Towards Automatic Risk Analysis and Mitigation of Software Applications

Towards Automatic Risk Analysis and Mitigation of Software Applications

Leonardo Regano, Daniele Canavese, Cataldo Basile,
Alessio Viticchié, and Antonio Lioy

Politecnico di Torino, Dip. di Automatica e Informatica, Italy
{leonardo.regano,daniele.canavese,cataldo.basile,alessio.viticchie,lioy}@polito.it

Abstract. This paper proposes a novel semi-automatic risk analysis approach that not only identifies the threats against the assets in a software application, but it is also able to quantify their risks and to suggests the software protections to mitigate them. Built on a formal model of the software, attacks, protections and their relationships, our implementation has shown promising performance on real world applications. This work represents a first step towards a user-friendly expert system for the protection of software applications.

Keywords: software protection, software risk analysis, software attacks

1 Introduction

Software is pervasive in our life. We rely on software applications for our leisure and to ease our work, regardless of our fields of activity. In addition, software is one of the pillars of the world economy that moves billions to trillions of dollars. Developers have to protect their applications from tampering and avoid that confidential data in their software are disclosed. In short, companies have to protect the assets in their software, assets that are exposed to very powerful attacks, known as Man-at-the-End (MatE) attacks, from crackers that fully control the execution environment of the software to protect.

When the software must be protected, the human experience is the leading factor and almost the only one. While big companies have ad hoc teams to decide how to protect their applications or they can pay specialized companies, small and medium enterprises cannot afford the costs for properly protecting their software. By remaining vulnerable, it can damage the companies themselves, generating monetary losses, and all of us, becoming a vector for various kind of malware. Automatic or assisted techniques are needed to help software developers in protecting their applications.

In this paper we propose a novel risk analysis approach to (1) identify the threats against the assets in target applications, (2) quantify their risks against them and (3) suggest potential mitigations. In this context, the *mitigations* are the protections applied to each asset in order to reduce their exposure to the identified risks. This work represents a first step towards an expert system that

can drive the software developers in all the delicate phases of software protection. While the ambition is to make software protection another standard, almost push-button activity like the compilation, in the short term, our approach can be an interesting solution for small and medium enterprises.

A preliminary version of this work has been already published [1], focusing only on the automatic threats identification. With respect to our previous approach, the main improvements in this paper are the risk quantification and the proposal of mitigations. In addition, we greatly improved the expressivity of our model, thus leading to an increased accuracy of the attacks identification phase. A formal modelling of attacker purposes, strategies, and approaches to tampering, allows a more sophisticated analysis that also associates discovered attacks to the protections that would reduce their likelihood and consequences. Together with the formalization of a more sophisticated vulnerability identification system, building and validating our novel risk analysis model has required the impact's assessment of both attacks and protections on the software assets. For a better validation of our model, we have collected assessment information by means of questionnaires proposed to experts in software protection.

This paper is structured as follows. Section 2 presents our approach, its inputs, outputs and work-flow. Section 3 introduces a reference application that we will use to practically describe our achievements. Section 4 describes our formal model for describing an application for risk analysis purposes. Section 5 introduces the threats and mitigation identification engines whose performance is detailed in Section 6. Section 7 presents previous works in the field. Finally, Section 8 draws conclusions and sketches future research directions.

2 Approach

This section presents a general overview of our approach, whose work-flow is sketched in Fig. 1.

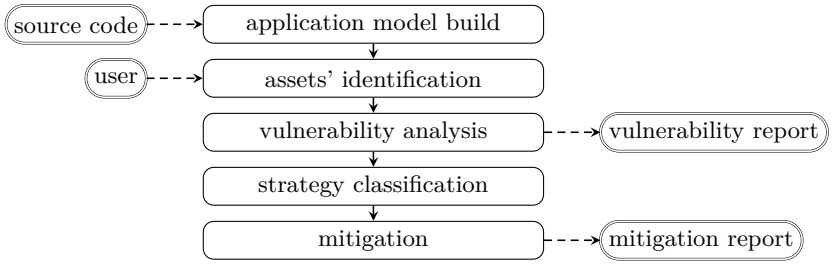

Fig. 1. General work-flow of our approach.

The input to our system is the set of the source files of the application to protect. Currently, only the C and C++ languages are supported, but the

same core ideas can be easily extended to any other programming language. The first phase, **application model build**, consists of a static analysis of the source code, which is parsed and analyzed to create an abstract model of the application. We named *application part* a generic piece of code of the application under analysis. The application parts are either *data*, that include (global and local) variables, class fields and function/method parameters, and *code areas*, which consist of functions, methods, classes or a particular code region in a function/method. Code areas have a hierarchical structure that allow them to contain other application parts. For instance, a class (a code area) can contain a method (another code area) that, in turn, encloses a local variable (a datum). Apart the containment association, application parts are also linked together via a set of other relationships, such as the call graph information. More information about these relationships is available in Section 4.

