Automatic Selection of Protections to Mitigate Risks Against Software Applications

Daniele Canavese, Leonardo Regano, Bjorn De Sutter, *Member, IEEE,* and Cataldo Basile, *Member, IEEE*

arXiv:2506.18470v1 [cs.CR] 23 Jun 2025

This work has been submitted to the IEEE for possible publication. Copyright may be transferred without notice, after which this version may no longer be accessible.

Abstract—This paper introduces a novel approach for the automated selection of software protections to mitigate Man-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 game-theoretic model. 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.

Index Terms—software protection, Man-at-the-End attacks, software risk mitigation, software potency and resilience

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 Man-At-The-End (MATE) attacks [1].

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 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 [2], 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 [2].

Other research highlighted a significant skill gap [3]: 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 SMEs.

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,

- *D. Canavese is with Institut de Recherche en Informatique de Toulouse. E-mail: daniele.canavese@irit.fr*
- *L. Regano is with the Dipartimento di Ingegneria Elettrica ed Elettronica, Università degli Studi di Cagliari, Cagliari, Italy. E-mail: leonardo.regano@unica.it*
- *B. De Sutter is with the Computing Systems Lab of Ghent University. E-mail: bjorn.desutter@ugent.be*
- *C. Basile is with the Dipartimento di Automatica e Informatica, Politecnico di Torino, Torino, Italy. E-mail: cataldo.basile@polito.it*

Corresponding Author: L. Regano

This work was partially supported by project SERICS (PE00000014) under the NRRP MUR program funded by the EU - NGEU, by ICO, Institut Cybersécurité Occitanie, funded by Région Occitanie, France, by the European research project Horizon Europe DUCA (GA 101086308), by the European FP7 research project ASPIRE (GA 609734) and CNRS IRN EU-CHECK.

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 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 [4], [5], [6]. All converge to the need for formal definitions of potency and resilience, the criteria introduced by Collberg *et al.* [7], that allow estimating the effectiveness of SPs when applied on specific portion 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 a method to suggest a set of SPs to apply to different parts of the software to delay attackers without degrading the 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;
- an approach for finding timely solutions to the above model.

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.

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 [8]:

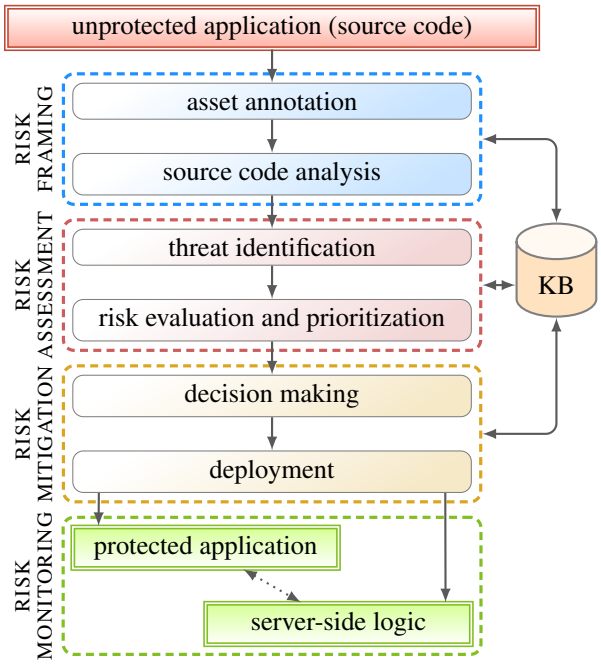

Fig. 1. The ESP workflow.

1) *risk framing*: to establish the scenario in which the risk must be managed;
2) *risk assessment*: to 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*: to determine and implement appropriate actions to mitigate the risks;
4) *risk monitoring*: to verify that the implemented actions effectively mitigate the risks.

Basile *et al.* [2] 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. Figure 1 presents the semi-automated workflow of the ESP. The work presented in this paper is a major contribution of the ESP.[1] Its complete code is available,[2] as well as a technical report on its inner workings [9], a user manual [10], 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.

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

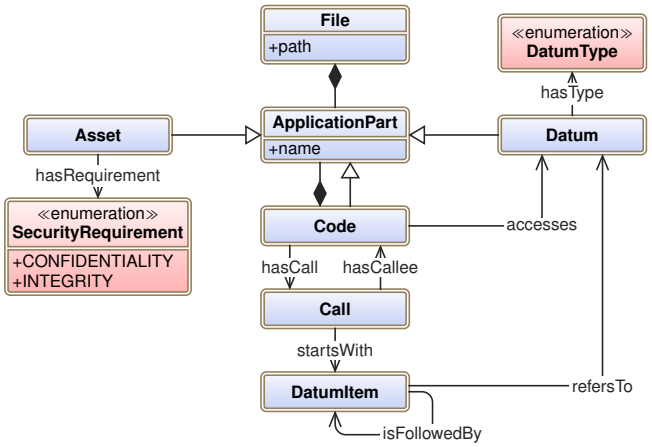

Fig. 2. The ApplicationPart class in the SP meta-model used in the ESP.

