SAPIENZA
UNIVERSITÀ DI ROMA

Enhancing the Quality of Information Supporting the Cyber Risk Management Process in Self-Protecting Systems

Faculty of Information Engineering, Informatics, and Statistics, SAPIENZA – University of Rome
PhD Program in Engineering in Computer Science (36° cycle)

Marco Cuoci
ID number 1630470

Advisor
Prof. Silvia Bonomi

Co-Advisor
Prof. Giuseppe Santucci

Academic Year 2024-2025

Thesis defended on 24 September 2024
in front of a Board of Examiners composed by:

Prof. Alberto Pretto, University of Padova (chairman)
Prof. Giuliana Vitiello, University of Salerno
Prof. Valeria Cardellini, University of Rome, "Tor Vergata"

Enhancing the Quality of Information Supporting the Cyber Risk Management Process in Self-Protecting Systems
Sapienza University of Rome

This thesis has been typeset by LaTeX and the Sapthesis class.

Author’s email: cuoci@diag.uniroma1.it

Abstract

The Cyber Risk Management process relies on multiple sources of information, some of which derive from the monitored environment, some of which is stored in external repositories. The availability and the quality of these sources of information plays a critical role during Cyber Risk Management, directly influencing the quality in terms of accuracy and completeness of the related processes. This is especially relevant for ICT systems designed around self-protection (i.e. self-protecting systems), which is currently a desired property of many modern ICT systems as it enriches its features with the ability to detect and react to security threats at run-time. Recently, several solutions leveraging the attack graph model have been proposed to design and implement such self-protecting systems. While such systems take a first step towards effective self-protection, they do not consider: (i) the possibility of having non complete information in the external repositories, (ii) the possibility of having non accurate information in the inventories derived from the environment, and (iii) the limitations in terms of accuracy-scalability trade-off imposed by the usage of the attack graph model.

This thesis represents a first step towards a solution to enhance the quality of information supporting the cyber risk management process in self-protecting systems, and provides the following major contributions: (i) A study of the external publicly available vulnerability repositories, in order to understand their structure, their semantics and how all these repositories can be integrated in a unified structure, able to provide the cyber risk management process with complete, accurate information. (ii) A computational pipeline able to enhance the accuracy of the inventories derived from the environment by reducing the number of false positives contained within, as well as explicitly addressing and instrumenting the accuracy-scalability trade-off imposed by the attack graph model. (iii) A comprehensive evaluation of the proposed methodologies on a case study.[1]

[1] A repository with the implementation and further documentation of the proposed methodologies is available at https://github.com/Marcvs101/enhancing-cyber-risk-management

Contents

Chapter 1

Introduction

State of the art self protecting systems promise to deliver a solution to the challenge of achieving a fully automated system capable of performing Risk Management at runtime, however the intricacies of ICT (Information and Communication Technology) systems are not making this an easy task.

Most notably, much effort has gone towards developing methodologies and techniques to adapt the Risk Management process to the specific domain of ICT, leading to its contextualization into the Cyber Security Assessment process. And while efforts to automate the Cyber Security Assessment process with the intention of enhancing or implementing a self protecting system have been made, to the best of the author's knowledge most of these efforts do not consider the possibility and consequences of being supported by information which may not achieve a perfect degree of accuracy and completeness.

For this reason, it is of interest to study: (i) which information is involved in the Cyber Security Assessment process, with particular attention to the semantics of each information source as well as what is actually required by the process; (ii) methodologies to polish and combine the different sources in order to obtain higher quality and quantity of information and (iii) how the quality of information affects the subsequent analysis steps of the process.

1.1 Risk Management: ISO 31000

Since 13 November 2009, the Risk Management process has been standardized by the International Organization for Standardization in the ISO 31000 family [83, 7]. This standard comes from the consensus of hundreds of risk management professionals around the world through a process which lasted over four years and seven drafts. ISO 31000 has been intended to resolve the many inconsistencies and ambiguities existing between different approaches and definitions by providing a consistent vocabulary and methodology for assessing and managing risk. In particular, the standard provides: (i) a unified vocabulary; (ii) a set of performance criteria; (iii) a consistent methodology for identifying, analyzing, evaluating and treating risk; and (iv) guidance on how to implement the aforementioned methodology into the governance processes of any organization.

Consistency and coherence of the vocabulary used during Risk Management is fundamental to achieve greater clarity and wider understanding of the Risk Management process. For this reason, in addition to this family of standards, a revision of the existing ISO/IEC vocabulary for risk management in Guide 73:2002 [9] has been carried out.

The ISO 31000 family provides clear performance criteria in order to manage risks effectively and efficiently. These criteria are based on the following eleven principles: (i) creating and protecting value; (ii) being an integral part of all organizational processes; (iii) being part of decision making; (iv) explicitly addressing uncertainty; (v) being systematic, structured, and timely; (vi) being based on the best available information; (vii) being tailored; (viii) taking into account human and cultural factors; (ix) being transparent and inclusive; (x) being dynamic, iterative, and responsive to change; and (xi) facilitating continual improvement of the organization. In addition, the standard also provides a set of desired outcomes (i.e. controls) which are to be used as a way to measure the effectiveness of the implementation of a risk management process. The last performance criteria that the standard provides is a list of important characteristics of an advanced risk management process, in which: (i) an emphasis is placed on continual improvement in risk management through measurement; (ii) there is a comprehensive, defined, and accepted accountability for risks, controls, and risk treatment tasks; (iii) all decision making within the organization must involve to some appropriate degree the explicit consideration of risks and the application of risk management; (iv) there is continual communication with external and internal stakeholders as well as comprehensive and frequent reporting of risk management performance, as part of good governance; and (v) risk management is viewed as central to the organization's management processes, such that risks are considered as "effect of uncertainty on objectives".

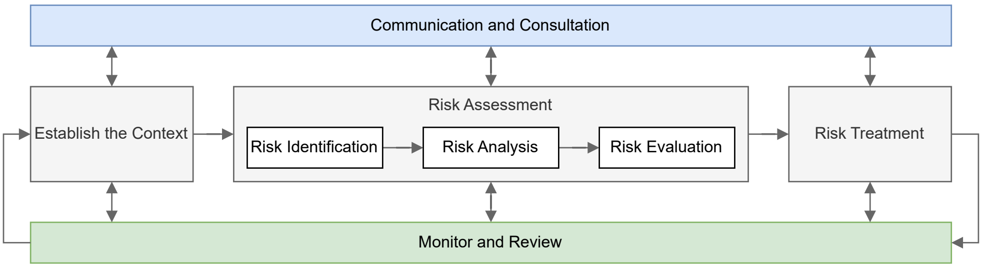


Figure 1.1. High level schema of the Risk Management process, as described in ISO 31000

The process for managing risk adopted by ISO 31000 derives partly from the process described in AS/NZS 4360:2004[1], and follows the structure shown in Figure 1.1. The process is iterative and is composed of a central stack which is responsible for establishing the context (internal and external), for identifying, analyzing and evaluating risks (Risk assessment) and finally, applying risk treatment, as well as two elements which act constantly throughout the process, namely: (i) Communication and consultation with internal and external stakeholders, in order to consider their input and to factor their involvement in the process; and (ii) Monitoring and review, in order to react to new risks and changes in the context (e.g. objectives, environment).

With regards to the central stack, the **first element**, **Establish the Context**, aims to define what is the desired goal of the organization, as well as which internal and external factors which may influence how these objectives are achieved.

The **Risk Assessment** is the **second element** of this stack and is itself composed of Risk Identification, Risk Analysis and Risk Evaluation. Given an established context, Risk Identification aims to understand what could happen, how, when and why. Risk Analysis then aims to study the consequences of each identified risk, and likelihood of these consequences. The standard does not require the analysis to adopt qualitative or quantitative approaches, as both have their own role depending on the type of risk, the information available and the purpose for which the output of process is to be used. Risk Evaluation aims to merge the output of the Risk Analysis into a single, unified prioritization value (i.e. Risk), the details of which have been defined during the establishment of context.

Lastly, **Risk Treatment** is the **third element** of the stack and aims to introduce new controls or improve existing controls in order to lower the overall risk which the organization is subject to. Since the standard defines risk as a combination of likelihood and impact of its consequences, the Risk Treatment element is similar to a process of optimization in which magnitude and likelihood of consequences are adjusted in order to achieve a net benefit. ISO 31000 lists seven general treatment actions which can be performed on risk: (i) avoiding the risk by avoiding the risk-inducing activity; (ii) accepting or increasing the risk in order to pursue opportunity; (iii) removing the source of risk; (iv) affecting the likelihood; (v) affecting the consequences; (vi) sharing the risk with another party; and (vii) accepting the risk via informed decision.

The last part of ISO 31000 provides guidance on how to implement the aforementioned methodology into the governance processes of any organization. This is necessary because in order to be effective, the Risk Management process must be integrated into an organization's decision-making processes. The standard describes not only which elements are needed in order to achieve an integration of the process in an organization's governance, but also goes out of its way to describe how an organization should create, implement and keep these elements up-to-date, while considering also the timescale needed to achieve alignment with the organization's culture and governance processes.

1.2 The Cyber Security Assessment Process

Since ISO 31000 is meant for a broad scope of application, thus generic by design, it has been necessary for the Risk Assessment Process to be contextualized into the Cyber Security Assessment Process (alternatively Cyber Risk Assessment Process) [72] to correctly adapt to the nuances of ICT systems. The Cyber Security Assessment Process focuses on cyber risks and threats, exploiting domain specific representations in order to quantify risk (calculated as a composition of likelihood and impact, as described by ISO 31000) and model threats. One of the main goals of this process includes the study of vulnerabilities and risks associated to an ICT system in order to produce and apply a response plan. This process requires a deep understanding of the ICT system in consideration as well as precise knowledge on the vulnerabilities that are associated to it. As a consequence, currently this process is done prevalently by human analysts with the assistance of tools that, at best, provide the analyst with increased situational awareness.

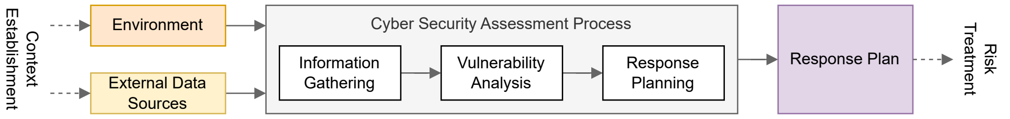


Figure 1.2. High level schema of how the Cyber Security Assessment Process achieves a response plan

Figure 1.2 highlights the basic high level schema of the process focused on producing a response plan. The Process begins with two products of the Context Establishment phase, in particular: (i) the environment (i.e. ICT assets to consider for assessing risk, attack surface) of the organization; and (ii) which external sources of information are to be used to integrate the elements necessary to support the process.

The **first step** of the process is then the **Information Gathering** step. During this step several tasks are performed, such as the identification and documentation of network asset vulnerabilities, the identification and usage of external sources of information such as sources of cyber threat intelligence and vulnerability repositories, the identification and documentation of internal and external threats and the identification of potential mission impacts.

The **second step** of the process is the **Vulnerability Analysis** step. During this step information about vulnerabilities, threats, mission impacts and likelihoods are factored together in order to determine the cyber risk of an organization.

The **third step** of the process is the **Response Planning** step. This step is tasked with producing a list of possible response strategies for each vulnerability, using the cyber risk calculated previously as one of many possible prioritization metrics.

1.3 Self-Protecting Systems

The research community has been engaged in addressing the challenge of achieving a fully automated system capable of performing Risk Management of ICT systems at runtime. This problem poses several technical and scientific challenges, especially in domains such as critical infrastructure, in which complex cyber and physical systems coexist [101, 84, 10, 67, 21, 42, 92, 110, 32, 22], and distributed systems, which strengthen reliability and enhance performance by mitigating possible bottlenecks in computation, at the cost of exposing applications to higher security risks [14, 59].

Moreover, the transition from closed and regulated environments to open ones (where entities cannot always be assumed trustworthy) complicates the detection of failures and anomalies, and rectifying such issues may demand considerable time and resources [111]. Consequently, there has been a growing focus towards self-* systems i.e., systems able to support *Self-configuration* (i.e., automatic configuration of components) [65, 69], *Self-healing* (i.e., automated discovery, and correction of faults) [82], *Self-optimization* (i.e., autonomous monitoring and control of resources to ensure optimal functioning for the defined requirements) [63] and *Self-protection* (i.e., proactive identification and protection from arbitrary attacks) [105, 108].

With particular reference on performing Risk Management at runtime in ICT systems, significant attention has been directed towards *self-protecting* systems, i.e., a particular class of autonomous systems capable of detecting and mitigating security threats at runtime [105].

