate our approach through a proof-of-concept implementation and expert evaluations, demonstrating that automated software protection is a practical and effective solution for risk mitigation in software.

1. Introduction

Software impacts many aspects of our lives these days. The business of companies developing software or creating or managing content and services with software depends to a large degree on the resistance of the software against so-called Machine-At-The-End (MATE) attacks (Falcarin et al., 2011).

In the MATE attack model, attackers have full access to the software and complete control over the systems on which they aim to reverse engineer the software and tamper with it to breach the security requirements of its assets. They can use various tools like simulators, debuggers, disassemblers, and decompilers. MATE attacks include reverse engineering (e.g., to steal algorithms or find vulnerabilities), tampering (e.g., to bypass license checks or cheat in games), or unauthorized execution (e.g., to run multiple copies with a single license).

Defenders can only rely on protections within the software or remote trusted servers to mitigate the MATE risks against their software. Hence, Software Protection (SP) refers to *protections deployed within that software* to secure its assets without relying on external services.

SP comes at a cost. It may add overhead to computation time, used memory and network bandwidth and may negatively impact the user experience. Mitigating risks from MATE attacks, hence, means selecting a set of SPs to be deployed on different parts of the application so that the attacker is delayed for a defender-defined time frame without degrading the performance over defender-defined acceptable levels.

As highlighted in the literature (Basile et al., 2023), SP today often lacks a formal risk analysis and relies heavily on security-through-obscurity. Experts manually select SPs, and their effectiveness and performance are assessed ex-post, i.e., only after deployment. Many challenges remain in achieving automated risk analysis of software. Formalization and automation are largely required as risk mitigation needs precision, i.e., the repeatability or reproducibility of obtained results (Basile et al., 2023).

Other research highlighted a significant skill gap (Goupil et al., 2022): there are not enough experts to protect all software that can benefit from rigorous SP; they are costly, hence SP is out of reach for Small and Medium-sized Enterprises (SMEs).

* Corresponding author.
E-mail addresses: daniele.canavese@cnr.it (D. Canavese), leonardo.regano@unica.it (L. Regano), cataldo.basile@polito.it (C. Basile), bjorn.desutter@ugent.be (B. De Sutter).

https://doi.org/10.1016/j.cose.2026.104959
Received 23 December 2025; Received in revised form 4 April 2026; Accepted 10 May 2026
Available online 16 May 2026

Automation is also needed as software vendors face time-to-market pressure. Every new version of an application needs to be protected. Part of the work on previous versions can probably be reused, but typically, the SPs at least need to be diversified. Additionally, software vendors may have to protect many versions, such as ports to different platforms, including mobile devices with limited computational resources. When proper protection would affect the application's usability due to SP's overheads, developers may decide to limit the features on those platforms. For example, media players with Digital Rights Management (DRM) will only access low-quality media versions if the platform does not allow full protection.

In this field, substituting human experts is not an easy job. The identification of the SP techniques to use, the parts of the software to protect and the configuration of the SPs are left to the 'feeling' of the team of experts operating on the code. Empirical studies aim at modeling the impact of protections against attacks (Ceccato et al., 2017; Viticchié et al., 2016b, 2020). All converge to the need for formal definitions of potency and resilience, the criteria introduced by Collberg et al. (1997), that allow estimating the effectiveness of SPs when applied on specific portions of a program. Moreover, even if human experts were available, latency would still be problematic. Automated tool support can cut the required time and effort.

In this context, our research aims to formalize, automate, and optimize the risk mitigation phase by developing methods to suggest a set of SPs to apply to different parts of the software to delay attackers without degrading performance over acceptable levels.

To address these questions and achieve our research goals, the contributions of this work are the following:

- a method to compute the effectiveness of protections when applied to software assets' requirements;
- a formal model for selecting the optimal SPs to mitigate risks against the vanilla application, constrained by an overhead threshold;
- a method for finding timely solutions to the above model.

Conceptually, the presented methods take as input a target application that is abstractly represented as a set of code *artifacts*, some of which are identified as *assets* for which *security requirements* must be preserved. The methods also receive a set of previously identified *attack paths* against those assets, and all the SPs available to counter them. To make decisions, the methods rely on data about attacks, such as the likelihood of successful completion and the level of attacker expertise required, and SPs, such as their effectiveness against specific attacks, and conflicts or synergies with other SPs. That information is application-independent and has been gathered a priori, in part through expert elicitation. Our methods (1) formalize the identification of candidate protections for each asset; (2) quantify the effectiveness of each protection solution through the *SP index*, a residual risk implementation that considers the impact attack paths and the mitigation of deployed protections; and (3) find the optimal SPs to safeguard the vanilla application, i.e., the ones that minimize the residual *SP index*, with a procedure inspired by game theory: the defender chooses protections under overhead constraints, while the attacker invests effort trying attack paths to compromise assets. We solve this problem with a bounded-depth minimax tree search algorithm, implemented as a depth-first exploration augmented with dynamic-programming optimizations, including pruning and caching.

The rest of the paper is organized as follows. Section 2 presents an overview of a MATE SP approach to manage MATE attack risks that we previously developed, to frame the novel contributions of this paper. Section 3 formally introduces the constructs used during the decision process. Section 4 describes the model, the algorithms, and the metrics for optimally selecting the mitigations. Section 5 describes how we consulted software protection experts and the inputs they provided for our approach. Section 6 presents a quantitative and qualitative validation of our models and tools. Section 7 relates our solution to the state of the art. Finally, Section 8 draws conclusions and sketches ideas for future work.

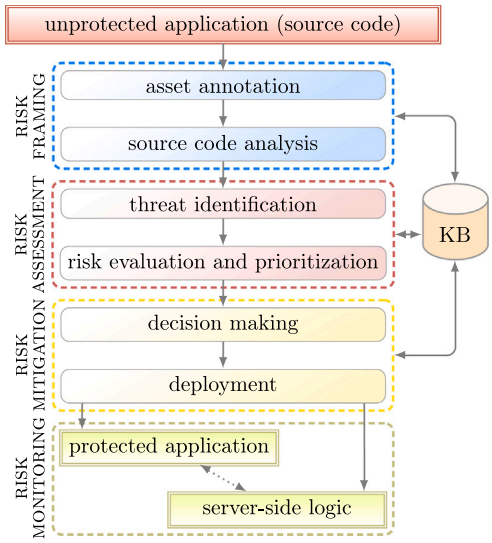


Fig. 1. The ESP workflow.

2. Overview of our approach

Protecting software against MATE attacks can be seen as a risk management process. The National Institute of Standards and Technology (NIST) has proposed an Information Technology (IT) systems risk management standard that identifies four main phases (Initiative, 2011):

1. *risk framing*: establish the scenario in which the risk must be managed;
2. *risk assessment*: identify threats against the system assets, vulnerabilities of the system, the harm that may occur if those are exploited, and the likelihood thereof;
3. *risk mitigation*: determine and implement appropriate actions to mitigate the risks;
4. *risk monitoring*: verify that the implemented actions effectively mitigate the risks.

Basile et al. (2023) discussed how this approach can be adopted for MATE SP. They argued that as much as possible of the four phases should be formalized and automated, and they presented results obtained with a prototype Expert system for Software Protection (ESP) that indeed automates much of the approach. Fig. 1 presents the semi-automated workflow of the ESP. This paper details a major contribution of the ESP, i.e. the automation of the risk mitigation phase.[1] Its complete code is available,[2] as well as a technical report on its inner workings (Basile et al., 2016b), a user manual (Coppens et al., 2016), and a demonstration video.[3] The ESP is primarily implemented in Java as a set of Eclipse plug-ins with a customized UI. It protects software written in C and needs source code access. The target users are software developers and SP consultants aiming to protect a given application.

2.1. Risk framing

In the risk framing phase, the ESP user must first annotate the code and data fragments in the application's C source code that require

[1] In the ASPIRE project and some cited papers, the ESP was called the ASPIRE Decision Support System (ADSS).

[2] https://github.com/daniele-canavese/esp/

[3] https://www.youtube.com/watch?v=pl9p5Nqsx_o

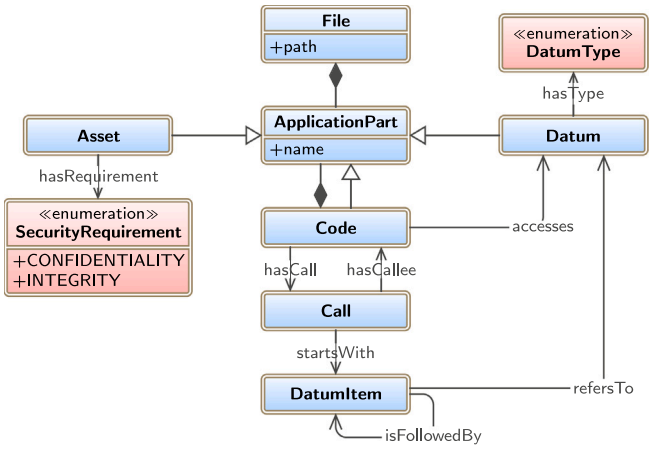


Fig. 2. The ApplicationPart class in the SP meta-model used in the ESP.

protection. In particular, the user can annotate variables, entire functions, or code regions (i.e., contiguous lines of code within the same function) as assets. These pragma and attribute annotations identify those fragments as assets and specify their security requirements, which currently include confidentiality and integrity. A formal specification of the annotation language is available online (Basile et al., 2016b).

Using the source code analysis capabilities of the Eclipse C Development Toolkit,[4] a formal representation of the whole application is then obtained from the source code, according to a software protection meta-model (Basile et al., 2019). In this meta-model, an application is modeled as a hierarchical structure of application parts that can be code regions or data elements (e.g., variables and parameters). The relations between those parts in the meta-model are captured in the model instances stored in a Knowledge Base (KB), including which code fragments access which data, and the call graph (see Fig. 2).

During the risk framing phase, a catalog of available SPs is also collected in the KB. This includes ordering requirements, restrictions, synergisms, and antagonisms of SPs. At the time of writing, the ESP supports Tigress, a source code obfuscator developed at the University of Arizona, and the ASPIRE Compiler Tool Chain (ACTC), which automates the deployment of SP techniques developed in the ASPIRE FP-7 project (Basile et al., 2016b; Coppens et al., 2016). Interestingly, the ACTC consists of two major obfuscation tools: a source-to-source rewriter to deploy several protections that are best implemented on source code, in which high-level semantic information is available, and a binary rewriter that disassembles object files to deploy complementary protections that can benefit from the additional degrees of freedom that assembly-level code rewriting comes with. For some protections, both protection tools collaborate. For example, for local attestation protections such as code guards, all necessary code is injected at the appropriate points in the source code, such that the compiler integrates them seamlessly into the existing application code, and the expected code checksum values are injected into the final executable by the binary rewriter, as they depend on the byte values of the compiled code. Table 1 summarizes the SP techniques supported by the ESP.

Furthermore, a catalog of possible attack steps is collected in the KB. This includes both preliminary and goal-oriented steps, such as setting up remote servers, locating and tampering with code or data statically or dynamically, executing code areas, injecting code, and tampering with the victim environment (Regano et al., 2016).

Finally, the KB is populated with all the information obtained through expert elicitation about the SPs and attack steps. Critically, this information is independent of the application and its assets. It is gathered from experts a priori, before targeting any specific application. It hence does not change from one application to another.

Example. As a concrete example of the information captured by the ESP in this phase, we consider one of the three use cases built by the industrial partners of the ASPIRE project. In particular, we consider a one-time Password (OTP) generator, which was already described in a previous publication detailing the ESP *risk mitigation* phase (Basile et al., 2015). The report produced by the ESP on this use case, comprising the assets, application parts, attacks, and mitigations, is available online[2], as is a demonstration video describing its architecture and assets.[5] The application runs on a client device and is used to generate OTPs for accessing a remote service, such as online banking or e-mail. During the provisioning phase, executed when the application runs for the first time, the client contacts a server and receives two random values, namely a seed and a counter. These values are transmitted in encrypted form so that only the client can decrypt and use them internally. In the same phase, the application also establishes the cryptographic material needed to protect the provisioning exchange and the obtained seed and counter, which will later be stored on the device. Whenever an OTP must be generated, the user is asked to enter a PIN. If the PIN verification succeeds, the application retrieves the protected data from storage, derives the needed keying material, generates the OTP from the seed and counter, and finally updates the counter in preparation for the next execution. Thus, from a high-level perspective, the most relevant elements for the attacker are the PIN verification logic, the OTP device key, the OTP counter value, and the key-derivation logic protecting the application secrets.

Starting from this attacker objective, the use-case owner identified, in the source code, the artifacts whose compromise could enable attacks, and annotated them in the application code as assets, together with the corresponding security requirement. For example, since an attacker may try to bypass the PIN-based access control by modifying the checking logic, the function `verify_pin` was marked as an asset with an integrity requirement. Likewise, since an attacker may try to recover the secret material needed to generate valid OTPs, the function `derive_storage_key`, which is used to generate at run time the key to decrypt the stored seed and counter, was marked as an asset with a confidentiality requirement. In total, 22 application parts were marked as assets for this use case by its owner.

2.2. Risk assessment

In the risk assessment phase, the threats to the assets are first identified. These threats are represented as a set of *attack paths* that attackers can try to execute. These paths, in turn, are ordered sequences of atomic attacker tasks called *attack steps*. Attack paths are equivalent to attack graphs (Phillips and Swiler, 1998) and can serve to simulate attacks, e.g., with Petri Nets (Wang et al., 2013). The attack steps that populate our KB originate from a study and taxonomy by Ceccato et al. (2017, 2019) and from data from industrial SP experts who participated in the ASPIRE project. In our ESP, the attack steps are rather coarse-grained, such as "locate the variable using dynamic analysis" and "modify the variable statically". Future work will address this limitation of our Proof of Concept (PoC) implementation.

The attack paths are built via backward chaining as presented in earlier work (Basile et al., 2015; Regano et al., 2016; Regano, 2019). An attack step can be executed if its premises are satisfied. It produces the results of its successful execution as conclusions. The chaining starts with steps that allow reaching an attacker's final goal (the breach of a security requirement) and stops at steps without any premise. The ESP then performs the risk evaluation and risk prioritization by assigning a *risk index* to each identified attack path. Every attack step in the

[4] https://projects.eclipse.org/projects/tools.cdt

[5] OTP use case demo: https://www.youtube.com/watch?v=O3H47zikzu8

Table 1
SPs supported by the ESP, with enforced security requirements and tools used to deploy the SPs. For each tool, we only mark techniques supported on our target platforms, i.e., Android and Linux on ARMv7 processors. For the SPs deployed by the ACTC, we distinguish between source-to-source (S2S) and binary-to-binary (B2B) transformations, while Tigress SPs involve only source-to-source transformations.

PROTECTION TYPE	REQUIREMENTS		TOOL		
	CONFIDENTIALITY	INTEGRITY	ACTC		TIGRESS
			S2S	B2B	S2S
anti-debugging	●	●	○	●	○
branch functions	●	○	○	●	○
call stack checks	○	●	○	●	○
code mobility	●	●	○	●	○
code virtualization	●	●	○	○	●
control flow flattening	●	○	○	●	●
data obfuscation	●	○	●	○	●
opaque predicates	●	○	○	●	●
remote attestation	○	●	●	○	○
white-box crypto	●	●	●	○	○

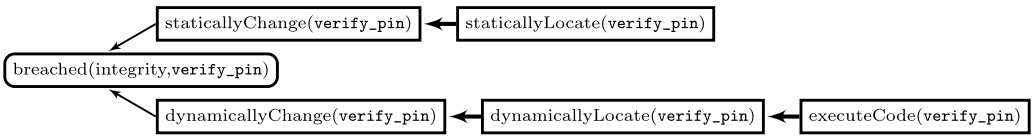


Fig. 3. Diagram showing the attack paths breaching the integrity of the `verify_pin` function of the OTP use case. Attack paths are represented as rectangles, while breaches of a security requirement are drawn as rounded rectangles. Links between attack steps are represented as thick arrows.

