Automatic estimation of the likelihood of cyberattacks in critical cyber-physical systems

No Author Given

[1] IBISC, Univ. Evry, Université Paris-Saclay, France
hanna.klaudel@univ-evry.fr
[2] IRT SystemX, Palaiseau, France
artur.rataj@irt-systemx.fr
[3] SafeTech Cybernetics, Palaiseau, France
witold.klaudel@outlook.fr

Abstract. We propose a model of a cyberphysical systems allowing an automatic estimation of a likelihood of cyberattacks. It differs from the most existing approaches by proposing (1) a construction of a model generated from the description of the architecture of the analyzed system and (2) an exhaustive search method for possible attacks.

1 Introduction

Identifying threats to the cyber-physical systems is challenging and difficult as they consist of many different entities often deployed in safety-critical sectors exposed to a wide range of cyberattacks. A cyberattack is understood as a sequence of unitary actions taken by an *attacker* to take control of some components of the system. This sequence starts with one or more entry points and ends with the loss of integrity of some system components leading in turn to damage targeted by the attacker. Identifying cyberattacks is crucial for optimising architectures and cybersecurity features in order to meet an acceptable level of risk. It is necessary to compute the cyberattack likelihood and so contributes to the estimation of the overall risk level.

When it comes to interpreting the concept of likelihood, many national cybersecurity agencies (e.g., NIST [15], ANSSI [14]) issue loosely defined recommendations and defer final decision to experts. In this work, we lean towards the ANSSI approach and we translate the likelihood into a cost that is inversely proportional to it. This allows us to calculate the cost of the entire attack as a sum of costs of all its unitary actions.

In most approaches, a formalism of attack-trees [13, 19, 9] (or variants of it like forests *e.g.* [18] or graphs [3, 5, 11, 7, 17]) is used to describe possible ways attacks can happen. Edges of an attack tree correspond to atomic attacks allowing the attacker to put the system into a desired insecure state. Attack trees offer a possibility of some quantitative analysis regarding costs, probabilities or likelihood, globally or for individual sub-attacks [2]. This allows them to be combined with cybersecurity features [1, 9], which can then be optimally adjusted to meet an acceptable level of risk.

Often the task of creating the attack tree falls to security specialists and remains manual. Since the manual creation of the attack tree is tedious and source of errors, many authors propose generating it automatically ([20, 18, 12, 8, 16, 6, 10, 5, 7]). As the basis for the attack tree generation, the authors choose a set of predefined generic atomic attacks (often called *atomic actions*) that the attacker can perform to put the system into the insecure state. The chosen set corresponds to known vulnerabilities present in the system or to zero-day flaws that the expert decides to consider. The languages proposed to describe atomic actions make it possible to define system state variables and attack propagation rules. Each action has pre-conditions and post-processing impacting the state of the system. Particular attention is paid to the reachability through the network of the attacked component by the attacking one. The evolution of the system state represents the attacker's progress towards the attack target. All authors cited above consider the system to be secure except for flaws explicitly introduced by atomic actions specified by experts. The main criticism against this approach is that it is prone to combinatorial explosion related to the number of atomic actions taken into account, which makes manual approaches non effective. This number is then often restrained in order to make it possible the generation of the attack tree. Also the above approaches are mostly suitable for audits of already existing systems but are of limited interest for the design phase and for the selection of the best choice of cybersecurity features for the system under construction.

In this paper we propose a model of a cyberphysical systems allowing an automatic estimation of cyberattack likelihood. It differs from the approaches cited above by proposing an automatically generated model following to a certain degree the approach initiated by Ritchey and Ammann in [17]. Our model is obtained from the description of the software and hardware architectures of the analyzed system and generic unitary attack propagation rules associated with classes of software components and communication protocols. Statically it forms a graph whose nodes are software components (hosted by hardware ones) connected by functional interaction links (physically connected through the network). In order to model attack propagation, nodes are in one of possible states such as functional, non-available, tainted-data or malware, which may change according to a set of attack propagation rules. Attacker's presence in the model is represented as a state different from functional of some software components. So, the model is fundamentally a network of automata in which possible attacks may be searched for using model checker and temporal logic queries from its initial state including the initial presence of an attacker.

The definition of automata constitutes another innovative aspect of the proposed model, as they result from an interpretation of the loss of classic cybersecurity properties such as integrity, availability and confidentiality. Losing integrity by a component leads to one of two possible states: tainted-data, when the component produces tainted-data reducing its propagation capabilities, or malware, when the component gains very strong propagation capabilities and

can impersonate all other states. The loss of availability leads to non-available state, while that of confidentiality results in updating the list of stolen secrets.