2.1 Risk Framing

In the risk framing phase, the ESP user must first annotate the code and data fragments of the C source code of the application that need protection. 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 [9].

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 [11]. In this meta-model, an application is modelled 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.

During the risk framing phase, a catalogue 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 [9], [10]. Table 1 summarizes the SP techniques supported by the ESP.

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 [12] and can serve to simulate attacks, e.g., with Petri Nets [13]. The attack steps that populate our KB originate from a study and taxonomy by Ceccato *et al.* [5], [14] 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 [15], [16], [17]. 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 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 [18]. 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.* [16], [17].

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 be applied on parts of the unprotected application such that the attacker will be delayed without degrading the application's performance beyond defender-defined acceptable levels.

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

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

PROTECTION TYPE	REQUIREMENTS		TOOL	
	CONFIDENTIALITY	INTEGRITY	ACTC	TIGRESS
anti-debugging	✓	✓	✓	○
branch functions	✓	○	✓	○
call stack checks	○	✓	✓	○
code mobility	✓	✓	✓	○
code virtualization	✓	✓	○[5]	✓
control flow flattening	✓	○	✓	✓
data obfuscation	✓	○	✓	✓
opaque predicates	✓	○	✓	✓
remote attestation	○	✓	✓	○
white-box crypto	✓	✓	✓	○

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.

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 a 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. Collberg *et al.* proposed the use of software complexity metrics originating from the domain of software engineering for assessing the potency of protections[7], [19], 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 [20] and cyclomatic complexity [18], examples of the latter are points-to-set sizes computed by data flow analysis tools [21], confusion factors of binary code disassemblers [22], and missing edges in function CFGs drawn by GUI disassemblers [23]. 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.

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 [24]. 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 significantly. This issue seems to be solvable with larger data sets and more advanced ML techniques.

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 Optimization Approach

The possibility, and in practice the necessity, of combining protections greatly increases the solution space. To explore it efficiently and to find (close to) optimal solutions in an acceptable time, the ESP uses a game-theoretic approach, 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 [25].

Figure 3 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, in this example, the same security requirements r. 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 are discussed in more detail as some of the major contributions of this paper.

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 then 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 [26]. 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 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 analyse 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 [26], [2] 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 [27] and reactive remote attestation [28], 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 [11] 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

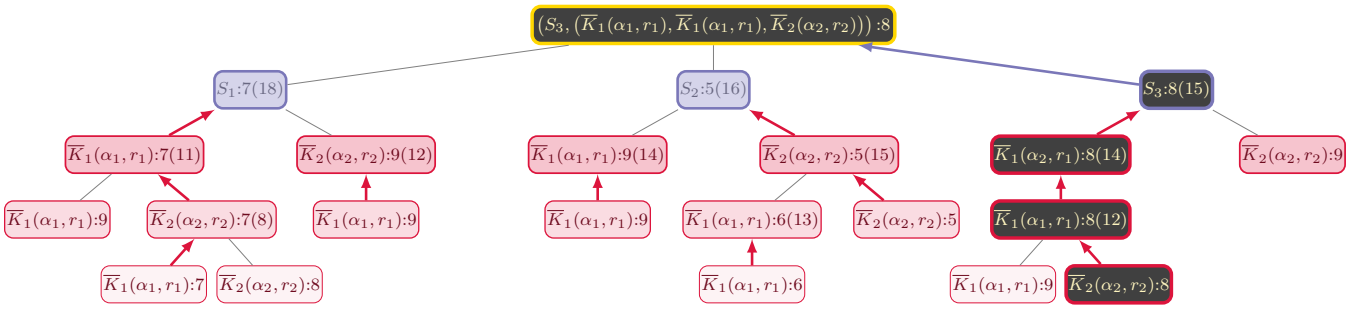

Fig. 3. Search tree example, computed with a mini-max approach and dynamic programming optimizations enabled.

the set of all source code lines that can directly or indirectly (via aliasing pointers) depend on the variable[6]. 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 $\text{weight} : \mathcal{O} \to \mathbb{R}_{\geqslant 0}$.