KB is associated with multiple attributes, including the complexity to mount it, the minimum skills required to execute it, the availability of support tools and their usability. Additional attributes can be added easily. Each attribute assumes a numeric value in a five-valued range. The values of complexity metrics and software features computed with the available analysis tools on the involved assets are used as modifiers on the attributes to assess the actual risks. For instance, an attack step labeled as medium complexity can be downgraded to lower complexity if the asset to compromise has a cyclomatic complexity below some threshold (McCabe, 1976). The risk index of an attack path is obtained by aggregating the modified attributes of its steps into a single value. Per attack step, our tool first aggregates all the step's modified attributes into a single attack step risk index. The attack path risk index is then computed from its steps' indices. For more details on this computation, we refer the reader to the existing work from Regano et al. (2016), Regano (2019).

Example. Continuing the OTP use case example introduced in the previous section, Fig. 3 provides an attack tree containing 2 attack paths generated by the ESP in this phase.[6] They endanger the integrity of the `verify_pin` function. Both attacks paths would allow the attacker to bypass the application's PIN-based access control to generate a OTP at their will. In one attack path, the attacker may statically analyze the application (e.g., manually inspecting the binary), to identify the function location in the application's binary. Then, they can change the binary (e.g., through a binary editing tool such as IDA Pro or Ghidra) in order to avoid the PIN check (e.g., returning a correct verification regardless of the inserted PIN). In the other attack path, they may execute the application attaching a debugger, stop the application when the PIN is requested, identify the function location in the binary (e.g.,inspecting the call stack), and then change the function's code through the debugger, again with the objective of avoiding the PIN check.

[6] In the use case report in the ESP code repository, these attack paths have respectively ID 730581298 and 402124250.

2.3. Risk mitigation

As is commonly done in MATE SP research, we assume that attacks cannot be prevented; they can only be delayed with the help of SPs. Hence, the mitigation process must select a set of SPs to apply to parts of the unprotected application, such that the attacker is delayed without degrading the application's performance beyond the defender-defined acceptable levels. The contribution of this paper is a method to automate this selection.

2.3.1. From risk index to software protection index

We model the delaying of attackers as lowering the risk index of their attack paths. The ESP has to find good candidate protection solutions to reduce those risk indices. To identify good candidate solutions, the ESP first searches for *suitable SPs*, i.e., SPs that are known qualitatively to impact attributes of the attack steps.

A solution is an ordered sequence of a number of SPs. In this context, an SP is not a conceptual construct or method such as "an opaque predicate". Instead, it refers to a concrete instantiation, meaning it is a concrete code transformation applied to a specific asset in a specific program by a specific SP tool that is configured with specific configuration parameters.

Each such SP is associated with a formula that can alter the attributes of each attack step. If an SP is deployed, the risk index of the attack steps and paths can hence be recomputed to assess the impact of the SP quantitatively.

For nearly three decades, software metrics have been used to model the strength of software protections quantitatively. In 1997, Collberg et al. proposed the use of software complexity metrics originating from the domain of software engineering for assessing the potency of protections (Collberg et al., 1997; Curtis et al., 1979), and others used quantitative metrics computed on the outputs of software analysis tools to assess the impact of protections on those tools' usefulness for attackers. Examples of the former are Halstead size (Halstead, 1977) and cyclomatic complexity (McCabe, 1976), examples of the latter are points-to-set sizes computed by data flow analysis tools (Foket et al., 2014), confusion factors of binary code disassemblers (Linn and Debray, 2003), and missing edges in function CFGs drawn by GUI

disassemblers (Van den Broeck et al., 2021). The first three example metrics can be considered general-purpose metrics, in the sense that they are relevant to many attack steps and are impacted by many protections. The last two examples are more special-purpose metrics, in the sense that they are relevant for only a limited set of attacker tools and that they are impacted by protections specifically designed for that reason. Our approach supports both general-purpose and special-purpose metrics.

Critically, our approach relies on potency, and hence on the metrics used to estimate potency, only to model the impact of the deployment of SPs on specific attack steps, not to model some form of inherent strength of the SPs. This is in line with the evolving understanding regarding the use of potency for evaluating SPs. Already in 2009, Nagra and Collberg (2009) revised the original definition of potency from 1997, now defining the potency of a SP in terms of considered program analyses. In 2025, De De Sutter et al. (2024) also defined a SP's potencies (plural) as its impacts on different attack steps. In our approach, this is achieved by considering a different combination of metrics for each type of attack step. Of all the used metrics, general-purpose and special-purpose, only the relevant ones are considered for each type of attack step.

In the ESP, the formulas used to recompute risk indices consider complexity metrics computed on the protected assets' code. Additional modifiers are activated when specific combinations of SPs are applied on the same application part. This way, the ESP models the impact of layered and synergetic SPs when recomputing the risk indices.

Candidate solutions must also meet cost and overhead constraints. Our PoC filters candidate SPs using five overhead criteria: client and server execution time overheads, client and server memory overheads, and network traffic overhead.

Finally, the *SP index* associated with a candidate solution is calculated based on the recomputed risk indices of all discovered attack paths against all assets, weighted by the assets' importance. The SP index is the ESP's instantiation of what is generally called residual risk.

Computing the SP index by recomputing the risk index requires knowledge of the metrics on the protected application. As applying all candidate solutions would consume an infeasible amount of resources, we have built a Machine Learning (ML) model to estimate the metrics delta after applying specific solutions without having to build the protected application (Canavese et al., 2017). The ESP's ML model has been demonstrated to accurately predict variations of up to three SPs applied on a single application part. With more SPs the accuracy starts to decrease; retraining the models with larger data sets, or leveraging more advanced ML techniques, would mitigate this issue.

The ESP uses the same predictors to estimate the overheads associated with candidate solutions. Per SP and kind of overhead, the KB stores a formula for estimating the overhead based on complexity metrics computed on the unprotected application.

2.3.2. Game-theoretic-inspired optimization method

The possibility, and in practice the necessity, of combining protections greatly increases the solution space. To explore it efficiently and to find high-quality solutions (w.r.t. the model) in an acceptable time, the ESP uses a method inspired by game theory, simulating a non-interactive SP game. In the game, the defender makes one first move, i.e., proposes a candidate solution for protecting all assets. Each proposed solution yields a *base SP index*, with a positive delta over the risk index of the vanilla application that models the solution's *potency*.

Then, the attacker makes a series of moves corresponding to investments of an imaginary unit of effort in one attack path, which the attacker selects from the paths found in the attack discovery phase. Similarly to how potency-related formulas of the applied SPs yield a positive delta in the SP index, we use *resilience*-related formulas that estimate the extent to which invested attack efforts eat away parts of protections and hence of their potency, thus yielding a decreasing SP index called the *residual SP index*. This use of resilience aligns with the framing of the potency and resilience terms in a recent survey on SP evaluation methodologies (De Sutter et al., 2024).

Fig. 4 shows a game tree for a scenario with three candidate solutions (S_1, S_2, and S_3) and two possible attack paths (K_1 and K_2) on two assets (α_1 and α_2) with security requirements (r_1 and r_2, respectively). Each node on the second row models a candidate solution. In a node labeled $s : p(p')$, s is the candidate solution, p is its residual SP index, and p' its base SP index.

The lower nodes model attack states. For example, the leftmost node on the bottom row models the state reached after a pre-order traversal of the path to that node, i.e., in the state after the attacker has invested in K_1 on α_1, in K_2 on α_1, and in K_1 on α_2. In each node labeled $k(\alpha, r) : p(p')$, k is the latest attack step, α the asset it targets with requirement r, p the state's residual SP index considering all succeeding attack steps included in the node's subtrees, and p' the state's residual SP index considering the already executed steps. It can be seen that each additional attack step decreases the p' value as it eats away at the SP index, and that each node's p is the minimum of its childrens' p' values, because the defender makes the worst-case assumption that the attacker will choose the optimal attack path. For leaf nodes, $p = p'$ shows only one residual SP index.

The directed edges in the graph mark the best attack paths on each candidate solution. The dark nodes mark the best candidate solution, i.e., the one with the highest residual SP index, as well as the best attack path on that solution. The top node of the graph, in essence, summarizes this info.

In Section 4, this whole optimization process and the used formulas will be discussed in more detail as some of the major contributions of this paper.

Example. Considering the OTP use case again, the best solution[7] proposed by the ESP is composed of 23 SPs deployed on the application's assets. Among those, to counter the attack paths depicted in Fig. 3, both breaching the integrity of the `verify_pin` function, the ESP proposes the use of the ACTC anti-debugging SP (Abrath et al., 2016) to prevent the attacker from attaching a debugger, a necessary step for executing the dynamic attack path. Furthermore, the ESP counter both attacks by proposing the use of the ACTC remote attestation SP (Viticchié et al., 2016a) to monitor changes to the `verify_pin` function. In this way, if an attacker is able to remove or bypass the anti-debugging SP, or to locate the asset with static analysis tools or with manual inspection of the binary, the subsequent tampering with the `verify_pin` function code will be detected by the remote attestation SP, which will trigger a user-defined reaction (e.g., halting the application) preventing the attacker from obtaining a valid OTP.

2.3.3. Deployment of candidate solutions

The ESP then proposes the best solution, possibly with a low number of comparably scoring alternative solutions, to the user, who can make the final selection. The user can still manually adapt the solution, e.g., fine-tuning some SP configuration parameters, and have the ESP generate the corresponding configuration files for the protection tools.

At that point, the user can simply invoke those tools (at the moment Tigress and the ACTC, see Section 2.1) on the application to actually deploy the selected solution in it. The result of this step (and of the whole workflow) is the protected binary plus source code for the server-side components for selected online SPs.

Optionally, the ESP can also be asked to deploy additional asset-hiding strategies. In practice, SPs such as obfuscations are never completely stealthy (Regano et al., 2017). Instead, they leave fingerprints. If only the assets are obfuscated, those fingerprints facilitate attack steps that aim to locate the assets. The ESP supports three asset-hiding

[7] In the use case report in the ESP code repository, this solution has ID 3561428950.

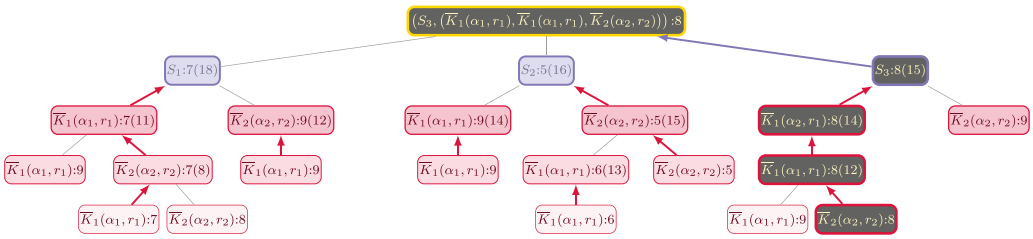


Fig. 4. Search tree example, computed with a mini-max algorithm and dynamic programming optimizations enabled. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

strategies to mitigate this and thus better hide the protected assets. In *fingerprint replication*, SPs already deployed on assets are also applied to other code parts to replicate the fingerprints such that attackers analyze more parts. With *fingerprint enlargement*, we enlarge the assets' code regions to which the SPs are deployed to include adjacent regions, so attackers need to process more code per region. With *fingerprint shadowing*, additional SPs are applied on assets to conceal fingerprints of the chosen SPs to prevent leaking information on the security requirements. We refer to existing papers (Regano et al., 2017; Basile et al., 2023) for more information on this aspect of the ESP's mitigation phase.

2.4. Risk monitoring

If the selected SPs include online SPs such as code mobility (Cabutto et al., 2015) and reactive remote attestation (Viticchié et al., 2016a), the ESP generates all the server-side logic, including the backends that perform the risk monitoring of the released application. This includes the untampered execution as checked with remote attestation and the communication with the code mobility server.

Our PoC does not automatically include the feedback and other monitoring data, such as the number and frequency of detected attacks, compromised applications, and server-side performance issues. The KB must be manually updated using GUIs to change risk framing data related to attack exposure and SP effectiveness. Issues related to insufficient server resources must also be addressed independently; the ESP only provides the logic, not the server configurations.

3. Formalization — the knowledge base

The KB contains the basic structures on which the algorithms operate that Section 4 will describe. These objects, relationships, and properties are based on the meta-model for SP (Basile et al., 2019) that was mentioned earlier in Section 2.1. This section discusses them in more detail in preparation for Section 4.

3.1. Artifacts

An *application artifact* a is a source code region, which consists of consecutive source code lines. The Application Artifacts (AAs) relevant to the mitigation form the artifact space $\mathcal{A}$. Two AAs are *joint* if they have at least one source element in common. This is denoted $a_1 \sqcap a_2$. Obviously, jointness is commutative ($a_1 \sqcap a_2 \iff a_2 \sqcap a_1$) and idempotent ($a \sqcap a$). In our model, we only consider AAs that are either completely disjoint, or one is included completely in the other, i.e., they are nested. We assume that a code normalization pre-pass has been performed, through which variable declarations and other statements, including subexpressions of interest, have all been put on separate lines. Hence variables correspond to proper AAs.

Variables need special care, however. For each variable, data flow analysis and alias analysis can identify the set of all source code lines that can directly or indirectly (via aliasing pointers) depend on the variable.[8] If this set of a variable's dependent AAs overlaps with some other AA, the variable is considered joint with that AA.

3.2. Security requirements and assets

A *security requirement*, denoted with r, is taken from the *requirement space* $\mathcal{R}$. Our experiments concerned with two security requirements: *confidentiality* and *integrity*. Our approach, however, is extensible and can support any other security requirement.

A *Protection Objective (PO)* is a pair $[r, a]$ that specifies the security requirement r of an AA a. All the POs belong to the *PO space* denoted with $\mathcal{O}$.

The *assets* of the application are the AAs that we ultimately need to protect. They appear in at least one PO pair and form the *asset set* $\mathbb{A} \subseteq \mathcal{A}$. For ease of notation, we will denote assets as $\alpha \in \mathbb{A}$, to easily tell them from non-assets artifacts $a \in \mathcal{A}$. All protection objectives are associated with a non-negative *weight* indicating their importance, which can be retrieved using the function $weight : \mathcal{O} \to \mathbb{R}_{\geq 0}$.

3.3. Protections

To protect the security requirements of the assets, the ESP will select *Concrete Software Protections (CSPs)* to be deployed on the assets. The CSPs are the protection instantiations implemented by the used SP tools and configured by the users of the tools. Each possible configuration of a supported SP is hence a CSP $p_{i,j}$.

The total *protection space* is the set $\mathcal{P}$ of all CSPs. We partition it into sets $P_i = \{p_{i,1}, \ldots, p_{i,n}\}$. Each such set models a single so-called *Abstract Software Protection (ASP)*: a set of CSPs that can be treated as one at certain points in our algorithms. For example, the SP types listed in Table 1 each correspond to one ASP.