The architecture description language used for model generation is not in the scope of the present paper. It includes most of cybersecurity features present in typical cyberphysical systems such as routing restrictions to isolate subsystems, operating systems enabling secure computing resource sharing, or application access control and security of network interactions to protect application resources. It is enriched with additional concepts such as application redundancy or security keys and the attack propagation patterns.

Paper structure: .....

2 Proposed approach of modelling

We consider the information system as a superposition of software and hardware architecture where software components are hosted by the hardware ones. Each software component exposes interfaces, which are necessary for its normal functioning but can be abused by an attacker in order to penetrate the system. Interface exposure depends on functional complexity of the system but also on hardware architecture constraints such as resource sharing and network routing.

We assume that an attacker can pass through any interface but with different costs. The definition of these costs is an important part of the system specification, which is used to generate the model to be queried to search for possible attacks. Attack discovery corresponds in this context to the search for sequences of interface crossings leading from the initial state of the system with one or more attackers positioned in software components to certain (other) undesirable target states. From the huge number of possible attacks we select using model checking queries those having a minimal cost or costs under some fixed bound. This bound corresponds to the maximum likelihood needed to maintain the acceptable level of risks.

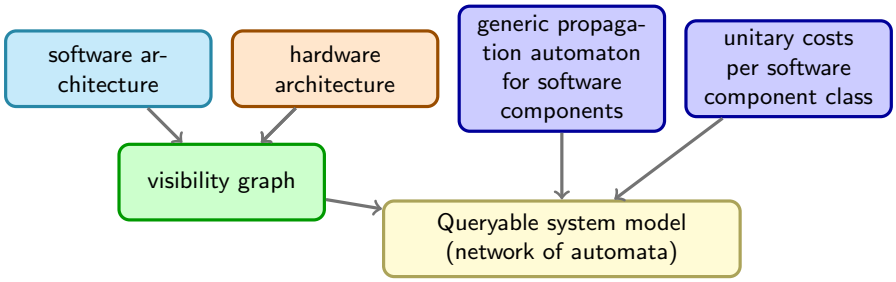

Fig. 1. Model construction elements and dependencies

As shown on Figure 1, a model generation goes through an intermediate construction called a *visibility graph*, which results from the synthesis of software and

hardware architectures and focuses only on the information flow allowing propagation of the attacker in the system. In order to obtain the queryable model, the visibility graph is completed by the definition of a generic automaton specifying the behavior of a software component under attack and the corresponding costs.

2.1 Hardware and software architecture

Hardware components are connected by undirected network links and the software ones are connected by directed *functional interactions* encapsulated in communication protocols, *e.g.* HTTP for web applications. Software interaction endpoints are attached to software components through software ports called *roles*, *e.g.* client and server for HTTP protocol. Software components are hosted by hardware components and functional interactions are routed through network links. We assume that each software component is assigned to exactly one hardware component. A privileged software component may also manage its hardware host, including subordinate software components within the same hardware host and possible network or intra-system routing rules.

The hardware link types correspond to the communication link type such as for example, Ethernet or CAN network. Hardware links are connected to hardware components through hardware ports. Hardware components can be equipped with operating systems but can also manage software components with the "bare metal" approach (without operating system). Each hardware component has its own *routing rules*, which determines the possible physical paths of interactions. These rules are quite general: they define sets of allowed interactions which can be routed through hardware ports to software components, from one hardware port to another, and also between software components sharing the same hardware. The first case describes the situation of a software components interacting with another one through the network, the second the behavior of actual network router, and the third inter-process interactions between software components sharing the same hardware.

Routing rules have to ensure a physical realization of all specified functional interactions. Often, due to technical and organizational constraints, routing rules are more permissive than necessary, which can result in creation of supplementary attack opportunities.

Example 1 (Toy example - software and hardware architectures). In order to illustrate the concepts introduced above let us consider a toy example of a system with a HTTP server and a user browser connected to it, see Figure 2. The hardware component server hosts two software components: HttpSrv for the HTTP functionalities and Linux as an operating system managing server. The userPc hardware hosts the Browser and Windows operating system. The router hardware hosts only the Routing firmware (with operating system privileges). The detailed software architecture of the hacker is not in the scope of our interest and is modeled as a unique software component: an operating system.