Similar to other self-* properties, self-protection enables the system to autonomously adapt to dynamic environment with minimal human intervention, thus ensuring responsiveness, agility, and cost-effectiveness. Typically, self-protecting systems consist of a *monitored environment* (i.e., the system to be protected) and of an *autonomic or protecting environment* implementing the protection and adaptation logic [102]. The autonomic environment is typically implemented as a feedback control loop based on the *Monitor-Analyze-Plan-Execute over a shared Knowledge* (MAPE-K) architecture [51, 18] comprising 5 conceptual modules:

- *Monitoring* (**M**) gathers data from the underlying monitored environment through probes (or sensors);

- *Analysis* (**A**) performs data analysis to determine whether an adaptation is necessary;
- *Planning* (**P**) devises a workflow of adaptation actions necessary to achieve the system's objectives if required;
- *Execution* (**E**) carries out the actions identified in the computed plan through actuators acting on the monitored system;
- *Knowledge Base* (**K**) stores data of the monitored environment, adaptation goals, and other pertinent states that are shared among the MAPE components.

The MAPE-K adaptation loop implicitly depends on an *adaptation function* that delineates the protection objectives. Within the cyber security domain, adaption involves analyzing the current *security posture* and taking actions to improve the security posture of the monitored environment. To achieve this, various security metrics are available [80] and can be combined to reflect the current security situation. However, among them, the most suitable candidate for governing the feedback loop is the *risk*, i.e., the quantification of the probability of a specific incident occurring and causing harm to an organizational asset.

Following this paradigm, in recent years, several dynamic risk estimation architectures have emerged, capitalizing on the attack graph model [38, 93, 62, 8], which represent potential attack steps in a network using a graph-based model.

Examples of architectures emerged in recent years are Dynamic Risk Management Systems [39, 38], which are able to assist the security analysts in comprehensively yet accurately mitigate possible and ongoing attacks on monitored ICT systems while upholding the system's mission integrity. According to the work of Granadillo et. al. [39, 38], a Dynamic Risk Management System (DRMS) integrated into a larger ICT system is able to promptly assess risks and address ongoing threats, by focusing on the technical aspects of the monitored environment. This is in contrast with the traditional approach to risk assessment outlined in ISO 31000, which emphasizes establishing a security architecture that spans organizational and business aspects. This contrast is essential because the disparity between technical and organizational levels often results in increased overhead and inertia during the risk assessment process, which is unacceptable in highly dynamic environments such as ICT systems, where both the system itself and the threat landscape are in continuous evolution.

Despite their unique features, these architectures adhere to the same principles underlying a MAPE-K loop, utilizing risk as the controlling variable. Figure 1.3 illustrates the key components characterizing these solutions.

Specifically, the autonomic environment collects information from the monitored environment and external data sources through a *Data Collection, Aggregation and Integration* component. Its objective is to generate data necessary for subsequent components for risk estimation, analysis, and response. Once input data is available, it is forwarded to an *Attack graph generation and risk estimation* component for the analysis phase. This component is responsible for the computation of the actual attack graph to identify potential dangerous attack paths and estimate the risk level based on the identified paths. Subsequently, a *response* component devises a series of mitigation actions that could be implemented to modify the system's exposure to risk. Finally, the execution phase is mediated by a *visual environment* enabling security experts to choose the most suitable response actions among those proposed and thereby bring closure to the feedback loop. A knowledge base is employed to store and manage all the information required to complete the control loop.

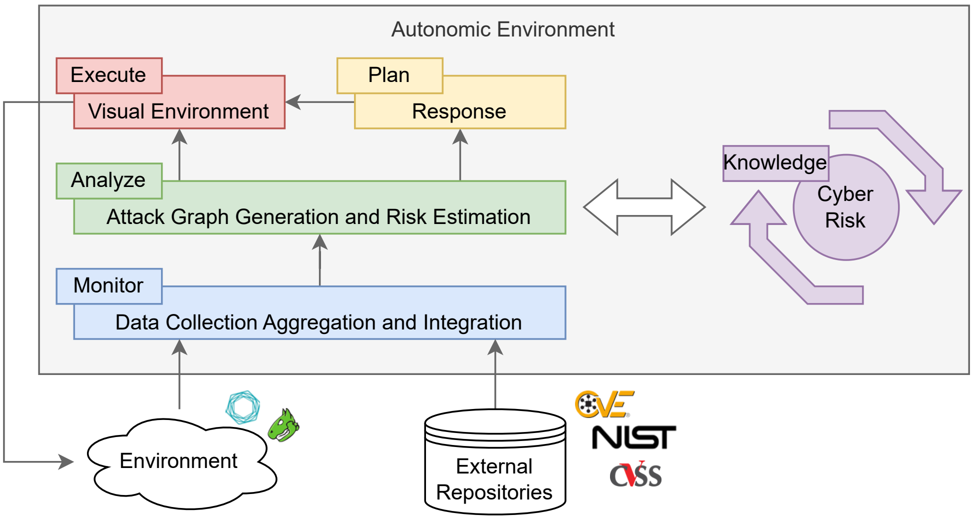


Figure 1.3. MAPE-K architecture for attack graph-based self-protecting systems.

1.4 Problem Statement and Contributions

Self-protecting systems based on the MAPE-K architecture represent the state-of-the-art in the field of Risk Management of ICT systems. While these systems have been proven to be highly capable, both in theory [38, 39], as well as in practice for curated environments [8, 93], no quality assessment has been carried out on the information feeding into these systems, which has always been assumed as trustworthy.

Understanding the sources of information that feed the system, assessing their quality and capabilities and attempting to remediate their flaws is fundamental for the correctness of the system, as the quality of a process can only be as good as the quality of the information it is provided with (*garbage in, garbage out*). For this reason, considerations must be made about the quality of the system, especially when considering the accuracy and completeness of the processes which are part of the system as well as the information used to feed them.

Considerations on Accuracy

A common factor of all implementations of autonomic environments utilizing the MAPE-K architectural framework is the need to ensure the utmost accuracy in risk estimation within the risk-driven feedback loop to guarantee the efficacy of the autonomic environment, and in turn of the self-protecting system. Due to the nature of the MAPE-K control loop, the accuracy of the response is directly correlated to the accuracy of the risk estimation. Risk estimation in state-of-the-art architectures is achieved through attack graph-based methodologies, which, in turn, hinge on the data processing conducted during the Data Collection, Aggregation and Integration (Monitoring) process.

This need for accuracy stands in contrast to the performance demands placed on the autonomous system. Indeed, while state-of-the-art methodologies such as attack graphs are powerful and potentially accurate models, they suffer from scalability issues, forcing to opt for the computation of an *approximated* Attack Graph [68, 70, 43]. This introduces a level of uncertainty into risk estimation, where accuracy is traded off for scalability. The situation is further complicated when considering that state-of-the-art implementations of the Data Collection, Aggregation and Integration (monitoring) process relies on probe-based mechanisms to extract information from the environment. Probe-based mechanisms are not always accurate, and may introduce false positives in subsequent processes, harming the accuracy of the risk analysis. Indeed, the introduction of false positives leads to a cumulative loss of accuracy, initially impacting the data collection phase and subsequentially affecting the attack graph and risk computation phase.

Considerations on Completeness

Currently, no official standard has been adopted to formally define and represent a vulnerability in a ICT network. This has lead to multiple efforts lead by both the scientific community as well as governmental and industrial communities, to define and adopt de-facto standards to represent ICT vulnerabilities, weaknesses, patterns of attack and threats. This effort, in turn, has lead to the establishment of multiple sources of data which are currently used during the cyber security assessment process. Each data source models different, disjoint concepts of the cybersecurity sphere, includes different attributes which depend on the modeled concept, and each source is curated by different entities (most prominent of which are the National Institute of Standards and Technology-NIST and MITRE Corporation). This leads to the consequence that no single data source holds all the information required to fully carry out the cyber security assessment process. As an example, the Planning phase of a state-of-the-art cyber risk management system implemented on the MAPE-K architecture should be able to devise a workflow of adaptation actions (e.g. mitigation actions, patches). This is usually carried out by performing a cost-benefit analysis to gauge the risk reduction of an adaptation action against the cost of applying such adaptation action on the ICT system. But this analysis is only possible if: (i) it is possible to obtain a list of available adaptation actions for a given vulnerability, and (ii) it is possible to give an estimation of the cost of application of each adaptation action. These elements are not always (if at all) present in each data source. This prompts the necessity for an analyst or an automated system to explore, consider and interpret each and every data source in order to gather the necessary information. However each and every data source has its own structure, semantics and intended target audience. In particular, most data sources that deal with cyber risk usually adopt formats aimed predominantly towards human consumption (i.e. natural text). This leads to the result that correctly traversing and interpreting each data source in search for relevant information is not trivial, especially for autonomic systems, and thus is still an open problem.

Thus, given these considerations, this thesis will aim to contribute to the following open problems:

1.4.1 Chapter 3: Integrating Sources of Data to Support Automatic Correlation

This chapter analyzes the most prominent publicly available data repositories used to feed the cyber risk management process, and proposes a methodology aimed at integrating multiple data sources to support automatic correlation and improve the overall level of knowledge available to the process. The main contributions of this chapter are:

- A systematic review of the most prominent publicly available repositories used to feed the cyber risk management process (NVD [75], CWE [31], CAPEC [29] and ATT&CK [94]), aimed at understanding and formalizing their structural properties. During this process, schemas will be presented to highlight each repository's capability and structure, as well as to highlight the presence of references between different repositories.

- The construction of a unified data model, a graph-based representation of elements from each different repository, which is able to explicitly represent and handle the structure and hierarchy of elements of the same repository as well as the links existing between elements of different repositories.
- The proposal of a methodology to be implemented in the *Data Collection, Aggregation and Integration* component of an autonomous system enabling the automated retrieval of critical information (e.g., which mitigation actions are available for a given vulnerability, what is the cost of applying each mitigation action) required in order to carry out the cyber risk management process by exploiting the unified data model in order to perform cross-repository searches while being mindful of each repository's internal structure and semantics.

Part of the work detailed in this chapter has been submitted in journal format to ACM Digital Threats: Research and Practice, as *Silvia Bonomi, Hérve Debar, Marco Cuoci* - "Integrated knowledge graph to support automatic correlation in the vulnerability analysis process". This submission received a major revision request and is currently undergoing rework for resubmission.

1.4.2 Chapter 4: Enhancing the Quality of Automatically Generated Inventories

This chapter focuses on two problems which currently characterize state-of-the-art self-protecting systems: (i) the improvement of the quality (in terms of accuracy) of the automatically generated inventories without increasing the degree of intrusiveness in the monitored system, and (ii) the analysis of the accuracy-scalability trade-off existing in attack graph-based self-protecting systems governed by risk estimation.

The first problem stems from the introduction of false positives during the *Data Collection, Aggregation and Integration* (Monitoring) phase of a state-of-the-art autonomic environment. The second problem stems from the need of improving the scalability of the self-protecting system and reducing the size of the input processed by the attack graph generator while being mindful of the loss of accuracy that results in doing so.

In order to address these two issues in a single highly configurable solution, this chapter proposes a computational pipeline to be implemented in the *Data Collection, Aggregation and Integration* component of an autonomic system. To this aim, two sub-components will be defined, namely *vulnerability filtering* and *vulnerability aggregation*, that can be used either in isolation or in combination, and for each component, several algorithms and methodologies will be proposed.

It is important to note that the proposed approach will be orthogonal to those available at the state-of-the-art, which only work on the self-protection bottleneck (i.e., the attack graph generation algorithm), at the price of finding approximate solutions without assessing the level of accuracy loss. Conversely, the proposed approach will work on the input data feeding the control loop *sanitizing and compressing* it, trying to avoid the loss of information (i.e., preventing as much as possible and quantifying the accuracy loss in the risk estimation).

Thus, the main contributions of this chapter are:

- A formalization of the problem of enhancing the quality of automatically generated inventories.

- Several methodologies and several algorithms aimed at tackling the problem of enhancing the quality of automatically generated inventories. To achieve this, human knowledge will be leveraged, since it will be assumed that an analyst or system administrator is able to validate a vulnerability by confirming or discarding the presence of the platforms that it affects. Since a human is involved in the process, all the proposals will also carry an additional goal to make this process as efficient as possible.

- A methodology aimed at tackling the problem of improving the scalability of the self-protecting system while being mindful of the loss of accuracy that results in doing so. In order to achieve this, several aggregation strategies able to aggregate semantically equivalent information and to reduce the size of the input processed by the attack graph generator will be proposed.

- A unified computational pipeline to be included in the Data Collection, Aggregation and Integration component of an autonomic system in order to provide a single, highly configurable solution to (i) enhance the quality of automatically generated inventories through the introduction of a *vulnerability filtering* sub-process and (ii) provide an *vulnerability aggregation* sub-process able to operate on the accuracy-scalability trade-off of the risk analysis in order to allow the analyst to tailor the analysis to better fit various use cases.

Part of the work detailed in this chapter has been:

- Published in conference format as: *Silvia Bonomi, Marco Cuoci, Simone Lenti*, “A Semi-automatic Approach for Enhancing the Quality of Automatically Generated Inventories” in proceedings of 2023 IEEE International Conference on Cyber Security and Resilience (CSR) [25].

- Accepted for publication in conference format as: *Silvia Bonomi, Marco Cuoci, Simone Lenti*, “A Version-based Algorithm for Quality Enhancement of Automatically Generated Vulnerability Inventories” (to appear) in proceedings of 2024 IEEE International Conference on Cyber Security and Resilience (CSR).