Once the initial application model has been automatically constructed, the user must finalize it by selecting which application parts actually need to be protected and the security properties that need to be guaranteed on them. Application parts that are associated to at least one security property are named *assets*. This is performed during the **assets' identification** phase. Identifying assets is pivotal in our approach as they are both the elements to protect (developers' perspective) and the targets of the attacks (attackers' perspective). We focus on four security properties in this work: integrity and confidentiality, which were already modelled in our previous paper, execution correctness and weak confidentiality, which are novel contributions.

For instance, some application parts can be marked with the *integrity* property, when they must be preserved from modifications. In this case, either the parts must be hard to modify or any modification must be detected. In other cases, the developers may want to guarantee the *execution correctness*, a stronger form of integrity that in addition requires that a code area must be called as expected. For instance, application parts marked with the execution correctness cannot be skipped, like authentication/license checks. Parts marked with the *confidentiality* property must be unintelligible for an attacker, such as keys to decrypt media streams or patented algorithms. Some data may be also tagged with the *weak confidentiality* property. This property is breached when the attacker is able to retrieve the datum at every moment of the application execution (thus for hard-coded data the weak confidentiality is the same as the normal confidentiality). This is mostly interesting for attacks that target the assets of a victim's application (i.e., not the attacker's copy) by means of a distributed approach aiming at continuously obtaining the data values. For instance, in case of an OTP (One-Time Password) generator[1] that generates the next password by hashing the value of a fixed seed and a counter modified at each generation, the variable storing the counter can be marked with weak confidentiality. To predict the next passwords (not only the next one), it does not suffice that the attacker obtains once the value of victim's counter, he has to obtain it just before every generation. Therefore, either he is able to access every time the victim's appli-

[1] See the example in [1] for more details on the OTP generator application.

cation to read the counter value, or he has to obtain one counter, understand the counter update function, and reproduce it on his copy.

During the assets' identification, the user is also given the opportunity to override the relationships that were previously automatically deduced or refine them with more precise associations. In addition, some important information may not be extracted by means of automatic tools as its correct identification would require knowledge that is not inferrable from the source code. For instance, correctly deducing that a function encrypts or decrypts some data is complex, especially if the code makes use of ad-hoc cryptographic libraries. Furthermore, some inferred relationships can be transformed into more accurate ones by means of a manual user intervention. As an example, a license verification function can invoke another function if and only if a previous license check is passed. These functions are automatically related by the automatic analysis with a simple *call* relationship. However our model supports a more expressive association, the *enables* one, used to indicate that a function can be only executed if another one has been "successfully" executed. To simplify this phase, we developed a simple yet effective domain specific language presented in Section 4. With this language, manually added data can be saved on disk, avoiding to ask the users to input them again at each analysis.

Once a valid and accurate application model has been constructed and validated by the user, during the **vulnerability analysis** phase, our system identifies all the attacks that can disrupt the security properties of the assets and produce a vulnerability report for the user. Attacks are sequences of simpler actions that an attacker must perform to mount it, the *attack steps*. Therefore, in our case, an attack is an ordered list of attack steps and will be thus called *attack path*. This simplification does not influence the accuracy of our analysis since we are interested in the effects of the attacks, regardless of their steps' order. For instance, two attack paths $(step_1, step_2, step_3)$ and $(step_1, step_3, step_2)$ are produced when both $step_2$ and $step_3$ can be executed at the same time or when their relative ordering does not matter. When producing the report to the user our approach will present only one of the two previous attack paths, by eliminating the clones. These sample attack paths are known as *unique attack paths*.

After having detected the attack paths, the **strategy identification** phase is performed. This phase consists in the classification of all the attack steps of the inferred attack paths in order to understand their purpose towards the goal. In our work, we classify attack steps in seven *strategy* types: static and dynamic code analysis, static and dynamic tampering, sniffing and spoofing, and compromission attacks. Note that, sniffing and spoofing also consider traditional network Man-in-the-Middle (MitM) attack steps while the compromission type includes code injection and attempts to control the application victim's copy[2].

[2] We have disinguished compromission from tampering as their purpose is different. When tampering with an application the attacker modifies the application code to achieve a goal or remove a protection on his own copy, compromission includes

Finally, the **mitigation** phase produces a mitigation report that lists all the protection techniques that can be used to mitigate (either block or render more difficult) all the attack paths. A *protection*, in our approach, is able to mitigate a set of attack step types with a particular level of efficacy (low, medium or high). We assume that an attack path is mitigated by a protection if it is able to mitigate at least one of its steps. This phase considers the following protections:

- *anti-debugging*, which makes more difficult to perform dynamic analysis by attaching a trusted debugger, preventing attackers from using their own [2];
- *algorithm hiding*, a set of obfuscation techniques against the reverse engineering, protecting a code's confidentiality and understandability [3];
- call stack checks, which verifies the execution correctness by checking that functions are called in the right order [4];
- *barrier slicing*, which enforces the integrity of data and code areas by moving them to a trusted server where they will be executed [5];
- *code guards*, which are checks added in an application to detect and react to integrity breaches [6];
- *code mobility*, which protects application parts from reverse engineering and analysis by removing them from the application to be installed at run time when they need to be executed [7];
- *data hiding*, which involves altering the data structures and the functions' data flow for ensuring data confidentiality [8];
- *remote attestation*, which protects application integrity by forcing the application to periodically send integrity proofs to a verification server [9].

