Towards a Formalisation of Expert's Knowledge for an Automatic Construction of a Vulnerability Model of a Cyberphysical System

Witold Klaudel and Artur Rataj
IRT SystemX, Palaiseau, France

Keywords: Security Modelling, Cyberphysical System, Attack Graph, Graph Algorithm, Probability.

Abstract: We present a method for a quantitative formulation of the knowledge of security experts, to be used in an evaluation of attack costs in a cyberphysical system. In order to make the formulation practical, we classify the attacker forms and its attack positions. Applying boiler-plate patterns, like that of an operating system, is also possible. The obtained cost model may allow an exhaustive analysis of hypothetical weaknesses, employed in the design phase of a critical system.

1 INTRODUCTION

Estimation of vulnerabilities of computer systems (Best et al., 2007; Gao et al., 2013) and in particular cyberphysical systems (CPS) (Al-Mohannadi et al., 2016) is a current topic due to the growing number of critical applications such as autonomous cars. Yet, we find that there is a lack of methods for formalising expert's knowledge into a quantitative form in order to produce an abstract vulnerability model, as opposed to one representing concrete vulnerabilities.

There is a number of schemes of security estimation (Everett, 2011) and attempts to systematise them (Dubois et al., 2010) which, in the case of information technology, result in labelling of hardware and software components with differently understood security levels (Matheu-García et al., 2019). Such labels represent a confidence and are often a statistical estimation of possible future vulnerabilities, and not a set of concrete fatalities. Thus, they do not allow a model of concrete exploits but an approximation of potential vulnerabilities, which is a form chosen in our approach. As opposed to frameworks like (Ekstedt et al., 2015), our method is not intended to perform searches for known security bugs. Such bugs, if known by the expert, would instead decrease the estimation of protection quality of corresponding security barriers. We find that modelling of concrete vulnerabilities, while essential for security audits, is of a limited utility in the phase of a product design. Consider for example a widely-known remote attack on the steering wheel of a car (Miller and Valasek, 2015) where there was a port open to everyone on a cellular network, giving access to a critical D-bus functionality. If this weakness had been known to the security expert, he would probably never have accepted it. Otherwise, being unknown, the weakness would not be part of the model anyway.

In order to facilitate the formalisation of expert's knowledge and make it reusable, we classify different security aspects, such as the attacker's state, his attack position or privileges of functions. Combined, they can form different attack scenarios. For example, having three changing attacker forms {malware, bad data, non-availability} we may already model a malware which injects an infected package into an update system. This, combined with attack positions, can be used to model e.g. a security key stolen from the update server. We also hierarchically classify functional components and communication protocols, so that the quantified knowledge can be a reusable asset not pertaining only to a single analysed system.

We focus on security, but as we support different states of the attacker, the method can also be applied to safety estimation. For example, a failing sensor might be seen as an "attack" initiated by an element producing abnormal data or becoming unavailable.

The structure of the paper is as follows: we start with a study of related work in Sec. 2. In Sec. 3 we discuss how the input model is translated into a graph of visibility between functions (Sec. 3.3) which is subsequently enhanced by the combined classifications (Secs. 3.4). This leads to a graph of costs (Sec. 3.5). Section 4 provides an example and Sec. 5 concludes the paper.

Klaudel, W. and Rataj, A.
Towards a Formalisation of Expert's Knowledge for an Automatic Construction of a Vulnerability Model of a Cyberphysical System.
DOI: 10.5220/0010207003910398
In *Proceedings of the 7th International Conference on Information Systems Security and Privacy (ICISSP 2021)*, pages 391-398
ISBN: 978-989-758-491-6

2 RELATED WORK

As the presented method is a workflow from model specification to attacks graphs, we divide the related work in a similar manner.

