Automatic Selection of Protections to Mitigate Risks Against Software Applications — supplemental material

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

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

This document provides supplemental material with further insights into the design and implementation of the technique for selecting the optimal Software Protections (SPs) to mitigate risks against software applications. It reports the questionnaire provided to software protection developers to characterize their protection tools to obtain the parameters for computing the Software Protection Index used during the decision process. Moreover, it lists and discusses the algorithms used by the optimization process, including a brief complexity analysis, whose results are only summarized in the main paper.

1. Questionnaire template

During the evolution of the ASPIRE FP7 project [1, 2], we asked SP developers and industry experts involved in the project to provide us with expert advice on the software protection processes, the software protections they developed, and the ones they use during their work activities. For example, Figure 1 reports a questionnaire for the SP developers about the protections they developed. We then used this data to build the knowledge base for our framework implementation, on which our decision support system relies for its decision making. Note that the terminology we use for some constructs has evolved since the ASPIRE project finished, so some terms in the questions listed here are semantically equivalent, but different from those in the original questionnaire.

2. Optimal selection: algorithms

This section presents the most interesting implemented algorithms, which deserve a thorough analysis to precisely determine their computational complexity as summarized in Section 6 of the main article.

*Corresponding author
Email addresses: daniele.canavese@cnr.it (Daniele Canavese), leonardo.regano@unica.it (Leonardo Regano), cataldo.basile@polito.it (Cataldo Basile), bjorn.desutter@ugent.be (Bjorn De Sutter)

1. Types of Protected Assets: Indicate the types of assets protected by your technique (e.g. code areas, variables)
2. Enforced Security Requirements: Identify the security requirements your protection technique enforces (confidentiality, integrity, or both).
3. Precedence Constraints: Specify any precedence constraints involving your protection technique:
   - Forbidden Precedences: List any protection techniques that must not co-exist on the same protected asset.
   - Discouraged Precedences: List protection techniques whose coexistence on the same asset is allowed but not recommended.
   - Encouraged Precedences: List protection techniques you recommend combining with yours for enhanced security.
4. Overhead Formulas: Indicate the overheads introduced by your protection technique, specifying the formulas to compute them on specific assets, based on their software metrics (e.g. Cyclomatic Complexity, Halstead Length).
   - Client-side Execution Time: Estimate the increased computation time of the protected application, due to the application of your protection on a specific asset.
   - Server-side Execution Time: If your protection requires server-side components, estimate the additional computation time on the server due to the application of your protection on a specific asset in the client application.
   - Client-side Memory: Estimate the increased memory footprint of the protected application, due to the application of your protection on a specific asset.
   - Server-side Memory: If your protection requires server-side components, estimate the additional memory footprint on the server due to the application of your protection on a specific asset in the client application.
   - Network Traffic: If your protection requires server-side components, estimate the additional network traffic exchange between the client and the server due to the application of your protection on a specific asset in the client application.
5. Impact on Software Metrics: Indicate the software metrics affected by your protection technique.
6. Mitigated Attacks: List and briefly describe the MATE attacks mitigated by your protection.
7. Potential Attacks Against Your Protection: Identify attacks that an attacker may mount on the protected application to undo or circumvent your protection.

Figure 1: Questionnaire completed by the SP developers.

```
INPUT: a Protection Objective (PO) [r, a] and the protection space 𝒫
OUTPUT: the set 𝔻_[r,a]
1 𝔻_[r,a] ← ∅
2 FOREACH P ∈ 𝒫 DO
3     IF compatible(P, a) ∧ protect(P, r) THEN
4         FOREACH p ∈ P DO
5             𝔻_[r,a] ← 𝔻_[r,a] ∪ {p(a)}
6         END
7     END
8 END
9 RETURN 𝔻_[r,a]
```

Algorithm 1: GetDSPs.

2.1. Preparatory stage algorithms

This section presents the algorithms needed for the preparatory stages of the optimization process, i.e., for stages described in Section 4.1 of the main article.

2.1.1. Determine Deployed Software Protections

Whenever the mitigations need to be selected, it is crucial to search among all available software protections for those ones that can protect the required security properties of the individual assets.

This search can be efficiently implemented with nested loops, as shown in Algorithm 1. Additional checks to allow pruning more DSPs can be easily added to the algorithm.

If $|P_i|$ denotes the total number of Concrete Software Protections (CSPs) available for the protection P_i, then this algorithm has a complexity of:

$$O\left(\sum_i |P_i|\right)$$

This is because the first for loop iterates over all the protections, while the second loop iterates over the protections' CSPs.