The Browser and the HttpSrv communicate through a router, which allows only the HTTP traffic towards the HttpSrv. The server and the userPC limit

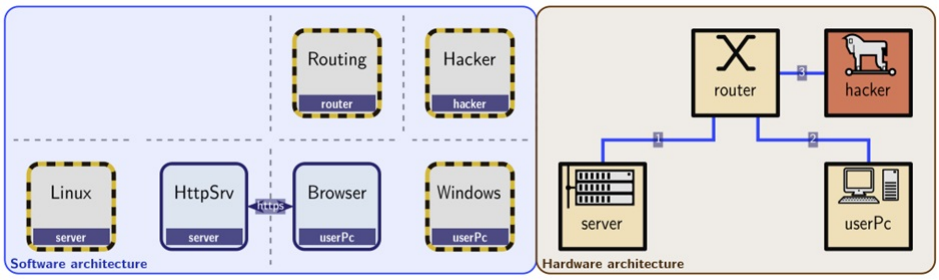

Fig. 2. Toy example of software and hardware architectures.

external communications to https interactions. The intruder (hacker), connected to the router, will try to compromise the system.

◊

2.2 Visibility graph

The nodes in the visibility graph are software components.

In order to define its edges, we need to introduce the auxiliary concepts of *implicit interaction* and *system role*, which will be used in its construction.

Implicit interactions relate to the sharing of hardware components and network links by software components and from the dependence of software components on operating systems managing hardware ones. Functional interactions are specified explicitly in the model, while implicit interactions are added automatically via templates following a chosen interpretation, like *e.g.* that of a typical operating system wrt the applications under its management or that of an application which attacks an other application through a functional role of the latter but which is not in the functional scope of the former.

System roles relate to the interfaces exposed by kernels[4] to their applications or to network interfaces which can be used to create an implicit interaction. All such implicit interactions are considered, provided they are authorized by the routing rules.

The edges of the visibility graph are obtained from all functional and implicit interactions. Each interaction is directed and gives rise to an edge connecting a node to the role of its target node (source role is no longer useful). Edges going to functional roles are labeled by the *attacker position* relative to this interaction:

- `peer` if the attacking node is a functional peer within that interaction;
- `mitm` (man in the middle) if the attacking node is on a possible routing path taken by the interaction;
- `side` if the routing rules merely allow the attacking node to see the target role but the node is not in `peer` or in `mitm` positions.

[4] In this paper, at this level of abstraction, we do not distinguish between kernels and operating systems and use indifferently both names.

Other edges are not labeled.

Definition 1. *A* visibility graph *is a tuple* $VG = (N, R, E)$ *where*

- N *is the set of nodes (software components);*
- R *is a set of* roles *of all the nodes;* $R = R_1 \cup \ldots R_{|N|}$ *where all* R_n *are pairwise disjoint and each role has a* type *(system or functional);*
- $E \subseteq N \times R$ *is the set of edges, each of them is labeled by the attacker position:* `peer`, `mitm` *or* `side`*;* $E = E_{\texttt{peer}} \cup E_{\texttt{mitm}} \cup E_{\texttt{side}}$.

The visibility graph is a view of the information system comprising nodes (software components), which can propagate their potentially corrupted statuses to their successors through the roles of the latter.

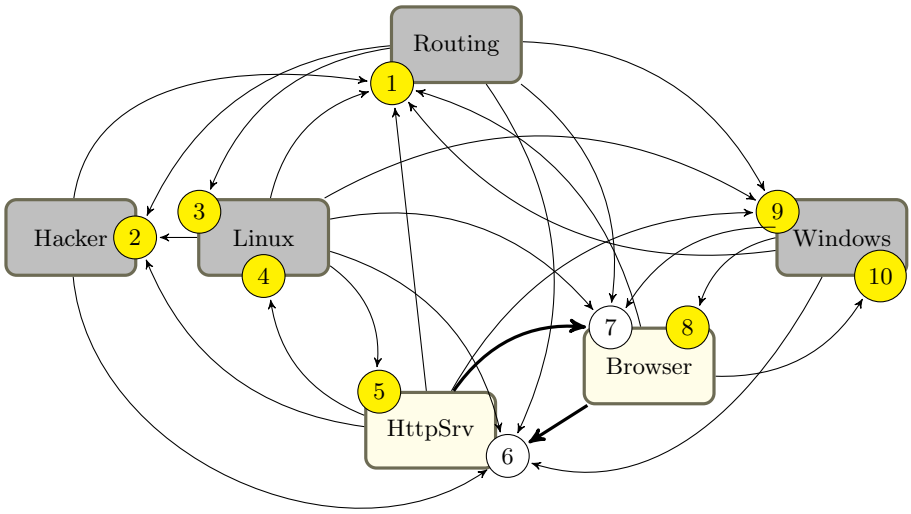

Fig. 3. The visibility graph of the toy example. Kernel components are depicted in gray, system roles are depicted in yellow while functional ones in white. Functional edges are thick while implicit are thin.