- To be submitted in conference format as: *Silvia Bonomi, Marco Cuoci, Simone Lenti, Alessandro Palma*, “A Computational Pipeline for Improving the Accuracy-Scalability Trade-off of Self-Protecting Systems based on Attack Graphs” to be submitted on July 15th in 19th International Conference on Risks and Security of Internet and Systems (CRISIS 2024).

1.4.3 Chapter 5: Analyzing the Accuracy-Scalability Trade-off in Attack Graph-Based Self Protecting Systems - A Case Study

This chapter serves as case study and as an experimental validation of the contributions of Chapter 4. In order to perform a thorough validation, which ensures the comparability of each result, a single case study will be constructed using a methodology developed to generate real network environments starting from a desiderata.

The presence of a single case study allows experimental results from each proposed algorithm and methodology to be compared correctly. A single case study is also crucial in the experimental evaluation of the whole computational pipeline, which highlights the pipeline's advantages and the contribution of each block to the definition of the accuracy-scalability trade-off.

Thus, this chapter provides the following contributions:

- An introduction to the accuracy-scalability trade-off existing in state-of-the-art attack graph-based self-protecting systems governed by the risk estimation.
- A methodology to provide real, statistically relevant testing environments through the use of virtualization.
- A single, coherent case study to perform a thorough experimental evaluation of all the methodologies and algorithms introduced in Chapter 4.
- The experimental evaluation of each methodology and algorithm introduced in Chapter 4, as well as their effect with respect to the accuracy-scalability trade-off. The experimental evaluation on the case study will also consider each sub-process of the computational pipeline in isolation, as well as their integration in the whole computational pipeline.

Part of the work detailed in this chapter has been:

- Published in conference format as: *Silvia Bonomi, Marco Cuoci, Simone Lenti*, "A Semi-automatic Approach for Enhancing the Quality of Automatically Generated Inventories" in proceedings of 2023 IEEE International Conference on Cyber Security and Resilience (CSR) [25].
- Accepted for publication in conference format as: *Silvia Bonomi, Marco Cuoci, Simone Lenti*, "A Version-based Algorithm for Quality Enhancement of Automatically Generated Vulnerability Inventories" (to appear) in proceedings of 2024 IEEE International Conference on Cyber Security and Resilience (CSR).
- To be submitted in conference format as: *Silvia Bonomi, Marco Cuoci, Simone Lenti, Alessandro Palma*, "A Computational Pipeline for Improving the Accuracy-Scalability Trade-off of Self-Protecting Systems based on Attack Graphs" to be submitted on July 15th in 19th International Conference on Risks and Security of Internet and Systems (CRISIS 2024).

Chapter 2

Related Work

In the literature, different works design self-protecting systems for different goals [79, 105, 108]. They focus on specific domains, as Yuan et al. [106] and English et al. [35], who propose self-protecting architectures to automatically detect and mitigate cyber threats of software (the former) and provide mitigation actions based on event correlation (the latter). They adapt the MAPE-K principles to dynamically adapt the system at runtime to respond to security threats. Similarly, Liang et al. [61] present a framework to enhance the self-protection of power systems by leveraging the distributed security features of blockchain technology. Finally, other works develop self-protecting strategies for securing data storage, such as Strunk et al. [95], who minimize the performance costs of system versioning, and Kocher et al. [54] who propose to distribute watermarking algorithms to self-adapt protection against piracy. These works highlight the challenges of self-protecting systems related to the risk estimation component that requires human intervention to be as accurate as possible, representing a potential bottleneck of autonomic systems [27]. Attack Graph-based approaches have been developed to address this limitation and automate risk estimation of self-protecting systems. For example, Kavallieratos et al. [48] map DREAD [91], a static threat modelling approach, to attack graph for assessing the risks of attack paths and making the analysis actionable for dynamic risk assessment. Similarly, Gonzalez-Granadillo et al. [38] provides a solution for dynamic cyber risk analysis leveraging an attack graph model where vulnerabilities are represented as CVEs and the assessment is performed starting from CVSS metrics [33]. Such system represents the current state of the art, providing the baseline over which every methodology and algorithm proposed in this thesis will be evaluated and compared against.

According to what has been introduced in Section 1.2, current state-of-the-art systems analyze the risk of an ICT system using mainly two sources of information: (i) external repositories, and (ii) inventories generated from the monitored environment. As it will be shown in the next sections, both of these sources of information and their involvement with the risk management process are the subject of several open research problems.

2.1 On publicly available external repositories

Understanding vulnerabilities affecting information systems is a long-standing activity, established for example by the PhD thesis of Krsul [55] at US CERT and the creation of the *Common Vulnerabilities and Exposures*[1] (CVE) [30] in 1999,

[1] https://cve.mitre.org/

followed by the NIST *National Vulnerability Database*[2] [75] in the US. More recently, the Common Vulnerability Scoring System (CVSS) [66] specified a framework for supporting the evaluation of risks associated with a CVE entry. These efforts have been further structured in order to provide additional information related to the principles of vulnerabilities through the Common Weakness Enumeration[3] (CWE) [31] dictionary, which classifies types of vulnerabilities, the Common Attack Patterns Enumeration[4] (CAPEC) [29], which provides a catalog of known patterns of attack, and the Adversarial Tactics, Techniques, and Common Knowledge[5] (ATT&CK) [94], which provides a guideline for classifying and describing cyberattacks and intrusions.

The information in these repositories is currently used for two main purposes: (i) to analyze, describe and identify vulnerabilities in software and (ii) to support the risk analysis of complex information systems. Grieco et al. [40] use machine learning techniques to predict types of vulnerabilities through large-scale analysis of software. Evans et al. [36] leverage CWE to understand the exploitability of vulnerabilities in software. Ohm et al. [76] propose an analysis of the impact of vulnerabilities affecting linked open source code. Hemberg et al. [41] build the BRON database and show how it can efficiently support cyber hunting activities. Kiesling et al. [52] build the SEPSES Knowledge Graph and show how it is able to support simple tasks. Zhang et al. [109] leverage NVD to propose a way to predict the overall occurrence of new vulnerabilities from the rate of past disclosure. Shen [90] uses the NIST Cybersecurity Framework (CSF) to analyze the impact of software vulnerabilities in the medical critical infrastructure. Fu et al. [37] leverage vulnerability information to manage risks, giving particular attention to the medical infrastructure. Grandillo et al. [38] presents a framework for automatic dynamic risk estimation which uses NVD and CVSS as external data sources.

In all the above-mentioned studies, the role played by publicly available repositories is crucial and the amount and quality of retrieved information represent critical factors to determine the accuracy of the solutions built on top of them. This is even more important in the critical infrastructure domain, at a time where regulations such as the *Network and Information systems Security* (NIS) directive [74] impact the way industries labelled as critical infrastructures, and the vendors which serve these industries must disclose vulnerability information to the regulators. Regulators worldwide are pushing towards the deployment of sectorial *Information Sharing and Analysis Centers* (ISAC), to establish trusted brokers that make available vulnerability information associated with sector-specific mitigation recommendations inside a trusted circle. Thus, the question of the quality and usage of this vulnerability information is highly relevant, as all the use cases presented herein rely on them to provide accurate analysis.

To the best of the author's knowledge, almost all the works dealing with NVD, CWE, CAPEC and ATT&CK repositories focus on how to use these repositories as data sources to feed learning algorithms (e.g., [11, 47, 12]) or to provide basic information for the risk analysis (e.g., [38]). The only works that have been found that actively aggregate the different repositories in order to increase the available information is a work from Hemberg et al. [41] that presents the BRON database and shows how it can efficiently support cyber hunting activities, and a work from Kiesling et al. [52] that presents the SEPSES Knowledge Graph and shows how it is able to

[2] https://nvd.nist.gov/
[3] https://cwe.mitre.org/
[4] https://capec.mitre.org/
[5] https://attack.mitre.org/

support simple tasks in the vulnerability assessment process. These works [41, 52] share the same perspective i.e., merging in a single graph all (heterogeneous) elements extracted from individual repositories to support complex analysis. In all three works, indeed, cross-references existing between repositories are exploited to build edges of graph based structures and all three works propose to store (even if in a different form) such resulting graph in a graph database to efficiently support queries (Neo4J in the work presented in Chapter 3, Arango DB in [41] and RDF in [52]).

The main difference between the two works and the work detailed in Chapter 3 is in the final objective behind the construction of the graph structure. In this work, the interest lies in understanding the properties and structures of every repository and of the resulting unified data model to understand if and how it is possible to extract enough information to support three analysis tasks, mainly performed during risk analysis: (i) identification of the actual vulnerable part within a host, (ii) identification of mitigation strategies to deal with discovered vulnerabilities and (iii) identification of the cost of applying a mitigation. In [41], the main focus is on threat hunting and identification (i.e., a task mainly used in incident management). To this aim, they selected the top 10 exploited vulnerabilities and used the graph structure to identify all possible threats linked to such 10 CVEs. The two works are thus complementary. [41] employs a "top-down" analysis approach starting from all possible threats and progressively filters them to identify only those related to the considered CVE. The work detailed in Chapter 3 takes a "bottom-up" approach as it starts from vulnerabilities (potentially detected by a network scanner) and tries to identify their effect on the system and how they can be mitigated. In [52], the main focus is the formalization of the knowledge base, as well as its maintainability. The resulting structure simplifies significantly the semantics and the structure of the underlying data sources, allowing simple selections to achieve simple goals. However forgoing the semantics of otherwise structured data (e.g. CPE configurations and strings) may not provide desired or accurate results. The work detailed in Chapter 3 aims to understand and include each repository's structure and semantics within the unified data model in order to maintain the best possible accuracy. So, to the best of the author's knowledge, no work that constructs a unified data model to support the Cyber Security Assessment process without forgoing potentially relevant information exists yet.

2.2 On the quality of automatically generated inventories

Much effort has been devoted to identifying known vulnerabilities in a monitored system in order to reduce its exposure. To the best of the author's knowledge, however, far less attention has been dedicated to the quality of produced inventories.

Kocaman et al. [53], Ushakov et al. [99], Tovarnak et al. [98] address the issue of building a device inventory containing Common Platform Enumeration (CPE) entities to retrieve information about vulnerabilities from NIST's National Vulnerability Database (NVD). The main difference between these works is in the CPE selection process. In particular, Kocaman et al. [53] assume the input to be already in CPE format, Ushakov et al. [99] try to reconstruct the CPE strings needed to perform the selection process from product names in system logs, Tovarnak et al. [98] construct a tree structure leveraging NIST's pattern matching specification in order to perform the selection process through graph queries. Neither of the three works deals with ensuring that the inventory produced is accurate and complete.

Several studies have been conducted regarding the vulnerability inventory in order to improve, enrich and provide access to information related to vulnerabilities for the purposes of supporting informed decisions. Russo et al. [86], Warehus et al. [103] and Yosifova et al. [104] use different machine learning, data mining, and natural language processing approaches to enrich every CVE entry with additional information and to provide valuable metrics to improve awareness during the cyber vulnerability assessment process. Other works focus on the support of situational awareness about the vulnerability inventories rather than on their quality, by providing suitable visualizations and analytics (as in Pham et al. [81] and Angelini et al. [15]) or by using data structures which enable further and more structured analyses (e.g., Kiesling et al. [52], Syed [97]).

2.3 On the accuracy and scalability of attack graph based methodologies

In the literature, there are different methodologies to analyze the cyber risks. They constitute the security standards and are OWASP [46], MEHARI (MEthod for Harmonized Analysis of RIsk) [28], and EBIOS [17], which are two-factor risk models as they evaluate the risk of an attack according to its likelihood and impact. Although they define the standard components that must be leveraged when evaluating the cyber risk (i.e., likelihood and impact), they are difficult to adapt to multi-step attacks due to the exploit dependencies that must be taken into account in the evaluation of the risk.

For this purpose, attack graph-based approaches have been born to estimate the risk of multi-step attacks. For example, Kavallieratos et al. [48] map DREAD [91], a static threat modeling approach, to attack graph for assessing the risks of attack paths and make the analysis actionable for dynamic risk analysis. Similarly, Gonzalez-Granadillo et al. [38] provides a solution for dynamic cyber risk analysis based on CVSS metrics [33]. One of the core aspects to underline when using these approaches is that the quality of the results depends on the quality of the input. While this is true for any system [20], it is particularly relevant when managing the risks of multi-step attacks because inaccurate results may provide inaccurate protection.

Regarding the accuracy and scalability of attack graph-based methodologies, the scientific literature analyzes these two issues separately, differentiating the issues of accuracy of the risk analysis based on the attack graph and the issues of the scalability of the graph itself.

Attack graph scalability
Let us note that the attack graph computation and analysis is currently the main bottleneck of attack graph-based self-protecting systems, deriving from the theoretical complexity of the graph exploration. A natural direction to improve the performance of such systems is by adopting scalable attack graph generation algorithms. Several approaches exist in the literature to deal with such issue like (i) computing only shortest paths [49], (ii) pruning the exploration up to a given path length [49], (iii) considering distributed [50, 77] or parallel algorithms [57, 58]. All of these strategies implicitly assume that input data used to compute the attack graph are accurate and trade the efficiency for an approximated solution. In addition, they do not assess and quantify the accuracy-scalability trade-off. Thus it is difficult to estimate the accuracy loss when they are used in self-protecting systems (e.g., [38]). The approaches proposed in Chapters 4 and 5 take an orthogonal perspective. Indeed, these proposals focus on the input data used by attack graph generation algorithms to improve their quality and reduce their size, still minimizing the accuracy loss.