After having revised the protection proposed by the mitigation report, the suggested protections can be applied on the assets. This stage has been implemented within the ASPIRE project by tagging all the variables and code areas with some special annotations that are later processed by the custom ASPIRE protection tool-chain.

3 Reference Example

We introduce here one of the applications we used to test our approach, the Linux Memory Game[3], an open source video game written in C based on the popular card game Memory. The game is played with a set of cards' pairs placed face down on a table. The player's goal is to find all the matching pairs with the minimum number of card turns. The game provides five skill levels: little one, beginner, skilled, master and daemon. When playing at the little one difficulty, all the cards are face up and visible (it is for children). On the other hand, in the highest difficulty level the cards are moved in different (and increasingly difficult) ways after every flip.

---

the cases where the attackers inject code to remotely control a pool of victims' applications.

[3] The source code is available at https://packages.debian.org/stretch/lmemory.

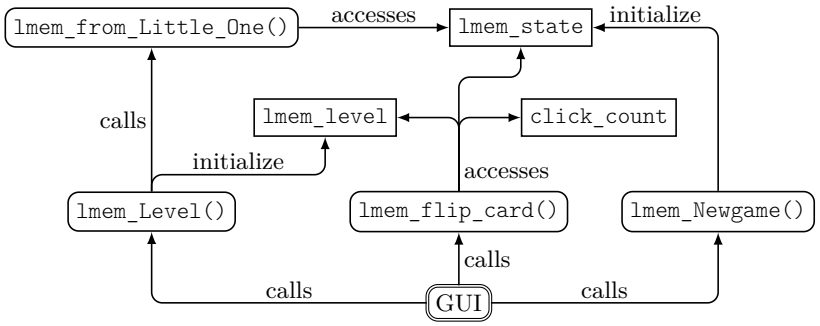

Fig. 2. Diagram showing the application parts relationships.

In our reference example, the main goal of the attacker is to win with the lowest number of flipped cards. On the other hand, the software developer must preserve the correctness of the game and the validity of the best scores, e.g., to keep the interest on it, as it happened for World of Warcraft.

Fig. 2 depicts a diagram showing the relationships (function calls, accesses and initializations of variables) between the most important application parts in the game. The card values are stored in the global variable `lmem_state`, a vector of integers whose elements are the cells in the card matrix, an asset. Its weak confidentiality must be safeguarded, as the attacker can play with all the cards visible, but also its integrity, since he can force a known card arrangement.

`lmem_state` is set by the `lmem_Newgame()` function, executed when a new game starts. `lmem_Newgame()` is also an asset, whose integrity must be preserved.

A critical datum is also `click_count`, an integer variable that counts the number of cards flipped. The developer must guarantee its integrity to avoid unwanted modifications (as an attacker can lower it to increase its final score).

The function `lmem_flip_card()` is executed every time a card is flipped and its logic can be summarized in the following steps: (1) when the user has clicked on a new card, turn its face up; (2) increase the click count; (3) if there are already two cards face up, then turn them face down; (4) if there is already a card face up and it matches with new card, remove both cards from the table; (5) rearrange all the face down cards according to the selected skill level.

The `lmem_from_Little_One()` function flips back all the cards and it is invoked by the `lmem_Level()` function, which in turn is called when the user changes the difficulty level from the little one level to a higher one. These three functions must be preserved from modifications since a plethora of attacks can be mounted against them, that is, their execution correctness must be guaranteed.

In the following sections, we will identify a set of attacks that can be used to alter the normal work-flow of the application. We will start our discussion by informally introduce some of them here.

The attacker might start by trying to discover the position of all the cards, even if they are face down, or to force a known card configuration. An attacker

can locate `lmem_state` by debugging a function that writes this vector, such as `lmem_Newgame()`, executed when a new game starts, thus easily recognizable via dynamic or static analysis. He can force some card values in `lmem_Newgame()` by using a known card configuration from a previously played game.

The attacker can also statically or dynamically change `lmem_Level()` to avoid the invocation of `lmem_from_Little_One()`. In this way the attacker can start with the little one difficulty level then switch to a higher one to be able to play with all the card faced up.

Finally, every step of `lmem_flip_card()` is vulnerable. The attacker can tamper the code to avoid the click count increase in order to obtain a reasonable score (step 2). He can stop cards from being turned face down (step 3). He can skip the card matching check, thus all the pairs of cards will be removed from the table, allowing an easy victory (step 4). Finally, he can also pretend to play at a higher skill level, whilst having a game at a lower difficulty (step 5).