This allows for a more concise expression of protection-related information in the KB, namely per ASP instead of per CSP, and it enables optimizations of the algorithms where they can reason per ASP instead of per CSP.[9]

[8] We assume that the used SP tools are conservative, in the sense that they will never deploy SPs in ways that alter the application's semantics. The AAs in our model are only used to determine which transformations will be requested from the tool. The alias analysis used to determine the set of a variable's dependent AAs can hence be unsound, and in the simplest case even be skipped. While this may yield a suboptimal selection of SPs when the selected ones are not applicable or composable, it cannot yield a non-conservatively protected program. Ideally, the SP tool and the decision support tool of course re-use exactly the same analyses, but this is no requirement.

[9] These benefits determine how to partition $\mathcal{P}$ into the ASPs P_i: if two CSPs can be considered equivalent with respect to the information and reasoning that is expressed at the level of ASPs, they can be added to the same partition. If not, they need to be stored in separate partitions and be considered different ASPs.

A CSP $p_{i,j}$ that is deployed on a specific AA a, is called a *Deployed Software Protection (DSP)* and denoted with $p_{i,j}(a)$. All the potential DSPs from the *DSP space* $\mathbb{D}$.

Most SPs can only be deployed on certain types of AAs. For instance, control flow flattening (László and Kiss, 2007) cannot protect variables but only code. We model whether an SP P_i is *compatible* with an artifact a with the Boolean function $compatible : 2^{\mathcal{P}} \times \mathcal{A} \rightarrow \{\top, \bot\}$. Furthermore, we model whether an SP affects a security requirement with the function $protect : 2^{\mathcal{P}} \times \mathcal{R} \rightarrow \{\top, \bot\}$. For instance, control flow flattening helps to preserve confidentiality but not integrity.

Dependencies between SPs applied to the same AA are captured with the following relations, which were inspired by the work by Heffner and Collberg (2004):

- *allowed precedence*: $P_1 \prec P_2$ indicates that P_1 *can* precede P_2, i.e., P_1 can be applied to some AA before P_2;
- *required precedence*: $P_1 \prec^R P_2$ denotes that P_1 *has to* precede P_2;
- *forbidden precedence*: $P_1 \not\prec P_2$ denotes that P_1 *cannot* precede P_2;
- *encouraged precedence*: $P_1 \prec^+ P_2$ indicates that P_1 *is suggested* to precede P_2, i.e., this order is particularly beneficial to the AA's protection. This implies $P_1 \prec P_2$;
- *discouraged precedence*: $P_1 \prec^- P_2$ denotes that P_1 *better not* precede P_2 because this combination negatively impacts the protection. This also implies $P_1 \prec P_2$.

Note that these relations only restrict the order in which SPs should be applied to some AA, not whether they need to be applied immediately after each other. For example, applying SPs P_1, P_2 and P_3 in that order to some AA is possible if $P_1 \prec^+ P_3$ holds. These relations can model various limitations, which may be due to an SP technique itself or to the used SP tool. For instance, the fact that some SP that can be applied at most once per asset (e.g., anti-debugging) can be formalized simply as $P \not\prec P$.

Dependencies can be expressed as regular expressions (Heffner and Collberg, 2004) and valid sequences of CSPs or ASPs can be generated accordingly. We exploited this property in our implementation.

3.4. Solutions

A *solution* S is an ordered list of DSPs $S = (p_1(a_1), p_2(a_2), \ldots)$ in the *solution space* $\mathcal{S}$.[10] The *vanilla solution* is the solution without any DSPs, and is represented as $\varnothing \in S$ for any application. DSPs and solutions are not inputs of algorithms in Section 4, they are outputs dynamically computed by our methods.

3.5. Metrics

Our methods and models rely on software metrics for estimating both the effectiveness of protection solutions and their overheads. General-purpose as well as special-purpose metrics are supported, and static metrics such as the mentioned as well as dynamic metrics such as profile information. All the metrics considered by the model are stored in a set M.

The optimization process has to examine numerous solutions. Since building binaries and measuring metrics is time-intensive, it is impractical to assess metrics on compiled binaries for all solutions to examine. Hence, in our model, we introduced an abstract function that predicts the value of the metrics after the application of the solution. Formally, the generic function $predict_m : S \times \mathcal{A} \rightarrow \mathbb{R}$ receives as input a solution and an artifact and returns the metric $m \in M$ of such artifact when it is protected with the solution's DSPs.

Our current PoC includes the following three general-purpose and three special-purpose metrics:

[10] CSPs have been indexed as $p_{i,j}$ when we were referring to partitions of the protection space. Here, the single indexed p_i is used to order CSPs in a solution.

Table 2
Metrics computed on the `verify_pin` function of the OTP use case, in both the vanilla version, and in a version obtained by deploying the SPs composing the ESP best solution for this use case.

Metric	Vanilla	Best solution
halstead	94	178
cyclomatic	6	12
instructions	31	73
instructions.remote	0	0
instructions.local	0	68
instructions.guarded	0	64

- *halstead*: the Halstead size of code AAs, i.e. their number of operators and operands (Halstead, 1977);
- *cyclomatic*: the cyclomatic complexity of code AAs, i.e. their number of linearly independent paths (McCabe, 1976);
- *instructions*: an AA's number of instructions (Halstead, 1977);
- *instructions.remote*: the number of instructions of an AA moved to a remote server by SPs such as code mobility (Cabutto et al., 2015) or client–server code splitting (Ceccato et al., 2007; Viticchié et al., 2020);
- *instructions.local*: the number of instructions of an AA that has been migrated from the main application process into additional local processes needed for protection purposes, such as the self-debugging code deployed as an anti-debugging protection (Abrath et al., 2016);
- *instructions.guarded*: the number of instructions of an AA guarded against tampering by tampering detection techniques such as remote attestation (Viticchié et al., 2016a).

Our PoC used the link-time rewriter framework Diablo (Van Put et al., 2005) to measure these metrics on the vanilla application.

Moreover, we have built a pool of ML models that implement the $predict_m$ function for the metrics supported in our PoC. Given a DSP $p(a)$ and the metrics values for the vanilla a, estimate the metrics values that would be obtained after the DSP has been applied (Canavese et al., 2017).

Example. Table 2 provides the metrics computed on the `verify_pin` function of the OTP use case, both on its vanilla version and on the version obtained by deploying the SPs contained in the best solution proposed by the ESP, i.e. the anti-debugging (Abrath et al., 2016) and remote attestation (Viticchié et al., 2016a) SPs provided by the ACTC. The *halstead*, *cyclomatic* and *instructions* metrics all increase due to the additional code introduced by the anti-debugging SP deployed on the asset. The anti-debugging also causes an increase in the *instructions.local* metric, because instructions are migrated to run in an external process, namely in the self-debugger that is key to this protection. Similarly, the *instructions.guarded* metric increases by the deployment of the remote attestation SP. Finally, we see no increase in the *instructions.remote* metric, because in this case no code from the `verify_pin` function itself was moved to a remote server.

3.6. Overheads

In our model, overheads are real numbers that correspond to ratios of performance metrics before and after protection with a given solution. Multiple performance metrics are supported because multiple types of metrics might be relevant (e.g., space, time, bandwidth) on different application parts (e.g., app initialization vs. later phases with real-time requirements, and client-side vs. remote server-side in the case of online SPs).

These overheads can be smaller than one, as some SPs can reduce metrics values. For instance, client–server code splitting can move AAs to a server (Ceccato et al., 2007), thus reducing the computational resources needed to execute the AA on the client.

Formally, the function $overhead_i : S \times 2^{\mathcal{A}} \to \mathbb{R}_{\geq 0}$ returns the type i overhead of a deployed solution on a set of artifacts. If this set is the whole program, the total overhead of type i is returned. By specifying these limits for sets of artifacts, the model supports expressing multiple, different constraints on different parts of the application.

The maximum allowed value for the overhead of type i on a set of artifacts A will be denoted by $\theta_i(A) \in \mathbb{R}_{\geq 0}$. To specify that we do not care about a specific type of overhead, we can write $\theta_i(A) = \infty$.

Our current PoC supports five types of overhead, the latter three of which are only considered when online SPs such as remote attestation are used in a solution:

1. client app computation time on sample inputs;
2. client app memory footprint on those inputs;
3. server computation time for online SPs;
4. server memory footprint for online SPs;
5. required network bandwidth.

To estimate the overhead $overhead_i$ function, our PoC measures the relevant metrics on the vanilla application and estimates them for the candidate solutions, because it cannot generate and measure that much binaries. The formulas used for this estimation were designed to estimate an upper bound on the overheads. Alternative techniques for obtaining more precise estimations are certainly useful (Alberto, 2021), but that is orthogonal to the rest of our methods.

3.7. Attacks

An *attack step* will be denoted by k, with all the known attack steps forming the set $\mathcal{K}$. When the generic operations implied by an attack step are performed on an AA $a \in \mathcal{A}$, we will write $k(a)$.

An *attack path* $K(\alpha, r)$ that endangers a security requirement r of an asset α is an ordered sequence of attack steps that the attacker executes on specific AAs a_i: $K(\alpha, r) = \big(k_1(a_1), k_2(a_2), \dots\big)$. It can be that $a_i \neq \alpha$ when a_i serves as a pivot to the attacker's goal α. The *attack path space* $\mathcal{K}(\alpha)$ includes all the attack paths against a specific artifact α.

The attack paths K in $\mathcal{K}$ and the attack steps k therein are abstract in the sense that they do not include a notion of effort. In other words, they model virtual attacks in which an attacker has infinite time to execute the steps of each attack, and therefore, succeeds in every step.

Since we aim at estimating how SPs delay attacks, we need to incorporate the concept of effort. Therefore, a *concrete attack path* $\overline{K}(\alpha, r)$ is the investment of a certain amount of effort in executing a sequence of *concrete attack steps* $\overline{k}_i(\alpha)$ to endanger the requirement r, which are the execution of the attack step k_i on the AA a_j for an imaginary unit of effort.[11] The set $\overline{\mathcal{K}}(\alpha, r)$ denotes the set of all possible concrete attack paths against the requirement r of α.

The concrete attack paths $\overline{K}(\alpha, r)$ against an asset are derived from the attack paths $K(\alpha, r)$. For an attack path $K(\alpha, r) = \big(k_1(a_1), k_2(a_2), \dots\big)$, such concrete attack paths are of the form $\overline{K}(a, r) = \Big(\overline{k}_1(a_1), \overline{k}_1(a_1), \dots, \overline{k}_2(a_2), \overline{k}_2(a_2), \dots\Big)$. When a step $\overline{k}_i(a_j)$ is repeated multiple times, it implies that more than one unit of effort is invested in it. A shorthand for a step $\overline{k}_i(a_j)$ that is repeated n times is to write $\overline{k}_i^n(a_j)$.

Not all concrete attack paths lead to the violation of a security property. For example, given the abstract attack path $K(\alpha, r) = \big(k_1(a_1), k_2(a_2)\big)$, investing in the concrete $\Big(\overline{k}_1(a_1), \overline{k}_2(a_2)\Big)$ may not be enough for compromising the security properties, while $\Big(\overline{k}_1(a_1), \overline{k}_1(a_1), \overline{k}_1(a_1), \overline{k}_2(a_2), \overline{k}_2(a_2)\Big)$ can instead lead to a successful attack.

We model the concept of *probability* of successfully mounting a concrete attack path on a protected asset α with the function $\Lambda : S \times \overline{\mathcal{K}}(\alpha, r) \to [0, 1]$. Such a path's success probability is computed on the success probability of its steps. To that extent, the *concrete attack step probability* $\pi_{\overline{k}^n(a)} \in [0, 1]$ is the probability of successfully executing the concrete attack step $\overline{k}$ on the unprotected asset α (i.e. on the vanilla application) with n imaginary units of effort.[12] In our PoC implementation, the base probability $\pi_{\overline{k}(a)}$ is a default value that only depends on the attack step k, not on the specific artifact a. The default value for each type of attack step modeled in our the KB was obtained through expert elicitation. The other values of n are obtained with formulas that asymptotically increase from a base probability to 1. Using a default value that is independent of the features of the attacked artifact is of course a simplification. In future work we foresee two options to improve this model: (i) in the framing phase of the overall approach, expert users of the ESP can override the value manually for specific assets if they know properties of the assets that make them easier or more difficult to attack; (ii) any user can invoke a set of analysis tools during the risk assessment phase that determines properties of the asset that make it easier to attack, e.g., with a toolbox such as TREX that automates the collection of different analyses results and simulation of different attack steps (Faingnaert et al., 2024).

Conceptually, π models the level of expertise of the envisioned attackers. By choosing the base probabilities $\pi_{\overline{k}(a)}$ in the framing phase of our approach, the ESP user can model different attacker capabilities, and hence configure the ESP to target different levels of adversaries. For example, when an ESP user is mostly interested in mitigating attacks by script kiddies, the base probability can be set to zero for attack steps that are considered out of script kiddies' reach, such as steps that require extensive manual configuration or customization. When guru-level adversaries are to be countered, high base probability values would be set instead. The default values elicited from experts for our PoC implementation assume guru-level adversaries, which is the worst-case scenario from a software vendor's perspective.

In addition, the *mitigation factor* $\zeta_{\overline{k}(a),p(a)} \in [0, 1]$ reduces the feasibility of executing the attack step $\overline{k}(a)$ when the DSP $p(a)$ is deployed.

The *synergy factor* $\omega_{\overline{k},p_i(a),p_j(a)} \in \mathbb{R}_{\geq 0}$ is used to model protections' precedences (see Section 3.3). If $p_i \in P_i$ and $p_j \in P_j$, then:

- $\omega_{\overline{k},p_i(a),p_j(a)} > 1$ if $P_i \prec^{-} P_j$;
- $\omega_{\overline{k},p_i(a),p_j(a)} < 1$ if $P_i \prec^{+} P_j$;
- $\omega_{\overline{k},p_i(a),p_j(a)} = 1$ otherwise (i.e. only $P_i \prec P_j$ holds).

In our PoC, the factors ζ and ω are constant parameters elicited from experts in the risk framing phase of our approach. In particular, $\zeta_{\overline{k}(a),p(a)}$ is fixed for each pair (k, p), while $\omega_{\overline{k},p_i(a),p_j(a)}$ is fixed for each pair (p_i, p_j). Hence, those factors do not vary with the characteristics of the protected artifact (nor the specific $\overline{k}$ for ω). Refining ζ and ω to take into account asset features is left for future work. The options to do so are similar to the two options for the success probabilities discussed above.

Finally, given a solution $S = \big(p_1(a_1), \dots, p_n(a_n)\big)$, the probability of a concrete attack path $\overline{K}(\alpha, r) = \Big(\overline{k}_i^{n_i}(a_i)\Big)_i$ is computed as follows:

$$\Lambda\left(S, \left(\overline{k}_i^{n_i}(a_i, r)\right)\right) = \prod_i \mu(a_i) \cdot \pi_{\overline{k}_i^{n_i}(a_i)}$$

where $\mu(a)$ is a function returning the combined effect of the mitigation (ζ) and synergy factors (ω) of all the SPs applied to a, so that:

$$\mu(a) = \prod_{p_x(a) \in S} \zeta_{\overline{k}(a),p_x(a)} \cdot \prod_{p_y(a) \in S} \omega_{\overline{k},p_x(a),p_y(a)}$$

[11] It is not useful for optimization purposes to give a precise value of the imaginary unit of effort, as is this just a fixed value to allow comparing the effectiveness of techniques.

[12] Our methods only require that the probabilities of successful attack steps are known to compute the protection indices. There is no need to formalize what it actually means to be successful, or even to define successful as the probability of success being 1.

4. Optimal selection

We leverage a method inspired by game theory to find the optimally protected application. In this scenario, we have two players: the defender, whose goal is to raise the application's security, and the attacker, who tries to diminish it. Our method works in two stages: a preparatory stage that precomputes the data structures needed for the second, exploratory stage that searches for the best solutions.