Example 2 (Toy example - visibility graph).

Figure 3 presents the visibility graph of the toy example. It defines the neighbourhood of all nodes according to the routing rules. System roles corresponding to kernel-network interfaces are depicted in yellow, kernel-application interfaces in light yellow, while functional ones in green.

Routing is a central point of the system and the routing firmware, when compromised, can attack the entire system. As a consequence, Routing has visibility on all roles of all software components except the kernel-application ones

(light yellow): it can attack Hacker, Linux and Windows through their network kernel roles (3,2,9), and HttpSrv and Browser through their functional roles (6,7). Edges from Routing to functional roles (6,7) are labeled `mitm` because the network path of bidirectional interaction between HttpSrv and Browser passes through Routing.

Due to restricted routing rules, the Hacker's visibility is limited to the functional role of HttpSrv (6) and the kernel-network role of Routing (1). Its attack position for role 6 is `side` because it is neither a partner nor on the path of the functional interaction between Browser and HttpSrv.

Linux visibility contains the kernel-network roles of Hacker, Routing and Windows (roles 1,2,9) and functional roles of HttpSrv and Browser (6 and 7). HttpSrv is also visible through the kernel-application role (5). The edge from Linux to role 7 is labeled `peer` because kernels can perfectly impersonate all software components under their responsibilities. The edge from Linux to role 6 is labeled `mitm` because a kernel (Linux) is present on the network path of the functional interaction between Browser and HttpSrv.

Windows visibility contains the kernel-network role of Routing (1) and functional roles of HttpSrv and Browser (6 and 7). Browser is also visible from Windows through the kernel-application role (8). The edge from Windows to role 6 is labeled `peer` due the impersonation capacity of Windows. The edge from Linux to role 7 is labeled `mitm` because a kernel (Windows) is present on the network path of the functional interaction between Browser and HttpSrv.

HttpSrv sees the kernel-network roles of Hacker (2), Routing (1) and Windows (9), and the kernel-application roles of Linux (4) and the functional role of Browser (7).

Browser sees the kernel-network role of Routing (1), the kernel-application role of Windows (10), and the functional role of HttpSrv (6).

One may observe that the notion of the attack position impacts only the edges pointing to the roles of functional interactions (in green in the figure). The difference between two types of system roles is used here in order to give the intuition behind the construction of the visibility graph but will not be useful in the final model. ◊

2.3 Attack propagation specification and costs

Given a visibility graph and some auxiliary information such as thresholds, secrets and roles categories, we are able to define the system model, which is essentially a network of identical automata $\{A_1, \ldots, A_{|N|}\}$ (one for each node of the visibility graph). As they are identical, we shall define only one generic automaton, and each copy of it will differ only by its current state in $L = \{\mathcal{F}, \mathcal{N}, \mathcal{T}, \mathcal{M}\}$ meaning respectively functional, not available, tainted data or malware. A node n is considered to be *active* if its automaton is in a state different from $\mathcal{N}$.

We define a system model in several steps:

– we define first an *attack propagation specification* AP, which encompasses the visibility graph and additional specific information such as secrets, thresholds, and roles categories;

- a *configuration* of AP is the state of automata of its nodes and the state of the secrets, which is subject to evolve during the execution. The state of the automaton of a given node is called its *status* and is given by the configuration;
- the generic automaton has transitions and updates that depend on AP and its configuration, firing a transition yield a new configuration;
- the firing of transitions is equipped with a *cost function* that evaluates the cost for the attacker to perform a transition;
- the semantics defines how firing of transitions from an initial configuration allows to build executions gathered in a state-space.