4 Application Modeling

Our approach aims at automatically inferring attacks and protections that mitigate them by means of a Knowledge Base (KB). In this context, the starting knowledge is the application itself. The application code is theoretically the best source of information, since it models the complete application behavior. However, it also contains low level details that are not interesting for our kind of analysis. Therefore, a more abstract form is better suited for our purposes, that is the application meta-model.

Fig. 3 sketches the UML class diagram of the meta-model describing a generic application in our approach.

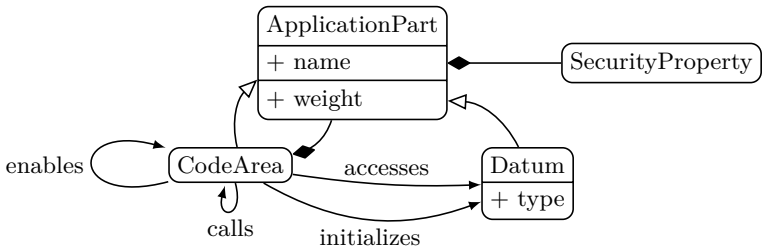

Fig. 3. UML class diagram of the application meta-model.

In our meta-model, an application is essentially a container of several application parts that can be either a datum or a code area. A code area can recursively contain one or more other application parts.

Every application part has a name (e.g., the variable or function name) and a *weight*, a non-negative real number used to explicitly indicate its importance (the greater the weight the more important the part). Weights are only meaningful

for the assets and are used to compute the risk values during the risk assessment phases. Application parts are associated to a non-empty list of *security properties* to ensure, which are also considered as the targets of the attacker.

The Data class stores an additional *type* attribute stating their data type (e.g., integer or string). This field is used to discard the unsuitable protections since some techniques might be applicable only to some kind of instances (e.g., data obfuscation for integer variables only).

Code areas are complex types with the following relationships:

- a code area *accesses* a datum when it reads or writes its content;
- a code area *initializes* a datum with a value (it represents the first writing of a value in a variable);
- a code area *calls* another code area;
- a code area *enables* another area if its execution depends on the (successful) execution of another area.

Our meta-model is simple and could be constructed by hand, but this task can be very time consuming as real applications might have hundreds or thousands of application parts. For this reason, the application model build phase in our approach automatically extract the application model from its source code and instantiates its components accordingly. As anticipated in Section 2, static analysis can lead to inaccurate or incomplete results that need to be complemented by the user input for validation and refinement purposes. To help users store this additional information, we developed a Domain Specific Language, the Application Description Language (ADL), which expresses the same concepts as the application meta-model, but in a more human readable form. Its Backus-Naur syntax rules are shown in Figure 4.

Application ::= *ApplicationPart**;
ApplicationPart ::= *Datum* | *CodeArea*;
Datum ::= *DatumType ID*("{"
    ("properties" *SecurityProperty* ("," *SecurityProperty*)*
    "weight" FLOAT)? "}" | ";");
CodeArea ::= "codeArea" ID (("{"
    ("properties" *SecurityProperty* ("," *SecurityProperty*)*
    "weight" FLOAT)?
    (*Relation* ID ("," ID)*)* "}") | ";");
Relation ::= "accesses" | "initializes" | "calls" | "enables";
DatumType ::= "integer" | "integerArray" | ...;
SecurityProperty ::= "confidentiality" | "weakConfidentiality" | "integrity" |
    "executionCorrectness";

Fig. 4. Grammar of the Application Description Language (ADL).

The terminal symbols ID and FLOAT respectively represents a valid C/C++ identifier and a sign-less floating point value.

As an example, Fig. 5 reports the description in ADL of the `lmem_flip_card()`, `lmem_Newgame()` and `lmem_state` assets for the Memory game.

```
code lmem_flip_card {
        properties integrity
        accesses lmem_level, click_count, lmem_state }
code lmem_Newgame {
        properties integrity
        initializes lmem_state }
```

Fig. 5. Assets description in ADL.

5 Vulnerability Analysis and Mitigation

In this section we present the vulnerability analysis and mitigation reporting steps of our approach. A preliminary work of the vulnerability analysis is available in a previously published paper [1], whose main ideas are summarized below:

- Facts are stored into a Knowledge Base (KB), initially populated with information concerning the application obtained from the application model.
- Breaching the assets properties becomes the goal of the attackers. Goals are modelled as properties. In this paper, the properties are: confidentiality, weak confidentiality, integrity and execution correctness.
- Attack step are modelled as rules of inference $P \rightarrow C\,(id)$, where id is an *identifier* of the attack step, that is its name, P is a set of *premises*, that is a set of facts in the KB that must be true in order to trigger the step, C is a set of *conclusions*, that is a set of additional facts that hold after the attack step is performed. Note that some attack steps (e.g. setting up a remote server) do not breach any security property, they are just preliminary actions needed to breach some properties (like the confidentiality of some data).
- Some inferences are not attack steps, relating different steps/facts and application parts (i.e., if x is in the KB do not perform the attack step y).