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 $P_i = \{p_{i,1}, ..., p_{i,n}\}$. 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 approach and algorithms. For example, the SP types listed in Table 1 correspond to ASPs.

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 CP.[7]

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 [29] cannot protect variables but only code. We model whether a SP P_i is *compatible* with an artifact a with the Boolean function $\text{compatible} : 2^{\mathcal{P}} \times \mathcal{A} \to \{\top, \bot\}$. Furthermore, we model whether a SP affects a security requirement with the function $\text{protect} : 2^{\mathcal{P}} \times \mathcal{R} \to \{\top, \bot\}$. 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 [30]:

- *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

6. 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.

7. It are these benefits that 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.

some AA is possible if $P_1 <^+ P_3$ 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 $P \not< P$.

Dependencies can be expressed as regular expressions [30] 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 $S = (p_1(a_1), p_2(a_2), \dots)$ in the *solution space* $\mathcal{S}$.[8]

The *vanilla solution* is the solution without any DSPs, and is represented as $\varnothing \in \mathcal{S}$ 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 approach 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 $\text{predict}_m : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ 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:

- halstead: the Halstead size of code AAs, i.e. their number of operators and operands [20];
- cyclomatic: the cyclomatic complexity of code AAs, i.e. their number of linearly independent paths [18];
- instructions: an AA's number of instructions [20];
- $\text{instructions.remote}$: the number of instructions of an AA moved to a remote server by SPs such as code mobility [27] or client-server code splitting [31], [6];
- $\text{instructions.local}$: 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 [32];
- $\text{instructions.guarded}$: the number of instructions of an AA guarded against tampering by tampering detection techniques such as remote attestation [28].

Our PoC used the link-time rewriter framework Diablo [33] to measure them on the vanilla application.

Moreover, we have built a pool of ML models that implement the predict_m function for the PoC supported metrics. 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 [24].

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 [31], thus reducing the computational resources needed to execute the AA on the client.

Formally, the function $\text{overhead}_i : \mathcal{S} \times 2^{\mathcal{A}} \to \mathbb{R}_{\geqslant 0}$ 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 $\theta_i(A) \in \mathbb{R}_{\geqslant 0}$. 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 [34], but that is orthogonal to the rest of our approach.

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 opera-

8. 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.

tions 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: $K(\alpha, r) = (k_1(a_1), k_2(a_2), \dots)$. 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 each attack steps, 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.[9] 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 $K(\alpha, r) = (k_1(a_1), k_2(a_2), \dots)$, 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 $K(\alpha, r) = (k_1(a_1), k_2(a_2))$, 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 $\Lambda : \mathcal{S} \times \overline{\mathcal{K}}(\alpha, r) \to [0, 1]$. 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. The base probability $\pi_{\overline{k}(a)}$ has been explicitly provided by our experts through interviews and questionnaires; the other values of n are obtained with formulas that asymptotically increase from a base probability to 1. π considers the attacker's expertise by changing the base probability. For instance, a guru-level attacker will have a higher probability of success than a script kiddie[10].

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* $\omega_{\overline{k}, p_i(a), p_j(a)} \in \mathbb{R}_{\geqslant 0}$ 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 ζ and ω are constant values determined using the experts' assessments, even if in the most general case, they may depend on the artifact.

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

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

4 Optimal Selection

We leverage a game theoretical procedure 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 approach 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:

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

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

9. 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.

10. Our approach only requires 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.

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 $\mathrm{art} : \mathcal{A} \to 2^{\mathcal{A}}$, 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 $\mathrm{art}(\alpha_i) \sqcap \mathrm{art}(\alpha_j)$ does not hold, protecting the two assets will have no interference.

By contrast, if $\mathrm{art}(\alpha_i) \sqcap \mathrm{art}(\alpha_j)$ 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:

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

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. $\bigcap_i \mathbb{A}^{\{i\}} = \varnothing$). 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 $(\varnothing, \varnothing)$, which is the vanilla solution without any attacks; note also that the vanilla solution is valid for any application.

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.

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'

```
IF T = NIL THEN                                   // the defender's turn
    p' ← −∞
    FOREACH S ∈ 𝒮 DO
        T̃, p̃ ← EXPLORE(𝒦_𝔸, 𝒮, (S, ∅), d − 1)
        IF p̃ > p' THEN
            p' ← p̃
            T' ← T̃
        END
    END
ELSE IF d = 0 THEN                                // a terminal node
    T' ← T
    p' ← index(T)
ELSE                                              // the attacker's turns
    p' ← ∞
    FOREACH K(α) ∈ 𝒦_𝔸 DO
        T̃, p̃ ←
          EXPLORE(𝒦_𝔸, 𝒮, (S, K̄_𝔸 ∪ K(α)), d − 1)
        IF p̃ < p' THEN
            p' ← p̃
            T' ← T̃
        END
    END
RETURN T' and p'
```

Algorithm 1: EXPLORE.