4.1. Preparatory stage

The *preparatory stage* is computationally inexpensive but helps improve the optimization process. It executes the algorithms for preparing the list of DSPs that can be considered for protection, as well as algorithms that partition the AAs into sets named Code Correlation Sets (CCSs), to speed up the optimization of the DSPs selection that will be applied on them.

Given the pair $[r, a]$, which includes a PO we want to enforce, we identify the set $\mathbb{D}_{[r,a]}$ of all the compatible DSPs implementing such PO, which is useful to explore the solution space:

$$\mathbb{D}_{[r,a]} = \{p(a) : p \in P \wedge compatible(P, a) \wedge protect(P, r)\}$$

The full DSP space can be trivially computed as the union of the individual $\mathbb{D}_{[r,a]}$.

Attack paths include attack steps that operate on different AAs (see Section 3.7). When two attack paths include attack steps operating on the same AA, one should consider countering all the involved attack steps for deciding how to protect the common AA. The idea is to partition the AAs into sets, named the CCSs, that permit dividing the attack paths so that we can protect artifacts in each set independently. The optimization problem is then split into smaller problems that are more manageable, as the game we propose is, in the worst case, exponential.

Formally, given an asset α and all the attack paths $\{K_i(\alpha, r)\}_i$ against it that have been determined during the risk assessment phase (see Section 2.2), we introduce the function $art : \mathcal{A} \to 2^{\mathcal{A}}$, which returns the set of all the artifacts involved in at least one attack step to compromise at least one security property of α. In other words, this function returns the AAs targeted by at least one attack step in any of the $K_i(\alpha, r)$.

If $art(\alpha_i) \sqcap art(\alpha_j)$ does not hold, protecting the two assets will have no interference.

By contrast, if $art(\alpha_i) \sqcap art(\alpha_j)$ holds, during the mitigation, the defender may have to deploy protections on common artifacts to mitigate both attack paths.

However, it is not enough to separate the attack paths that have no shared AAs using *art*; it is needed to build the closure to be able to partition the optimization problem. Hence, we use the *art* function to partition the artifact space $\mathcal{A}$ into a collection of non-empty *CCSs* $= \{\mathbb{A}^{\{1\}}, \mathbb{A}^{\{2\}}, \dots, \}$ where:

$$\mathbb{A}^{\{i\}} = \{\alpha_j \in \mathbb{A} : \exists \alpha_k \in \mathbb{A}^{\{i\}}, art(\alpha_j) \sqcap art(\alpha_k)\}.$$

Given this recursive construction, when protecting assets in a CCS, the assets in all the other CCSs will not be affected. Being partitions, CCSs satisfy the *coverage* (i.e. $\bigcup_i \mathbb{A}^{\{i\}} = \mathbb{A}$) and the *disjointness* (i.e. $\bigcap_i \mathbb{A}^{\{i\}} = \emptyset$). Hence, a solution S can be split into a set of *partial solutions* $S^{\{1\}}, S^{\{2\}}, \dots$ one for each CCS.

4.2. Exploratory stage

The goal of the *exploratory stage* is to find the optimal candidate solutions. A *state* $T = \left(S, \overline{K}_{\mathbb{A}}\right)$ is a pair consisting of a solution S and an ordered sequence of attack paths $\overline{K}_{\mathbb{A}} = \left(\overline{K}(\alpha_1, r), \overline{K}(\alpha_2, r), \dots\right)$ against the assets $\mathbb{A}$ to protect. We will indicate with $\mathcal{T}$ the *state space* so that $T \in \mathcal{T}$. The simplest state is $(\emptyset, \emptyset)$, which is the vanilla solution without any attacks; note also that the vanilla solution is valid for any application.

INPUT: the attack path space $\mathcal{K}_{\mathbb{A}}$, the solution space $\mathcal{S}$, a state $T = \left(S, \overline{K}_{\mathbb{A}}\right)$, and the maximal depth d
OUTPUT: the optimal state T' of the sub-tree rooted in T and its protection index p'

```
1  IF T = NIL THEN                                   // the defender's turn
2      p' ← −∞
3      FOREACH S ∈ 𝒮 DO
4          T̃, p̃ ← EXPLORE(𝒦_𝔸, S, (S, ∅), d − 1)
5          IF p̃ > p' THEN
6              p' ← p̃
7              T' ← T̃
8          END
9      END
10 ELSE IF d = 0 THEN                                // a terminal node
11     T' ← T
12     p' ← index(T)
13 ELSE                                              // the attacker's turns
14     p' ← ∞
15     FOREACH K(α) ∈ 𝒦_𝔸 DO
16         T̃, p̃ ← EXPLORE(𝒦_𝔸, S, (S, K̄_𝔸 ∪ K(α)), d − 1)
17         IF p̃ < p' THEN
18             p' ← p̃
19             T' ← T̃
20         END
21     END
22 RETURN T' and p'
```

Algorithm 1: EXPLORE.

Inspired by game theory, we devised an imaginary turn-based game with two players: the *attacker*, who will invest effort trying a variety of attack paths to compromise the security requirements of the application assets, and the *defender*, who will explore various solutions to protect them. Unlike traditional games like chess and checkers, however, the first turn is due to the defender, while all the remaining turns are for the attacker. This simulates the situation where a software house tries to publicly release a protected application and attackers have a certain amount of time to try multiple attacks before the value of assets in it decreases to irrelevant values.

This scenario can be represented with a tree such as the one depicted in Fig. 4. The first level of the tree contains the (blue) solutions (i.e. the defender moves, i.e., the candidate solutions). All the other (red) nodes are concrete attack paths (i.e. the attacker moves). Hence, any path from the root is a state as it includes a specific candidate solution and zero or more concrete attack paths that the attacker may mount to compromise the application when that candidate solution is applied. Every solution is associated with a base SP index (in parentheses) that is reduced every time the attacker executes a new attack path, yielding a residual SP index with the lowest value being reached at the leaves. The (black) *optimal state* contains the *optimal solution* that, after the attack phase, maintains the maximum residual SP index. Although the optimal solution is the most interesting information, the other information in the state is the sequence of the most dangerous (black) attack paths, which can also be useful when performing a more educated risk assessment of the application to protect.

Exploring the graph with depth-first search allows estimating residual SP indices under the attacker model and selecting the solution with the highest residual SP index. The simplest way to explore such trees is to use the recursive Algorithm 1 based on the traditional minimax depth-first exploration strategy used in chess programming (Borel, 1921; Claude, 1950). It receives the attack paths against the assets, the

solution space, the state to analyze T, and the maximum remaining depth of the tree to visit (which corresponds to the attack path moves still available to the attacker). It returns the optimal state T' of the sub-tree rooted in T and its SP index as outputs.

The SP index is a real number stating how safe a state is. As a rule of thumb, the defender wants to maximize the SP index of the application, while the attacker wants to minimize it.

To start the search from the tree root, the first call must be $\textsc{Explore}(NIL, \mathcal{K}_{\mathbb{A}}, S, d)$, with $d \geq 1$; this will return the optimal state after analyzing the entire tree. In the initial call, the loop of Lines 3–9 explores all the children of the root note, i.e., all possible moves in the defender's turn. For each of those moves, which correspond to the potential solutions, the recursive call on Line 4 explores the subtree corresponding to the attackers' answers. The answer with the highest residual SP index is then selected on lines 5–8 since the defender's goal is to maximize the application's security.

In the recursive calls, the code on Lines 10–21 is executed to optimize the attackers' answers and the compute the attacker subtrees. The algorithm first checks if the current state is a terminal node at Line 10. When a terminal node is found, the exploration stops, and the current state and its residual SP index are returned. Otherwise, at Line 13, the algorithm recursively explores all the attack paths and returns the state with the smallest residual SP index, as the attacker's goal is to compromise the application's security. For example, in the tree in Fig. 4, the optimal state is $\left(S_3, \left(\overline{K}_1(\alpha_1, r_1), \overline{K}_1(\alpha_1, r_1), \overline{K}_2(\alpha_2, r_2)\right)\right)$ with a protection index of 8. When the attacker plays, the attack with the lowest residual SP index is chosen as the 'winning move', and residual SP index is propagated upward until it reaches the blue (defender) nodes. Dually, the defender's goal is to pick the state with the highest residual SP index, and the optimal solution is propagated to the root. Algorithm 1's performance can be vastly improved by adopting a series of well-known dynamic programming optimizations. Namely, we implemented the following techniques:

- alpha–beta pruning (Slagle and Dixon, 1969): this variation skips large portions of the tree without impacting the final result;
- aspiration windows (Kaindl et al., 1991): this optimization explores a minimal portion of the tree by guesstimating the protection index range of the optimal state — this technique is particularly useful when securing a new version of an already analyzed application; thus, when the optimal solution of a similar model is known in advance;
- transposition tables (Breuker et al., 1997): they cache-like objects that store previously computed values related to the protection indices;
- futility pruning (Schaeffer, 1986), extended futility pruning (Heinz, 1998) and razoring (Birmingham and Kent, 1988): these reduction techniques aggressively prune forward some states if they seem unpromising.

When Algorithm 1 is adapted to incorporate these optimizations, the optimized algorithm can decide not to explore some children or sibling nodes, thus making the search tree asymmetric; in these cases, the protection index is computed also in some non-terminal nodes and used to perform an estimation whether or not it is useful to further explore a subtree.

Another simple yet effective optimization exploits the CCSs. Instead of building a single tree for the whole application, we build a tree for each CCS $\mathbb{A}^{\{1\}}, \mathbb{A}^{\{2\}}, \ldots$ as each CCSs are independent pieces of the application. The global optimal solution combines together all the optimal solutions for each CCS, and its residual SP index is computed accordingly. The Explore algorithm can be used without any modification; however, one caveat must be reported for our PoC. The requirements on overheads are global properties of an entire application and cannot be easily split according to the CCSs. Solutions to this issue are under investigation, in our PoC, we explicitly associated an overhead threshold θ_i with each CCSs. These thresholds were explicitly asked to the industry experts we consulted (as will be discussed in Section 5).

4.3. Iterating the solution space

The Explore algorithm requires an efficient manner to iterate through the solution space S, which can be too big to fit into memory.

In our PoC, we decided to explore the solution space S by generating the next solution the mini-max algorithm must process. The generation algorithm receives as input a solution S, the POs, and DSPs spaces, and a user-defined integer constant σ specifying the maximum number of DSPs to use per each PO. It returns the next candidate solution to explore S' or NIL if S has been fully explored.

The algorithm is iterative and does not require storing the entire solution space in memory. However, it needs a starting point. The vanilla solution $\varnothing$ is always a valid starting solution for any application. Alternatively, based on a request of the industrial partners in the ASPIRE project, the ESP offers expert users the option to feed the algorithm with their manually crafted candidate solutions. This allows them to use the ESP to gather feedback on those solutions. Importantly, this feature was not evaluated experimentally: in all assessments reported in Section 6, the exploration of candidate solutions started from the vanilla solution, not from a user-provided solution.

The function that generates the next solution works in three steps.

1. First, it generates a permutation of the DSPs in the input solution, i.e., it changes the order of the protections in the input solution. It uses the lexicographic permutations with restricted prefixes algorithm (Knuth, 2011) to correctly take into account protections' precedence and may also exclude discouraged combinations. Further, combinations that exceed the overhead thresholds are discarded.
2. Then, it fuzzes the DSPs in that permutation, that is, it adds, removes, or replaces some DSPs with some other ones and checks that all the precedences are satisfied.
3. Finally, it selects a subset of the DSPs at the previous step and returns the solution including them (which may contain more, less or the same number of DSPs of the input solution.

Trivially, the algorithm returns NIL, signaling that the solution space has been fully explored.[13]

4.4. The software protection index

The *SP index* is computed with the function $index : \mathcal{T} \to \mathbb{R}$. Similarly to Collberg's potency, the key property of our protection index is that if a state T_1 is more secure than another state T_2, then $index(T_1) > index(T_2)$ must hold, thus permitting comparison and ranking of candidate solutions within the model. In addition, the sign of an SP index allows us to infer some traits of a state $T = \left(S, \overline{K}_{\mathbb{A}}\right)$:

- if $index(T) = 0$, the state T is without protections from the attacks in $\overline{K}_{\mathbb{A}}$, as for the vanilla application $index((\varnothing, \varnothing)) = 0$,
- if $index(T) > 0$, the state T is mitigating the risks against the application, also when the attacker is investing in the attacks in $\overline{K}_{\mathbb{A}}$;
- if $index(T) < 0$, the security of T is compromised by some attacks in $\overline{K}_{\mathbb{A}}$.

Security is a multi-faceted aspect of a protected application; thus, to compute the SP index, we decided to use multiple quantifiable security characteristics named *security measures*.

The functions $measure_i : \mathbb{A} \times \mathcal{T} \to \mathbb{R}_{\geq 0}$ return the value of the ith security measure of an asset in a particular state. We identified four security measures that stem from how SPs work:

[13] Even if the solutions are not saved, the used algorithms do not generate duplicates. Hence, we know the generation has been completed by only maintaining counters.

- $measure_{CC}$: The *code comprehension* measure estimates how hard it is to understand (local) code. SPs such as obfuscations increase it, while attacks such as deobfuscation attempt to decrease it.
- $measure_{CT}$: The *code transfer* measure estimates how much code has been moved to a remote trusted server, thus making it unavailable for reverse engineering on a local machine. For instance, code mobility raises this measure, while attacks on the application's dependence on the remote server (e.g., by reconstructing its funtionality locally) lower it.
- $measure_{TD}$: The *tampering detection* measure evaluates how effective a protection is in detecting an integrity failure. As an example, remote attestation boosts this measure while circumventing or bypassing such a protection reduces it.
- $measure_{TA}$: The *tampering avoidance* measure assesses how effective a protection is in making (static or dynamic) tampering harder. For instance, anti-debugging increases this value, while removing such a technique decreases it.

The code comprehension and transfer measures are related to code confidentiality, while tampering detection and avoidance are related to integrity.

All these measures have different relations with the complexity metrics and protections selected, which are captured by our formulas. An increase in all the static metrics values, e.g., after obfuscation, has a positive impact on protection, as it is supposed to make code comprehension tasks harder. Decreasing the code size due to the application of some server-side protections, like client–server code-splitting, has a positive impact on the code transfer measure. On the other hand, the application of remote attestation is unrelated to the static complexity metrics as it only depends on the technique used and the number and types of attestation checks inserted. The general formula for computing the SP index of a state $T = (S, \overline{K}_{\mathbb{A}})$ is:

$$index(T) = \sum_{\alpha \in \mathbb{A}} \left(weight(\alpha) \cdot \left(\sum_i measure_i(\alpha, T) \right) \right).$$

We computed the ith measure using the equation:

$$measure_i(\alpha, T) = \tau_i \cdot measure'_i(\alpha, T) - \rho_i \cdot H(\epsilon_i - measure'(\alpha, T)).$$

This formula leverages $measure'_i(\alpha, T)$, an *adjusted measure* that takes into account the effects of DSPs and attack paths on the asset α and the Heaviside step function H. The adjusted measure is multiplied by $\tau_i \in \mathbb{R}_{\geq 0}$, a custom weight introduced to allow us to fine-tune the importance of each measure. The second part of the formula subtracts a large constant $\rho_i \in \mathbb{R}_{\geq 0}$ whenever the adjusted measure is less than $\epsilon_i \in \mathbb{R}_{\geq 0}$. This subtraction allows marking states for which assets have been breached so that the search algorithm will avoid them.[14]

To compute the adjusted measures, we will make use of the equation

$$measure'_i(\alpha, T) = \left(\prod_{K(a) \in \overline{K}_{\mathbb{A}}} (1 - \Lambda(S, K(a)) \right) measure_i(\alpha, (S, \varnothing))$$

This equation uses $measure_i(\alpha, (S, \varnothing))$, the ith adjusted security measure computed only on the solution S without any attack path. The attack path's influence is instead taken into consideration with the multiplicative factor using the Λ function.