Once the Knowledge Base has been populated, all the attack paths are obtained by means of backward programming, which starts from the attack steps that breach the goals and progressively adduces facts that make the premises of attack steps true until the axioms are reached (i.e., attack steps or facts that have no premises). Attack paths can be extracted in an automatic way with any inferential engine of choice.

The previous attack path discovery model has been improved by adding new inference types that allow the detection of a larger set of attacks. The most important improvement consists in making the model more expressive by introducing the concept of attack strategy. Strategies are formal ways to determine

the behavior of an attacker and they are modelled by changing the way the backward reasoning process work. In some cases, strategies enable a different set of attack steps. For instance, if the attacker has to tamper with the victim's copy, he has to perform several network-oriented attack steps (e.g. creating fake servers, tampering with the victim's OS, injecting malware). More strategies can be enabled at the same time. Moreover, certain facts are derived only if certain attack strategies are applied and certain premises are enabled or disabled based on the strategy. Enabling more sophisticated strategies (that may include more attack steps and render premises more sophisticated require) may have a significant impact on the performance.

As a first instance of strategy, together with MatE attacks, we have modeled attack strategies depicting distributed scenarios, where an attacker tries to gain access to data of an application running on a victim PC. As an example, we have introduced an attack step modeling a code injection attack, where the attacker modifies a code of the victim application to send to him all the data accessed by the code. We also added strategies to breach the weak confidentiality. As anticipated, to breach this property, the attacker must know the value of the datum on the victim application at every moment of the application execution. In our internal model, we modelled additional preconditions: the attacker must not only obtain the value of the datum (e.g., with the code injection attack step described before), but he must also retrieve, either with static or dynamic attacks, all the code areas that access (and therefore potentially modify) the datum. Referring to the example application in Section 3, we have marked `lmem_state`, which contains the position of the cards, with the weak confidentiality. To breach it, the attacker must not only locate the datum in memory, but he must also execute the `lmem_Newgame` code before reading the datum value, because, as we can see in Fig. 2, `lmem_Newgame` initializes `lmem_state`. We also take into account the call graph of the application, thanks to the calls relationships: in particular, if a code has been executed, i.e., if the execute code step is present in the attack path, all the codes called by the first one are considered as executed.

We can also model indirect attack strategies, when the attacker has to start tampering with other parts of the application to achieve his goal. For instance, some code areas are executed only if first enabled by the execution of other code areas. An example may be an application that contains a license check function that, if performed successfully, enables the execution of the rest of the application. In our internal model a code area is enabled not only if the enabler code area has been executed before, but also if it have been changed, covering cases such as the license check in which the attacker, instead of using a working license, modifies the license check code to bypass it.

Another strategy has been defined to represent attacks against the execution correctness: an attacker can breach the execution correctness of a code not only modifying the latter, but also avoiding all the calls to it.

Finally, we greatly improved the accuracy of the attack path inference rules, which now leverage a greater number of relationships between the application parts, such as the ones described in Section 4. Furthermore, we added several

Symbol	Meaning
statLoc	staticallyLocate
dynLoc	dynamicallyLocate
statCh	staticallyChange
dynCh	dynamicallyChange
ch	changed
avC	avoidedCalls
c1	lmem_Level
c2	lmem_from_Little_One
br	breached
exCor	executionCorrectness

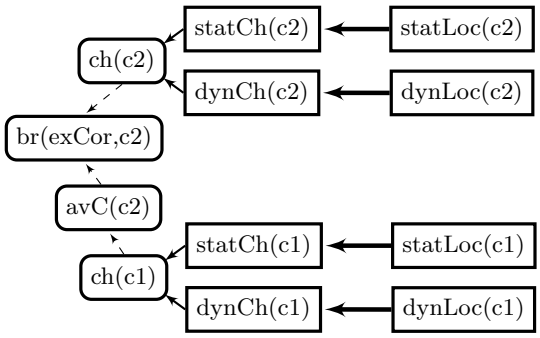

Fig. 6. Diagram showing the attack paths breaching the execution correcteness of the `lmem_from_Little_One` function.

new kind of attack steps, such as a new one representing the execution of a single code area, needed to better handle the relationships such as the initializes association and the dynamic attacks in general.

As as example, the attack where the attacker can play at higher level of difficulty with open cards by avoiding the call by `lmem_Level()` to `lmem_from_Little_One` (Section 3) can be automatically obtained by the attack deduction phase if `lmem_from_Little_One` is marked with the execution correctness property. In this case if the corresponding strategy is enabled it produces the different attack steps in Fig. 6. Attack paths are represented as rectangles, while attack facts are drawn with rounded rectangles. Links between attack steps are represented as thick arrows, while links between attack steps and the produced facts are draw with normal arrows. Facts deduced by other facts are linked with dashed arrows.