Given the cardinality of the set of protections, the complexity added by this algorithm is usually negligible.

2.1.2. Compute the Code Correlation Sets

Code correlation sets are a useful construct to divide the optimization problem into smaller, independent ones.

Computing the Code Correlation Sets (CCSs) can be performed iteratively and efficiently with the GetCCSs function reported in Algorithm 2. Note that the algorithm might require recomputing the same $\text{art}(\alpha_i) \sqcap \text{art}(\alpha_j)$ (Line 7) multiple times. A cache-based approach can be used to boost the performance, but we omitted this optimization in Algorithm 2 for the sake of readability.

The worst case here manifests when we have a single CCS. In that case, every asset is related to all the other assets. This scenario forces the algorithm to compute all the

```
INPUT: the asset space 𝔸
OUTPUT: the code correlation sets 𝔸^{1}, ..., 𝔸^{n}
n ← 1
𝔸^{n} ← {α}                          // α is any asset in 𝔸
A ← 𝔸 ∖ 𝔸^{n}
WHILE A ≠ ∅ DO
    FOREACH α_i ∈ A DO
        FOREACH α_j ∈ 𝔸^{n} DO
            IF art(α_i) ⊓ art(α_j) ≠ ∅ THEN
                𝔸^{n} ← 𝔸^{n} ∪ {α_i}
                A ← A ∖ {α_i}
            END
        END
    END
    IF A is not changed THEN
        n ← n + 1
        𝔸^{n} ← {α}                  // α is any asset in 𝔸
        A ← A ∖ 𝔸^{n}
    END
END
RETURN 𝔸^{1}, ..., 𝔸^{n}
```

Algorithm 2: GetCCSs.

pair comparisons between every $\text{art}(\alpha)$, leading to a quadratic complexity w.r.t. to the number of application artifacts $|\mathcal{A}|$:

$$O\left(|\mathcal{A}|^2\right)$$

2.2. Exploratory Stage Algorithms

This section presents the algorithms needed for the exploratory stages of the optimization process, i.e., for stages described in Section 4.2 of the main article.

2.2.1. Exploring the Solution Space

Determining the optimum solution requires identifying the candidate solutions and evaluating their protection index.

Algorithm 3 determines the next solution to consider, by iterating over the solution space $\mathcal{S}$. An iterative approach is needed since storing the entire solution space in memory is most likely unfeasible due to its sheer size.

The algorithm returns the next valid solution to analyze or NIL if the solution space has been fully explored.

Note that the GetNextSolution function will always be called in a loop by the Explore function (see Algorithm 7) to generate the whole solution space $\mathcal{S}$. For this reason, in the following paragraphs, we will not estimate the complexity of a single

INPUT: a solution S, the PO space $\mathcal{O}$, the Deployed Software Protection (DSP) space $\mathbb{D}$, and the maximum number of DSPs per PO σ
OUTPUT: the next solution S' or NIL if all solutions have been generated

```
   // step 1: reorder the DSPs in S
1  S' ← SHUFFLE_DSPs (S)
2  IF S' ≠ NIL THEN
3  |  RETURN S'
4  END
   // step 2: replace DSPs in S
5  S', N ← REPLACE_DSPs (S, O, D)
6  IF S' ≠ NIL THEN
7  |  RETURN S'
8  END
   // step 3: grow a larger solution
9  S' ← GENERATE_DSPs(N, σ)
10 IF S' ≠ NIL THEN
11 |  RETURN S'
12 END
   // step 4: S has been fully explored
13 RETURN NIL
```

Algorithm 3: GETNEXTSOLUTION.