To compute $measure_i(\alpha, (S, \varnothing))$, we use the utility function $H'(x) = H(x) \cdot x$ to simplify some formulas, a variety of complexity metrics and the notion of Collberg's potency (Collberg et al., 1997) $\mathcal{P} : \mathbb{A} \times S \to \mathbb{R}$. The potency is a value stating how well an artifact is protected and, given the metric m (see Section 3.5), it can be expressed as:

$$\mathcal{P}_m(a, S) = \frac{predict_m(a, S)}{predict_m(a, \varnothing)} - 1.$$

Using these definitions, we computed the four adjusted security measures with the following formulas[15]:

$$\begin{cases} measure_{CC}(\alpha, (S, \varnothing)) = H'(\mathcal{P}_{\text{halstead}}(\alpha, S) + \mathcal{P}_{\text{cyclomatic}}(a, S)) \\ measure_{CT}(\alpha, (S, \varnothing)) = \dfrac{predict_{\text{remote.instructions}}(\alpha, S)}{predict_{\text{instructions}}(\alpha, \varnothing)} \\ measure_{TD}(\alpha, (S, \varnothing)) = \dfrac{predict_{\text{guarded.instructions}}(\alpha, S)}{predict_{\text{instructions}}(\alpha, S)} \\ measure_{TA}(\alpha, (S, \varnothing)) = \dfrac{predict_{\text{local.instructions}}(\alpha, S)}{predict_{\text{instructions}}(\alpha, S)}. \end{cases}$$

5. Expert elicitation

To instantiate the quantitative optimization approach that we described conceptually in the previous sections, many functions, formulas, factors, parameters, and weights need to be instantiated, i.e., concrete values need to be chosen during the framing phase of our overall approach to populate the KB. However, known models of MATE attacker behavior (Ceccato et al., 2017, 2019) and reverse engineering in general (Mantovani et al., 2022; Votipka et al., 2019) are qualitative. In other words, the literature offers no established, comprehensive quantitative models of how SPs affect the attacker's performance as needed for our approach. We hence collected the necessary inputs from SPs developers and industry experts involved in the ASPIRE project; from other experts from the project consortium partners, i.e., that were not performing or assisting the research in the project); and from experts in the Advisory Board.

Importantly, all of the expert elicitation used to determine the values to populate the KB took place before the evaluations of the ESP were executed as reported in Section 6. In other words, the information obtained from the experts was not biased by experience with (early versions of) our PoC implementation. Moreover, all elicited information was aggregated before populating the KB, and the same aggregated (and hence domain-neutral) values were used for all use cases reported in Section 6.2.

5.1. Consulted experts

We can broadly distinguish two types of experts that have been consulted through structured interviews.

SPs Developers. This category of experts developed SPs, like obfuscation tools, code guards, software attestation techniques, and code mobility, as reported in a project deliverable (Basile et al., 2016b). These experts were primarily asked to answer surveys about the SPs they developed, the security requirements they help preserve, the attackers' activities their techniques impact, and the dependencies with other SPs, i.e., limitations on composability and potential synergies between them. Moreover, they participated in the surveys related to the definition of the SP Index function.

[14] In chess, this is the equivalent of a checkmate. However, in chess, all checkmate configurations are equivalent, so their score can be set to $-\infty$. By contrast, we need to differentiate a state with a security breach from another state with two breaches, so we cannot set all their SP indices to the same value.

[15] $measure_{CT}$ is normalized against $predict_{\text{instructions}}(\alpha, \varnothing)$ because it estimates how much code from the vanilla artifact has been transferred to a remote trusted server; hence, the relevant baseline is the original unprotected code size. By contrast, $measure_{TD}$ and $measure_{TA}$ are normalized against $predict_{\text{instructions}}(\alpha, S)$ because they quantify protection coverage on the currently protected artifact. In particular, tampering-detection and tampering-avoidance protections may add instructions (e.g., attestation checks or anti-debugging code), and the measures should therefore be interpreted relative to the instruction count of the protected artifact rather than the vanilla one.

Industry experts. This category of experts includes researchers and practitioners, often with strong backgrounds in offensive tasks and domain knowledge of the use cases on which the project artifacts were evaluated. They worked on designing and analyzing the use case applications to protect, on selecting the proper SPs as mitigation (as human experts do when no decision support tools are available to automate the selection), on deploying the techniques, and eventually on evaluating empirically whether the protected binaries met the security requirements at an acceptable overhead. They were asked to answer surveys on the SPs they used for their jobs, following the same approach as the SPs developers. Moreover, they were interviewed to acquire empirical information, like expert evaluation of the effectiveness of SPs, complexity of attack steps, and relations among complexity metrics and attack steps and SPs. In short, they were the primary source of information for building the SP index formulas.

5.2. Elicitation coverage

The inputs obtained from the experts cover many areas.

Suitability to preserve security requirements (Section 3.3)

SPs developers' feedback was used to build the *compatible* function and, together with industry experts' feedback, to define the *protect* function.

Relations among SPs (Section 3.3)

SPs developers and industry experts helped to formally model the relations between SPs to the same AAs (allowed, required, forbidden, encouraged, discouraged). These relations have been assessed using *ad hoc* surveys where they were asked to evaluate them on a three-level scale. This information was complemented with the existing literature in the field and with the results of empirical experiments we conducted to assess the efficacy of selected SPs (Viticchié et al., 2016b,a). These data helped build the function $\omega^{-}_{k,p_i(a),p_j(a)}$, i.e., the *synergy factor*.

Metrics, SP's and attack complexity relations (Section 3.5)

SP developers were first asked to indicate the metrics that were affected by the application of their SPs. Then, together with industry experts, they were asked to estimate the impact of variations in the metrics on the complexity of specific attack steps. These data were used to build the $measure_i(\alpha, T)$, the parameters τ_i to relate the importance of the metrics, and $measure'_i$ functions.

Relations between metrics and SP overheads (Section 3.6)

SPs developers answered surveys to determine the association between the metrics and the overheads for the SPs they owned. This feedback was also used to build the formulas that estimate overheads based on the results of the ML predictors. Moreover, they helped determine the overhead thresholds and their split into CCSs.

Relations between SP s and attack steps (Section 3.7,3.5)

Experts were first asked to evaluate the complexity of individual attack steps, regardless of the presence of SPs. This information served to build the base attack steps probability $\pi^{-}_{k(a)}$. Then, they were asked to indicate the impact of the SPs in countering the attack steps, which resulted in the mitigation factor $\zeta^{-}_{k(a),p(a)}$. We also collected information about attack steps able to weaken specific SPs, which was used to estimate the effectiveness of SPs and helped build the formulas estimating the probability of successfully mounting an attack path against the security requirement of an asset, i.e., the Λ function. Moreover, this experts' feedback was used to build the resilience-related formulas used during the game-theoretic optimization to estimate the extent to which invested attack efforts eat away parts of the SP potency, thus decreasing the SP index.

Table 3
Computational complexity of the approach.

STAGE	ALGORITHM	COMPLEXITY
preparatory	determine deployed SPs	linear
preparatory	compute code correlation sets	quadratic
exploratory	explore search tree	exponential

Table 4
Code correlation sets in our use cases. For each use case, the columns indicate respectively the total number of POs contained in the use case, the total number of computed CCSs, and the range and mean of POs contained in each CCS.

USE CASE	PO	CCSs		
		COUNT	PO RANGE	PO MEAN
demo player	58	27	1–2	1.15
license manager	59	26	1–7	1.65
OTP generator	24	17	1–4	1.29

6. Validation

The first validation step we have performed is a theoretical complexity analysis. Table 3 summarizes the results; the full results are reported in the Supplemental material, where we also list the pseudo-code for all our relevant algorithms. In the worst case, the search tree algorithm has an exponential upper bound complexity. This is the case with and without enabling the dynamic programming optimizations reported in Section 4.2.

To assess whether this high theoretic complexity impacts the feasibility of the approach, Section 6.1 will present an experimental evaluation of the runtime usability of the optimization method and the introduced heuristics, showing that in practice, the PoC implementation completes in minutes. Next, Section 6.2 will report a summary of the qualitative evaluation by SP experts.

6.1. Quantitative evaluation

We tested our PoC (written in Java) on a virtual machine running on 4 cores of an 11th gen Intel® Core™ i9-11950H@2.60 GHz and 8GB RAM with Ubuntu 18.04.2 LTS and OpenJDK version 11.0.4 2019-07-16.

Figs. 5 and 6 show the time to find the best solution returned by the bounded-depth search in a variety of applications depending on the POs and the concrete attack paths. The times have been computed on search trees of depths 3, 4, 5, and 6. We have chosen these depth values considering that we used a tree depth of 3 in the qualitative validation reported in Section 6.2, with which we obtained results considered satisfactory by the SP experts involved in the validation. Furthermore, we considered the numbers of POs and attack paths ranging from 4 to 512, which we consider reasonable for real-world applications. For example, as reported in Table 4, the use cases devised to perform the qualitative validation range from 24 to 59 POs.[16] In addition, we enabled all the supported protections (the complete list is available in the Supplemental material).

The plots show that the trend is exponential in the number of POs, concrete attack paths, and search depth. The latter is the most impactful factor on the execution time since we are increasing the tuple length

[16] Notice that the number of POs in an application is larger than the number of high-level assets to protect in them. For example, when a high-level asset such as a *license manager* needs to have its integrity protected, the multiple functions that implement it all become individual POs. The number of high-level assets ranged from 5 to 8 in the use cases used in the qualitative evaluation.

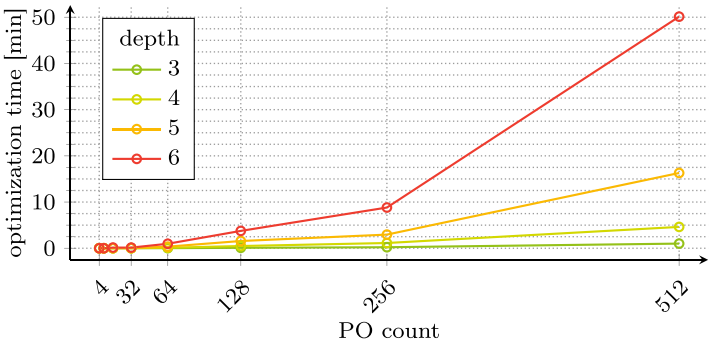


Fig. 5. Optimization time vs. number of POs.

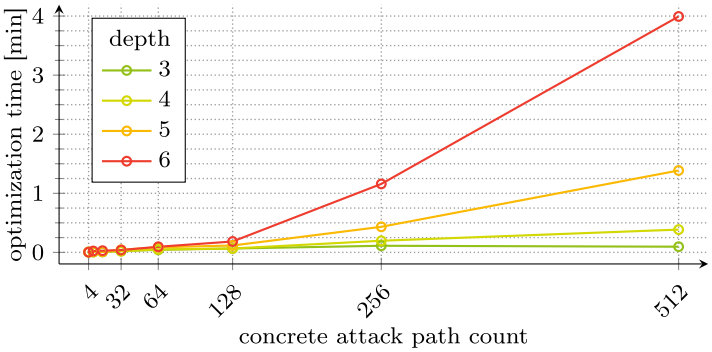


Fig. 6. Optimization time vs. number of attack paths.

of the concrete attack paths to be analyzed. Interestingly, the PO count affects the search time more than the concrete attack path count. This is because the number of POs affects the first tree level and, hence, indirectly, also the whole tree, while the number of concrete attack paths affects all the levels except the first one.

We note that the number of POs, concrete attack paths, and the search depth primarily influences the computational time. The actual size of the assets (e.g., the source lines of code) does not affect the running time, nor the number of AAs that are non-assets.

Furthermore, the use of CCSs can mitigate the exponential nature of the search tree algorithm, as this heuristic allows the execution of the algorithm multiple times on smaller sets of POs. Table 4 shows the size of the CCSs computed on our use cases used for the qualitative validation described in Section 6.2. The obtained CCS contain at most 7 POs, with a mean value of less than 2 POs per CCS in all use cases. Fig. 5 indicates that, for such a small number of POs, the execution time of the search tree algorithm was in the order of seconds.

6.2. Qualitative validation

This qualitative evaluation reports experts' opinions about our PoC implementation. It was collected from experts at industrial partners in the ASPIRE project consortium and its advisory boards. As reported in Section 2, our PoC covers the whole process of protecting a Vanilla Application (VA). In this section, we focus on evaluating the mitigation phase, which implements the technique described in Section 4. A comprehensive evaluation of the entire PoC is provided in another article (Basile et al., 2023), which frames our research within the IT Design Science Framework (Hevner et al., 2004) and seeks to apply the NIST Risk Management Framework (SP 800-39 (Initiative, 2011)) to advance towards a standardized MATE risk management approach. The IT Design Science Framework enabled us to scientifically validate our research results and the effectiveness of the automatic selection of software protections, despite certain limitations and threats to validity, as described in Section 6.4.

The evaluation process objects were three Android apps designed and implemented by the project's industrial partners to represent their commercial software: a One-Time Password (OTP) generator for home banking apps, a software licensing manager (SLM), and a DRM-enabled video player for protected content. These apps included security-sensitive assets in dynamically linked C libraries, which were only made available to academic partners. The high-level descriptions of applications and assets were disclosed in a project deliverable (Basile et al., 2016a) to confirm they are not toy examples. Demonstration videos of these apps, including descriptions of their architecture and of the assets embedded in them, are available online.[17] These three use cases only have in common that they are all examples of application/protocol-level security mechanisms, which are the prime examples of software functionality that embeds assets that come with confidentiality and integrity requirements. The three use cases do not share a single line of source code, they have been developed independently within three different companies to be representative of those companies' commercial software at the time of the ASPIRE project,[18] and they feature very different architectures. For example, the OTP and SLM use cases consist of libraries that are linked into end-user applications, while the DRM use cases consists of libraries loaded into the Android's DRM server processes to which end-user applications connect via standard Android APIs.

Each of the three ASPIRE industrial partners involved two experts to validate their own use cases: one internal expert (i.e., actively involved

[17] DRM: https://www.youtube.com/watch?v=iV-ATSShd7A, SLM: https://www.youtube.com/watch?v=MPyey16rh5I, OTP: https://www.youtube.com/watch?v=O3H47zikzu8

[18] The companies did not want to share the source code of their commercial products with academics, so they created use cases specifically for the project, with the goal of not risking leakage of confidential information while still getting evaluation and validation results that would generalize to their commercial offerings.

in the project) and one external expert (i.e., not participating in the project). The evaluation was organized into three consecutive phases:

1. *Early Internal Expert Assessment*: During the PoC development, the protection owners were involved in evaluating if their individual SPs were used on the proper assets and in the correct way to build solutions. Moreover, internal experts provided continuous feedback on the PoC models, reasoning processes, and results. Their feedback drove the development of the PoC, resulting in the alpha version, which was thoroughly tested by internal experts. In particular, they were involved in demos. When the PoC was stable enough, they used it to protect their use case, analyzing and commenting on the results, including the solutions proposed by the PoC and their protection indices.
2. *Final Internal Expert Assessment*: Near the end of the project, internal experts were asked to test the PoC's first stable version. They used the PoC's GUI to protect their use case. Moreover, they evaluated the tool's maturity, answering a set of open-ended questions. Such answers were then discussed in multiple calls among internal experts, protection owners, PoC developers, and the coordinator (including the authors of this paper). The internal experts' comments and suggestions were incorporated into the final version of the PoC.
3. *Assessment with External Experts*: The PoC's final version was finally tested by external experts, who had never before used the PoC nor had they any information on its internal reasoning processes. They analyzed the results of the PoC execution on their use cases, commenting on the solutions and the individual SPs chosen by the PoC to protect them. They provided their assessment results by answering a questionnaire given to internal experts in the previous phase.