Definition 2 (Attack Propagation Specification). *An* Attack Propagation Specification *is a tuple* $AP = (VG, \texttt{cat}, \texttt{thresh}, \texttt{open}, S)$ *where:*

- $VG = (N, R, E)$ *is a visibility graph,*
- $\texttt{cat}\colon R \to \{\texttt{man}, \texttt{opt}, \texttt{transp}\}$ *is the role* category*, which may be respectively mandatory, optional or transparent*[5]*;*
- `thresh` *encompasses two thresholds:*
  - $\texttt{thresh}_{\mathcal{N}}\colon N \to \mathbb{N}$ *is an* activity threshold *indicating for each node the minimum number of optional roles having active predecessors, which are necessary to preserve node's activity;*
  - $\texttt{thresh}_{\mathcal{T}}\colon N \to \mathbb{N}$ *is a* plausibility threshold *indicating for each node the minimum number of optional roles accepting plausible tainted data necessary to fool the node's informational redundancy and possibly change its status to* $\mathcal{T}$*;*
- $\texttt{open}\colon E \to \varphi$ *associates with each edge in* E *a Boolean formula over statuses of nodes and indicates the conditions for the edge to be enabled for communication;*
- S *is a finite set of globally shareable* secrets *with three functions:*
  - $\texttt{Store}\colon (N \times S) \to \mathbb{B}$ *indicating which secrets are stored in nodes,* i.e. *from which nodes they can be stolen,*
  - $\texttt{Protection}\colon (E \times S) \to \mathbb{B}$ *indicating which secrets protect communication sessions represented by edges;*

Each edge in E in the visibility graph represents a communication session between two software components, component n on role r and its predecessor $m \in \texttt{pre}(n) = \{m \mid \forall r \in R_n, (m, r) \in E\}$. In order the session to be `open`, at least one communication routing path should be available, *i.e.* the statuses of router nodes on this path should be different from $\mathcal{N}$. This is expressed statically by a Boolean formula φ obtained from the software and hardware architectures when generating the visibility graph. Each node whose status occurs in φ belongs to $\texttt{pre}(n)$ and is in the `mitm` attack position.

Example 3 (Toy example - `open` *formulas).* Let us look more closely at two edges in visibility graph from Figure 3:

[5] System roles category is always `transp` because they do not impact the behavior of software components

– The edge from *Browser* to role 6 of *HttpSrv* corresponds to the functional interaction between the software components and which is routed trough the *router* managed by the *Routing* software. In order to be open, the edge needs availability of three software components: $Windows$, *Routing* and $Linux$. The corresponding logical formula is then

$$L(Windows) \neq \mathcal{N} \wedge L(Routing) \neq \mathcal{N} \wedge L(Linux) \neq \mathcal{N}$$

where $L(.)$ denotes the status of the component.
– The edge from *Browser* to role 10 of $Windows$ corresponds to the implicit interaction between *Browser* and its managing software $Windows$. In our convention this interaction is always open and the corresponding formula is just `true`. ◊

Secrets in S refers to security keys protecting communication sessions, which can be stolen from the nodes where they are stored. Once stolen, they impact the cost calculation by eliminating costs of breaking communication protocols' security of edges (interactions) they protect. Stolen secrets are visible globally.

Definition 3 (Configuration). *A configuration of an* $AP = (VG, \texttt{cat}, \texttt{thresh}, \texttt{open}, S)$ *with* $VG = (N, R, E)$ *is a pair* $(\overrightarrow{q}, \overrightarrow{s})$*, where* $\overrightarrow{q} \in L^{|N|}$ *is the vector of states of the automata and* $\overrightarrow{s} \in \mathbb{B}^{|S|}$ *the current value of secrets, which may be stolen (*`true`*) or not (*`false`*).*

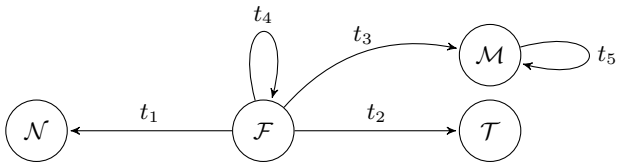

Fig. 4. General shape of automaton A_n of node $n \in N$.

Definition 4 (Generic automaton). *The behavior of each node* $n \in N$ *is defined by the automaton* $A_n = (L, T, \gamma, \delta)$ *shown in Figure 4, where:*

– L *is the set of its states;*
– $T = \{t_1, t_2, t_3, t_4, t_5\}$ *is its set of transitions as shown in Figure 4,*
– $\gamma \colon T \to \mathbb{B}$ *is a guard of each transition, and*
– $\delta \colon T \times \mathbb{B}^S \to \mathbb{B}^S$ *is an update of secrets' global knowledge.* ◊

Intuitively, the status change of node n depends on its own status and on the status of its predecessors in $\texttt{pre}(n)$. More precisely, the impact of a predecessor m on n may vary depending on the role of n and its category and type, the node m is connected to. It may also depend on the attacker position labeling the edge pointing to a role r of n.