GETNEXTSOLUTION call, but the complexity of exploring $\mathcal{S}$, which requires multiple GETNEXTSOLUTION invocations. The GETNEXTSOLUTION algorithm calls internally three functions (SHUFFLE$_{\text{DSPs}}$, REPLACE$_{\text{DSPs}}$, and GENERATE$_{\text{DSPs}}$). They are not called sequentially, though. GETNEXTSOLUTION will call the SHUFFLE$_{\text{DSPs}}$ function for the first consecutive invocations until it gets a NIL. When this happens, the next GETNEXTSOLUTION call will trigger the REPLACE$_{\text{DSPs}}$ function call (see Line 5). However, the following GETNEXTSOLUTION invocation will call the SHUFFLE$_{\text{DSPs}}$ function again, restarting the whole procedure from the beginning. A similar approach is leveraged by the GENERATE$_{\text{DSPs}}$ function (see Line 9). This invocation approach is equivalent to fully exploring a tree with tree levels, where the solutions, generated by calling GETNEXTSOLUTION multiple times, are the leaves. The first-level nodes are generated by the GENERATE$_{\text{DSPs}}$ function; the children of each first-level node are produced by the REPLACE$_{\text{DSPs}}$ function, while the children of the second-level nodes are created by the SHUFFLE$_{\text{DSPs}}$ function. That means that the total complexity of exploring $\mathcal{S}$ can be estimated by multiplying the complexities of SHUFFLE$_{\text{DSPs}}$, REPLACE$_{\text{DSPs}}$, and GENERATE$_{\text{DSPs}}$, thus leading to:

$$O\left(\sigma^{|\mathcal{O}|} \cdot \sigma^{\sigma \cdot |\mathcal{O}|} \cdot \psi^{n \cdot \psi}\right)$$

We can simplify this upper bound by removing the $\sigma^{|\mathcal{O}|}$ since it is dominated by $\sigma^{\sigma \cdot |\mathcal{O}|}$, thus obtaining:

```
INPUT: a solution S
OUTPUT: the next solution S' or NIL if all solutions have been generated
split S into the partial solutions S^{1}, ..., S^{n}
FOREACH i ∈ [1, n] DO
    X ← S^{i}
    S^{i} ← GETPERMUTATION_DSP(S^{i})
    IF S^{i} ≠ NIL THEN
        S' ← ⋃_{1≤j≤n} S^{j}
        RETURN S'
    ELSE IF i < n THEN
        S^{i} ← first permutation with the DSPs of X
    END
END
RETURN NIL
```

Algorithm 4: SHUFFLE$_{\text{DSPs}}$.

$$O\left(\sigma^{\sigma\cdot|\mathcal{O}|}\cdot\psi^{n\cdot\psi}\right)$$

in which n is the maximum number of partial solutions per solution, ψ is the maximum number of DSPs per partial solution, and σ is the maximum number of DSPs per PO.

2.2.2. Reordering the DSPs in a solution

The SHUFFLE$_{\text{DSPs}}$ function is the first core function used by the GETNEXTSOLUTION algorithm. Its job is to reorder the elements (e.g., the DSPs) of a solution, effectively allowing us to iterate over all its possible permutations.

Algorithm 4 lists the pseudo-code for our SHUFFLE$_{\text{DSPs}}$ function. The for-each loop iterates over all the partial solutions, while the GETPERMUTATION$_{\text{DSP}}$ function computes the next valid permutation of DSPs for each partial solution. From a functional point of view, the for-each loop is a variation of the mixed-radix generation algorithm for computing all the tuples (of a partial solution in this case) [3]. On the other hand, we implemented the GETPERMUTATION$_{\text{DSP}}$ function with the lexicographic permutations with restricted prefixes algorithm [3]; this algorithm receives in input a permutation and outputs the next valid one or NIL if all permutations have been visited. Since this is a well-known algorithm in the scientific literature, we will not list its pseudo-code here. The GETPERMUTATION$_{\text{DSP}}$ function allows us to perform various checks on the prefixes of a permutation before being fully generated, thus allowing us to immediately skip all the solutions with that prefix. For instance, if we identify that the first DSP in a solution cannot precede the second DSP, then we can immediately skip all the solutions starting with such DSPs.

If we have n partial solutions, then the complexity of the SHUFFLE$_{\text{DSPs}}$ algorithm is:

$$O\left(\left|S^{\{1\}}\right|!\cdot\left|S^{\{2\}}\right|!\cdot\ldots\cdot\left|S^{\{n\}}\right|!\right)$$

INPUT: a solution S, the PO space $\mathcal{O}$ and the DSP space $\mathbb{D}$
OUTPUT: the next solution S' or NIL if all solutions have been generated and the number of DSPs per PO N

```
// split the DSPs according to their POs
N ← ∅
FOREACH [r, a] ∈ O DO
    A_[r,a] ← ∅
    FOREACH p(a) ∈ S DO
        P ← protection of p
        IF compatible(P, a) ∧ protect(P, r) THEN
            A_[r,a] ← A_[r,a] ∪ {p(a)}
        END
    END
    N ← N ∪ {|A_[r,a]|}
END
// compute a new combination for each PO
FOREACH [r, a] ∈ O DO
    X ← A_[r,a]
    A_[r,a] ← GetCombination_DSP (A_[r,a], D_[r,a])
    IF A_[r,a] ≠ NIL THEN
        S' ← first solution with A_[r,a], ∀ [r, a] ∈ O
        RETURN (S', N)
    ELSE IF i < n THEN
        A_[r,a] ← first combination with the DSPs of X
    END
END
RETURN (NIL, N)
```

Algorithm 5: REPLACE$_{\text{DSPs}}$.