As a consequence, the techniques proposed in this work can also be combined with existing solutions.

To move a step towards real-time attack graph generation, other solutions propose to compute attack paths based on alerts coming from detection systems. Among them, Nadeem et al. [71] propose a generation system that translates alert events to episode sequences that are then used to build a finite automaton representing the structure of the attack graph. Similarly, Ning et al. [73] and Moskal et al. [70] aggregate intrusion alerts based on their frequency and features to generate smaller-sized attack graphs and accelerate risk analysis. Also in this case, performance is traded off for accuracy (i.e., the attack graph is built from potentially inaccurate data extracted from alerts).

Different solutions leverage Artificial Intelligence (AI) to predict attack paths and avoid their complete enumeration. For example, Hu et al. [43] use data mining to correlate alerts and construct the attack graph accordingly, while Li et al. [60] leverage deep neural networks to generate attack graphs. These solutions focus on defining an approximation of attack graphs to avoid the complete generation and improve the scalability. On one side, they analyze the trade-off between the approximation degree of the generated graphs and the computational complexity. On the other hand, they do not analyze the impact on the risk estimation and scalability trade-off, which is crucial for self-protecting systems [56, 105, 108] to adjust response planning dynamically.

On the accuracy of attack graph input data

State-of-the-art solutions do not consider pre-processing the inputs of the attack graph computation algorithm. Instead, they collect data from different sources (e.g., different vulnerability scanners, sensors, IDSs) and aggregate them into a repository which is finally normalized to accomplish the data format necessary for the *Analysis* component [45, 50] of a self-protecting system following the MAPE-K architecture. To the best of the author's knowledge, the main contribution looking at the quality of vulnerability inventories (i.e., one of the attack graph inputs) is proposed by Bonomi et al. [25], who leverage the security expert's knowledge to detect and identify false positives coming from vulnerability scanners. They propose to require the human analyst to *validate* the presence of vulnerabilities by reducing as much as possible the number of interactions. The core idea behind such methodology is to model the dependencies between vulnerabilities and vulnerable platforms in the network (i.e., the hosts' configurations defined according to NIST's National Vulnerability Database [100]) and use such dependency model to drive the interaction with the user. Indeed this contribution is part of this thesis, and is featured in detail in Chapter 4, which will also propose several advancements to perfect the methodology and to consider and evaluate the accuracy of risk estimation in the computational pipeline proposed in the same chapter.

Chapter 3

Integrating Sources of Data to Support Automatic Correlation

Of all the aspects surrounding the Cyber Security Assessment process, the analysis of vulnerabilities is one of the most critical aspects. Moreover, vulnerability analysis not only has a strong impact on risk management but also fundamentally affects other processes such as incident management, as it provides information which is necessary for the process and for the decision activities that follow.

Understanding vulnerabilities affecting ICT systems is a long-standing activity, as established by the PhD thesis of Krsul [55] at US CERT and consolidated by the creation of the *Common Vulnerabilities and Exposures*[1] (CVE) in 1999, followed by the creation of National Institute for Standards and Technology's (NIST) *National Vulnerability Database*[2] in the US. More recently, efforts to develop standardized frameworks to support the evaluation of risk associated to each vulnerability have gained traction, leading to the adoption of many methodologies, such as the Common Vulnerability Scoring System (CVSS) [66]. These efforts have been further expanded to provide additional information related to the principles of vulnerabilities through the Common Weakness Enumeration (CWE) [31] dictionary, which classifies types of vulnerabilities.

With self-protecting systems being still in their infancy, vulnerability assessment is still an activity typically performed by cyber security analysts. Analysts manually check and analyze reports obtained from various sources, such as vulnerability scanners, to estimate each vulnerability's risk by evaluating the likelihood of each exploit and the magnitude of the consequences of the successful exploitation of each vulnerability.

During this task, knowledge extracted from publicly available repositories is leveraged (e.g., NIST NVD [75], the MITRE CWE [31], and the available ICS-CERT advisories, just to cite a few) and is correlated between multiple, possibly heterogeneous and large, sets of complementary and sometimes redundant information. This activity is extremely time-consuming for a human and not trivial for an automated system as the relevant information needs to be searched, downloaded and then analyzed by accessing multiple data sources which are often provided as plain text. In addition, the accuracy of the analysis is also impacted by the background and expertise of the analyst and by its familiarity with data sources that need to be used. Thus, there is a non-negligible effort that is required for a human or automated agent to understand the type and the semantics of data that is needed to complete

[1] https://cve.mitre.org/

[2] https://nvd.nist.gov/

the task.

Recently, effort is being spent in designing and developing automatic or semi-automatic correlation engines to support security operators as well as potential autonomic systems, in order to reduce the effort of finding the relevant information. As an example, there exist architectures for dynamic risk management designed during the Panoptesec project [8] and the PANACEA EU research project [93] where vulnerabilities are detected through vulnerability scanners and then analysed to estimate the related level of risk, by correlating information coming from the organization with data coming from NVD.

However, most of the existing solutions currently focus on how to correlate data observed from the monitored environment with data stored in a single external data source (typically the NVD). To the best of the author's knowledge, few works look at the integration of the knowledge across different external repositories and very few consider more than two information sources together and how to leverage the existing relationships between data.

Contributions

This chapter takes a step in the direction of integrating multiple data sources to improve the overall level of knowledge and to support automatic correlation of information. In particular, the chapter provides the following contributions:

1. A systematic review of several well-known publicly available repositories such as NVD [75], CWE [31], CAPEC [29] and ATT&CK [94, 23], aimed at understanding and formalizing their structural properties. During this process, schemas have been made to highlight each repository's capability and structure, as well as to highlight the presence of references between different repositories; The result is a graph-based conceptual model of every repository where nodes represent elements present in each repository and edges represent the existing relationships (between elements) defined inside each repository;

2. A unified data model, a graph-based representation of elements from each different repository, which is able to explicitly represent and handle the structure and hierarchy of elements of the same repository as well as the links existing between elements of different repositories; The unified data model is built on top of the existing repositories. It allows for correlation and analysis of all relevant concepts stored across all the included repositories. This unified data model will be then proven to support complex analysis tasks;

3. The formalization of three relevant and complex analysis tasks (usually performed during the vulnerability analysis in the risk management process), with the final aim to show how they can be efficiently supported by the unified data model. Algorithms that leverage the unified data model to easily find solutions will be proposed. In particular, the following analysis tasks will be considered:

    - Vulnerability classification to highlight the actual vulnerable system part and the system context needed to exploit a vulnerability,
    - Identification of possible mitigation actions given a vulnerability, and
    - Identification of the cost of applying a mitigation to a vulnerability.

To perform and validate the analysis, an exemplary unified data model has been instantiated starting from a dataset that includes (i) the whole CWE repository, (ii) the whole CAPEC repository and (iii) a subset of the NVD database considering vulnerabilities extracted from security advisories published by the ICS-CERT. For the first task an analysis strategy spanning a single repository has been proposed. This has been done to highlight the importance of having a deep knowledge of the data, the structure and the semantics of each data source. Concerning the second task a solution has been proposed, enabled by the integration of concepts in different repositories (i.e, NVD and CWE). Lastly, for the third task a methodology that elaborates the results and the principles behind tasks one and two has been proposed, in order to prove the ability of the unified data model to produce new, relevant information starting from previous computations across multiple repositories.

3.1 Repositories for Vulnerability Assessment

This section introduces the main publicly available repositories i.e., NVD [75], CWE [31], CAPEC [29, 23], and MITRE ATT&CK [94], that are typically used as primary sources to support security operators and autonomous systems during the vulnerability assessment and risk management tasks. Particular emphasis will be given in the description of:

- The information made available in the repository;
- The structure of each repository and, in particular, how data is organized and classified internally to the repository (i.e., *internal data relationships*);
- Existing relationships between data stored in different repositories (i.e., *external data relationships*).

The section aims to highlight the importance of the correct interpretation of data and their relationships. Indeed, this is fundamental to design algorithms and methodologies to better support security operators and automated systems in reducing the time required for the analysis and in improving the accuracy of the information provided. Section 3.3.1 will describe a naive solution that could have been designed without deep knowledge of the repositories structure and then an improvement will be shown by designing a solution that is fully aware of the semantics and of the structure of the data.

3.1.1 Common Vulnerabilities and Exposures (CVE) and the NIST National Vulnerability Database (NVD)

The *Common Vulnerabilities and Exposures* (CVE) [30] is a dictionary of publicly disclosed cybersecurity vulnerabilities and exposures managed by MITRE corporation. The CVE dictionary consists of a set of independent entries each one having: (i) an identification number; (ii) a description; and (iii) at least one public reference to external sources providing information about the issue. A CVE identifier follows a simple syntax: `CVE-`*`YEAR-5-digits-number`*, where *`YEAR`* identifies when the identifier has been *created* and *`5-digits-number`* is a sequential number to separate different entries of the same year. When someone suspects a vulnerability in a piece of software, he/she requests to MITRE such an identifier. MITRE assigns the identifier (*`YEAR`* and *`5-digits-number`*), but does not publish it immediately. The identifier is only published when the vulnerability is confirmed. In some cases, this process may take a significant amount of time, and there exist identifiers which are never confirmed and thus never published.

In addition to the dictionary maintained by MITRE, CVE entries are collected, organized and published in the *National Vulnerability Database* (NVD) [75], managed by NIST. NVD is built upon and fully synchronized with the CVE list so that any update to CVE appears immediately in NVD. Inside NVD, each entry is structured as follows:

- *CVE identifier*: it is the unique CVE identifier associated with the vulnerability.
- *CVE assigner*: it contains the indication of the entity/body that detected and notified the vulnerability.
- *CVE description*: it is typically a textual description of the vulnerability and of the effect of a possible exploit.
- *Relevant References*: it provides a set of references that allow enriching the knowledge about the vulnerability context. In particular, it is worth mentioning the following references:
  - Reference to software weaknesses categorized and classified in an external repository i.e., the *Common Weakness Enumeration* (CWE) repository (discussed in detail in Section 3.1.2).
  - Reference to known affected software configurations. This information is categorized by assigning to the CVE entry one (or more) *Common Platform Enumeration* (CPE) string(s) organized in configurations, used to identify information technology systems, software, and packages affected by the vulnerability.
  - A severity score and the associated CVSS vector.
- *CVE history*: each CVE has typically associated temporal information related to its history such as the date of the publication and the date of the last update.

Common Platform Enumeration (CPE) Strings

CPE strings identify, with a variable degree of accuracy, the vulnerable components of an ICT system, as well as components which are not vulnerable per se, but may be required in order express the vulnerability. According to the NIST documentation on CPE version 2.3 [26, 78], a well-formed CPE string is obtained by the concatenation of sub-strings, one for each attribute listed in Table 3.1 and separated by ":".

Among them, three attributes are mandatory i.e., *part*, *vendor* and *product*. When an optional attribute is not specified, it is replaced in the CPE string with $*$, which according to he documentation translates to *Any*. An example of a well-formed CPE string is *cpe:2.3:o:microsoft:windows_98se:-:*:*:*:*:*:*:** that refers to the operating system Microsoft Windows 98 and where the part attribute is equal to *o*, the vendor attribute is equal to *microsoft*, the product is *windows_98se* and all other (optional) attributes have not been specified.

The last column of Table 3.1 highlights the number of distinct sub-strings existing in the CPE dictionary (as of September 2023) used to label individual attributes (e.g., looking to the first row we have that only 3 values are used to identify the *part* attribute i.e., "a", "o" and "h").

Let us note that, a specific system component could potentially be identified by multiple CPE strings. This is due to the presence of optional attributes generating an overlap between entries in the CPE dictionary, with certain strings being included in others as particular cases e.g., we may have two possible CPE strings for the same product, one specifying only mandatory attributes (e.g., Microsoft Windows XP) and another one with a specific version (e.g., Microsoft Windows XP service pack 2).

Table 3.1. Attributes used in the definition of a CPE string

Attribute	Description	Distinct
part	domain of the element represented by the CPE (“a”=application, “h”=hardware, “o”=OS)	3
vendor	vendor of the product	30684
product	product name	139604
version	version of the product	107581
update	specific update of the product	10668
edition	specific edition of the product	1077
language	language used in UI of the product (encoded accordingly with [RFC5646[24]])	38
sw_edition	market (or class of end users) that the product is tailored to	440
target_sw	software computing environment used by the product	538
target_hw	instruction set architecture on which it runs	65
other	any other information that may pertain to the CPE	24

The CPE dictionary is built and curated manually by human analysts and, as such, may contain errors e.g., typos in product names or a bad split between product names and versions when both are tightly coupled in the common denomination proposed by the vendor. Thus, on one hand, it is a rich source of information for security analysts but on the other hand, it raises a few issues concerning the quality of stored data.