2.1 Models

Model-based engineering (MBE) defines the development procedure from ideas to product (here, a CPS) and is a proven approach to abstraction and modularisation (Kossiakoff et al., 2011) which increases understanding and thus facilitates analysis (Eppinger et al., 1994). The lowest practical abstraction for model specification are formal automatons, like Markov processes or probabilistic timed automata (Beauquier, 2003). It is a flexible but sometimes tedious way of defining custom systems. For example (Hildmann and Saffre, 2011) study the presence of malicious strategies in a distributed system of energy production with a model specified directly in the language of Prism (Kwiatkowska et al., 2011). This gives access to a vast set of properties calculable in Prism, but is paid by a potentially high cost of a manual model creation. An example of a formalism at a higher level of abstraction is the language AADL (Feiler and Gluch, 2012) dedicated to concurrent systems. (Ibrahim et al., 2020) applied it to model objects as different as a nuclear power plant or a vehicular network system. However, AADL does not explicitly support attack costs or probability, thus finding of basic properties of interest here is not supported. Many custom approaches exist. For example, a modelling software Securicad (Ekstedt et al., 2015) uses its own model representation, which operates on lists of concrete, predefined attacks (also possible in our framework, but not discussed here).

2.2 Attack Graphs

An attack graph (AG) is a formal representation of possible attack paths in a system (Sheyner et al., 2002; Zeng et al., 2019; Jha et al., 2002). A node in an AG may represent not only attacker's position but a general system state (Sheyner et al., 2002). An edge represents a unitary attack step (Zeng et al., 2019); edges are typically directed, which may express an asymmetry of attacks costs, depending on which of the two nodes is the attacking one.

An AG can also be represented as an automaton [15]. AADL, and a number of other architectural languages, have toolchains which allow translation to such automata – the mentioned (Ibrahim et al., 2020) translates its models to Lustre (Pilaud et al., 1987), for which model checking tools exists. In (Ekstedt et al., 2015) AGs use a custom representation where costs are represented by probabilities and delays.

3 PROPOSED WORKFLOW

The workflow starts with a graph of functions and their interactions. It should also be completed with information resulting from the physicality of the model: kernels and routers should also be present as functions, even if they do not normally make part of a pure functional architecture. Routing rules, and thus the resulting network visibility, should be known.

We will start with a function definition (Sec. 3.1). Then we describe completion of the functional interaction with system-level interactions (Sec. 3.2) and the so-called intercepting attacks (Sec. 3.3). Finally, we propose an interpretation which uses the classification in question (Sec. 3.4) which, accompanied by vulnerability costs, allow the generation of an attack graph (Sec. 3.5).

3.1 Functions and Dependencies

Functions are calculating entities. Each is of some class like for example `PosixCriticalApp`. Classes are proposed and precised as a consensus between experts which facilitates a coherent estimation of vulnerabilities. A more fine-grained classification is possible if increased precision is desired. For example, there might be need for a refinement of a general class `PosixCriticalApp` by introducing its two subclasses: of a simple proxy `HttpBasicProxy` and of a proxy with advanced security mechanisms `HttpEnterpriseProxy`.

Dependencies between functions are defined by functional interactions (FI). A FI joins *ports* of two functions and is directed which conveys a dependency in a producer-consumer relation. An FI has a specific protocol class. Ports have individual names which define their roles. For example, an interaction `https` has two port named `client` and `server`. Communication via an FI must be realised by some infrastructure (like the kernel of an operating system and/or a network router).

See that the direction of a FI may be independent from the roles of ports. For example, a server of geographical data with malware may infect its client and thus, it would be the client which is a dependent consumer. Similarly, the server, if put out of order, may cause a cascading unavailability in a tree of recursive consumer functions which directly or indirectly depend on the server. But the functional dependency

may also be opposite if the same protocol is used: consider a HTTP client which infects a database via a HTTP server. A bidirectional service is represented by two parallel opposing FIs and depicted by a single line with two arrowheads.