To help security engineers to select the most vulnerable assets and properly decide what to protect in their applications, we also provide a measure about how dangerous is an attack path. Of course, custom formulas can be used in alternative to adapt to target applications and their business models.

Given all the deduced attack paths AP_i, all their attack steps $AS_{i,j}$, and all the assets a_k, we define the *risk* $\Omega^{\epsilon}_{\mathrm{AP}_i}$ of an attack path AP_i against attackers with expertise ϵ as:

$$\Omega^{\epsilon}_{\mathrm{AP}_i} = \pi(\mathrm{AP}_i, \epsilon)\ \Gamma_{\mathrm{AP}_i} \tag{1}$$

The $\pi(\mathrm{AP}_i, \epsilon)$ in Eq. 1 is the probability that an attacker with expertise ϵ is able to successfully complete of the attack path. Note that, in estimating the probability to successfully mount an attack path, we consider the probabilities that an attacker with expertise ϵ is able to successfully execute all the attack steps needed to complete it, that is, $\pi(\mathrm{AP}_i, \epsilon) = f\left(\pi(\mathrm{AS}_{i,j}, \epsilon)\right)$. The idea behind this formula is that an attacker, to successfully complete an attack path, must correctly undertake all the steps constituting it. We have tested our approach with $f = \min$ (the one that gave us best results), with an unrealistic but very conservative $f = \max$, and with $f = \cdot$.

Γ_{AP_i} is a quantitative measure of the damage resulting from a successful attack path, calculated as the sum of the damage $\Gamma_{\mathrm{AS}_{i,j}}$ from each attack step:

$$\Gamma_{\mathrm{AP}_i} = \sum_j \Gamma_{\mathrm{AS}_{i,j}} = \sum_j \sum_k (W_{a_k} b(a_k, \mathrm{AS}_{i,j}))$$

where W_{a_k} is the user-defined asset weight, and $b(a_k, \mathrm{AS}_j)$ is a function, deduced by our inference system, that returns the fraction of the security properties of the asset a_k that are breached by the attack step AS_j, that is, it returns 1 if all the security properties are compromised and 0 if none[4]. Finally, the values obtained with the risk formulas are mapped on a three values score (low, medium, high) with an ad hoc mapping.

Our inference system also suggests a list of protections that can be implemented to mitigate the risk of an attack. Protections can be applied to code areas and data, and are associated to a mitigation level, which is a measure of how much they reduce the probability of success of a particular attack step. To assess the mitigation of the risk associated to an attack path against application assets after the application of a set of protections selected by the user from the proposed ones, we recalculate the risk by using an updated value of the probability to successfully perform every attack step in the presence of the selected protections. To this purpose, we override the function $\pi(\mathrm{AS}_{i,j}, \epsilon)$ into $\pi(\mathrm{AS}_{i,j}, \epsilon, \{p_l\}_l)$ that also considers that $\{p_l\}_l$ protections are deployed to protect the asset.

The values of the expertise levels needed by the attacker to undertake an attack step, and the mitigation levels of the protections (and corresponding probabilities) that we have used in our experiments have been obtained by means of questionnaires proposed to 20 software protection experts in the ASPIRE project (https://aspire-fp7.eu/) consortium and advisory board.

6 Experimental Results

We have implemented our approach in Java 8 as a set of plug-ins for the Eclipse Mars platform. In addition we used XText 2.9.0 for developing the ADL parser and SWI-Prolog 7.2.3 for the attack path engine. In order to speed up the computation, the inference engine is implemented as a multi-threaded Prolog program so that multiple assets can be evaluated in parallel.

We tested our framework on the Linux Memory Game application described in Section 3. It is written in C and contains 53 application parts (10 global variables and 43 functions). We have marked 6 application parts as assets. The tests have been performed on an Intel i7-4980HQ 2.80 GHz with 16 GB of memory under Linux Debian 4.5.0, allocating 4 CPU cores for the vulnerability analysis phase. Table 1 summarizes our results on the application where $\mathcal{C}^{\mathrm{W}}$ stands for weak confidentiality, $\mathcal{I}$ for integrity and $\mathcal{E}$ for execution correctness.

[4] In practice, we assume that the whole asset weight assigned by the user is gained by the attacker when all the security properties are compromised.

assets		attacks		risks		chosen protection
part	property	total	uniques	initial	mitigated	
`click_count`	$\mathcal{I}$	2	2	medium	low	anti-debugging
`lmem_from_Little_One()`	$\mathcal{E}$	12	6	high	low	code guards
`lmem_flip_card()`	$\mathcal{E}$	37	16	high	low	remote attestation
`lmem_level`	$\mathcal{I}$	3	3	medium	low	code mobility
`lmem_state`	$\mathcal{C}^{\mathrm{W}}$	20	8	high	low	code guards
`lmem_state`	$\mathcal{I}$	4	4	high	low	anti-debugging
`lmem_Newgame()`	$\mathcal{E}$	32	14	high	low	remote attestation
TOTAL		110	53	–	–	

Table 1. Memory game attack statistics.