Importantly, apart from the use case applications themselves (i.e., their annotated source code), the KB was populated with almost exactly the same information for all three use cases: no domain-specific or application-specific tuning of any values collected through expert elicitation (Section 5) was performed during the use cases' framing. All use cases were protected with the ACTC, and all were protected considering the same attack model, i.e., the same, guru-level attacker capabilities. The only differences in the framing was that some ACTC-supported protections (e.g., white-box crypto) were disabled for some of the use cases, because the protection's industrial owner did not want to share their source code with the other industrial project partners. This implies that each use cases was in practice evaluated using a combination of two protection tools, namely the ACTC source-to-source rewriter and its binary rewriter, with different protections enabled/disabled per use case.

The experts accessed the PoC outputs, an HTML report including the AAs and assets, the attack paths, and the 10 candidate solutions with the highest solution protection index.[19] The PoC reports on the three app use cases, almost identical[20] to the ones analyzed by the experts, are available on GitHub.[21]

The questionnaire answers provided in the second and third phases of the evaluation showed that, in summary, the internal and external experts considered the PoC promising. The degree of automation of the risk management phases, particularly the mitigation phase, was perceived as useful in supporting their daily tasks. They noted that the tool could be powerful in the hands of experts due to the high configurability of the internal reasoning processes, which can lead to choices of SPs for the target application with a quality comparable to a completely manual solution. Conversely, in the hands of software developers without an SP background and consequently unable to properly fine-tune the PoC parameters, the experts determined that the PoC would not attain the same degree of security for the target application.

The evaluation of the candidate solutions proposed by the PoC was positive. The experts highlighted that for each use case, the ESP's selection of SPs was specifically tailored for that use case.In other words, the selected SPs were instantiated for the specific AAs constituting the application and for the specific attack paths that are relevant for that application, in contrast to the generic protection recipes found in cookbooks provided by industrial protection tool vendors.

The experts also confirmed that the resulting protected applications conserved their original semantics after applying any of the proposed Candidate Solutions (CSs). Furthermore, they agreed on the acceptability of the computational overhead introduced by the chosen SPs, as the use cases protected with the proposed CSs remained usable without excessive delays.

Most importantly, they reported a high level of obtained asset protection since the SPs included in the CSs were considered able to protect the use cases appropriately against all the attack paths (see Section 2.2) generated by the PoC, and, as far as this was possible, against real attacks performed by professional pen testers.

During the ASPIRE project, each of the three use cases was pen tested by a different team of two external, professional pen testers with experience in the application domain of the use case. The pen tested application versions were protected with a set of SPs manually chosen by the internal experts. The pen testers could not successfully attack the DRM player use case within their available time frame and reported a significant delay in attacking the two other use cases. Reports summarizing their activity and their results are available in two public ASPIRE deliverables (Basile et al., 2016a; Ceccato, 2016). These pen tests also served as the basis for a taxonomy and models of how professional reverse engineers tackle protected applications (Ceccato et al., 2019).

The pen testing experiment is not a direct validation of the attack paths modeled by the ESP or of the SPs selected with the methods presented in this paper. Due to resource and scheduling constraints within the ASPIRE project, pen testing could only be done on versions protected with manually selected SPs, not with ESP-selected ones. Furthermore, due to resource constraints, the modeling of attack steps in the framing phase (i.e., their modeling in the Knowledge Base) was more coarse-grained than real attack steps, and it was limited in scope. However, to the extent the attack steps we framed for the ESP validation overlapped with the ones observed during the pen tests, and to the extent internal experts had manually chosen SPs to counter those attack steps,[22] the ESP selected the same or highly similar SPs. While this does not provide complete or direct validation of the ESP and the methods presented in this paper, it does provide evidence of their practical utility.

6.3. Sensitivity analysis

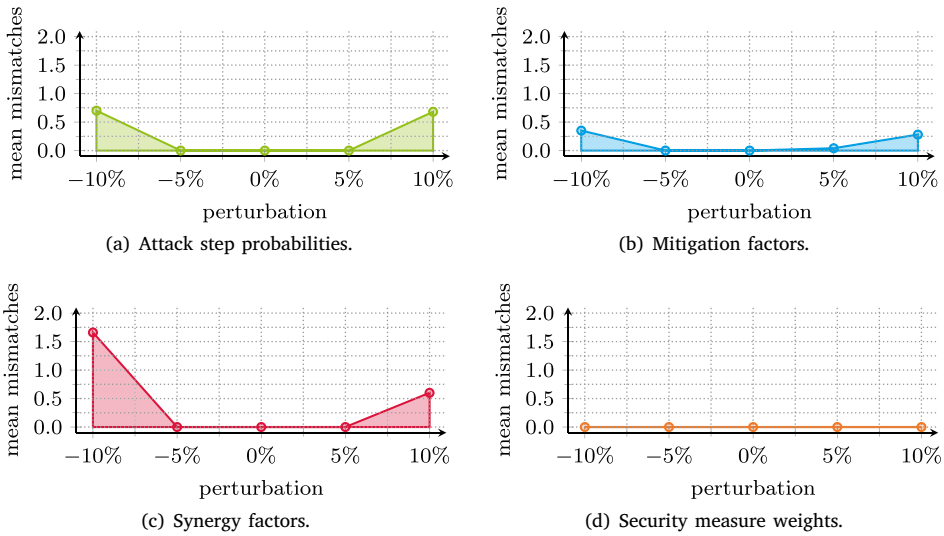
To assess the robustness of the proposed decision-support framework with respect to the expert-specified parameters used to instantiate the ESP, we performed a sensitivity analysis on our PoC. In particular, we repeatedly executed the PoC while perturbing one parameter at a time by ±5% and ±10%, relative to its original value. For each run,

[19] These solutions, produced by implementing the technique described in Section 4, are listed in the report as *Level 1 Protections*. The results listed as *Level 2 Protections* are not relevant for this article, since they are produced by an additional reasoning phase of the PoC, where additional protections are applied to non-sensitive code to hide the target application assets and confuse the attacker, following an approach described in a previous publication (Regano et al., 2017).

[20] To comply with industrial partners' confidentiality requirements, we renamed code and data identifiers of their use cases.

[21] https://github.com/daniele-canavese/esp/tree/master/reports

[22] The internal experts could foresee some of the attack steps that the pen testers would try, but they did not predict all of them.



(a) Attack step probabilities.

(b) Mitigation factors.

(c) Synergy factors.

(d) Security measure weights.

Fig. 7. Sensitivity analysis.

the protection solution returned by the PoC was compared against the reference solution obtained without perturbation. The comparison was performed by simply aligning the two solutions and counting the number of DSP changes.

We considered four parameter classes, all depending on experts' inputs: attack step probabilities, mitigation factors, synergy factors, and security measure weights. For each class, we individually perturbed the relevant parameters (i.e., the probability of each attack step, the mitigation factor of each DSP, the synergy factor for each DSP pair, and each security measure's weight), reran the PoC, and calculated the mean number of changes for each perturbation level. Fig. 7 reports the results. In all plots, the value at perturbation 0% is the reference point and is therefore always equal to 0 by construction.

The results indicate that our approach is robust under moderate parameter variations. In three of the four analyses, the mean number of changes remains either null or very small for perturbations of ±5%, suggesting that limited inaccuracies in the elicited parameters do not significantly alter the selected solutions. Moreover, even when changes occur, they typically affect only a small portion of the optimal solution.

For the attack step probabilities and mitigation factors, the perturbation has a limited effect, visible only at the largest magnitudes. This behavior suggests that the optimization is driven more by the overall structure of the attack paths and the interplay among protections than by small local variations in a single attack step probability or the strength of an individual SP.

The synergy factors exhibit the highest sensitivity among the analyzed parameters. For instance, when the perturbation is -10%, the mean number of changes reaches about 1.66. This behavior is consistent with the semantics of synergy in our model. In many runs, decreasing this value caused the PoC to swap the order of two protections applied to the same asset. Since our sensitivity-analysis procedure compares the original and perturbed solutions position by position, such a swap contributes a score of 2, because both positions become mismatches. Thus, the larger values observed for the synergy factors do not necessarily indicate radically different protection strategies, but rather frequent local reorderings among closely related protections whose relative desirability changes when synergy is weakened. In fact, if the ordering amongst the DSPs of each solution is ignored, the 1.66 mean mismatches decrease to 0.33. Given that a solution contains, on average, 16 DSPs, this implies that the ESP modifies on average 2% of the solution's DSPs, further confirming the stability of our approach.

For the security measure weights, the mean number of changes is always 0, showing the solutions are insensitive to reasonable weight variations. This likely reflects that the optimization depends more on attack-related values SP-related factors, and feasibility constraints.

6.4. Threats to validity

We reviewed our evaluation protocol using the threats to validity checklist proposed by Wohlin et al. (2000), considering construct, internal, external, and conclusion validity threats. Furthermore, we also consider threats to instantiation validity (Lukyanenko et al., 2014). The threats reported in this section primarily concern the risk mitigation phase implementation and evaluation described in this paper; broader threats related to the overall PoC and to the general evaluation setting are reported in the threats to validity discussion in the previously published work on the overall risk management approach for MATE SP (Basile et al., 2023).

Construct validity. The strength of the proposed protection solutions depends strongly on the parameters and coefficients used to instantiate the ESP model (e.g., synergy and mitigation factors). In the current PoC, most of these values are derived from the expert elicitation described in Section 5, which inevitably introduces subjectivity. We tried to reduce this threat to validity by eliciting information from multiple experts, with different industrial as well as academic backgrounds, and by aggregating their information as much as possible. Furthermore, to assess and mitigate this risk that the produced solutions are too sensitive to the exact parameter values, we performed a sensitivity analysis, with a positive outcome.

A second construct-related threat is that the SP index relies on complexity metrics and formulas as proxies for attacker effort and protection impact. This may under- or overestimate the actual effort required by attackers for specific workflows, and hence under- or overestimate the involved risks. Therefore, the security improvements reported by our approach should be interpreted as model-driven estimates of increased attack difficulty, rather than as direct empirical measurements of attack time, reverse-engineering effort, or success probability under controlled adversarial experiments. We note that the use of complexity metrics to measure protection strength is a common practice in literature (De Sutter et al., 2024), despite few if any of the commonly used metrics having been validated in the context of MATE SP. We limited the threats to validity of their use by (i) not using them as a direct metric of SP strength, instead only using them where appropriate (according to the consulted experts) to model the impact on different types of attack steps; (ii) making the use of the metrics

transparent in the reported solutions, such that expert users can assess the appropriateness thereof and manually intervene; (iii) avoiding that the PoC tools output anything that might be mistakenly interpreted as a real-world effort estimation. During the assessment with external experts, none of them raised the concern that the reports were unclear in this regard.

Finally, the current PoC uses a catalog of attack steps that is explicitly coarse-grained (e.g., high-level tasks such as locating or modifying a variable), which may hide important prerequisites and tool-dependent subtasks and thus affect the fidelity of both the risk assessment and mitigation phases.

Internal validity. The correctness of the optimization inputs depends on how accurately the KB is populated for a target application. The ESP requires users to annotate assets and their security requirements, and then performs source code analysis to build a formal representation of the program. Consequently, incomplete or incorrect annotations, or imprecision in the underlying analyses (e.g. data-flow reasoning used to relate variables to dependent artifacts), can lead to missing or spurious dependencies, which may in turn alter the set of protection objectives, the inferred attack paths, and the computed indices. We limited this threat to validity by letting multiple researchers check the annotations, and by manually verifying the correctness of the source code analyses on the relevant artifacts.

In addition, when online protections are used, the PoC generates monitoring logic but does not automatically incorporate monitoring feedback (e.g. detected attacks, compromised instances, or server-side performance issues) into subsequent decisions; instead, the KB must be updated manually. In general, this may limit internal validity over time because the modeled program exposure and SP effectiveness can drift unless the KB is actively maintained. We limited this threat to validity by making worst-case assumptions about possible attacks and by not relying on security-through-obscurity. Concretely, our use case scenarios assumed from the start that the defender faces guru-level attackers that already have all useful knowledge and expertise in advance, so their capabilities do not evolve over time, and hence no adaptations need to be made over time.

External validity. The just described reliance on worst-case assumptions to mitigate threats to internal validity comes at the cost of external validity threats. In industrial practice, for example, time-to-market pressures can lead to the initial releases of some software being protected with weaker protections to fend of all-to-easy attacks initially, rather than aiming to defend for the worst-case attackers from the start. In such scenarios, updates to the framing after the initial release will be required, which should be driven at least in part by information gathered with the monitoring logic. Our approach does not yet incorporate that.[23] Similarly, if the used protections depend (even lightly) on security-through-obscurity (this is not uncommon in practice) updates to the framing can become necessary when the monitoring of the software reveals that the obscurity has been lifted by adversaries.

In addition, the current PoC and its empirical evaluation are tied to the available protection toolchains (Tigress and ACTC) and platforms (Android/Linux ARMv7) supported by the ESP at the time of writing, which limits generalization to other ecosystems and protection stacks. In particular, since each CSP is modeled as a tool-specific instantiation, integrating other protection tools in the state of the art would require revisiting the related model parameters (e.g. security measures and overhead). Such reparametrization may change the SP indexes and the solutions rankings in a given ecosystem, but in general would not undermine the usefulness of the proposed approach.

We emphasize that, when designing the use of qualitative experts' judgments, we managed the risk of costly reparametrization to ensure the framework's use across different domains, application types, and protection toolchains. First, the three use cases already feature three very different software architectures, and second, we already experimented with three different protection flows: Tigress, the ACTC's source-to-source rewriting flow, and the ACTC's binary rewriting tools. Many of the parameters used to populate the KBs to address our use cases could remain the same, as the considered attacker capabilities were the same (under worst-case assumptions). Of course, if the framing of a SP task at hand changes, i.e., if new protections, new protection tools, or new attack strategies are considered in scope, then additional parametrization is necessary to capture the newly considered scope. We consider this unavoidable. Such additional parametrization has been designed to remain local. For example, if some protection is replaced by another, only that protection's parameters need to be revised. Or, if an additional attack option is included, only the parameters related to that attack option need to be provided. Both adaptations require the involvement of experts on both protections and attacks, given that the impact of protections on attacks needs to be captured, but still, the adaptations remain local. Our framework and models already allow the minimization of the reparametrization where this is appropriate. For example, by capturing protection features at multiple levels (CSPs that instantiate ASPs), it is possible to let different implementations (e.g., in different tools) share their common features. In practice, providing parameters for an additional protection or protection tool is no more difficult than creating a configuration file or additional configuration file entries based on a template file.

Finally, if our approach might be overfitting the three use cases, this would also endanger the general applicability of our framework. In this regard, we note that when experts tried the ESP on their use cases, the knowledge base on which the ESP operates was populated with generic parameter values and with the source code of their use case application. The generic parameter values were the same for all use cases. They were obtained by aggregating information elicited from experts with backgrounds in different domains *before* the PoC implementation of the ESP started and much before the use cases were ready. Apart from the application source code that the ESP obviously needs as input, no inputs or parameters of the methods presented in this paper have been tuned in any way for specific case studies.