We model an operating system by adding a kernel function, discussed in detail in Sec. 3.2. In order to model processes (as understood in POSIX-compliant operating systems or in Autosar (Fürst et al., 2009)) functions can be grouped into a process. A process may have additional vulnerabilities due to decreased security measures, like the presence of a common memory.

3.2 System Interactions

System interactions (SI) model boiler-plate vulnerabilities which result from traits of operating systems or hardware. Because these traits are repetitive (e.g. a kernel can practically always kill a user process) our method adds them to the model using predefined templates.

Details of determining SIs depend on the organisation (presence of an operating system etc.) in a physical entity, further called a terminal. A terminal can represent a multitude of hardware, like a machine running a cartography server, a battery control module, a network switch or even a simple sensor. Each function is associated with some terminal. Let us discuss some examples of organisation within a terminal.

Kernel-based Operating System. To model such a system, we use the template illustrated in Fig. 1 which shows all potentially possible candidates for SIs. The candidates are checked against actual physical links and routing rules in the model. The ones which are actually realisable become SIs. Protocols of SIs are

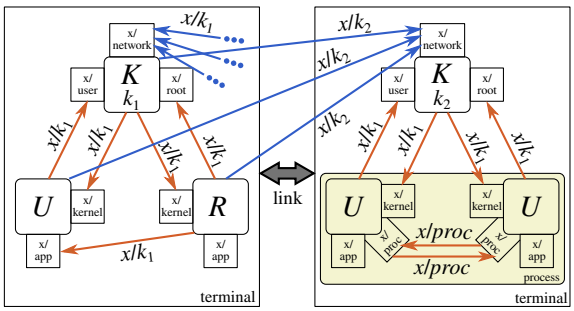

Figure 1: A template for system attacks within an operating system with a kernel: K is a kernel, U a user application and R a user application with root privileges, interactions are depicted with arrows and decorated with protocols.

named after the scheme $x/\langle$kernel class$\rangle$ where the prefix $x/$ denotes a special protocol uniquely for system interactions, as opposed to function–specific protocols of FIs. The naming scheme reflects that the quality of the kernel is a primary factor in deciding about the cost of system attacks in a computer system managed by that kernel.

The template in Fig. 1 reflects several common traits of an operating systems:

- a compromised K may easily attack U or R, e.g. by killing them or by watching their memory - it is represented by ports $x/$kernel;
- as opposed to R, K is a managing software which thus controls ports belonging to any application in its system, which adds to K's vulnerability by making it listen to the network and is represented by its ports $x/$network;
- any application may attack the kernel via the API of the latter (ports $x/$user); normally these attacks tend to be very expensive as they rely on e.g. unpatched kernels which should be rare in critical systems;
- R may attack U thanks to its root privileges (ports $x/$app);
- due to common resources, functions within a process have additional ways of attacking one another via ports $x/$proc.

An example of filtering of canditates for SI: R (from the left computer system in Fig. 1) is limited by local routing rules (if K is not compromised) and thus the respective SI between R and U is not always realisable.

No Kernel. An embedded system often consists of a set of functions without a kernel which would otherwise protect one function from another. This lack of routing rules results in a total local visibility of ports within the set. To generate SIs, the same template is used as before (Fig. 1), but all functions are R (have root privileges) and the name of the kernel class in protocol definition is empty.

Other templates similar to that in Fig. 1 can be added, e.g. to model hypervisors.

3.3 Visibility Graph

A visibility graph $\mathcal{V}$ is an intermediate structure before the construction of an attack graph which completes it with costs. In $\mathcal{V}$ nodes are functions and edges are the so-called visibility vectors (VV), derived from FIs, SIs and the so-called interception interactions (II) (introduced at the stage of $\mathcal{V}$) representing the interception attacks (like man-in-the-middle).

A VV is a one-hop directed adjacency between a function f_1 and a port p of a function f_2 and can be interpreted as a potential unitary step within an attack. The attacker at f_1 must see p via at least one of possible means, like a network route or a system call. This is true for SIs, thanks to the filtering of SI candidates. As of FIs, any FI without the visibility of p of the interaction's consumer makes the whole model ill-defined, because functional interactions, being part of the system by design, must by principle be also be physically realisable. All VVs are additionally decorated with a further discussed attack position of the attacker.