The whole analysis process completed in 344.1 seconds (less than 6 minutes) with all the attack strategies enabled. Note that the computation time is heavily influenced by the indegree and outdegree of the code areas in the call graph. A higher number of relationships surrounding an application part leads to a higher number of search combinations, which the vulnerability analysis has to try and take into account. For instance, `click_count` is directly used by only 2 application parts and its analysis takes only 1.4 seconds, while `lmem_flip_card()` is related to 16 application parts and its analysis takes 322.4 seconds.

The framework found 110 attacks on the assets, which reduces to only 53 unique attacks if we discard the attack step order (see Section 2). For the risk analysis we opted to use an 'amateur' expertise level due to the type of the example application. The risks columns in the table respectively list the risk of the most dangerous attack than can be mounted against an asset and the mitigated risk by using the strongest protection technique supported by our approach (also shown in the table).

Most of the attacks have a high risk to be performed even by an amateur attacker, but using the techniques suggested by our framework all of them can be nearly avoided reducing their risks to low. For instance, the attack path $(\, statLoc(c2), statCh(c2), ch(c2), br(exCor, c2)\,)$ (Fig. 6) against the execution correctness of `lmem_from_Little_One()` has a high risk to be executed. However, our implementation has detected that using the code guards technique will lower the risk to low since the attack step $statCh(c2)$ will be severely hindered.

7 Related works

The vulnerability assessment is a common issue for different and interdisciplinary fields of research. In the same way, knowledge base decision support systems are employed in several fields that need to take a decision based on a large amount of pre-collected information. For this reason, several projects have been proposed in literature. However, to our knowledge there are no works that completely matches what we have discussed so far. In this section we present some of the

works that are relevant to our discussion. Works can be categorized as ontology-based, Petri net and graph based, and web-based systems.

7.1 Ontology-based systems

Applied to risk management, Ekelhart *et al.* presented a ontology-based system that aims at acting as a decision support system [10]. The work proposed by the authors relies on a methodology, called AURUM (*AUtomated Risk and Utility Management*), which is used to perform risk estimation, risk reduction and defense cost estimation. The expert system proposed by the authors is able to support decisions in risk analysis, mitigation and safeguard evaluation.

Fenz *et al.* have proposed an expert system aiming at semi-automatically inferring the needed controls to protect a system using an ontology. The expert system, named FORISK [11], is the result of an extension of two their previous works [12], [13]. It is a formal representation of information security standards, risk determination and automated identification of countermeasures.

7.2 Petri nets

Dalton *et al.* have shown that it is possible to model and probabilistically analyse attack trees by using Petri nets [14]. This method aims at automatically simulate the system behavior and, alongside the attack tree methodology, identify the proper countermeasures.

Dahl *et al.* introduced an interval timed coloured Petri net based mechanism that is able to analyse multi-agent and multi-stage attacks [15]. The proposed method automatically identifies vulnerabilities in network-based systems.

Coloured Petri nets have been also used by Wu *et al.* to model hierarchical attacks. Attacks are subdivided into high level and low level attacks. The former ones represent all the paths and the system vulnerabilities exploited to perform the attack. Based on this modelling it is possible to deduce attack cost estimation and risk measurements. The latter ones use separated coloured Petri nets to describe the details of the attack transitions. This level enhance the attack understanding and the effective countermeasure identification.

Yao *et al.* have recently proposed a Petri net based mechanism to analyse SDN threats [16]. In the proposed mechanism they use Petri nets to model the SDN structure and data flow. Then, they employs attack trees to model the attacks. Anyway, they only proposed a method for modelling attacks without delivering any mechanism to deduce countermeasures or cost evaluations.

7.3 Web-based and Bayesian network

Xie *et al.* based their network security analysis under uncertainty on Bayesian networks [17]. This approach aims at improving the enterprise security analysis. They built the Bayesian network on the security graph model. They tested and validated the approach using attack semantics and experimental studies. They also demonstrated that the system does not suffer parameter perturbation.

A Bayesian network based risk management framework have been proposed by Poolsappasit *et al.* [18]. The framework allows system administrators to evaluate the chance of network compromises, foreseeing how to mitigate and managing them. The system relies on a genetic algorithm that can perform single and multiple objective optimization of security administrator objectives.

A comparison of common methods for security information sharing has been performed by Steffan *et al.* [19]. They evaluated the capabilities of the systems in supporting avoidance and discovering of vulnerabilities. Finally, they suggested a method based on collaborative attack modelling. The proposed methodology combines graph-based attack modelling with a collaborative web-based tool.

Basset *et al.* developed a method to analyse network security based on probabilistic graphs [20] modeling actors, events and attributes of a network as nodes. Then, an estimate is associate to each node; it represents the ease of realizing the event, condition or attribute of the node. Attacks are modelled as paths in the graph that reach compromising conditions. They finally associate a probability to each edge in the attack paths thus allowing the final attack chance.