This scenario can be represented with a tree such as the one depicted in Figure 3. 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 a depth-first search algo-

rithm allows one to determine how the solution resists the attack paths and then choose the solutions that resist the best. 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 [35], [36]. 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 EXPLORE$(\text{NIL}, \mathcal{K}_{\mathbb{A}}, \mathcal{S}, d)$, with $d \geqslant 1$; 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 Figure 3, 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 [37]: this variation skips large portions of the tree without impacting the final result;
- aspiration windows [38]: 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 [39]: they cache-like objects that store previously computed values related to the protection indices;
- futility pruning [40], extended futility pruning [41] and razoring [42]: 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 $\mathcal{S}$, which can be too big to fit into memory.

In our PoC, we decided to explore the solution space $\mathcal{S}$ by generating the next solution the mini-max algorithm must process. The generation algorithm receives in 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 $\mathcal{S}$ 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 but our approach also took into account the experts' requests to feed the algorithm with their initial candidate solutions.

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 [43] to correctly take into account protections' precedence and may also exclude dis-

couraged 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, signalling that the solution space has been fully explored[11].

4.4 The Software Protection Index

The *SP index* is computed with the function $\text{index} : \mathcal{T} \to \mathbb{R}$. 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 $\text{index}(T_1) > \text{index}(T_2)$ must hold, thus permitting us to find the optimal one. In addition, the sign of a SP index allows us to infer some traits of a state $T = (S, \overline{K}_{\mathbb{A}})$:

- if $\text{index}(T) = 0$, the state T is without protections from the attacks in $\overline{K}_{\mathbb{A}}$, as for the vanilla application $\text{index}((\varnothing, \varnothing)) = 0$,
- if $\text{index}(T) > 0$, 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 $\text{index}(T) < 0$, 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 $\text{measure}_i : \mathbb{A} \times \mathcal{T} \to \mathbb{R}_{\geqslant 0}$ return the value of the i-th security measure of an asset in a particular state. We identified four security measures that stem from how SPs work:

- 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:

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

We computed the i-th measure using the equation:

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

This formula leverages $\text{measure}'_i(\alpha, T)$, 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 $\tau_i \in \mathbb{R}_{\geqslant 0}$, 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 $\rho_i \in \mathbb{R}_{\geqslant 0}$ whenever the adjusted measure is less than $\epsilon_i \in \mathbb{R}_{\geqslant 0}$. This subtraction allows marking states for which assets have been breached so that the search algorithm will avoid them[12].

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

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

This equation uses $\text{measure}_i(\alpha, (S, \varnothing))$, the i-th 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.

11. 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.

12. 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.

To compute $\text{measure}_i(\alpha,(S,\varnothing))$, we use the utility function $\text{H}'(x)=\text{H}(x)\cdot x$ to simplify some formulas, a variety of complexity metrics and the notion of Collberg's potency [7] $\mathfrak{P}:\mathbb{A}\times\mathcal{S}\rightarrow\mathbb{R}$. 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:

$$\mathfrak{P}_m(a,S)=\frac{\text{predict}_m(a,S)}{\text{predict}_m(a,\varnothing)}-1.$$

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

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

5 Expert Consultation

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. However, known models of MATE attacker behavior [5], [14] and reverse engineering in general [44], [45] 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.

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 [9]. 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.

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 Consultation Coverage

The inputs obtained from the experts cover many areas.

Suitability to preserve security requirements (Sec. 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 (Sec. 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 [4], [28]. These data helped build the function $\omega_{\overline{k},p_i(a),p_j(a)}$, i.e., the *synergy factor*.

Metrics, SP's and attack complexity relations (Sec. 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 $\text{measure}_i(\alpha,T)$, the parameters τ_i to relate the importance of the metrics, and $\text{measure}'_i$ functions.

Relations between metrics and SP overheads (Sec. 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.

USE CASE	PO	CCSs		
		COUNT	RANGE	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

TABLE 2
Code correlation sets in our use cases.

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

TABLE 3
Computational complexity of the approach.

Relations between SPs and attack steps (Sec. 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 concrete attack steps probability $\pi_{\overline{k}^n(a)}$. Then, they were asked to indicate the impact of the SPs in countering the attack steps, which resulted in the mitigation factor $\zeta_{\overline{k}(a),p(a)}$. 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.