At most two VVs are derived from an interaction:

- a forward attack, from the interaction's producer to its consumer;
- a reverse attack, from the consumer the producer; does not exist for IIs, as in their case, by definition, the attacker is already at the source; reverse attack are often unviable, e.g. if the source port is physically not receiving any information.

Actual viability of an attack via a VV depend on further discussed attack costs which are specified as infinite or non-existing if an attack is not viable (the latter quality differs from the former only in disallowing the presence of associated interactions which in turn improves finding errors in model specification).

During the construction of a $\mathcal{V}$, IIs are searched for by exhaustively checking each FI for its possible interceptions, the condition being the visibility of the FI's producer or consumer port by an intercepting attacker. We do not repeat the process for SIs, as the attacker never attacks a SI from the side, because if such an interaction were possible, it should already be introduced directly to a respective template as another SI.

In order to differentiate attack costs more finely, each VV has a class determining the attacker's attack position (AP). Let us define a physical communication path (PCP) as an (unordered) set of all terminals which may potentially take part in the transport of data of an FI, including the host terminals of two concerned peer functions. Depending on the physical position of the attacker, we divide VVs into the following classes:

- AP S (source): attacker is at one of the two end points of the attacked interaction; thanks to this, he may e.g. steal the security keys more easily and use them on the other peer; all VVs derived from FIs and SIs are of the class S, because the other classes model only intercepting attacks;
- AP P (path): concerns only IIs; a man-in-the-middle-scheme where the attacker function is in a terminal belonging to PCP of a FI to be attacked and the function is also a kernel (thus, a network-managing function) and therefore it may listen to and easily intercept the routed interaction;
- AP A (access): concerns only IIs; not of the class P, but (by definition of a VV) the target (attacked) port is seen; the attacker has a smaller number of possibilities, it may though perform e.g. a DOS.

In case of an II, the attack difficulty always includes both the intercepting attack on a respective FI and a subsequent attack on a peer function of the FI.

A compromised kernel, due to its privileges, may have several impersonation capabilities (here understood as posing as another function) which may trick other functions, in particular kernels which manage the routing rules. We put a line here concerning the impersonation limits which, while not perfect, seems simple and reasonable to us: the impersonation in question may work around routing rules in remote terminals (like a network router), as a kernel may set/alter a network packet's origin data (MAC etc.) as it wishes and routing hardware typically controls just that. Conversely, the impersonation cannot give an attacker a better AP, even if the compromised kernel makes it seem that it is a peer of an FI or a router on a PCP, because:

- an advantage of AP S is associated with confidential information in a VV's peer of the function to attack, and here we assume that it cannot be stolen if the attacker has merely an AP of class P or A (which does not always hold – consider e.g. (Mavrogiannopoulos et al., 2012) where a function stores secure data in the kernel);
- similarly, an advantage of AP P stems from the actual physical position of the attacker, which cannot thus be replaced by impersonation.

The line in question would not work if a peer within a FI authenticated its other peer only via the packet's origin data. In that case, an impersonation would help in gaining advantages of the AP S. We assume, however, that it is an unlikely practice in a critical system.

3.4 Compromised States

We augment the modelling with variable states of the attacker. This will allow further fine-tuning of costs. Namely, the attacker, during its propagation through the system, puts functions into different compromised states (CS):

- malware (M) where the attacker is a live code;
- bad data (B) where the attacker hides in a passive blob of data;

- unavailability (N) where the attacker disables a function.