From the attacked node's role perspective, the status of its predecessor may be interpreted as functional data, non available data, tainted plausible data, tainted non-plausible data, malware code injection, remote stealing of secret keys, or it may be not considered; the latter occurring if the predecessor is not able to impact the status of attacked node. Tainted plausible data is understood as a tainted data, which is accepted as correct by the role plausibility check. Each of these interpretations leads to some unitary cost participating in the calculation of the global cost of the attacked node status change. Two situations have to be discussed: the `open` formula evaluates to `true` and the `open` formula evaluates to `false`, *i.e.* the predecessor can communicate or not with the node n. For the the `open` formula evaluation to `false` only predecessors in the `peer` position are considered and are always interpreted as non available data. For the the `open` formula evaluation to `true` the following cases exist:

- status $\mathcal{F}$ is always interpreted as functional data;
- status $\mathcal{N}$ is always interpreted as non available data if the predecessor attacker position is `peer`, otherwise it is not considered;
- status $\mathcal{T}$ may be one of the following if the attacker position is `peer`:
  - Tainted plausible or tainted non-plausible, or
  - Malware code injection, with three type of code: non-operational, fully operational, or tainted data.

  otherwise it is not considered;
- status $\mathcal{M}$ in any attacker position can give any of the above interpretations and a supplementary one: remote stealing of secret keys.

Given the above interpretations, we are able to define the conditions of the change of status of node n in the automaton of n, *i.e.* the transitions guards.

1. Transition t_1 from status $\mathcal{F}$ to $\mathcal{N}$ is enabled if
   - there exist an non-available data or tainted non-plausible data interpretation pointing to at least one of the mandatory roles of n, or
   - there exist a set of non-available data or non-plausible tainted data interpretations pointing to a subset R' of optional roles of n, $R' \subseteq R_o$ such that $|R_o \setminus R'| < \texttt{thresh}_{\mathcal{N}}$, or
   - there exists at least one interpretation of type malware code injection pointing to one of the roles of n. This means that the attacker intends to inject the restricted malicious code forcing the status $\mathcal{N}$ for n.
2. Transition t_2 from status $\mathcal{F}$ to $\mathcal{T}$ is enabled if
   - there is a set of interpretations I for which the conditions for passing to status $\mathcal{N}$ are not fulfilled and
     - there exists in I at least one tainted plausible data interpretation pointing to one of the mandatory roles of n, or
     - there exists a subset of I of tainted plausible data interpretations pointing to a set R' of optional roles such that $|R'| \geq \texttt{thresh}_{\mathcal{T}}$, or
   - there exists at least one interpretation of type malware code injection pointing to one of the roles of n. This means that the attacker intends to inject the restricted malicious code forcing the $\mathcal{T}$ status for n.

3. Transition t_3 from status $\mathcal{F}$ to $\mathcal{M}$ is enabled if there exists at least one interpretation of type malware code injection pointing to one of the roles of n. This means that the attacker intends to inject the fully operational malicious code (malware) to the node n.
4. Transition t_4 from status $\mathcal{F}$ to $\mathcal{F}$ is enabled if there are locally stored not already stolen secrets and if there exists at least one interpretation of type remote stealing of secret keys pointing to one of the roles of n.
5. Transition t_5 from status $\mathcal{M}$ to $\mathcal{M}$ is enabled if there are locally stored not already stolen secrets. It is the only transition which does not rely on states of predecessors of n.

The triggering of transitions comes with costs which encompass the following different aspects:

- $\kappa_E \colon (E \times \{\texttt{theft}, \texttt{destr}\}) \to \mathbb{N}$ defines unitary costs of breaking communication protocols corresponding to edges for two types of break: session theft (`theft`) or session destruction (`destr`);
- $\kappa_R \colon ((R \times L \setminus \{\mathcal{F}\} \times \{\texttt{code}\}) \cup (R \times \{\texttt{data}, \texttt{steal}\})) \to \mathbb{N}$ defines unitary costs of breaking node's functionalities bypassing role protections. For `code`, it gives the cost of malicious code injection (non-operational, fully operational, or tainted data only) forcing the transition of the node respectively to status $\mathcal{N}$, $\mathcal{M}$ or $\mathcal{T}$. For `data` it gives the cost of producing tainted plausible data, while for `steal` the cost of stealing secrets by remote malware;
- $\kappa_N \colon N \to \mathbb{N}$ defines the cost of stealing secrets by the node itself when its status is already $\mathcal{M}$ (malware).

Definition 5 (Computation of costs of transitions).