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 practical 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.

12. The ACTC provides limited support for code virtualization, meaning that it is not reliably applicable to all code fragments. Hence, the ESP does not consider it a potential protection instance.

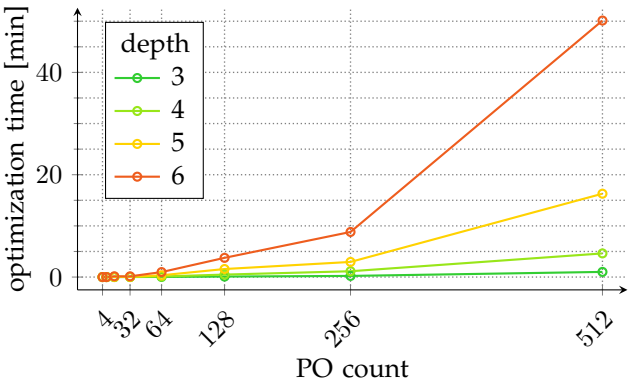

Fig. 4. Optimization time vs. number of POs.

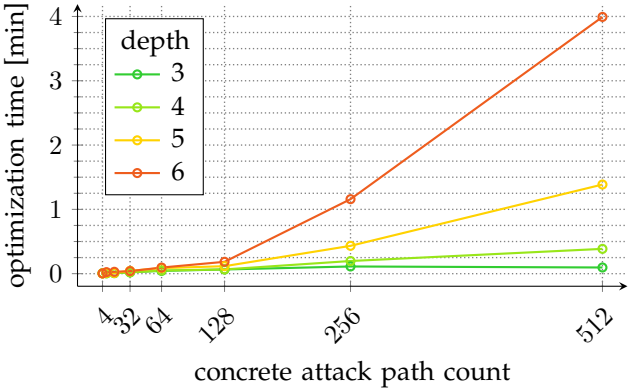

Fig. 5. Optimization time vs. number of attack paths.

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.

Figures 4 and 5 show the time to find the optimal solution 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 2, the use cases devised to perform the qualitative validation range from 24 to 59 POs[13]. 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

13. 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.

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 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 2 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. Figure 4 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 the experts from the 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. The complete evaluation of the whole PoC is available in another article [2] that framed our research in the IT Design Science Framework [46] towards the adoption of the NIST Risk Management Framework (SP800-39 [8]) to progress towards a standardized MATE risk management approach.

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 generator for home banking apps, an app licensing software, and a Digital Rights Management (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 [47] to confirm they are not toy examples.

Each of the three ASPIRE industrial partners involved two experts to validate their own use cases: one internal expert (i.e., actively involved 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 PoC development, leading to the alpha version of the PoC, which was tested comprehensively by the 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 between the internal experts, the protection owners, the PoC developers and the coordinator (including this paper's authors). 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 the questionnaire provided to the internal experts in the previous phase.

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.[14] The PoC reports on the three app use cases, almost identical[15] to the ones analyzed by the experts, are available on GitHub[16].

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 of the mitigation, was perceived as useful to support 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

14. 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 [26].

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

16. https://github.com/daniele-canavese/esp/tree/master/reports

to a completely manual solution. Conversely, in the hands of software developers without a 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 the ESP's selection of SPs was specifically tailored for the use cases. Indeed, all the POC's decision processes are based on a formal model of the AAs constituting the use case code. Thus, attack paths are specific to the target use case. Consequently, since the generation of candidate solutions considers both the AAs formal model and the application-specific attack paths, the resulting choice of SPs is customized for the targeted use case.

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, since the use cases protected with the proposed CSs were still usable without excessive delays. Also, they reported a high level of obtained asset security 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 also against the real attacks performed by professional pen testers[17] and it has been the basis for a journal article [14].

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.* [7] 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 [25].

Our use of complexity metrics to compute protection indices is our implementation of the conceptual 2009 proposal of Nagra and Collberg [49] 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.* [25] 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 [50], [51]. 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 [52], 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 [53].

A recent survey by Kaur *et al.* enumerates works for automated risk mitigation in computer networks [54], distinguishing between approaches for the automated isolation of infected devices and tools for automated recommendation and implementation of risk mitigation procedures [55]. MATE software protection differs considerably from network security, however. MATE attack modelling 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 analysing 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 [2], we proposed two possible approaches for MATE risk mitigation. The first is performing single-pass mitigation, where a human or a tool

17. During the ASPIRE project, two external pen testers were tasked to attack the three use cases, each protected with a set of SP manually chosen by the internal experts. They 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. A report of their activity is available in two ASPIRE public deliverables [47], [48].