Conclusion validity. The qualitative validation described in Section 6.2 was necessarily limited in scale and context: it was performed within the ASPIRE project on three industrial Android use cases, with two experts per partner (one internal and one external), making it difficult to draw statistically reliable conclusions about the approach's general effectiveness.

The development process also included extensive iterative feedback from internal experts during PoC construction, which is valuable for engineering maturity but may introduce expectation or confirmation biases compared to fully independent evaluation settings.

Our quantitative and qualitative conclusions are conditioned by practical solution space exploration limits and approximations. The search-tree algorithm is a worst-case exponential one, and in the evaluation, it was run with bounded search depths. As a result, optimality should be interpreted as optimal within the explored portion of the search space under the enabled optimization described in Section 4.

Moreover, in the PoC we avoid building the protected application for all the evaluated solutions by using ML predictors (Canavese et al., 2017) to estimate the metrics delta. The predictors' accuracy is good up to roughly three CSP applied to a single Attack Path (AP), and decreases significantly beyond that. Thus, the ESP may fail to evaluate the SP index of solutions accurately with more than three CSP on a

[23] To some extent, this is a chicken-and-egg issue. In the current practice, without automated decision support, one of the main reasons for not using the strongest available SPs for the initial release of some software is that under time-to-market pressure, human SP experts do not have sufficient time to determine the best protection solution manually. With automated decision support, that would no longer be an issue.

single AP. Finally, CCS-based decomposition improves scalability, but overhead constraints are inherently global: in the PoC, global overhead constraints are enforced approximately by assigning per-CCS overhead budgets based on expert input. This approximation may misestimate the feasibility of some combined solutions, i.e., locally feasible per-CCS choices may violate the global overhead bound, or vice versa.

Instantiation validity. Instantiation validity concerns whether our implemented artifact (the risk-mitigation automation in the ESP) is a faithful instance of the theoretical object of interest, i.e. a semi-automated decision support system for selecting software protections. A first threat is the large instantiation space: the same mitigation goals could be realized through alternative KB schemas, solver paradigms, and workflows, whereas we committed to a specific implementation inspired by game theory. In particular, we did not experimentally compare the proposed minimax solver against alternative optimization paradigms; indeed, the evaluation focused on feasibility and expert-perceived utility of the ESP rather than relative optimality against other solvers. A second threat arises from auxiliary features and emergent properties characteristic of complex IT artifacts: UI design, workflow integration, and how recommendations are presented and interpreted can significantly impact outcomes beyond the mitigation logic itself. We attempted to mitigate this threat by covering these aspects of our implemented artifact when the experts tested the PoC implementation in the second and third phase of the qualitative validation (Section 6.2). Third, artifact cost and project constraints limited our ability to implement and compare multiple alternative instantiations, which reduces our ability to rule out design-specific effects. Finally, portability over time is a threat: changes in platforms and SP and reverse engineering toolchains may require re-validating that this instantiation still matches the intended decision-support construct in new environments.

7. Related work

Our work relates to existing work in software protection and risk management.

7.1. Evaluation of software protection strength

Our approach's use of complexity metrics is in line with the 1997 proposal of Collberg et al. (1997) to evaluate the potency of protections in terms of complexity metrics. Since then, complexity metrics have been frequently used in literature to evaluate the strength of novel obfuscations (De Sutter et al., 2024).

Our use of complexity metrics to compute protection indices is our implementation of the conceptual 2009 proposal of Nagra and Collberg (2009) to define potency in terms of extra resources needed for an attacker's analyses to reveal properties of a protected program. Nagra and Collberg define potency in relation to specific analyses to reveal specific properties, which is an improvement over the 1997 definition, but they leave it open how those analyses can be composed of sequences of individual attack steps and how the impact of protections on such compositions should be evaluated. With our approach, we propose a method to specifically solve that issue.

For each type of attack step, our approach uses distinct formulas in terms of complexity metrics to compute how that specific step's required effort is impacted by the deployed SPs, and how much that attack step can counter that impact, i.e., reduce the protection index. By using distinct formulas for each type of attack step, our approach captures that different metrics are relevant for the different attack steps to be considered.

By considering only the attack steps that are relevant for the given POs, i.e., the given assets and their security requirements, and with those distinct formulas, we instantiate the recommendation of De Sutter et al. (2024) to evaluate the strength of protections in terms of concrete attacks. By considering both how SPs yield base protection indices, and how attack steps can reduce them to yield residual protection indices, our approach also adopts their recommendation to perform complete evaluations, i.e., to consider both the potency the and resilience of SPs.

7.2. Automated IT risk management

Research in the automation of risk management procedures in IT systems is rather old, with multiple expert systems for network intrusion detection and auditing being proposed from 1986 onwards (Hoffman, 1986; Denning and Neumann, 1985). More recent research mixes expert systems with AI/ML approaches. The work by Depren et al. uses Self Organizing Maps and decision trees for breach detection (Depren et al., 2005), feeding these results to an expert system for further interpretation, while the approach by Pan et al. uses neural networks for detecting attacks leveraging zero-day vulnerabilities and an expert system to identify known attacks (Pan et al., 2005).

A recent survey by Kaur et al. enumerates works for automated risk mitigation in computer networks (Kaur et al., 2023), distinguishing between approaches for the automated isolation of infected devices and tools for automated recommendation and implementation of risk mitigation procedures (Husák et al., 2022). MATE software protection differs considerably from network security, however. MATE attack modeling needs to include manual tasks and human comprehension of code, which are not considered in network security. For example, in network security, the development of zero-day exploits (using tools also found in the MATE toolbox) is handled as an unpredictable event, which side-steps the complexity of analyzing and predicting human activities. This entirely prevents us from reusing of existing assessment models developed for the network security scenario.

7.3. MATE software protection risk mitigation

In a previous paper (Basile et al., 2023), we proposed two possible approaches for MATE risk mitigation. The first is performing single-pass mitigation, where a human or a tool is able to find in a single pass the best SP solution, taking into account also attacks against the protected application. Considering the complexity of the SP decision process, we deem the automation of this approach unfeasible given the current state of research and the currently available computational resources. The second approach is iterative mitigation, where multiple steps in the SP decision process are performed. In this approach, a first SP solution is evaluated on the VA. Then, possible attacks are evaluated on this solution in order to refine it with additional SP. Multiple rounds of refinement are possible. The procedure presented in Section 4 can be considered a first attempt at automating this procedure since solutions are found iteratively, taking into account the effect of possible attacks against the protected application in terms of a decrease in the protection index. Indeed, this is only an estimation of the actual resilience of selected protections against attacks. In this sense, the approach could be improved by generating multiple versions of the application protected with the SP solutions with the highest protection index, and automating attack paths found in the risk assessment phase to find the most resilient solution. It should be noted that, given the size of the solution space, it would be practically impossible to perform such a test on all possible solutions. Thus, the game-theoretic-inspired approach would still be useful even with an available implementation of such an automated attack procedure.

In industrial practice, companies provide so-called cookbooks with SP recipes. For each asset, users of their tools are advised to manually select and deploy the prescribed SPs in an iterative, layered fashion as long as the overhead budget allows for additional SPs. Automated approaches are either overly simplistic or limited to specific types of SPs, and hence only support specific security requirements. Collberg et al. (1997), and Heffner and Collberg (2004) studied how to decide which obfuscations to deploy in which order and on which fragments given an overhead budget. So did Liu (2016), Liu et al. (2017). They differ in their decision logic and in the metrics they use to measure SP effectiveness. Importantly, however, their used metrics are fixed and limited to specific program complexity and program obscurity metrics, without adapting them to the identified attack paths. Coppens et al. proposed an

iterative software diversification approach to counter a concrete form of attack, namely diffing attacks on security patches (Coppens et al., 2013). Their work measured the performance of concrete attack tools to steer diversification and reduce residual risks. All of the mentioned works are limited to obfuscations. In all works, measurements are performed after each round of transformations, much like in the second approach we discussed above.

To improve the user-friendliness of manually deployed SP tools, Brunet et al. proposed composable compiler passes and reporting of deployed transformations (Brunet et al., 2019). Holder et al. evaluated which combinations and orderings of obfuscating transformations yield the most effective overall obfuscation (Holder et al., 2017). However, they did not discuss the automation of the selection and ordering according to a concrete program and security requirements.

7.4. Software protection tools

Multiple tools, both commercial and free and open source software (FOSS), are available to automatically *deploy* SP techniques to protect selected AA on a target application. Our PoC were originally designed in the context of the ASPIRE project, which also developed the ASPIRE Compiler Tool Chain (ACTC), an FOSS toolchain for protecting native ARM Android/Linux libraries (Basile et al., 2016a). Tigress[24] is another popular automatic SP tool, that is freely available for research. Tigress is developed by the University of Arizona. This tool performs source-to-source transformations and supports multiple SP techniques. The techniques we support for our mitigation phase are the ones of these two tools (Regano, 2019). For the protection of natively compiled C/C++ programs, only one additional tool is popular in research according to a recent survey (De Sutter et al., 2024), namely Obfuscator-LLVM (Junod et al., 2015) and more recent derivatives thereof. Those operate on the LLVM Intermediate Representation of the target code to deploy multiple SP techniques.

Many commercial SP solutions are available, such as the ones from Irdeto,[25] GuardSquare,[26] VMProtect[27] and Oreans (Code Virtualizer[28] and Themida[29]). However, scarce information can be derived from commercial descriptions of these tools on their inner workings and the implemented SP techniques.

There is also research interest (Lind et al., 2017; Shen et al., 2020) in automating the deployment of hardware-based SPs, such as Intel SGX and ARM Trustzone. Adapting an application code to support such HW solution is no trivial task, as target application code must comply with multiple requirements (e.g., the use of a modified C Standard Library for SGX-based applications).

8. Conclusions and future works

This paper presented an approach for automatically selecting protections to mitigate risks against assets in software applications. Starting from a vanilla application with annotated assets and previously identified attack paths, the approach employs a method inspired by game theory to select a set of protections that maximizes the residual SP Index under the proposed model and overhead constraints, simulating a scenario where a defender uses protection to delay potential attackers. The game is solved using a heuristic based on a mini-max depth-first exploration strategy, enhanced with dynamic programming optimizations. To compare candidate solutions, we introduce the Software Protection Index, which evaluates the effectiveness of protection against specific attack paths. We developed a proof-of-concept tool that implements our approach, which experts validated throughout the ASPIRE project. The expert assessment indicates that the approach can be practical as decision support for selecting protections under constraints, within the modeling assumptions and limitations discussed in Section 6.4.

Future work will see technical improvements in the decision-making process. Better heuristics in the solution solver, some inspired by chess, like killer moves, smarter solutions and attack paths visit order, can further improve performance.

Applying the most recent advances in ML and AI should allow better prediction of metrics used in the computation of the Software Protection Index and overhead estimations. Furthermore, the model for estimating overheads can also be made more precise; we would like to enable the protection experts to express global overheads to be translated into the artifact-specific overheads used by our model.

Moreover, we aim to refine the software protection index to make it a practical yet general implementation of potency and resilience, using more metrics, including the dynamic ones like entropy of memory access patterns and instruction traces, and results from dynamic taint analysis.

We also plan to conduct a controlled empirical validation of the model predictions, for example through attacker studies or penetration-testing campaigns that measure attack time, reverse-engineering effort, and attack success across alternative protection solutions.

Finally, another interesting research area is the automatic generation of more comprehensive attack paths using Large Language Models (LLMs) with Retrieval-Augmented Generation. Indeed, more precise attack paths during the risk assessment phase could help generate even better solutions.

CRediT authorship contribution statement

Daniele Canavese: Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Resources, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Leonardo Regano:** Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Methodology, Investigation, Formal analysis, Data curation. **Cataldo Basile:** Writing – review & editing, Writing – original draft, Validation, Supervision, Resources, Investigation, Funding acquisition, Formal analysis, Conceptualization. **Bjorn De Sutter:** Writing – review & editing, Writing – original draft, Validation, Supervision, Project administration, Funding acquisition, Conceptualization.

Compliance with ethical standards

Expert elicitation. The work reported in this paper includes expert elicitation activities (surveys, structured interviews, and questionnaires) with software protection developers and industry experts involved in the ASPIRE FP-7 project. In particular, the experts involved in the qualitative validation were drawn from the industrial partners of ASPIRE; their contributions were provided in their professional role (i.e. as part of their job activities) and focused on technical assessments of protections, attacks, and tool outputs. No sensitive personal data is reported in the manuscript, and feedback is discussed at an aggregate level, in line with the contractual obligations of the project's Consortium Agreement.

Confidentiality and research artifacts. The qualitative validation used three industrial Android applications provided by project partners. The PoC reports used in the evaluation are available in a public repository, whereas the original applications and related artifacts cannot be disclosed due to industrial confidentiality requirements. To comply with the confidentiality clauses of the Consortium Agreement, code and data identifiers were anonymized in the published reports.

[24] https://tigress.wtf
[25] https://irdeto.com
[26] https://www.guardsquare.com
[27] https://vmpsoft.com
[28] https://www.oreans.com/CodeVirtualizer.php
[29] https://www.oreans.com/Themida.php

Funding

This work was partially supported by project SERICS (PE00000014) under the NRRP MUR program funded by the EU - NGEU. This work was partially 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.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Appendix A. Supplementary data

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.cose.2026.104959.

Data availability

A statement on data availability is available in the manuscript in the "Compliance with ethical standards" section.

References

Abrath, B., Coppens, B., Volckaert, S., Wijnant, J., De Sutter, B., 2016. Tightly-coupled self-debugging software protection. In: Proc. of the 6th Workshop on Software Security, Protection, and Reverse Engineering. SSPREW '16, ACM, pp. 7:1–7:10. http://dx.doi.org/10.1145/3015135.3015142.

Alberto, S., 2021. Towards the prediction of performance degradation of obfuscated code.

Basile, C., Canavese, D., d'Annoville, J., De Sutter, B., Valenza, F., 2015. Automatic discovery of software attacks via backward reasoning. In: Proc. 1st Int'L Workshop on Software Protection. SPRO '15, IEEE Press, pp. 52–58. http://dx.doi.org/10.1109/SPRO.2015.17.

Basile, C., Canavese, D., Regano, L., 2016a. ASPIRE Validation. Deliverable D1.06. ASPIRE EU FP7 Project.

Basile, C., Canavese, D., Regano, L., Falcarin, P., De Sutter, B., 2019. A meta-model for software protections and reverse engineering attacks. J. Syst. Softw. 150, 3–21. http://dx.doi.org/10.1016/j.jss.2018.12.025.

Basile, C., De Sutter, B., Canavese, D., Regano, L., Coppens, B., 2023. Design, implementation, and automation of a risk management approach for man-at-the-end software protection. Comput. Secur. 132, 103321. http://dx.doi.org/10.1016/j.cose.2023.103321.

Basile, C., et al., 2016b. ASPIRE Framework Report. Deliverable D5.11, ASPIRE EU FP7 Project. URL https://aspire-fp7.eu/sites/default/files/D5.11-ASPIRE-Framework-Report.pdf.

Birmingham, J., Kent, P., 1988. Tree-searching and tree-pruning techniques. In: Computer Chess Compendium. Springer, pp. 123–128.

Borel, E., 1921. La théorie du jeu et les equation intégrales à noyau symétrique gauche." comptes rendus de l'académie des sciences, 173: 1304–08. Translated by lj savage in. Econometrica 21, 97–100.