B should be used only in specialised cases: the idea is, that the attacker stealthy uses a legal (i.e going via functional dependencies) transmission mechanism which can be relatively cheap if it replaces expensive compromise of all functions on the same path into M. An example might be the update system, transporting a compromised binary via a number of proxies, or a faulty speed sensor reporting an invalid speed (see that only in the former case a final transition from B to M seems likely). If the mechanism does not exist or cannot be clearly evaluated for some protocol, the attacks which include the B state should be impossible (which is specifiable via a non-existing attack cost).

In order to justify the function states, we present an interpretation of the resulting attacks types, shown in Fig. 2. The interpretation is arbitrary and does not preclude other interpretations from being compatible with the further presented verification methods. In particular, it is also arbitrary that costs of some edges are equated to 0 in the figure, which denotes an extremely easy attack.

As illustrated in Fig. 2, CSs can be mapped to the elements of the CIA triad. VV's AP divides them further into six categories in total (only in the attacking function, as it is where AP plays a role). The categories M_s and M_p belong, amongst others, to the *compromised* part of the triad, because a live code may take advantage of e.g. a security key stolen from a compromised interaction peer (AP S) or from eavesdropping (AP P) which may allow to e.g. prepare bad data (M_s to B', a primed CS denotes an attacked functions).

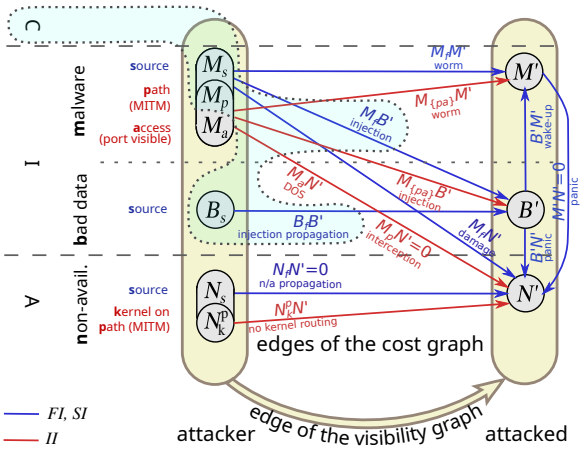

Figure 2: Example of possible attack types, with interpretation. Also illustrates the transformation from $\mathcal{V}$ to $\mathcal{C}$.

Let us add more detail to the interpretation of edges depicted in the figure.

- From M:
  - $M_{\{spa\}}M'$ could be a typical worm;
  - $M_{\{spa\}}B'$ is a concealed injection of tainted dangerous data;
  - M_sN' could symbolise a panicked kernel or another function forced into a non-active state;
  - M_pN' is a physical blocking of an interaction, which renders its functionally dependent consumer inactive; it is an easy attack to perform by e.g. a switch;
  - M_aN' could be a DOS attack.
- From B: passive attacker cannot intercept an interaction, thus only AP S.
  - B_sB' is a further, passive propagation of bad data, assumed to passes all easily all verifications, because it did so already when initially injected ($M_{\{spa\}}B'$); not possible for SI, because the transfer of bad data, as said, concerns only legal mechanisms.
- From N: unavailability, by disabling infrastructure may sometimes intercept an interaction.
  - N_sN': attacker has once induced unavailability, which then propagates by itself in cascade via FIs due to recursively broken consumer–producer dependencies;
  - N_k^pN': a panicked kernel stops routing and thus breaks all concerned interactions via IIs having AP P.
- Inside the attacked function:
  - $B'M'$: a wake-up of malware from a binary blob into an active form (typical to update subsystems);
  - $B'N'$: malformed data consumed by a function finally inactivates it; it is a passive equivalent of M_sN';
  - $M'N'$ is an in-application equivalent of $M_{\{s\}}N'$.

A kernel manages network interfaces and inter–process communication, thus compromising it causes the following changes in visibility:

- M disables restrictions imposed by local routing rules within a terminal, as this is the kernel which controls them;
- N disables an activity of all local network interfaces, i.e. links from/to the respective terminal can no more be used, except for further propagation of N.