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 game-theoretic 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 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.* [7], and Heffner and Collberg [30] studied how to decide which obfuscations to deploy in which order and on which fragments given an overhead budget. So did Liu *et al.* [56], [57]. 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 [58]. 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 [59]. Holder *et al.* evaluated which combinations and orderings of obfuscating transformations yield the most effective overall obfuscation [60]. 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 [47]. Tigress[18] 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 [17]. For the protection of natively compiled C/C++ programs, only one additional tool is popular in research according to a recent survey [25], namely Obfuscator-LLVM [61] 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[19], GuardSquare[20], VMProtect[21] and Oreans (Code Virtualizer[22] and Themida[23]). 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 [62], [63] 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 game-theoretic method to choose the optimal set of protections, 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

18. https://tigress.wtf
19. https://irdeto.com
20. https://www.guardsquare.com
21. https://vmpsoft.com
22. https://www.oreans.com/CodeVirtualizer.php
23. https://www.oreans.com/Themida.php

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 final assessment confirmed that automated software protection is a viable means for developers and experts to mitigate application risks.

Future work will see technical improvements in the decision-making process. Better heuristics in the game-theoretic 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.

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.

REFERENCES

[1] P. Falcarin, C. Collberg, M. Atallah, and M. Jakubowski, "Guest editors' introduction: Software protection," *IEEE Software*, vol. 28, pp. 24–27, March 2011.

[2] C. Basile, B. De Sutter, D. Canavese, L. Regano, and B. Coppens, "Design, implementation, and automation of a risk management approach for man-at-the-end software protection," *Computers & Security*, vol. 132, p. 103321, 2023.

[3] F. Goupil, P. Laskov, I. Pekaric, M. Felderer, A. Dürr, and F. Thiesse, "Towards understanding the skill gap in cybersecurity," 2022.

[4] A. Viticchié, L. Regano, M. Torchiano, C. Basile, M. Ceccato, P. Tonella, and R. Tiella, "Assessment of source code obfuscation techniques," in *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 11–20, IEEE, 2016.

[5] M. Ceccato, P. Tonella, C. Basile, B. Coppens, B. De Sutter, P. Falcarin, and M. Torchiano, "How professional hackers understand protected code while performing attack tasks," in *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pp. 154–164, IEEE Computer Society, 5 2017.

[6] A. Viticchié, L. Regano, C. Basile, M. Torchiano, M. Ceccato, and P. Tonella, "Empirical assessment of the effort needed to attack programs protected with client/server code splitting," *Empirical Software Engineering*, vol. 25, no. 1, p. 1 – 48, 2020.

[7] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," Computer Science Technical Reports 148, Dep. of Computer Science, University of Auckland, New Zealand, 7 1997.

[8] Joint Task Force Transformation Initiative, "SP 800-39. managing information security risk: Organization, mission, and information system view," tech. rep., National Institute of Standards & Technology, 2011.

[9] C. Basile *et al.*, "ASPIRE Framework Report," Deliverable D5.11, ASPIRE EU FP7 Project, 2016.

[10] B. Coppens *et al.*, "ASPIRE Open Source Manual," Deliverable D5.13, ASPIRE EU FP7 Project, 2016.

[11] C. Basile, D. Canavese, L. Regano, P. Falcarin, and B. De Sutter, "A meta-model for software protections and reverse engineering attacks," *Journal of Systems and Software*, vol. 150, pp. 3–21, 2019.

[12] C. Phillips and L. P. Swiler, "A graph-based system for network-vulnerability analysis," in *Proceedings of the 1998 Workshop on New Security Paradigms*, NSPW '98, pp. 71–79, ACM, 1998.

[13] H. Wang, D. Fang, N. Wang, Z. Tang, F. Chen, and Y. Gu, "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)*, pp. 837–844, IEEE Computer Society, nov 2013.

[14] M. Ceccato, P. Tonella, C. Basile, P. Falcarin, M. Torchiano, B. Coppens, and B. De Sutter, "Understanding the behaviour of hackers while performing attack tasks in a professional setting and in a public challenge," *Empirical Software Engineering*, vol. 24, no. 1, pp. 240–286, 2019.

[15] C. Basile, D. Canavese, J. d'Annoville, B. De Sutter, and F. Valenza, "Automatic discovery of software attacks via backward reasoning," in *Proc. 1st Int'l Workshop on Software Protection*, SPRO '15, pp. 52–58, IEEE Press, 2015.