NVD lists, for every stored vulnerability, its linked CPEs and supports queries to filter CVEs based on the vendor, the severity or the publication date. In addition, NVD labels CPE strings with additional information that specifies if the considered product is itself vulnerable or if it represents an environmental condition necessary for the exploit.

Inside NVD, CPE strings associated with a specific vulnerability are organized in a set of alternative configurations (i.e., logically related through a OR operator). Each configuration then is represented by a AND – OR tree called *configuration tree*. As per NIST IR 7698 [100], a configuration tree defines which combinations of software, hardware and operating system are necessary for the vulnerability to exist. Figure 3.1 shows a graphical representation of possible topologies of configurations trees.

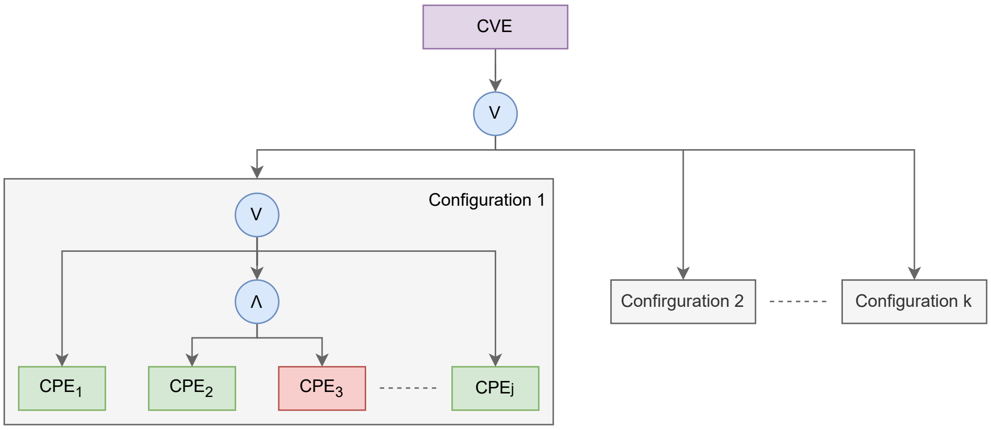


Figure 3.1. Structure of CPE configurations stored in NVD. Red and green denote vulnerable / non-vulnerable status

Let us note that, NVD (as it is now) is just a repository and it has not been designed to embed a correlation engine able to perform a deep analysis among data (e.g., it is not currently able to analyze different CPE configurations or correlate CVE and CWE to suggest mitigation actions). This task is indeed delegated to (i) the vulnerability analyst that manually browses the available data and correlates him/her self such information or (ii) external tools that download the data and implement all the correlation logic internally.

Common Vulnerability Scoring System (CVSS)
NVD complements the description of every vulnerability by reporting one or more severity scores computed according to the *Common Vulnerability Scoring System* (CVSS) [5] and the corresponding CVSS strings. CVSS provides a way to consider the main characteristics of a vulnerability and to produce a numerical score reflecting its severity. The numerical score is then translated into a qualitative representation (such as low, medium, high, and critical) to help organizations to properly assess and prioritize their vulnerability management processes. The severity score reported in NVD corresponds (in almost all entries) to the vulnerability *base score* i.e., a numerical value between 0 and 10 that is computed by scoring all the base metrics defined in the CVSS specification. CVSS also allows to compute an *environmental score* and a *temporal score* that can be used to (i) contextualize and reweigh the severity of the vulnerability in the specific environment and (ii) keep into account the time elapsed from when the vulnerability has been detected and when it is analysed.

CVSS evolved across three different versions, namely CVSS version 2.0, 3.0 and 3.1, defined and managed by FIRST [6], that coexist within the NVD ecosystem. The main difference between these versions is in the set of metrics and corresponding values that are considered in the computation of the score. Independently from the considered version, only the *base metrics* are always evaluated and stored within each NVD entry. This is necessary as by definition the estimation of *environmental metrics* and *temporal metrics* are highly dependant on the ICT environment that the vulnerability resides in.

Let us note that, while at the time of writing almost all NVD entries have associated a CVSS version 2.0 base score, only newer entries report a CVSS 3.* base score. Currently, NIST is not planning to update legacy CVSS 2.0 attributes into the newer CVSS 3.0 and 3.1 for old entries. For this reason, CVSS 2.0 is still kept as an alternative in newer entries as a measure to ensure both retro-compatibility and uniformity across the NVD repository.

For the sake of completeness, CVSS *base metrics* are reported according to specification 2.0 [3], as well as 3.0 [4] in tables 3.2 and 3.3 respectively. The CVSS 3.1 specification [5] introduces minor modifications in the definitions and formulas used to compute the numerical scores in order to clarify and improve the existing standard.

Figure 3.2 shows a possible UML conceptual representation for data stored in the NVD database and its related information. As it can be seen, the central concept of a NVD entry is represented by the CVE vulnerability which is then related to CPE configurations and CVSS scores (that could be expressed according to specification v2, v3 or both).

3.1.2 Common Weakness Enumeration (CWE)

The *Common Weakness Enumeration* (CWE) repository [31] is a categorization of software weaknesses and vulnerabilities. It is managed by MITRE and it is

Table 3.2. CVSS 2.0 base metrics

Metric	Description	Values
Access Vector (AV)	This metric reflects how the vulnerability is exploited	Local (L) Adjacent Network (A) Network (N)
Access Complexity (AC)	This metric measures the complexity of the attack required to exploit the vulnerability once an attacker has gained access to the target system	Low (L) Medium (M) High (H)
Authentication (Au)	This metric measures the number of times an attacker must authenticate to a target in order to exploit a vulnerability	None (N) Single (S) Multiple (M)
Confidentiality Impact (C)	This metric measures the impact on confidentiality of a successfully exploited vulnerability	None (N) Partial (P) Complete (C)
Integrity Impact (I)	This metric measures the impact on integrity of a successfully exploited vulnerability	None (N) Partial (P) Complete (C)
Availability Impact (A)	This metric measures the impact on availability of a successfully exploited vulnerability	None (N) Partial (P) Complete (C)

sustained by a community project. Its main goal is to report details about flaws in software to support the identification, mitigation, and prevention of those flaws. To this aim, every CWE entry has associated a numerical identifier and an explanatory name. A CWE entry may contain references to other CWE entries with the result of having information organized in a graph-based structure that can be queried and analyzed i.e., CWE entries can be seen as vertexes of a graph and relationships between entries can be seen as edges of the CWE graph. Such relationships are made explicit in the repository from the fields *Relationships* and *MemberOf Relationships* defined for each CWE entry. Given this graph-oriented view of the CWE repository, in the following, the term “node” may be used to refer to a CWE entry. Additionally, CWE entries report historical information that allows the user to analyze how the issue evolved over time.

CWE entries are organized in a taxonomy and every entry can be one of the following three different types: (i) *view*, (ii) *category* and (iii) *weakness*.

A *weakness* entry provides details about the associated issue, including a textual description, common consequences of the weakness exploit, examples and possible mitigations. CWE weakness nodes may also provide pointers to vulnerabilities (i.e., by referencing CVE identifiers). This class of entries includes the majority of CWE nodes. Weakness nodes can be also connected to highlight specific semantic relationships. As an example, a *parent-of* (or *child-of*) relationship between cwe_1 and cwe_2 models the fact that cwe_1 represents a more (resp. less) general problem than cwe_2. Let us note that while in theory weakness nodes may have associated multiple useful information, however, many fields are optional and thus may not be populated for each entry.

Category nodes represent a way to group and structure different specialized weaknesses under a common denominator. A category node represents a higher-level concept aggregating together specific weaknesses. As an example, if we consider the category *CWE-1218 Memory Buffer Errors*[3] representing weaknesses related to memory buffer management errors within a software system, we can see that it

[3] https://cwe.mitre.org/data/definitions/1218.html

Table 3.3. CVSS 3.0 and 3.1 base metrics

Metric	Description	Values
Attack Vector (AV)	This metric reflects the context by which vulnerability exploitation is possible	Network (N) Adjacent (A) Local (L) Physical (P)
Attack Complexity (AC)	This metric describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability	Low (L) High (H)
Privileges Required (PR)	This metric describes the level of pivileges an attacker must possess before successfully exploiting the vulnerability	None (N) Low (L) High (H)
User Interaction (UI)	This metric captures the requirement for a human user, other than the attacker, to partecipate in the successful compromise of the vulnerable component	None (N) Required (R)
Scope (S)	The Scope metric captures whether a vulnerability in one vulnerable component impacts resources in components beyond its security scope	Unchanged (U) Changed (C)
Confidentiality Imapct (C)	This metric measures the impact to the confidentiality of the information resources managed by a software component due to a successfully exploited vulnerability	None (N) Low (L) High (H)
Integrity Impact (I)	This metric measures the impact on integrity of a successfully exploited vulnerability	None (N) Low (L) High (H)
Availability Impact (A)	This metric measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability	None (N) Low (L) High (H)

abstracts 9 detailed weaknesses (i.e., CWE-120, CWE-123, CWE-124, CWE-125, CWE-131, CWE-786, CWE-787, CWE-788, CWE-805), each describing a specific type of error leading to problems with memory buffers.

Categories can also be linked between each other to model concepts with different levels of abstractions or to map already classified issues from one class to another (e.g., category node CWE-723 Broken Access Control is linked to the category node CWE-275 Permission Issues through a *parent-of* (in the graph represented as *Member_Of*) relationship meaning that CWE-723 is a more general issue than CWE-275).

A *view* node identifies a perspective abstracting from the specific weakness and it is usually associated to the type of aggregation it represents (e.g., *Architectural Concepts* or *Research Concepts*) and its target audience (e.g., *Academic Researchers* or *Assessment Vendors*). Generally, views are used to navigate and filter the CWE repository and they result in a tree-based structure (called *CWE tree*) where the view is the root and categories and weaknesses are its children. It is worth noting that CWE categories and weaknesses may belong to multiple views and thus, they may be part of more than one CWE tree.

Figure 3.3 shows the UML conceptual representation of the information stored in the CWE repository.

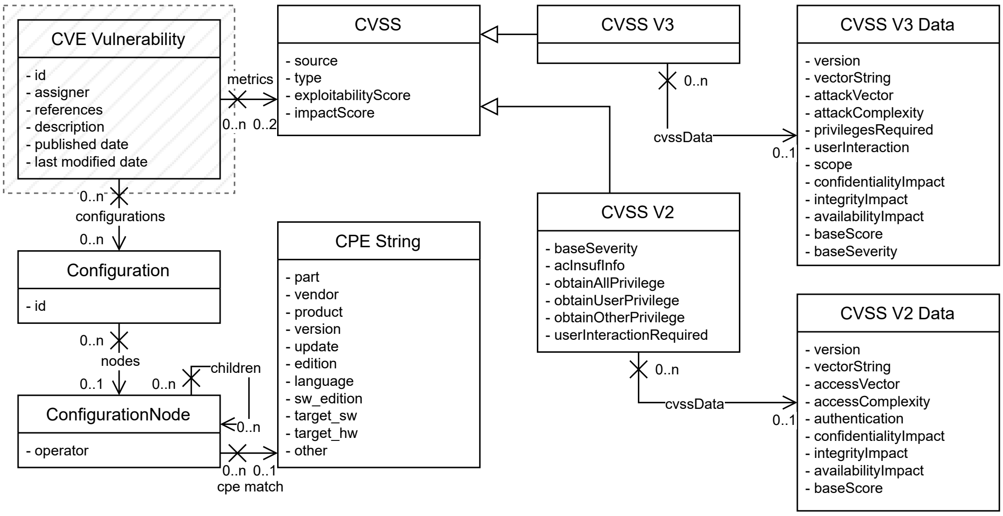


Figure 3.2. UML conceptual representation of the vulnerability data model.