3.5 Cost Graph

An attacker pays the cost of infiltrating a system. It is represented by a cost graph $\mathcal{C}$, that is, an AG where a unitary attack step is decorated with a cost expressing its difficulty. A cost graph is automatically derived from the visibility graph by augmenting it with information about CS: every node f (a function) in $\mathcal{V}$ is transformed into three nodes in $\mathcal{C}$: f_M, f_B and f_N, indexed with a respective CS; every edge (a VV) in $\mathcal{V}$ is transformed into a set of edges joining the triplets, using the pattern in Fig. 2. We assume that the total cost of an attack is a sum of costs of unitary steps. It may be realistic if it is about monetary costs, but otherwise it may require transforming functions in order to be realistic.

3.5.1 Valuation of Costs

The following factors influence the cost:

- the combination protocol class/attacked port class (e.g. `http`/`client`); this cost element is expressed in the number of stars from 0 to 5 with a step of $1/4$, representing the difficulty from very easy to extremely expensive;
- a quality multiplier of the class of the attacked function (one which contains the attacked port) which is a real value with 1 being the neutral default; for example, a more costly database may have a higher quality;
- attack position;
- CS of both the attacking and the attacked function: see that the pair, together with the attack position, determine the attack types enumerated in the interpretation in Sec. 3.4.

All these factors are discrete and thus number of cost values is finite and can be stored in a simple (cost) table. Creating it might be, though, a rather involved process due to its size. It is, however, a crucial element of a realistic vulnerability modelling. The effort is a formalisation of expert's knowledge and thanks to the generality of classes on the level of abstraction on which we operate, such a table may serve as a general library representing a consensus within a group of security experts, reusable in a number of models.

Using the number of stars is intuitive but does not reflect well summing of costs of unitary attacks. A very easy attack of one star does not have as much as 1/4 costs of a four-star attack, but the ratio is likely much larger. Thus, summing of four one-star attacks cannot produce a value which represents a four-star attack. To accommodate for that, if s is the number of stars, q is the quality and c is the cost, let

$$c = \exp(as)q$$

where a is the strength of the exponential growth to be decided. For $a = 1$ which we use in the further example, the ratio in question would be $\exp(4)/\exp(1) \approx 20$.

3.5.2 Artifacts of $\mathcal{C}$

Because $\mathcal{C}$ should be static and reasonably small for efficiency, the transition from $\mathcal{V}$ to $\mathcal{C}$ introduces certain artifacts to the modelling. Let us consider if they are not too troublesome, comparing to the abstractions which have already been introduces (like the quantisation to only three CS). Firstly, the division of nodes when transforming $\mathcal{V}$ to $\mathcal{C}$ makes it possible for DST to return to the already compromised function if the respective CS is different from the original one. It adds certain possibilities but also minor modelling problems (further, the symbol $\curvearrowright$ denotes 'returning to the same function', and by viable we understand possibly profitable for an optimal attacker):

- $B \curvearrowright \{M,N\}$: having passed in the past via a function as bad data was a secret operation not damaging that node, thus it did not change the costs of its future compromise into M or N, so no problem with the evaluation of costs;
- $N \curvearrowright \{M,B\}$ is never possible (see Sec. 3.4);
- $M \curvearrowright N$ is not viable, because a direct attack $M'N'$ is cheap (Fig. 2);
- $M \curvearrowright B$ is problematic cost-wise because it is a reasonable assumption that an already compromised function would accept bad data at a smaller cost; however, the only reason of trying this would be a future attempt of infecting a non-compromised function with the same bad data, because just passing bad data between functions compromised already is not viable; otherwise, i.e. if there is a non-compromised function to be infected, it means that the bad data must be prepared well enough, which may at least partially justify the higher cost in question.