For simplicity's sake, we can assume that every partial solution has at most ψ DSPs. This allows us to simplify the formula and find an approximation as:

$$O\left((\psi!)^n\right)$$

Using Stirling's approximation for the factorial, we can also compute a lax upper bound as:

$$O\left(\psi^{n\cdot\psi}\right)$$

2.2.3. Replacing the DSPs in a solution

The second step of the GETNEXTSOLUTION algorithm is performed by the REPLACE$_{\text{DSPs}}$ function. The REPLACE$_{\text{DSPs}}$ function replaces some of the DSPs outputted by the SHUFFLE$_{\text{DSPs}}$ function with other techniques to help protect the same POs.

The REPLACE$_{\text{DSPs}}$ function is listed in Algorithm 5. It works in two consecutive phases. The first for-each loop splits the DSPs in the solution S according to the POs

they safeguard creating a map named A. At the same time, the loop computes the vector N that contains the size of each element in A; this value will be used later by the GENERATE$_{\text{DSPs}}$ function. The second loop iterates over the map A and computes the next combination of DSPs in each $A_{[r,a]}$ using the mixed-radix generation algorithm approach, like in the SHUFFLE$_{\text{DSPs}}$ function. Once a valid combination is generated, the algorithm returns the first suitable order of such DSPs as the next feasible solution.

The GETCOMBINATION$_{\text{DSP}}$ function receives in input a combination and a space. It returns the next combination to visit or NIL if all the combinations have been produced. In our case, we adopted Chase's sequence algorithm [3] function due to its efficiency.

The REPLACE$_{\text{DSPs}}$ algorithm operates on DSPs split according to their POs and not on the CCSs for balancing mitigations. Computing a new combination on a partial solution can lead to a solution leaving some POs unprotected, while with this approach, all the POs are guaranteed to have at least one DSP (if there is at least one suitable in the DSP set $\mathbb{D}$).

The complexity of the first loop is trivial:

$$O\left(|\mathcal{O}| \cdot |S|\right)$$

For the second loop, we assume that every PO can be protected with σ DSPs for simplicity's sake.

The number of combinations outputted by the GETCOMBINATION$_{\text{DSP}}$ function can be computed using a binomial coefficient. The worst case scenario for the binomial coefficient is $\binom{\sigma}{\lfloor \sigma/2 \rfloor}$. This leads to the second loop's complexity formula:

$$O\left(\binom{\sigma}{\lfloor \sigma/2 \rfloor}^{|\mathcal{O}|}\right)$$

For the two loops combined we obtain

$$O\left(|\mathcal{O}| \cdot |S| + \binom{\sigma}{\lfloor \sigma/2 \rfloor}^{|\mathcal{O}|}\right)$$

The first polynomial, however, can be omitted since the second exponential one shadows it, thus obtaining:

$$O\left(\binom{\sigma}{\lfloor \sigma/2 \rfloor}^{|\mathcal{O}|}\right)$$

Using the Stirling approximation again, we can find another simpler lax bound as:

$$O\left(\sigma^{\sigma \cdot |\mathcal{O}|}\right)$$

2.2.4. Adding DSPs to a solution

The final function used by the GETNEXTSOLUTION algorithm in its third step is GENERATE$_{\text{DSPs}}$, whose task is to add additional DSPs to the current solution. Algorithm 6 reports the pseudo-code of the GENERATE$_{\text{DSPs}}$ function.