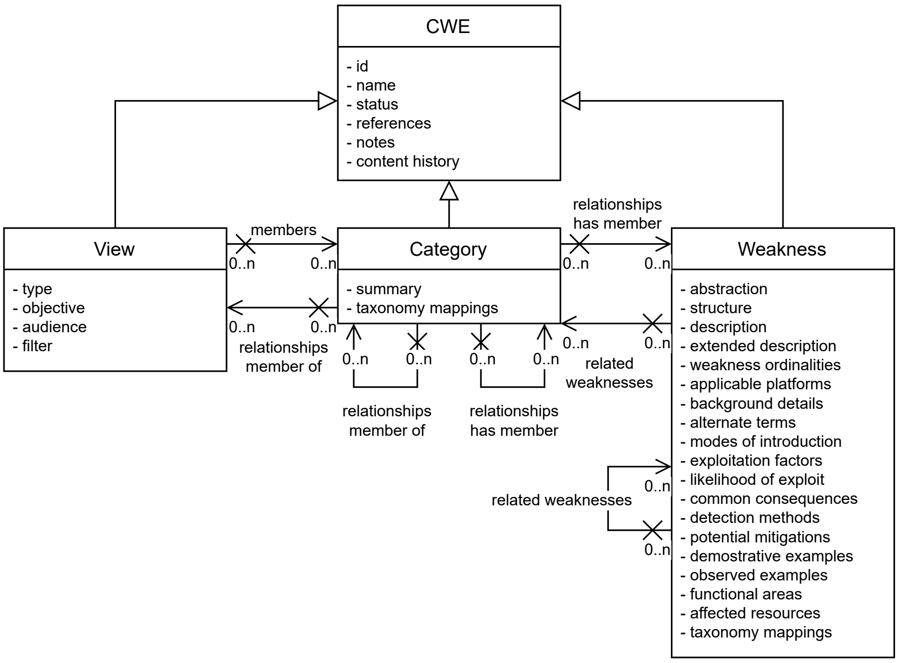
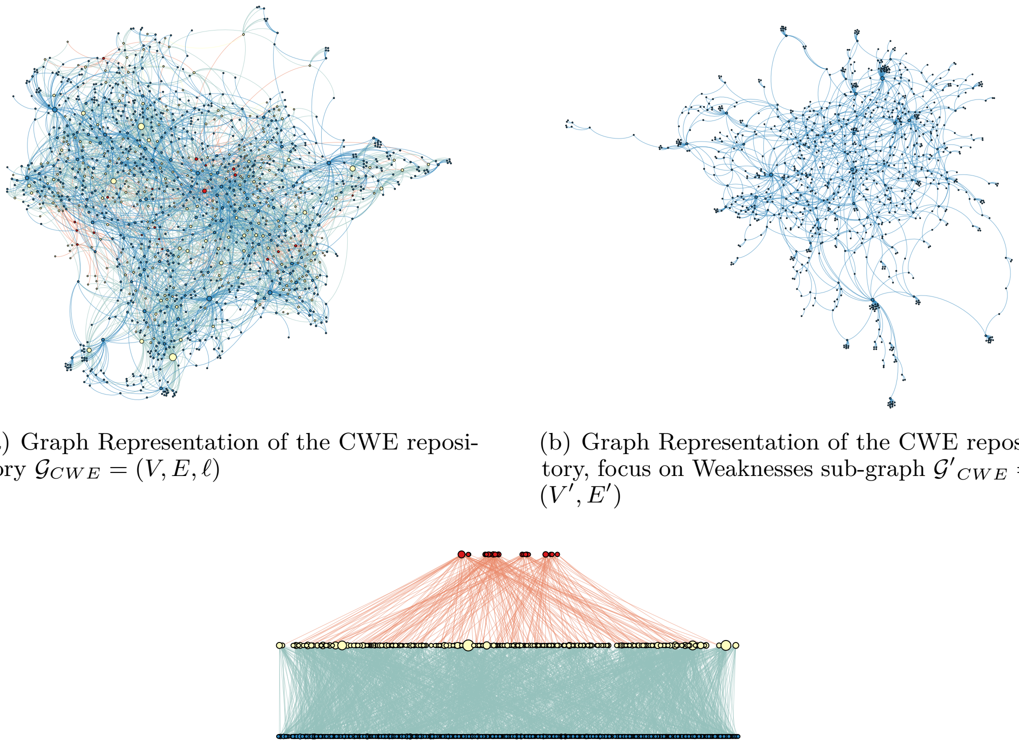


Figure 3.3. UML conceptual representation of the CWE data model



(a) Graph Representation of the CWE repository $\mathcal{G}_{CWE} = (V, E, \ell)$

(b) Graph Representation of the CWE repository, focus on Weaknesses sub-graph $\mathcal{G}'_{CWE} = (V', E')$

(c) Graph Representation of the CWE repository, focus on the sub-graph including only links between Views, Categories and Weaknesses $\mathcal{G}''_{CWE} = (V'', E'')$

Figure 3.4. Graph Representation of the CWE repository $\mathcal{G}_{CWE} = (V, E, \ell)$ including all CWE published up to September 2023.

Considerations on the CWE repository

As anticipated, information stored in the CWE repository and its relationships can be represented as a graph of concepts where the set of vertexes is the set of CWE nodes and the set of edges is represented by any existing relationship between nodes (i.e., weakness-weakness such as *parent_of* or *child_of*, category-weakness, category-category and view-category). More formally, if we denote $\mathcal{W}$ the set of weakness nodes, $\mathcal{C}$ the set of category nodes and $\mathcal{V}$ the set of view nodes, we can define the CWE graph as $\mathcal{G}_{CWE} = (V, E, \ell)$ where

- V is the set of vertexes defined as $V = \mathcal{W} \cup \mathcal{C} \cup \mathcal{V}$;
- E is the set of all existing relationships between CWE nodes (i.e., given two nodes $v_i, v_j \in V$ there exists an edge $e_{i,j} \in E$ if and only if v_i is related to v_j by some relation in CWE;
- ℓ is a label function that assigns the type of relationship to any existing edge. Existing labels (i.e., relationships) at the time of the writing are: child_of, parent_of, starts_with, can_precede, can_follow, required_by, requires, can_also_be, peer_of.

Figure 3.4 shows the graphical representation of $\mathcal{G}_{CWE}$ representing the whole CWE repository and interesting sub-graphs. In particular: (i) Figure 3.4(a) shows the graph representing the whole set of nodes stored in the repositories and their relationships; (ii) Figure 3.4(b) shows the CWE sub-graph obtained when considering only weaknesses nodes; and (iii) Figure 3.4(c) highlights the CWE sub-graph obtained considering only relationships between nodes of different types.

Contrary to the expectation, the "hierarchical organization" of data (i.e., the partition of nodes in views, categories and weaknesses) does not induce a pure tri-partite graph, but it rather constructs a complex structure where the number of relationships between nodes in the same class is not negligible. A second interesting point is the structure of the CWE graph obtained when considering only relationships between nodes of different types (i.e., when we consider only edges labelled as category-weakness and view-category). This sub-graph is indeed tri-partite (by definition) and highlights the presence of a few categories having a high degree.

Table 3.4 summarizes the main relevant metrics of the CWE graph $\mathcal{G}_{CWE}$ and for few interesting sub-graphs defined as: (i) $\mathcal{G}_{view}$ is the sub-graph obtained when considering only nodes of type view i.e., $\mathcal{G}_{view} = (\mathcal{V}, E')$ where $E' \subset E$ is the set of edges $e_{i,j} = (v_i, w_j)$ such that $v_i, v_j \in \mathcal{V}$, (ii) $\mathcal{G}_{ctg}$ is the sub-graph obtained when considering only nodes of type category i.e., $\mathcal{G}_{ctg} = (\mathcal{C}, E')$ where $E' \subset E$ is the set of edges $e_{i,j} = (v_i, w_j)$ such that $v_i, v_j \in \mathcal{C}$ and (iii) $\mathcal{G}_{weak}$ is the sub-graph obtained when considering only nodes of type weakness i.e., $\mathcal{G}_{weak} = (\mathcal{W}, E')$ where $E' \subset E$ is the set of edges $e_{i,j} = (v_i, w_j)$ such that $v_i, v_j \in \mathcal{W}$.

Table 3.4. Summary of $\mathcal{G}_{CWE}$ Graph Properties and of its sub-graphs (snapshot of the CWE repository as of September 2023)

	$\mathcal{G}_{\mathbf{CWE}}$	$\mathcal{G}_{\mathbf{view}}$	$\mathcal{G}_{\mathbf{ctg}}$	$\mathcal{G}_{\mathbf{weak}}$
# Vertexes	1420	53	409	958
# Edges	6231	0	105	1571
# Isolated Vertexes	70	53	275	25
Average Degree	9.29	0.00	1.57	3.37
Max Degree	157	0	9	53
Average In-degree	4.78	0.00	1.08	4.55
Max In-degree	78	0	3	51
Average Out-degree	4.74	0.00	2.44	1.70
Max Out-degree	157	0	9	8

It can be observed that in the whole CWE graph $\mathcal{G}_{CWE}$ there are only 70 isolated nodes (i.e., 5% of the vertexes) i.e., nodes not connected to other nodes. These nodes represent, in most of the cases, deprecated concepts still stored in the repository. When considering the induced sub-graphs, it can be seen that (i) all nodes in the view sub-graph are isolated meaning that views are independent each other, (ii) in the category sub-graph 67% of nodes are isolated and the remaining ones are weakly connected meaning that most of the categories are independent each other and just a few of them models similar concepts and (iii) in the weakness sub-graph only 3% of nodes are isolated while the others are generally well connected among them. These properties let us think that CWE can be used as basic knowledge for building complex correlation algorithms exploiting the graph relationship to infer potentially hidden relevant information (an example will be discussed in the next section about how to use the CWE tree structure to find possible mitigation actions when they are not directly available in the considered CWE node).

3.1.3 Common Attack Pattern Enumeration and Classification (CAPEC)

Common Attack Pattern Enumeration and Classification (CAPEC) [29] is a publicly available catalogue of common attack patterns reporting on how adversaries may exploit weaknesses or flaws in applications. Similarly to CWE, it is managed by MITRE and is maintained by a community project.

The structure of the CAPEC repository is very similar to CWE's with entries organized in a graph-based structure and classified in *views*, *categories* and *attack patterns*.

An *Attack Pattern* entry provides a detailed description of the attack pattern, specifying common attributes characterizing an attack and approaches employed by adversaries to exploit known weaknesses.

Attack patterns also define the challenges that an adversary may face and are generated from in-depth analysis of specific real-world exploit examples. Each attack pattern captures knowledge about how specific parts of an attack are designed and executed and gives guidance on ways to mitigate the attack's effectiveness.

View and *category* nodes are built following the same rationale used in CWE: *view* nodes identify a perspective to support the traversal of the CAPEC repository, while *category* nodes model high-level concepts that serves as a point of aggregation for Attack Patterns.

Let us note that CAPEC entries may be referenced by CWE entries (i.e., starting from a specific CWE entry it is possible to understand the related attack patterns). Unfortunately, the vice-versa is not true and CAPEC does not provide any reference back to CWE or to CVE. Thus, it is not easy to understand, for a given attack pattern, which are the vulnerabilities or weaknesses that are necessary to proceed with the exploit.

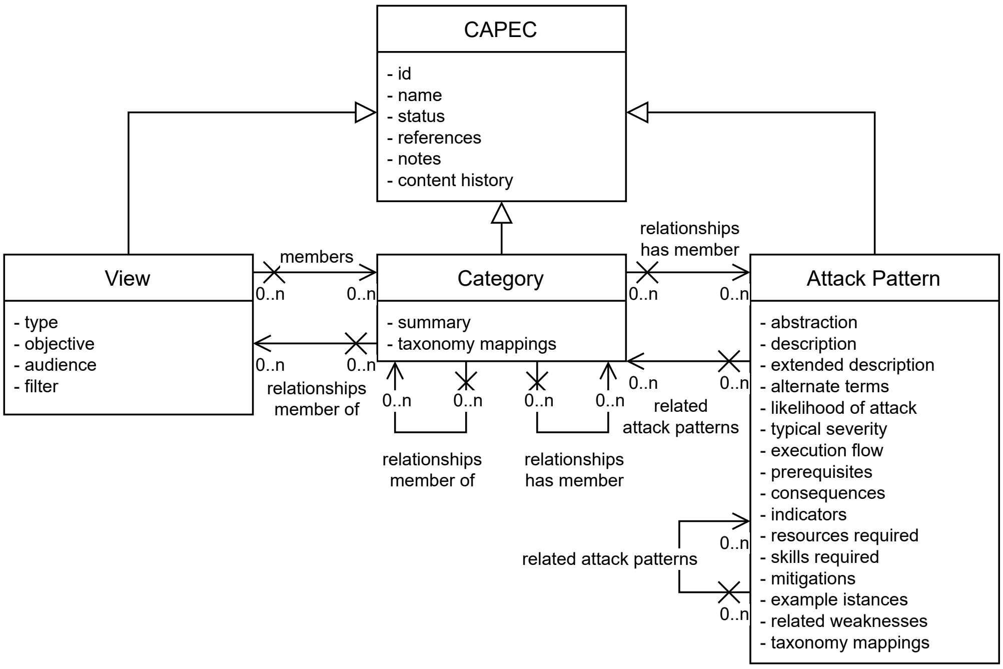


Figure 3.5. UML conceptual representation of the CAPEC data model

Figure 3.5 shows the UML conceptual representation of data stored in the

CAPEC repository.

Let us note that while CAPEC is widely used by analysts for manual analysis, using it as data source for automated analysis tools is far from being trivial as most of the information stored is represented in plain text.

Considerations on the CAPEC repository

As per the case of CWE, the CAPEC repository can be interpreted as a graph where vertexes are represented by CAPEC nodes (i.e., views, categories and attack patterns) and edges are represented by relationships among them. More formally, if we denote $\mathcal{A}$ the set of attack pattern nodes, $\mathcal{C}$ the set of category nodes and $\mathcal{V}$ the set of view nodes, we can define the CAPEC graph as $\mathcal{G}_{CAPEC} = (V, E, \ell)$ where

- V is the set of vertexes defined as $V = \mathcal{A} \cup \mathcal{C} \cup \mathcal{V}$;
- E is the set of existing relationships between CAPEC nodes (i.e., given two nodes $v_i, v_j \in V$ there exists an edge $e_{i,j} \in E$ if and only if v_i is related to v_j by some relation in CAPEC;
- ℓ is a label function that assigns the type of relationship to any existing edge.