The two visibility changes due to a compromised kernel, discussed in Sec. 3.4, are realised in the static $\mathcal{C}$ as follows: any additional visibility thanks to reaching the state k_M of a kernel k is reflected by respective edges stemming from k_M, and any visibility lost by reaching k_N is not reflected at all. Let us analyse potential problems here:

- Lost visibility: Reaching k_N means that the attacker, from now on, can only passively propagate unavailability, as its code does not exists anymore

(see the lack of edges in Fig. 2), yet by definition, the lost of visibility does not affect such a propagation (Sec. 3.4).

- Extra visibility: After reaching k_M of a terminal t, the routing rules are compromised which may change t into a more permissive router. It is not reflected by the static $\mathcal{C}$ but neither does it produce modelling artifacts, because taking advantage of the increased permissiveness in the future is never viable. For once, it never concerns attacks via FI, as these have their legal paths so they do not need the increased permissiveness. FIs entail in turn confidentiality advantages of the B state (see the description of B_sB' in Sec. 3.4). Neither it concerns SI, as these can be impersonalised by another SI attacking directly from k_M. As of II, their only advantage of AP P exists only on a PCP of a FI, thus such II do not use the increased permissiveness as well.

4 EXAMPLE

Let us consider two attacks of a certain system. Both begin in a function `Attacker` which exists in an attacker's terminal connected to the diagnostic port, and end in `F4`. In the case of the first attack, `F4` should only be made unavailable, while the other attack must infect it with malware. Let there be also a network switch `Router` through which pass all FI with `F3`. The costs table which produces $\mathcal{C}$ is not shown for brevity.

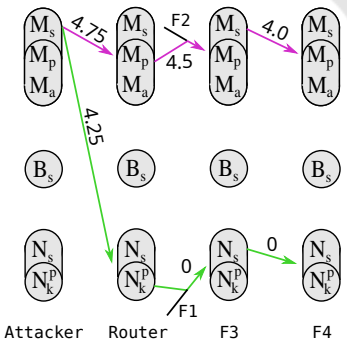

Figure 3: Paths for attacks: 1st (violet), 2nd (green). Angled edges depict intercepting attacks and are labelled with the non-attacked function of the intercepted interaction. As depicted in Fig. 2, arrowheads point to the target function only with the precision of one of three compromised states, while tails also take into account the attack position.

Fig. 3 illustrates example paths. We see that the two respective attacks go via the same functions but compromise them in a different way. In the case of the second attack, after disabling the router, there is only a zero-cost, passive propagation of unavailability: a producer `F1` cannot send data anymore via `Router` to its consumer `F3`, which in turn makes the consumer unavailable. The first attack cannot do this, as the attacker's malicious code must be transferred until the end. Thus, malware is installed in subsequent functions, with an intercepting attack on the interaction F2→F3.

5 DISCUSSION

We have shown how a number of simple classifications, like the attack position (Sec. 3.3) and the classes of compromised states (Sec. 3.4) produce together a versatile and intuitive interpretation (Sec. 3.4) which facilitates the quantification of security experts' knowledge.

Our further work includes a translation from a system documentation given in the form of architectural layers, as practices in the industry (Braga and Sztajnberg, 2004), in order to analyse real-life systems. We will also support a probabilistic analysis on the basis of a distribution of costs formed by opinions of several experts.

REFERENCES

Al-Mohannadi, H., Mirza, Q., Namanya, A., Awan, I., Cullen, A., and Disso, J. (2016). Cyber-attack modeling analysis techniques: An overview. In *2016 IEEE 4th International Conference on Future Internet of Things and Cloud Workshops (FiCloudW)*, pages 69–76. IEEE.

Beauquier, D. (2003). On probabilistic timed automata. *Theor. Comput. Sci.*, 292(1):65–84.

Best, B., Jurjens, J., and Nuseibeh, B. (2007). Model-based security engineering of distributed information systems using umlsec. In *29th International Conference on Software Engineering (ICSE'07)*, pages 581–590. IEEE.

Braga, C. and Sztajnberg, A. (2004). Towards a rewriting semantics for a software architecture description language. *Electronic Notes in Theoretical Computer Science*, 95:149–168.