8 Conclusions and Future Work

In this paper we have extended the work initially proposed in [1]. Our approach semi-automatically constructs a representation of an application source code, searches the attacks against software assets and identifies the protections that can mitigate them, performing a risk analysis and protection evaluation cycle.

We have shown that our implementation is able to infer a great number of attacks that could be probably be identified by manual inspection of the code in a very long time. Based on this information, it assesses the risks against the application assets. Furthermore, our inference engine suggests if a protection can mitigate an attack, hence it is able to propose how to protect an application in an automatic way and estimates the residue risk, with minimal user intervention.

For the future, we aim to boost the performance of our approach and support more complex inference rules. We are planning to add the support to suggest an optimal sequence of protections that can be applied to an application, taking into account their synergies (protections that are known to work well when applied on the same assets) and their suggested order of application of the protections.

Acknowledgment

The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement number 609734.

References

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

2. Shields, T.: Anti-debugging - a developers view. Technical report, Veracode (2009)
3. Anckaert, B., Madou, M., De Sutter, B., Bus, B.D., Bosschere, K., Preneel, B.: Program obfuscation: a quantitative approach. In: Proceedings of QOP 2007: the 3rd Workshop on Quality of Protection. (2007) 15–20
4. De Sutter, B.: D2.08 ASPIRE Offline Code Protection Report (2015)
5. Ceccato, M., Preda, M.D., Nagra, J., Collberg, C., Tonella, P.: Barrier slicing for remote software trusting. In: Proceedings of 7th IEEE International Working Conference on Source Code Analysis and Manipulation 2007. (2007) 27–36
6. Chang, H., Atallah, M.J.: Protecting software code by guard. In: Proceedings of CCS 2001: the 8th Conference on Computer and Communications Security. (2001) 160–175
7. Falcarin, P., Carlo, S.D., Cabutto, A., Garazzino, N., Barberis, D.: Exploiting code mobility for dynamic binary obfuscation. In: Proceedings of the WorldCIS 2011: 1st World Congress on Internet Security. (2011) 114–120
8. Collberg, C., Thomborson, C., Low, D.: A taxonomy of obfuscating transformation. Technical report, University of Auckland (1997)
9. Coker, G., Guttman, J., Loscocco, P., Herzog, A., Millen, J., O'Hanlon, B., Ramsdell, J., Segall, A., Sheehy, J., Sniffen, B.: Principles of remote attestation. International Journal of Information Security **10** (2011) 63–81
10. Ekelhart, A., Fenz, S., Neubauer, T.: Ontology-based decision support for information security risk management. In: Proceedings of ICONS 2009: the 4th International Conference on Systems. (2009) 80–85
11. Fenz, S., Neubauer, T., Accorsi, R., Koslowski, T.: Forisk: Formalizing information security risk and compliance management. In: Proceedings of DSN-W 2013: the 3d Conference on Dependable Systems and Networks Workshop 2013. (2013) 1–4
12. Ekelhart, S.F.A.: Formalizing information security knowledge. In: Proceedings of CCS 2009: the 4th International Symposium on Information, Computer, and Communications Security. (2009) 183–194
13. Fenz, S., Ekelhart, A., Neubauer, T.: Information security risk management: In which security solutions is it worth investing? Communications of the Association for Information Systems **28** (2011) 329–356
14. Dalton, G.C., Mills, R.F., Colombi, J.M., Raines, R.A.: Analyzing attack trees using generalized stochastic petri nets. In: Proceedings of IAW 2006: the 4th Information Assurance Workshop. (2006) 116–123
15. Dahl, O.M., Wolthusen, S.D.: Modeling and execution of complex attack scenarios using interval timed colored petri nets. In: Proceedings of IWIA 2006: the 4th International Workshop on Information Assurance. (2006) 157–168
16. Yao, L., Dong, P., Zheng, T., Zhang, H., Du, X., Guizani, M.: Network security analyzing and modeling based on petri net and attack tree for sdn. In: Proceedings of ICNC 2016: the 5th International Conference on Computing, Networking and Communications. (2016) 1–5
17. Xie, P., Li, J.H., Ou, X., Liu, P., Levy, R.: Using bayesian networks for cyber security analysis. In: Proceedings of DSN 2010: the 40th International Conference on Dependable Systems and Networks. (2010) 211–220
18. Poolsappasit, N., Dewri, R., Ray, I.: Dynamic security risk management using bayesian attack graphs. Dependable and Secure Computing, IEEE Transactions on **9** (2012) 61–74
19. Steffan, J., Schumacher, M.: Collaborative attack modeling. In: Proceedings of SAC 2002: the 17th ACM symposium on Applied computing. (2002) 253–259
20. Bassett, G.: System and method for cyber security analysis and human behavior prediction (2016) Patent US 9292695.