Figure 3.6 shows the graphical representation of $\mathcal{G}_{CAPEC}$ representing the whole CAPEC repository and interesting sub-graphs. In particular: (i) Figure 3.6(a) shows the graph representing the whole set of nodes stored in the repositories and their relationships; (ii) Figure 3.6(b) shows the CAPEC sub-graph obtained when considering only weaknesses nodes; and (iii) Figure 3.6(c) highlights the CAPEC sub-graph obtained considering only relationships between nodes of different types.

Looking at the structure of $\mathcal{G}_{CAPEC}$ it can be observed that relationships between nodes in the same class play a significant role in the CAPEC graph too. In addition, it can also be seen that most of the structural properties of the CWE graph also apply to $\mathcal{G}_{CAPEC}$. However, $\mathcal{G}_{CAPEC}$ is much smaller than $\mathcal{G}_{CWE}$, making its analysis and interpretation easier.

Table 3.5 summarizes the main metrics for the CAPEC graph $\mathcal{G}_{CAPEC}$ and for few interesting sub-graphs defined as: (i) $\mathcal{G}_{view}$ is the sub-graph obtained when considering only nodes of type view i.e., $\mathcal{G}_{view} = (\mathcal{V}, E')$ where $E' \subset E$ is the set of edges $e_{i,j} = (v_i, w_j)$ such that $v_i, v_j \in \mathcal{V}$, (ii) $\mathcal{G}_{ctg}$ is the sub-graph obtained when considering only nodes of type category i.e., $\mathcal{G}_{ctg} = (\mathcal{C}, E')$ where $E' \subset E$ is the set of edges $e_{i,j} = (v_i, w_j)$ such that $v_i, v_j \in \mathcal{C}$ and (iii) $\mathcal{G}_{atk\_pat}$ is the sub-graph obtained when considering only nodes of type attack pattern i.e., $\mathcal{G}_{atk\_pat} = (\mathcal{A}, E')$ where $E' \subset E$ is the set of edges $e_{i,j} = (v_i, w_j)$ such that $v_i, v_j \in \mathcal{A}$.

Table 3.5. Summary of $\mathcal{G}_{CAPEC}$ Graph Properties (snapshot of the CWE repository as of September 2023)

	$\mathcal{G}_{\mathbf{CAPEC}}$	$\mathcal{G}_{\mathbf{view}}$	$\mathcal{G}_{\mathbf{ctg}}$	$\mathcal{G}_{\mathbf{atk\_pat}}$
# Vertexes	706	13	78	615
# Edges	984	0	0	727
# Isolated Vertexes	120	13	78	61
Average Degree	3.36	0.00	0.00	2.62
Max Degree	53	0	0	24
Average In-degree	4.05	0.00	0.00	3.62
Max In-degree	29	0	0	24
Average Out-degree	1.85	0.00	0.00	1.44
Max Out-degree	52	0	0	8

Looking at the number of isolated nodes, we can observe that, differently from the CWE graph, categories are independent of each other (i.e., there are no connections between two categories). Thus, CAPEC seems to be really closer to a taxonomy than CWE (see Figure 3.6). Finally, we can also observe the range of nodes' degree when considering attack patterns and their relationships is quite high (it ranges between 0 and 24 with an average of 2.62). This suggests that there are few central attack patterns in the graph while others are generally independent (or loosely related) concepts.

3.1.4 Adversarial Tactics, Techniques, and Common Knowledge (ATT&CK)

The MITRE Adversarial Tactics, Techniques, and Common Knowledge (ATT&CK) repository [94] is a globally-accessible knowledge base of adversary tactics and techniques based on real-world observations. ATT&CK provides information about cyber adversary behaviour with particular emphasis on the attack lifecycle and the platforms that specific attackers are known to target. ATT&CK focuses on how external adversaries compromise and operate within computer networks and it consists of the following core concepts:

- *Tactic*: denotes the short-term, tactical adversary goals during an attack and basically represents *what* an attacker wants to obtain;
- *Technique*: describes how an adversary can achieve its tactical goals and thus it represents *how* objectives can be achieved;
- *Sub-technique*: specializes the concept of technique by describing more specific means by which adversaries achieve tactical goals;

The relationship between tactics and techniques/sub-techniques can be visualized in the ATT&CK Matrix which is currently what analysts use to prevent and detect specific attack categories.

Given a technique, it is typically associated to one or multiple *groups* i.e., sets of related intrusion activity that are tracked by a common name in the security community. Groups identification and their association with techniques are done manually by analysts that do their best to ensure the accuracy, completeness and consistency of the relationships tracked in ATT&CK. Groups are also mapped to reported Software used, and the Technique implemented by that Software is tracked separately on each Software page.

A technique (or sub-technique) has also typically associated a *mitigation* and in the ATT&CK syntax a mitigation represents a security concept and classes of technologies that can be used to prevent a technique or sub-technique from being successfully executed.

It is interesting to see that this repository is also organized with a graph-based logic and that the $\mathcal{G}_{ATT\&CK}$ graph can be built similarly to what has been done for CWE and CAPEC by defining a node type for each relevant concept and an edge for any type of relationship existing between concepts.

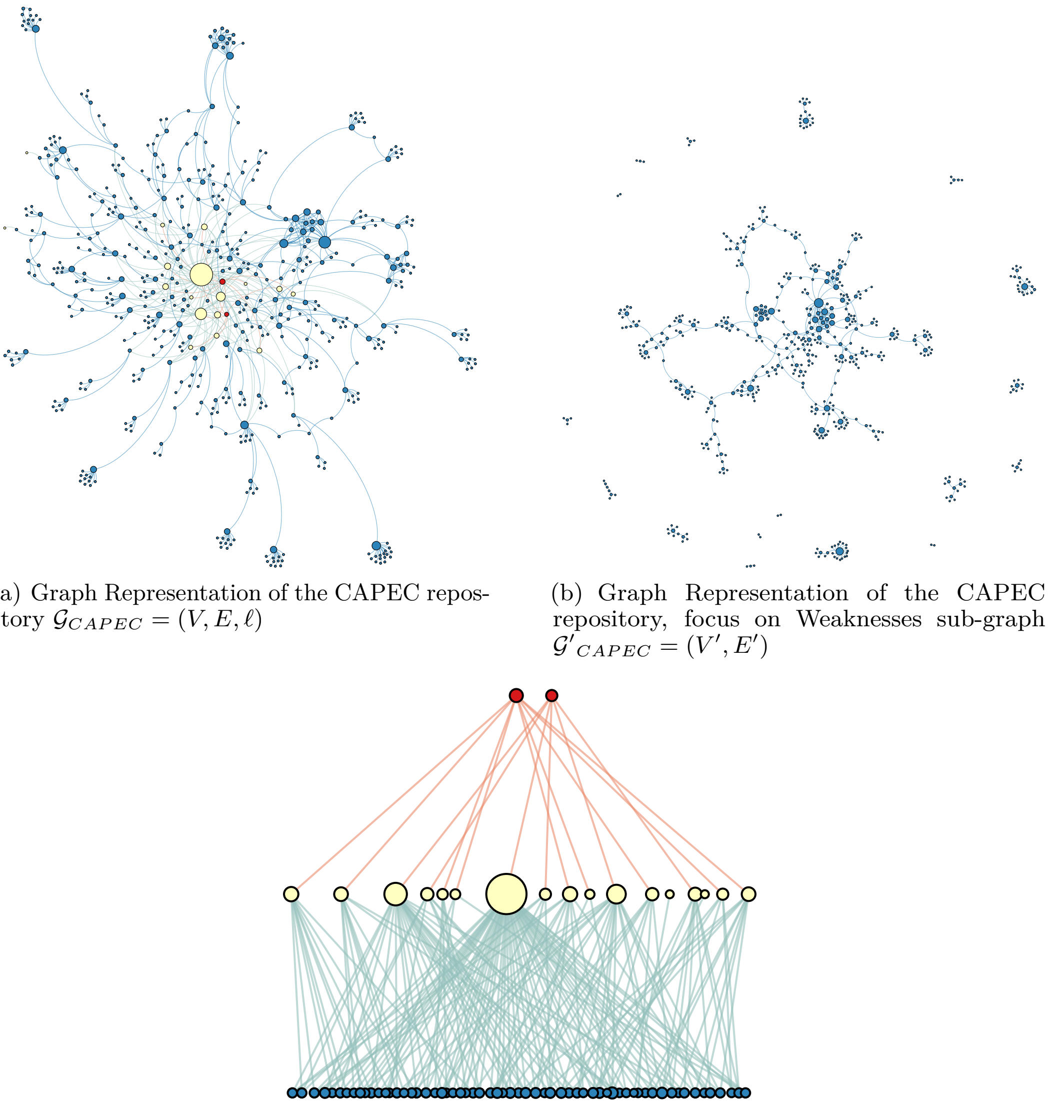
Figure 3.7 shows a possible conceptual data model for the ATT&CK repository.

Access to data stored in the ATT&CK repository can be done in different ways ranging from using the tools available on the website (e.g., the ATT&CK Navigator) to the download of the whole repository as an excel file or using the Structured Threat Information Expression (STIX) language and serialization format used to exchange cyber threat intelligence (CTI). At the time of writing, the ATT&CK dataset is available in STIX version 2.0 and 2.1.

It is important to note that STIX format representation is richer than every other available format, even when compared to the information available on the official web pages.

Table 3.6 summarizes the main metrics for the ATT&CK graph[4] and some of its relevant sub-graphs.

[4] The attributes in the graph derived from the STIX 2.1 representation of ATT&CK



(a) Graph Representation of the CAPEC repository $\mathcal{G}_{CAPEC} = (V, E, \ell)$

(b) Graph Representation of the CAPEC repository, focus on Weaknesses sub-graph $\mathcal{G}'_{CAPEC} = (V', E')$

(c) Graph Representation of the CAPEC repository, focus on the sub-graph including only links between Views, Categories and Weaknesses $\mathcal{G}''_{CAPEC} = (V'', E'')$

Figure 3.6. Graph Representation of the CAPEC repository $\mathcal{G}_{CAPEC} = (V, E, \ell)$ including all CAPEC published up to September 2023.

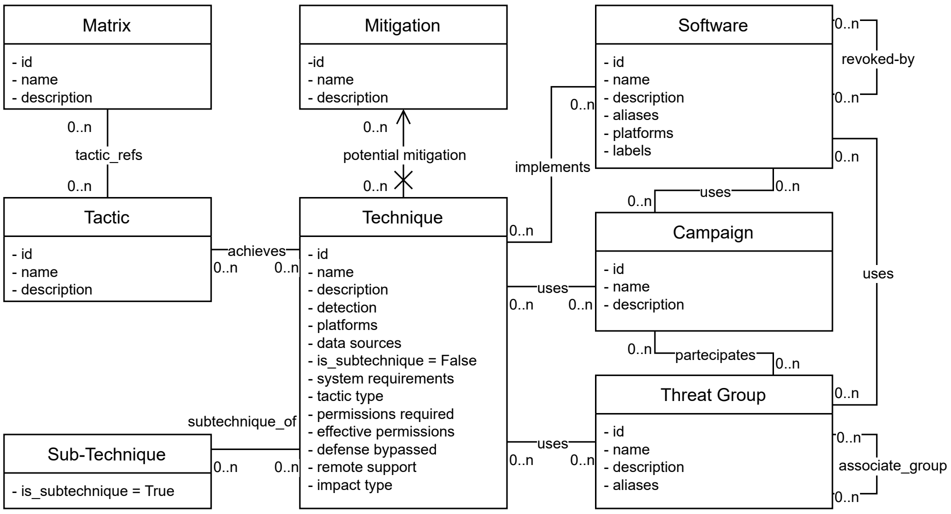


Figure 3.7. ATT&CK Graph Structure

Table 3.6. Summary of ATT&CK Graph Properties (snapshot of the CWE repository as of September 2023)

	Full	**Matrix**	**Tactics**	**Techniques**	**Threat Groups**	**Mitigations**	**Software**	**Caimpaigns**
# Vertexes	2280	4	55	1192	152	107	748	22
# Edges	17625	0	0	673	5	0	1	0
# Isolated Vertexes	249	4	55	405	142	107	746	22
Average Degree	17.36	0.00	0.00	1.71	1.00	0.00	1.00	0.00
Max Degree	421	0	0	16	1	0	1	0
Average In-degree	10.52	0.00	0.00	1.02	1.00	0.00	1.00	0.00
Max In-degree	266	0	0	2	1	0	1	0
Average Out-degree	15.43	0.00	0.00	2.21	1.00	0.00	1.00	0.00
Max Out-degree	421	0	0	16	1	0	1	0

3.2 Integrating the Repositories

Section 3.1 provided a detailed overview of the characteristics of NVD, CWE, CAPEC and ATT&CK repositories and analysed their structural properties independently.