[16] L. Regano, D. Canavese, C. Basile, A. Viticchié, and A. Lioy, "Towards automatic risk analysis and mitigation of software applications," in *Information Security Theory and Practice*, pp. 120–135, Springer International Publishing, 2016.

[17] L. Regano, *An Expert System for Automatic Software Protection*. PhD thesis, Politecnico di Torino, 2019.

[18] T. J. McCabe, "A complexity measure," *IEEE Transactions on software Engineering*, vol. SE-2, no. 4, pp. 308–320, 1976.

[19] B. Curtis, S. Sheppard, P. Milliman, M. Borst, and T. Love, "Measuring the psychological complexity of software maintenance tasks with the Halstead and McCabe metrics," *IEEE Transactions on Software Engineering*, vol. SE-5, no. 2, pp. 96–104, 1979.

[20] M. H. Halstead, *Elements of Software Science*. Elsevier, 1977.

[21] C. Foket, B. De Sutter, and K. De Bosschere, "Pushing Java type obfuscation to the limit," *IEEE Trans. on Dependable and Secure Computing*, vol. 11, pp. 553–567, 2 2014.

[22] C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly," in *Proceedings 10th ACM conference on Computer and communications security*, (New York, NY, USA), pp. 290–299, ACM, 2003.

[23] J. Van den Broeck, B. Coppens, and B. De Sutter, "Obfuscated integration of software protections," *Int'l Journal of Information Security*, vol. 20, pp. 73–101, 2 2021.

[24] D. Canavese, L. Regano, C. Basile, and A. Viticchié, "Estimating software obfuscation potency with artificial neural networks," in *Security and Trust Management* (G. Livraga and C. Mitchell, eds.), (Cham), pp. 193–202, Springer International Publishing, 2017.

[25] B. De Sutter, S. Schrittwieser, B. Coppens, and P. Kochberger, "Evaluation methodologies in software protection research," *ACM Comput. Surv.*, vol. 57, Dec. 2024.

[26] L. Regano, D. Canavese, C. Basile, and A. Lioy, "Towards optimally hiding protected assets in software applications," in *Proc. Int'l Conf. on Software Quality, Reliability and Security*, pp. 374–385, IEEE Computer Society, 2017.

[27] A. Cabutto, P. Falcarin, B. Abrath, B. Coppens, and B. De Sutter, "Software protection with code mobility," in *Proc. of the 2nd ACM Workshop on Moving Target Defense*, MTD '15, pp. 95–103, ACM, 2015.

[28] A. Viticchié, C. Basile, A. Avancini, M. Ceccato, B. Abrath, and B. Coppens, "Reactive attestation: Automatic detection and reaction to software tampering attacks," in *Proceedings of the 2016 ACM Workshop on Software PROtection*, SPRO '16, p. 73–84, ACM, 2016.

[29] T. László and Ákos Kiss, "Obfuscating c++ programs via control flow flattening," *Annales Universitatis Scientiarum Budapestinensis de Rolando Eötvös Nominatae. Sectio Computatorica*, vol. 30, 06 2007.

[30] K. Heffner and C. Collberg, "The obfuscation executive," in *Information Security* (K. Zhang and Y. Zheng, eds.), (Berlin, Heidelberg), pp. 428–440, Springer Berlin Heidelberg, 2004.

[31] M. Ceccato, M. Dalla Preda, J. Nagra, C. Collberg, and P. Tonella, "Barrier slicing for remote software trusting," in *7th IEEE Int'l Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 27–36, IEEE Computer Society, 2007.

[32] B. Abrath, B. Coppens, S. Volckaert, J. Wijnant, and B. De Sutter, "Tightly-coupled self-debugging software protection," in *Proc. of the 6th Workshop on Software Security, Protection, and Reverse Engineering*, SSPREW '16, pp. 7:1–7:10, ACM, 2016.

[33] L. Van Put, D. Chanet, B. De Bus, B. De Sutter, and K. De Bosschere, "DIABLO: a reliable, retargetable and extensible link-time rewriting framework," in *Proc. Fifth IEEE Int'l Symposium on Signal Processing and Information Technology*, pp. 7–12, IEEE Computer Society, 12 2005.

[34] S. Alberto, "Towards the prediction of performance degradation of obfuscated code," Master's thesis, Politecnico di Torino, 2021.

[35] E. Borel, "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*, vol. 21, pp. 97–100, 1921.

[36] S. Claude, "Programming a computer for playing chess," *Philosophical Magazine, Ser*, vol. 7, no. 41, p. 314, 1950.