Breuker, D., Uiterwijk, J., Van Den Herik, H., 1997. Information in transposition tables. Adv. Comput. Chess 8, 199–211.

Van den Broeck, J., Coppens, B., De Sutter, B., 2021. Obfuscated integration of software protections. Int'L J. Inf. Secur. 20, 73–101. http://dx.doi.org/10.1007/s10207-020-00494-8.

Brunet, P., Creusillet, B., Guinet, A., Martinez, J.M., 2019. Epona and the obfuscation paradox: Transparent for users and developers, a pain for reversers. In: Proceedings of the 3rd ACM Workshop on Software Protection. Association for Computing Machinery, pp. 41–52. http://dx.doi.org/10.1145/3338503.3357722.

Cabutto, A., Falcarin, P., Abrath, B., Coppens, B., De Sutter, B., 2015. Software protection with code mobility. In: Proc. of the 2nd ACM Workshop on Moving Target Defense. MTD '15, ACM, pp. 95–103. http://dx.doi.org/10.1145/2808475.2808481.

Canavese, D., Regano, L., Basile, C., Viticchié, A., 2017. Estimating software obfuscation potency with artificial neural networks. In: Livraga, G., Mitchell, C. (Eds.), Security and Trust Management. Springer International Publishing, Cham, pp. 193–202. http://dx.doi.org/10.1007/978-3-319-68063-7_13.

Ceccato, M., 2016. ASPIRE Security Evaluation Methodology. Deliverable D4.06. ASPIRE EU FP7 Project.

Ceccato, M., Dalla Preda, M., Nagra, J., Collberg, C., Tonella, P., 2007. Barrier slicing for remote software trusting. In: 7th IEEE Int'L Working Conference on Source Code Analysis and Manipulation. SCAM, IEEE Computer Society, pp. 27–36.

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: 2017 IEEE/ACM 25th International Conference on Program Comprehension. ICPC, IEEE Computer Society, pp. 154–164. http://dx.doi.org/10.1109/ICPC.2017.2.

Ceccato, M., Tonella, P., Basile, C., Falcarin, P., Torchiano, M., Coppens, B., De Sutter, B., 2019. Understanding the behaviour of hackers while performing attack tasks in a professional setting and in a public challenge. Empir. Softw. Eng. 24 (1), 240–286. http://dx.doi.org/10.1007/s10664-018-9625-6.

Claude, S., 1950. Programming a computer for playing chess. Philos. Mag. Ser 7 (41), 314.

Collberg, C., Thomborson, C., Low, D., 1997. A Taxonomy of Obfuscating Transformations. Computer Science Technical Reports 148, Dep. of Computer Science, University of Auckland, New Zealand.

Coppens, B., De Sutter, B., Maebe, J., 2013. Feedback-driven binary code diversification. ACM Trans. Archit. Code Optim. (TACO) 9 (4), 1–26. http://dx.doi.org/10.1145/2400682.2400683.

Coppens, B., et al., 2016. ASPIRE Open Source Manual. Deliverable D5.13. ASPIRE EU FP7 Project URL https://aspire-fp7.eu/sites/default/files/D5.13-ASPIRE-Open-Source-Manual.pdf.

Curtis, B., Sheppard, S., Milliman, P., Borst, M., Love, T., 1979. Measuring the psychological complexity of software maintenance tasks with the Halstead and McCabe metrics. IEEE Trans. Softw. Eng. SE-5 (2), 96–104. http://dx.doi.org/10.1109/TSE.1979.234165.

De Sutter, B., Schrittwieser, S., Coppens, B., Kochberger, P., 2024. Evaluation methodologies in software protection research. ACM Comput. Surv. 57 (4), http://dx.doi.org/10.1145/3702314.

Denning, D., Neumann, P.G., 1985. Requirements and Model for IDES – a Real-Time Intrusion-Detection Expert System. Tech. rep.,, SRI International, Menlo Park, CA, USA.

Depren, O., Topallar, M., Anarim, E., Ciliz, M.K., 2005. An intelligent intrusion detection system (IDS) for anomaly and misuse detection in computer networks. Expert Syst. Appl. 29 (4), 713–722.

Faingnaert, T., Zhang, T., Van Iseghem, W., Everaert, G., Coppens, B., Collberg, C., De Sutter, B., 2024. Tools and models for software reverse engineering research. In: Proceedings of the 2024 Workshop on Research on Offensive and Defensive Techniques in the Context of Man At the End (MATE) Attacks. CheckMATE '24, Association for Computing Machinery, New York, NY, USA, pp. 44–58. http://dx.doi.org/10.1145/3689934.3690817.

Falcarin, P., Collberg, C., Atallah, M., Jakubowski, M., 2011. Guest editors' introduction: Software protection. IEEE Softw. 28 (2), 24–27. http://dx.doi.org/10.1109/MS.2011.34.

Foket, C., De Sutter, B., De Bosschere, K., 2014. Pushing Java type obfuscation to the limit. IEEE Trans. Dependable Secur. Comput. 11 (6), 553–567. http://dx.doi.org/10.1109/TDSC.2014.2305990.

Goupil, F., Laskov, P., Pekaric, I., Felderer, M., Dürr, A., Thiesse, F., 2022. Towards understanding the skill gap in cybersecurity. In: Proceedings of the 27th ACM Conference on on Innovation and Technology in Computer Science Education Vol. 1. ITiCSE '22, Association for Computing Machinery, New York, NY, USA, pp. 477–483. http://dx.doi.org/10.1145/3502718.3524807.

Halstead, M.H., 1977. Elements of Software Science. Elsevier.

Heffner, K., Collberg, C., 2004. The obfuscation executive. In: Zhang, K., Zheng, Y. (Eds.), Information Security. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 428–440.

Heinz, E.A., 1998. Extended futility pruning. ICGA J. 21 (2), 75–83.

Hevner, A.R., March, S.T., Park, J., Ram, S., 2004. Design science in information systems research. MIS Q. 75–105.

Hoffman, L.J., 1986. Risk analysis and computer security: bridging the cultural gaps. In: Proceedings of the 9th National Computer Security Conference. National Institute of Standards and Technology, pp. 156–161.

Holder, W., McDonald, J.T., Andel, T.R., 2017. Evaluating optimal phase ordering in obfuscation executives. In: Proceedings of the 7th Software Security, Protection, and Reverse Engineering / Software Security and Protection Workshop. In: SSPREW-7, Association for Computing Machinery, http://dx.doi.org/10.1145/3151137.3151140.

Husák, M., Sadlek, L., Špaček, S., Laštovička, M., Javorník, M., Komárková, J., 2022. CRUSOE: A toolset for cyber situational awareness and decision support in incident handling. Comput. Secur. 115, 102609. http://dx.doi.org/10.1016/j.cose.2022.102609.

Initiative, J.T.F.T., 2011. SP 800-39. Managing Information Security Risk: Organization, Mission, and Information System View. Tech. rep., National Institute of Standards & Technology.

Junod, P., Rinaldini, J., Wehrli, J., Michielin, J., 2015. Obfuscator-LLVM – software protection for the masses. In: Wyseur, B. (Ed.), Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15, Firenze, Italy, May 19th, 2015. IEEE, pp. 3–9. http://dx.doi.org/10.1109/SPRO.2015.10.

Kaindl, H., Shams, R., Horacek, H., 1991. Minimax search algorithms with and without aspiration windows. IEEE Trans. Pattern Anal. Mach. Intell. 13 (12), 1225–1235. http://dx.doi.org/10.1109/34.106996.

Kaur, R., Gabrijelčič, D., Klobučar, T., 2023. Artificial intelligence for cybersecurity: Literature review and future research directions. Inf. Fusion 97, 101804. http://dx.doi.org/10.1016/j.inffus.2023.101804.

Knuth, D.E., 2011. The Art of Computer Programming, Volume 4A: Combinatorial Algorithms, Part 1. Addison-Wesley.

László, T., Kiss, Á., 2007. Obfuscating c++ programs via control flow flattening. Ann. Univ. Sci. Budapest. Sect. Comput. 30.

Lind, J., Priebe, C., Muthukumaran, D., O'Keeffe, D., Aublin, P.L., Kelbert, F., Reiher, T., Goltzsche, D., Eyers, D., Kapitza, R., Fetzer, C., Pietzuch, P., 2017. Glamdring: Automatic Application Partitioning for Intel SGX. In: Proceedings of USENIX Annual Technical Conference. USENIX Association, pp. 285–298.

Linn, C., Debray, S., 2003. Obfuscation of executable code to improve resistance to static disassembly. In: Proceedings 10th ACM Conference on Computer and Communications Security. ACM, New York, NY, USA, pp. 290–299. http://dx.doi.org/10.1145/948109.948149.

Liu, H., 2016. Towards better program obfuscation: Optimization via language models. In: Proc. 38th Int'L Conference on Software Engineering Companion. ICSE '16, Association for Computing Machinery, pp. 680–682. http://dx.doi.org/10.1145/2889160.2891040.

Liu, H., Sun, C., Su, Z., Jiang, Y., Gu, M., Sun, J., 2017. Stochastic optimization of program obfuscation. In: Proceedings of the 39th International Conference on Software Engineering. ICSE '17, IEEE Press, pp. 221–231. http://dx.doi.org/10.1109/ICSE.2017.28.

Lukyanenko, R., Evermann, J., Parsons, J., 2014. Instantiation validity in IS design research. In: Tremblay, M.C., VanderMeer, D., Rothenberger, M., Gupta, A., Yoon, V. (Eds.), Advancing the Impact of Design Science: Moving from Theory To Practice. Springer International Publishing, Cham, pp. 321–328.

Mantovani, A., Aonzo, S., Fratantonio, Y., Balzarotti, D., 2022. RE-Mind: a first look inside the mind of a reverse engineer. In: 31st USENIX Security Symposium (USENIX Security 22). USENIX Association, Boston, MA, pp. 2727–2745, https://www.usenix.org/conference/usenixsecurity22/presentation/mantovani.

McCabe, T.J., 1976. A complexity measure. IEEE Trans. Softw. Eng. SE-2 (4), 308–320.

Nagra, J., Collberg, C., 2009. Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection. Pearson Education, London, UK.

Pan, Z.S., Lian, H., Hu, G.Y., Ni, G.Q., 2005. An integrated model of intrusion detection based on neural network and expert system. In: 17th Int'L Conf. on Tools with Artificial Intelligence. IEEE Computer Society, pp. 672–673. http://dx.doi.org/10.1109/ICTAI.2005.36.

Phillips, C., Swiler, L.P., 1998. A graph-based system for network-vulnerability analysis. In: Proceedings of the 1998 Workshop on New Security Paradigms. NSPW '98, ACM, pp. 71–79. http://dx.doi.org/10.1145/310889.310919.

Regano, L., 2019. An Expert System for Automatic Software Protection (Ph.D. thesis). Politecnico di Torino.

Regano, L., Canavese, D., Basile, C., Lioy, A., 2017. Towards optimally hiding protected assets in software applications. In: Proc. Int'L Conf. on Software Quality, Reliability and Security. IEEE Computer Society, pp. 374–385. http://dx.doi.org/10.1109/QRS.2017.47.

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. http://dx.doi.org/10.1007/978-3-319-45931-8_8.

Schaeffer, J., 1986. Experiments in search and knowledge. University of Waterloo.

Shen, Y., Tian, H., Chen, Y., Chen, K., Wang, R., Xu, Y., Xia, Y., Yan, S., 2020. Occlum: Secure and Efficient Multitasking Inside a Single Enclave of Intel SGX. In: Proceedings of APLOS 2020: International Conference on Architectural Support for Programming Languages and Operating Systems. ACM, pp. 955–970. http://dx.doi.org/10.1145/3373376.3378469.

Slagle, J.R., Dixon, J.E., 1969. Experiments with some programs that search game trees. J. ACM 16 (2), 189–207. http://dx.doi.org/10.1145/321510.321511.

Van Put, L., Chanet, D., De Bus, B., De Sutter, B., De Bosschere, K., 2005. DIABLO: a reliable, retargetable and extensible link-time rewriting framework. In: Proc. Fifth IEEE Int'L Symposium on Signal Processing and Information Technology. IEEE Computer Society, pp. 7–12. http://dx.doi.org/10.1109/ISSPIT.2005.1577061.

Viticchié, A., Basile, C., Avancini, A., Ceccato, M., Abrath, B., Coppens, B., 2016a. Reactive attestation: Automatic detection and reaction to software tampering attacks. In: Proceedings of the 2016 ACM Workshop on Software PROtection. SPRO '16, ACM, pp. 73–84. http://dx.doi.org/10.1145/2995306.2995315.

Viticchié, A., Regano, L., Basile, C., Torchiano, M., Ceccato, M., Tonella, P., 2020. Empirical assessment of the effort needed to attack programs protected with client/server code splitting. Empir. Softw. Eng. 25 (1), 1–48. http://dx.doi.org/10.1007/s10664-019-09738-1.

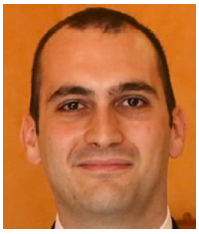
Viticchié, A., Regano, L., Torchiano, M., Basile, C., Ceccato, M., Tonella, P., Tiella, R., 2016b. Assessment of source code obfuscation techniques. In: 2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation. SCAM, IEEE, pp. 11–20. http://dx.doi.org/10.1109/SCAM.2016.17.

Votipka, D., Rabin, S., Micinski, K., Foster, J.S., Mazurek, M.L., 2019. An observational investigation of reverse engineers' process and mental models. In: Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems. http://dx.doi.org/10.1145/3290607.3313040.

Wang, H., Fang, D., Wang, N., Tang, Z., Chen, F., Gu, Y., 2013. Method to evaluate software protection based on attack modeling. In: Int'L Conf. on High Performance Computing and Communications (HPCC) & Int'L Conf. on Embedded and Ubiquitous Computing. EUC, IEEE Computer Society, pp. 837–844. http://dx.doi.org/10.1109/HPCC.and.EUC.2013.120.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M., Regnell, B., Wesslén, A., 2000. Experimentation in Software Engineering - An Introduction. Kluwer Academic Publishers.

Daniele Canavese received an M.Sc. degree in 2010 and a Ph.D. in Computer Engineering in 2016 from Politecnico di Torino, where he worked as a research assistant for more than ten years. He is currently a post-doc researcher at the IMATI (Istituto di Matematica Applicata e Tecnologie Informatiche), in Genova (Italy). His current research interests include using artificial intelligence and machine learning techniques for security management, software protection systems, public-key cryptography, and models for network and traffic analysis.



Leonardo Regano received an M.Sc. degree in 2015 and a Ph.D. in Computer Engineering in 2019 from Politecnico di Torino, where he worked as a research assistant for eight years. He is currently an assistant professor at the Department of Electrical and Electronic Engineering, University of Cagliari (Italy). His current research interests focus on software security, artificial intelligence and machine learning applications to cybersecurity, security policies analysis, and software protection techniques assessment.

Cataldo Basile is an associate professor at the Politecnico di Torino, from which he received an M.Sc.in 2001 and a Ph.D. in Computer Engineering in 2005. His research concerns software protection, software attestation, policy-based security management, and general models for detecting, resolving, and reconciling security policy conflicts.

Bjorn De Sutter is associate professor at Ghent University in the Computer Systems Lab. He obtained his M.Sc. and Ph.D. degrees in Computer Science from the university's Faculty of Engineering in 1997 and 2002. His research focuses on techniques to aid programmers with non-functional aspects such as performance and software protection to mitigate reverse engineering, software tampering, code reuse attacks, fault injection, and side channel attacks. He co-authored over 100 peer-reviewed papers.