Dubois, É., Heymans, P., Mayer, N., and Matulevičius, R. (2010). A systematic approach to define the domain of information system security risk management. In *Intentional Perspectives on Information Systems Engineering*, pages 289–306. Springer.

Ekstedt, M., Johnson, P., Lagerström, R., Gorton, D., Nydrén, J., and Shahzad, K. (2015). Securicad by foreseeti: A cad tool for enterprise cyber security management. In *2015 IEEE 19th International Enterprise Distributed Object Computing Workshop*, pages 152–155. IEEE.

Eppinger, S. D., Whitney, D. E., Smith, R. P., and Gebala, D. A. (1994). A model-based method for organizing tasks in product development. *Research in engineering design*, 6(1):1–13.

Everett, C. (2011). A risky business: Iso 31000 and 27005 unwrapped. *Computer Fraud & Security*, 2011(2):5–7.

Feiler, P. H. and Gluch, D. P. (2012). *Model-based engineering with AADL: an introduction to the SAE architecture analysis & design language*. Addison-Wesley.

Fürst, S., Mössinger, J., Bunzel, S., Weber, T., Kirschke-Biller, F., Heitkämper, P., Kinkelin, G., Nishikawa, K., and Lange, K. (2009). Autosar–a worldwide standard is on the road. In *14th International VDI Congress Electronic Systems for Vehicles, Baden-Baden*, volume 62, page 5.

Gao, J.-b., Zhang, B.-w., Chen, X.-h., and Luo, Z. (2013). Ontology-based model of network and computer attacks for security assessment. *Journal of Shanghai Jiaotong University (Science)*, 18(5):554–562.

Hildmann, H. and Saffre, F. (2011). Influence of variable supply and load flexibility on demand-side management. In *Proc. 8th International Conference on the European Energy Market (EEM'11)*, pages 63–68.

Ibrahim, M., Al-Hindawi, Q., Elhafiz, R., Alsheikh, A., and Alquq, O. (2020). Attack graph implementation and visualization for cyber physical systems. *Processes*, 8(1):12.

Jha, S., Sheyner, O., and Wing, J. (2002). Two formal analyses of attack graphs. In *Proceedings 15th IEEE Computer Security Foundations Workshop. CSFW-15*, pages 49–63. IEEE.

Kossiakoff, A., Sweet, W. N., Seymour, S. J., and Biemer, S. M. (2011). *Systems engineering principles and practice*, volume 83. John Wiley & Sons.

Kwiatkowska, M., Norman, G., and Parker, D. (2011). PRISM 4.0: Verification of probabilistic real-time systems. In *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 585–591. Springer.

Matheu-García, S. N., Hernández-Ramos, J. L., Skarmeta, A. F., and Baldini, G. (2019). Risk-based automated assessment and testing for the cybersecurity certification and labelling of iot devices. *Computer Standards & Interfaces*, 62:64–83.

Mavrogiannopoulos, N., Trmač, M., and Preneel, B. (2012). A linux kernel cryptographic framework: decoupling cryptographic keys from applications. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, pages 1435–1442.

Miller, C. and Valasek, C. (2015). Remote exploitation of an unaltered passenger vehicle. *Black Hat USA*, 2015:91.

Pilaud, D., Halbwachs, N., and Plaice, J. (1987). LUSTRE: A declarative language for programming synchronous systems. In *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages (14th POPL 1987). ACM, New York, NY*, volume 178, page 188.

Sheyner, O., Haines, J., Jha, S., Lippmann, R., and Wing, J. M. (2002). Automated generation and analysis of attack graphs. In *Proceedings 2002 IEEE Symposium on Security and Privacy*, pages 273–284. IEEE.

Zeng, J., Wu, S., Chen, Y., Zeng, R., and Wu, C. (2019). Survey of attack graph analysis methods from the perspective of data and knowledge processing. *Security and Communication Networks*.