[37] J. R. Slagle and J. E. Dixon, "Experiments with some programs that search game trees," *J. ACM*, vol. 16, p. 189–207, apr 1969.

[38] H. Kaindl, R. Shams, and H. Horacek, "Minimax search algorithms with and without aspiration windows," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 13, p. 1225–1235, dec 1991.

[39] D. Breuker, J. Uiterwijk, and H. Van Den Herik, "Information in transposition tables," *Advances in Computer Chess*, vol. 8, pp. 199–211, 1997.

[40] J. Schaeffer, *Experiments in search and knowledge*. University of Waterloo, 1986.

[41] E. A. Heinz, "Extended futility pruning," *ICGA Journal*, vol. 21, no. 2, pp. 75–83, 1998.

[42] J. Birmingham and P. Kent, "Tree-searching and tree-pruning techniques," in *Computer chess compendium*, pp. 123–128, Springer, 1988.

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

[44] A. Mantovani, S. Aonzo, Y. Fratantonio, and D. Balzarotti, "RE-Mind: a first look inside the mind of a reverse engineer," in *Proc. 32st Usenix Security Symposium*, 2022. To appear.

[45] D. Votipka, S. Rabin, K. Micinski, J. S. Foster, and M. L. Mazurek, "An observational investigation of reverse engineers' process and mental models," in *Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems*, 2019.

[46] A. R. Hevner, S. T. March, J. Park, and S. Ram, "Design science in information systems research," *MIS quarterly*, pp. 75–105, 2004.

[47] C. Basile, D. Canavese, and L. Regano, "ASPIRE Validation," Deliverable D1.06, ASPIRE EU FP7 Project, 2016.

[48] M. Ceccato, "ASPIRE Security Evaluation Methodology," Deliverable D4.06, ASPIRE EU FP7 Project, 2016.

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

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

[51] D. Denning and P. G. Neumann, "Requirements and model for ides – a real-time intrusion-detection expert system," tech. rep., SRI International, Menlo Park, CA, USA, 08 1985.

[52] O. Depren, M. Topallar, E. Anarim, and M. K. Ciliz, "An intelligent intrusion detection system (IDS) for anomaly and misuse detection in computer networks," *Expert Systems with Applications*, vol. 29, no. 4, pp. 713–722, 2005.

[53] Z. S. Pan, H. Lian, G. Y. Hu, and G. Q. Ni, "An integrated model of intrusion detection based on neural network and expert system," in *17th Int'l Conf. on Tools with Artificial Intelligence*, pp. 672–673, IEEE Computer Society, 11 2005.

[54] R. Kaur, D. Gabrijelčič, and T. Klobučar, "Artificial intelligence for cybersecurity: Literature review and future research directions," *Information Fusion*, vol. 97, p. 101804, 2023.

[55] M. Husák, L. Sadlek, S. Špaček, M. Laštovička, M. Javorník, and J. Komárková, "Crusoe: A toolset for cyber situational awareness and decision support in incident handling," *Computers & Security*, vol. 115, p. 102609, 2022.

[56] H. Liu, "Towards better program obfuscation: Optimization via language models," in *Proc. 38th Int'l Conference on Software Engineering Companion*, ICSE '16, pp. 680–682, Association for Computing Machinery, 2016.

[57] H. Liu, C. Sun, Z. Su, Y. Jiang, M. Gu, and J. Sun, "Stochastic optimization of program obfuscation," in *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pp. 221–231, IEEE Press, 2017.

[58] B. Coppens, B. De Sutter, and J. Maebe, "Feedback-driven binary code diversification," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 4, pp. 1–26, 2013.

[59] P. Brunet, B. Creusillet, A. Guinet, and J. M. Martinez, "Epona and the obfuscation paradox: Transparent for users and developers, a pain for reversers," in *Proceedings of the 3rd ACM Workshop on Software Protection*, pp. 41–52, Association for Computing Machinery, 2019.

[60] W. Holder, J. T. McDonald, and T. R. Andel, "Evaluating optimal phase ordering in obfuscation executives," in *Proceedings of the 7th Software Security, Protection, and Reverse Engineering / Software Security and Protection Workshop*, SSPREW-7, Association for Computing Machinery, 2017.

[61] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin, "Obfuscator-LLVM – software protection for the masses," in *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15, Firenze, Italy, May 19th, 2015* (B. Wyseur, ed.), pp. 3–9, IEEE, 2015.

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

[63] Y. Shen, H. Tian, Y. Chen, K. Chen, R. Wang, Y. Xu, Y. Xia, and S. Yan, "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*, pp. 955–970, ACM, March 2020.