- *For $t \in \{t_1, t_2, t_3, t_4\}$ in the automaton of n, a set of predecessors $M \subseteq \texttt{pre}(n)$ and a choice of interpretations I of their statuses satisfying the enabling conditions of t, the cost of t, $\kappa_{M,I}(t)$, is the sum for all $m \in M$ if $\texttt{open}((m,n)) = \texttt{true}$ of:*
  - *$\kappa((m,r), s, a)$ corresponding to the cost of breaking communication protocol between m and n on its role r, with s representing the current possession of secrets, and a being one of two possible attacks on sessions (`theft` or `destr`);*
  - *$\kappa(r, \texttt{data})$, $\kappa(r, \texttt{steal})$ or $\kappa(r, l, \texttt{code})$ depending on one of possible attack types (`data`, `steal`, `code`) through role r, where the first is the cost of producing tainted data accepted by n's plausibility check, the second is the cost of stealing secrets by remote malware and the last is the cost of malicious code injection of type (non-operational, fully operational, or tainted data) leading to the corresponding status l ($\mathcal{N}$, $\mathcal{M}$, $\mathcal{T}$, respectively),*
- *For transition t_5, $\kappa(t_5) = \kappa(n)$.* $\Diamond$

One may observe that a given transition in the automaton of n may be enabled for various choices of predecessors of n and of the interpretations of its statuses, each of them leading potentially to a different cost. In that case only the minimal cost is retained.

Definition 6 (Semantics). *The semantics of an AP from an initial configuration* $c_0 = (\overrightarrow{q_0}, \overrightarrow{s_0})$ *is a transition system* $(\texttt{Config}_{AP}, \xrightarrow{\kappa}, c_0)$ *where* $\texttt{Config}_{AP}$ *is the set of all configurations reachable from* c_0 *by executing transitions defined as follows: A transition* $(\overrightarrow{q}, \overrightarrow{s}) \xrightarrow{\kappa} (\overrightarrow{q'}, \overrightarrow{s'})$ *exists if*

- *there exists an enabled transition* $t_n = (\eta, \eta')$ *with minimal cost* κ *in the automaton of* n*, and all other states in* $\overrightarrow{q}$ *are unchanged,* i.e. $\overrightarrow{q}[n] = \eta$*,* $\overrightarrow{q'}[n] = \eta'$*, and for all* $i \in [1, |N|], i \neq n, \overrightarrow{q'}[i] = \overrightarrow{q}[i]$*, and*
- $\overrightarrow{s'} = \delta_n(t_n, \overrightarrow{s})$, i.e. *the secrets are updated.* ◊

Hanna: ———— we are here———— still in progress———

Hanna: Donner l'intuition sur ce qu'on peut en faire à partir du moment où on l'a

3 Case study

Hanna: Décrire l'exemple de la Figure 5 et les résultats de requêtes.

Uppaal [4]

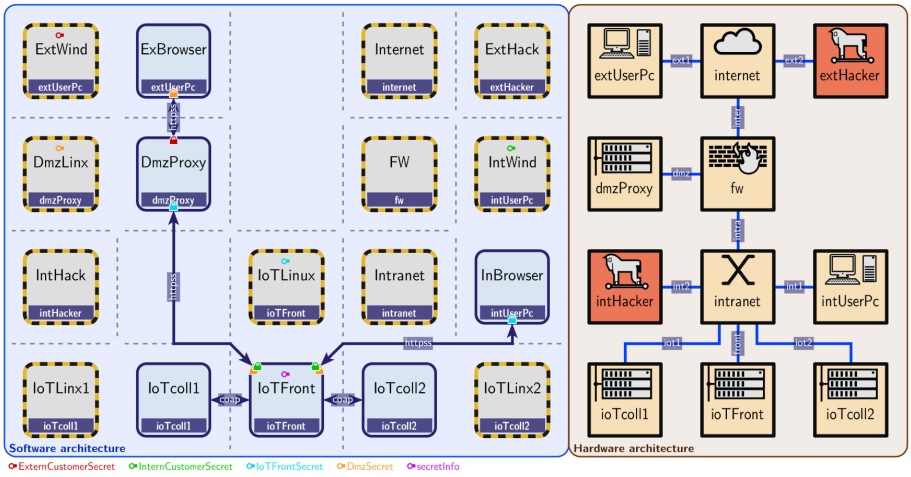

Fig. 5. A bigger example.

4 Conclusion

4.1 Proposed approach in the life cycle of the analyzed architectures

Witold: Il faut en parler dans le papier, comparer avec d'autres approches et démontrer qu'on peut facilement faire ce que d'autres font, c'est à dire prendre en compte la connaissance des bugs en diminuant les coûts de nos protections ou en mettant les couts infinis pout bloquer certains passages considérés comme non franchissables. Ceci est déjà mentionné dans l'intro mais pas assez développé ce qui est normal. Une fois la démarche décrite on peut y revenir dans les conclusions.