The for-each loop iterates over all the partial solutions, while the GETPERMUTATION$_{\text{DSP}}$ function computes the next valid permutation of DSPs for each partial solution. From

INPUT: the number of DSPs per PO N and the maximum number of DSPs per PO σ
OUTPUT: the next solution S' or NIL if all solutions have been generated

1 $N' \leftarrow \text{GETTUPLE}_{\text{INTEGER}}(N, \sigma)$
2 IF $N' \neq NIL$ THEN
3 | $S' \leftarrow$ first solution with N' DSPs per PO
4 | RETURN S'
5 END
6 RETURN *NIL*

Algorithm 6: $\text{GENERATE}_{\text{DSPs}}$.

a functional point of view, the for-each loop is a variation of the mixed-radix generation algorithm [3] for computing all the tuples (of a partial solution in this case). Furthermore, the $\text{GETPERMUTATION}_{\text{DSP}}$ function implements the lexicographic permutations with restricted prefixes algorithm [3];

This algorithm receives as input a permutation and outputs the next valid one or NIL if all permutations have been visited. Since this is a well-known algorithm in the scientific literature, we will not list its pseudo-code here. The $\text{GETPERMUTATION}_{\text{DSP}}$ function allows us to perform various checks on the prefixes of a permutation before being fully generated, thus allowing us to immediately skip all the solutions with that prefix. For instance, if we identify that the first DSP in a solution cannot precede the second DSP, then we can immediately skip all the solutions starting with such DSPs.

This function is actually a wrapper around the $\text{GETTUPLE}_{\text{INTEGER}}$ function used to compute the next tuple of integers, stating how many DSPs per each PO must be used. This function receives a tuple of integers as input and the maximum allowed integer value. In our case, the minimum is implicitly set to 1 to protect each PO, if possible. It returns the next valid tuple of numbers or NIL if all the tuples have been explored. The $\text{GETTUPLE}_{\text{INTEGER}}$, in our case, was implemented with the loop-less reflected mixed-radix Gray generation algorithm [3]. When a valid integer tuple is found, the first valid solution with that number of DSPs per PO is returned. On the other hand, if all the integer tuples have been generated, then the $\text{GENERATE}_{\text{DSPs}}$ algorithm returns NIL since the solution space $\mathcal{S}$ has been fully explored.

If we have $|\mathcal{O}|$ POs, then the complexity of this algorithm is just:

$$O\left(\sigma^{|\mathcal{O}|}\right)$$

2.2.5. Exploring the state space

The main algorithm for exploring the state space is implemented via the EXPLORE function.

Algorithm 7 reports our basic algorithm for exploring the state tree. Note that the FOREACH statement at line 3 iterates the solution space $\mathcal{S}$, obtaining each solution S by calling the function GETNEXTSOLUTION. This algorithm explores a tree where the first level contains all the solutions, while the other levels contain concrete attack paths.

If we fix a tree depth of ζ, and denote with $\left|\overline{K}\right|$ the number of all the concrete attack

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 $T = \mathit{NIL}$ THEN // the defender's turn
  $p' \leftarrow -\infty$
  FOREACH $S \in \mathcal{S}$ DO
    $\tilde{T}, \tilde{p} \leftarrow \text{EXPLORE}(\mathcal{K}_{\mathbb{A}}, \mathcal{S}, (S, \varnothing), d-1)$
    IF $\tilde{p} > p'$ THEN
      $p' \leftarrow \tilde{p}$
      $T' \leftarrow \tilde{T}$
    END
  END
ELSE IF $d = 0$ THEN // a terminal node
  $T' \leftarrow T$
  $p' \leftarrow \text{index}(T)$
ELSE // the attacker's turn
  $p' \leftarrow \infty$
  FOREACH $K(\alpha) \in \mathcal{K}_{\mathbb{A}}$ DO
    $\tilde{T}, \tilde{p} \leftarrow \text{EXPLORE}(\mathcal{K}_{\mathbb{A}}, \mathcal{S}, (S, \overline{K}_{\mathbb{A}} \cup K(\alpha)), d-1)$
    IF $\tilde{p} < p'$ THEN
      $p' \leftarrow \tilde{p}$
      $T' \leftarrow \tilde{T}$
    END
  END
RETURN T' and p'

Algorithm 7: EXPLORE.

paths, then the number of states to be explored is:

$$O\left(|\mathcal{S}| \cdot \left|\overline{K}\right|^{\zeta}\right)$$

Plugging in the upper bound previously found for the GETNEXTSOLUTION function (used to iterate over the solution space), allow us to obtain this complexity formula:

$$O\left(\psi^{n \cdot \psi \cdot \sigma \cdot |\mathcal{O}|} \cdot \left|\overline{K}\right|^{\zeta}\right)$$

References

[1] Basile, C., et al.: ASPIRE Framework Report. Deliverable D5.11, ASPIRE EU FP7 Project (2016), https://aspire-fp7.eu/sites/default/files/D5.11-ASPIRE-Framework-Report.pdf

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

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