References

1. Zaruhi Aslanyan and Flemming Nielson. Pareto efficient solutions of attack-defence trees. volume 9036, pages 95–114, 04 2015.
2. Alessandra Bagnato, Barbara Kordy, Per Håkon Meland, and Patrick Schweitzer. Attribute decoration of attack–defense trees. *International Journal of Secure Software Engineering*, Apr 2012.
3. Mridul Sankar Barik, Anirban Sengupta, and Chandan Mazumdar. Attack graph generation and analysis techniques. *Defence science journal*, 66(6):559, 2016.
4. Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on UPPAAL. In *Lecture Notes in Computer Science*, volume 3185, pages 200–236. Springer, 2004.
5. I. Chokshi, N. Ghosh, and S. K. Ghosh. Efficient generation of exploit dependency graph by customized attack modeling technique. In *2012 18th International Conference on Advanced Computing and Communications (ADCOM)*, pages 39–45, Los Alamitos, CA, USA, dec 2012. IEEE Computer Society.
6. Francesco Di Maio, Roberto Mascherona, and Enrico Zio. Risk analysis of cyber-physical systems by gtst-mld. *IEEE Systems Journal*, 14(1):1333–1340, 2019.
7. Kerem Kaynar. A taxonomy for attack graph generation and usage in network security. *Journal of Information Security and Applications*, 29:27–56, 2016.
8. Kerem Kaynar and Fikret Sivrikaya. Distributed attack graph generation. *IEEE Transactions on Dependable and Secure Computing*, 13(5):519–532, 2016.
9. Barbara Kordy, Sjouke Mauw, Saša Radomirović, and Patrick Schweitzer. Attack–defense trees. *Journal of Logic and Computation*, 24(1):55–87, 06 2012.
10. Rajesh Kumar, Stefano Schivo, Enno Ruijters, Buğra Mehmet Yildiz, David Huistra, Jacco Brandt, Arend Rensink, and Mariëlle Stoelinga. Effective analysis of attack trees: A model-driven approach. In *FASE 2018*, pages 56–73. Springer International Publishing, 2018.
11. Harjinder Singh Lallie, Kurt Debattista, and Jay Bal. A review of attack graph and attack tree visual syntax in cyber security. *Computer Science Review*, 35:100219, 2020.
12. Heng Li, Yongjun Wang, and Yuan Cao. Searching forward complete attack graph generation algorithm based on hypergraph partitioning. *Procedia Computer Science*, 107:27–38, 2017. Advances in Information and Communication Technology: Proceedings of 7th International Congress of Information and Communication Technology (ICICT2017).
13. Sjouke Mauw and Martijn Oostdijk. Foundations of attack trees. In *Information Security and Cryptology-ICISC 2005: 8th International Conference, Seoul, Korea, December 1-2, 2005, Revised Selected Papers 8*, pages 186–198. Springer, 2006.
14. Agence nationale de la sécurité des systèmes d'information. Ebios risk manager, 2019. Accessed: November 4, 2023.

15. National Institute of Standards and Technology. Identifying and estimating cybersecurity risk for enterprise risk management, 2021. Accessed: November 4, 2023.
16. K. Piwowarski, K. Ingols, and R. Lippmann. Practical attack graph generation for network defense. In *2006 22nd Computer Security Applications Conference*, pages 121–130, Los Alamitos, CA, USA, dec 2006. IEEE Computer Society.
17. R.W. Ritchey and P. Ammann. Using model checking to analyze network vulnerabilities. In *Proceeding 2000 IEEE Symposium on Security and Privacy. S&P 2000*, pages 156–165, 2000.
18. Martin Salfer, Hendrik Schweppe, and Claudia Eckert. Efficient attack forest construction for automotive on-board networks. In Sherman S. M. Chow, Jan Camenisch, Lucas C. K. Hui, and Siu Ming Yiu, editors, *Information Security*, pages 442–453, Cham, 2014. Springer International Publishing.
19. Bruce Schneier. Attack trees. *Dr. Dobb's journal*, 24(12):21–29, 1999.
20. O. Sheyner, J. Haines, S. Jha, R. Lippmann, and J.M. Wing. Automated generation and analysis of attack graphs. In *Proceedings 2002 IEEE Symposium on Security and Privacy*, pages 273–284, 2002.