Let us remark that such repositories are managed independently from each other and they have been initially thought to target different audiences:

- CVE and NVD are dedicated to providing detailed information about vulnerabilities and their features from the system perspective and they serve as primary source of information for vulnerability evaluation in risk assessment.
- CWE has the aim of providing a common language and measuring stick for security tools; it serves as a baseline for weakness identification, mitigation and prevention efforts and thus it is more oriented to software developers and testers.
- CAPEC provides a comprehensive dictionary of known attack patterns taken by adversaries to exploit known weaknesses. It is mainly used by analysts, developers, testers but also from educators to enhance understanding, awareness and defenses.
- ATT&CK is used as a foundation for the development of specific threat models and to understand how an attacker may launch its offensive to the system. Thus, it is an important source of information for both blue team and red team exercises but also for risk analysts to correctly evaluate the likelihood of specific threat sources.

However, despite their independent target, structure and management, such repositories are not completely unrelated. Contrarily, they address and report about strictly related concepts from different point of views that, when considered together, enlarge the perspective on the security situation providing a deeper and more detailed view about it. Currently, every repository provides references to entries of the other repositories but the correlation and the analysis is still usually carried out by a human that manually browses and queries the data requiring typically minutes (or hours in case of complex correlations). In particular:

- CVE entries have, among their references, pointers to CWE identifiers to support the analyst in understanding the type of issue that may lead to the existence of the vulnerability;
- CWE entries have, among their references, pointers to CAPEC entries and (in a negligible number of cases) pointers to CVEs. This is done to point out the relationships between a specific software issue or flaw and one (or more) attack patterns that can be triggered from it;
- CAPEC entries have, among their references, pointers to MITRE ATT&CK entries and (in a negligible number of cases) pointers to CVEs. This pointer helps the analysis in linking an abstract attack pattern with concrete attack techniques and tools.

Let us note that the presence of such explicit links paves the way to a natural integration in to a single knowledge base that can be used to simplify the analysis of a given situation from multiple points of view.

In addition, when entering into the detail of the data model behind every repository, it can be noticed that there are “attributes” (e.g., mitigation) that are specified, with different flavours, in multiple repositories and that represent a relevant concept by themselves from the security situation analysis point of view. Similarly, there exists few attributes that induce an implicit relation between concepts stored in different repositories. As an example, let us consider the *affected resources* attribute associated to a weakness entry in CWE. Such attribute has the aim of specifying the set of potential system resources affected by the considered weakness. Such resources are described in CWE in plain text but conceptually they correspond to the notion of platform that can be formalized through the concept of CPE.

In the following, a data model will be proposed in order to support the integration of the considered repositories in to a unique *knowledge graph* [34] and an architecture will be proposed to instantiate and keep the knowledge graph up to date. Since both in academia and in industrial applications no consensus has been reached on the definition of the term *knowledge graph*, throughout the scope of this work the definition proposed by Ehrlinger and Wöß [34] will be used. According to the definition proposed by Ehrlinger and Wöß [34], “a knowledge graph acquires and integrates information into an ontology and applies a reasoner to derive new knowledge”. This definition asserts that the knowledge graph retains all the properties and characteristics of an ontology, and expands upon it by: (i) integrating more than one sources of information, and (ii) supporting the application of a reasoning engine in order to derive new information. This definition is also adopted by IBM[5], for which *knowledge graphs* “are typically made up of datasets from various sources, which frequently differ in structure”, and “the data integration efforts around knowledge graphs can also support the creation of new knowledge, establishing connections between data points that may not have been realized before”. According to both definitions, the structure of a *knowledge graph* is made up of three components called *nodes* (representing the objects or entities we are interested in), *edges* (representing the relationships existing between pairs of nodes) and *labels* (assigning a specific semantic to any relationship).

When modeling the proposed *knowledge graph*, the following additional requirements have been considered:

- The *knowledge graph* should be easily and quickly updated reflecting updates in the input repositories. To this aim, it has been decided to preserve as much as possible the internal structure of every repository (i.e., it has been decided to preserve all the originally existing semantic relationships) as well as to build edges between concepts that are already linked using cross-references. While this results in a trivial construction of the core part of the proposed knowledge graph, it is important to ensure that the resulting knowledge graph can be instantiated and populated efficiently and effectively;

- “Relevant” concepts for the analysis of the situation (e.g., CPE configurations, mitigation actions) modeled in the input repositories as attributes must be translated into entities (i.e., nodes) of the knowledge graph. This has been decided in order to highlight these concepts

- The “hidden” relationships between concepts in different repositories have been made explicit. Analyzing the attributes of each relevant entity, it has been noticed that they could be semantically associated with existing concepts e.g., a weakness node has associated the *applicable platform* attribute (expressed as

[5] https://www.ibm.com/topics/knowledge-graph

plain text) and it has been noticed that this attribute could be translated in a relationship with a CPE node.

The result is a single graph that enables the correlation of heterogeneous information and the identification, as an example, of the possible attack techniques available to an attacker to intrude into a network, starting from the detected vulnerabilities on network hosts.

Figure 3.8 provides an overview of the conceptual data model behind the integrated knowledge graph i.e., the type of nodes, links and label that can be find in the proposed knowledge graph. Coloured shades[6] are used to identify concepts and related attributes (i.e., nodes of the knowledge graph) that are directly modelled and represented inside the original repositories while the white background has been left to identify concepts and attributes not directly available in the repositories. Relationships between concepts in Figure 3.8 represent relationships (i.e., edges and corresponding labels) in the knowledge graph.

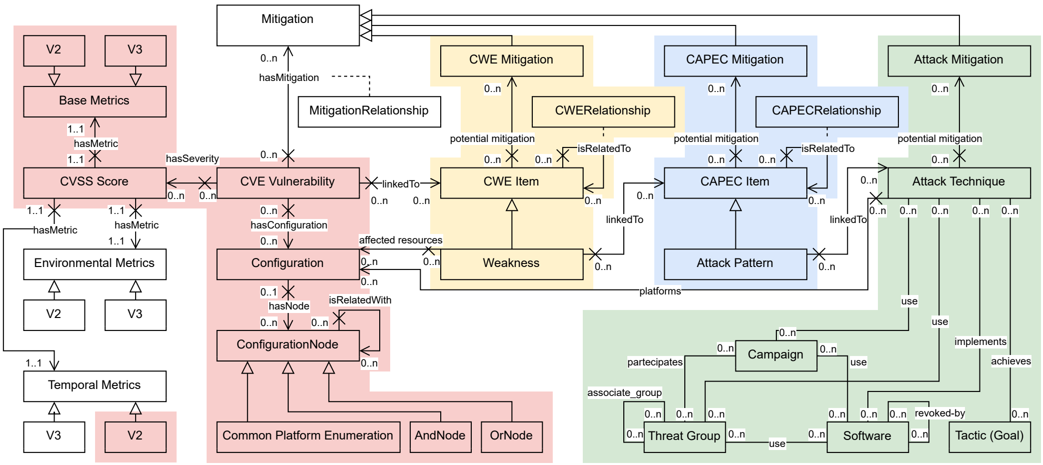


Figure 3.8. Conceptual Model of the integrated knowledge graph $\mathcal{KG}$

In the following, concepts, attributes and relationships that are not directly represented (or represented differently) in the source repositories will be discussed.

The CVE Vulnerability Concept

When modelling a CVE vulnerability, all the relevant attributes already available in NVD have been kept and two additional pieces of information have been added, namely *vulnerable part* and *contextual components* that are not explicitly stated in NVD but that could be inferred by analyzing CPE configuration trees associated to the CVE. More precisely, the attribute *vulnerable part* identifies which is the part in the device stack affected by the vulnerability (e.g., it specifies if the vulnerability affects the hardware, the operating system, an application, or multiple layers of the stack). The attribute *contextual components* represents the list of platforms that enable the vulnerability exploit and thus represent the context to materialize the vulnerability.

CVSS Score Concept

Let us note that the CVSS score specification specifies three types of metrics for every

[6] We used the following color coding: red identifies concepts ad relationships deriving from CVE and NVD, yellow is associated to CWE, light blue is associated to CAPEC and light green is associated to ATT&CK.

vulnerability: (i) *base metrics*, (ii) *temporal metrics* and (iii) *environmental metrics*. However, only the base metrics are available in NVD as the other two are context dependent. It has been decided to include in the proposed knowledge graph all the three types as it is reasonable to believe that such information is of high importance for the analyst because it can support dynamic and context-aware vulnerability analysis. In this case, three classes of metrics have been simply associated to the concept of CVSS score (base, temporal and environmental metrics) and the existence of two possible scoring mechanism have been modeled, depending on the specification that is being considered (v2.0 vs v3.1 if available, 3.0 otherwise).

Mitigation Concept
From the point of view of a security analyst or a risk assessor, it is important to identify if a possible mitigation for a detected vulnerability exists, as well as its degree of *relevance* (i.e. how relevant is the mitigation to the vulnerability). Thus, the notion of *mitigation* has been explicitly modeled and has been associated to a CVE vulnerability. In addition, considering that the notion of mitigation is recurrent in the repositories (even if it is in the form of an attribute and with a different flavour in every repository) it has been decided to further specialize it by modeling the concepts of *CWE mitigation*, *CAPEC mitigation* and *ATT&CK mitigation*. Let us note that CAPEC and CWE, by design, represent concepts at a different level of granularity with respect to NVD (i.e., CWE generalizes and aggregates multiple vulnerabilities under the same weakness and CAPEC defines patterns by linking different weakness). However, a mitigation for a weakness could still provide some indication on how to mitigate a specific problem. Thus, it has been decided to model this aspect by associating an attribute i.e., *relevance value*, to the relationship between CVE vulnerability and the associated mitigation.

Implicit Relationships
During the in-depth analysis of the attributes of each repository, it has been noticed that there could be additional (implicit) relationships between elements of different repositories. In particular, this is true for all the repositories that provide information about platforms or affected resources. In the proposed knowledge graph, the concept of platform has been introduced explicitly (i.e., the Common Platform Enumeration concept) and thus an implicit relationship between entities having platforms or affected resources attribute and the CPE concepts has been introduced.

This section detailed the structure of the ontology which is at the base for the proposed unified data model. In order to transform this ontology into a knowledge graph, as per the definition proposed by Ehrlinger and Wöß [34], the ontology must be able to support a correlation engine capable of extracting information which is not already present in the source repositories.

To this aim, the next section will detail how a correlation engine may be able to extract new information using (i) only the information in a single source repository, as well as (ii) information from multiple source repositories.

More specifically, the ability for a correlation engine to exploit the graph in order to solve following tasks which are part of the cyber security assessment process will be analyzed:

- *Given the set of vulnerabilities affecting a specific host, for each vulnerability get their vulnerable and contextual parts.*
- *Given the set of vulnerabilities affecting a specific host, get a set of possible mitigation actions that could be applied, ranked according to their relevance with respect to the considered vulnerability.*

- *Given the set of vulnerabilities affecting a specific host and a ranked set of possible mitigation actions that could be applied, get the cost derived from applying each mitigation action to the host.*

3.3 Computing the Integration

In order to construct and keep the knowledge graph presented in Section 3.2 up to date, a data collection, integration and correlation system has been implemented, as shown in Figure 3.9. The system is composed of a set of 4 parsers (one for each repository) that gather data from each source and transform it using a graph-based representation according with the data model presented in Figures 3.2 - 3.7. Once the input data is represented in a graph-based format, it can be processed in the *Graph Integration* step to extract data not directly available in the source repositories (e.g., identifying all the mitigation nodes) and to generate the corresponding graph according to the data model in Figure 3.8. The last step of the process is storing the processed data in a graph data base that will support the query answering.

Let us note that building and maintaining (at least the core of) the proposed integrated knowledge graph is theoretically possible and relatively easy due to the presence of cross-repository references. However, its effectiveness and usefulness depend on the amount of existing cross-repositories links and on the quality of the attributes that have been instantiated for each entry. In this chapter, the objective is to analyze such integrated knowledge graph to understand its strengths and its weaknesses and to point out the practical challenges in its population.

In order to do so, the specific tasks introduced in the previous section (i.e., finding vulnerable and contextual parts of each vulnerability, finding all the possible mitigation for a given host, evaluating the cost of applying a mitigation) will be considered, in order to highlight the challenges and peculiarities of correlating information (i) within a single repository, and (ii) between multiple repositories. Lastly, it will be shown how the correlation engine may be able to work using the products of previous correlations in order to generate new information which is still relevant with respect to the cyber security assessment process.

3.3.1 Task 1: Identify the vulnerable part and the contextual part

This section will be dedicated to showing how a correlation engine may be able to extract information relevant to the cyber security assessment process, by only acting on a single repository of the proposed knowledge graph.

In particular, the problem chosen to demonstrate this capability is the following:

Given the set of vulnerabilities affecting a specific host, for each vulnerability get their vulnerable and contextual parts.

This means showing how it is possible to instantiate, for every CVE vulnerability, (i) the *vulnerable part* attribute i.e., the part of the system that is affected from the vulnerability (e.g., hardware, application or operating system) and (ii) the *contextual components* attribute (i.e., the components that are necessary as a precondition to exploit the considered CVE).

Let us note that the vulnerable part of a vulnerability is an information not directly available in the NVD's CVE description but it must be (manually) inferred by the analyst analyzing the CPE configuration trees associated with the vulnerability. Thus, to retrieve this information and to store it explicitly as an attribute associated

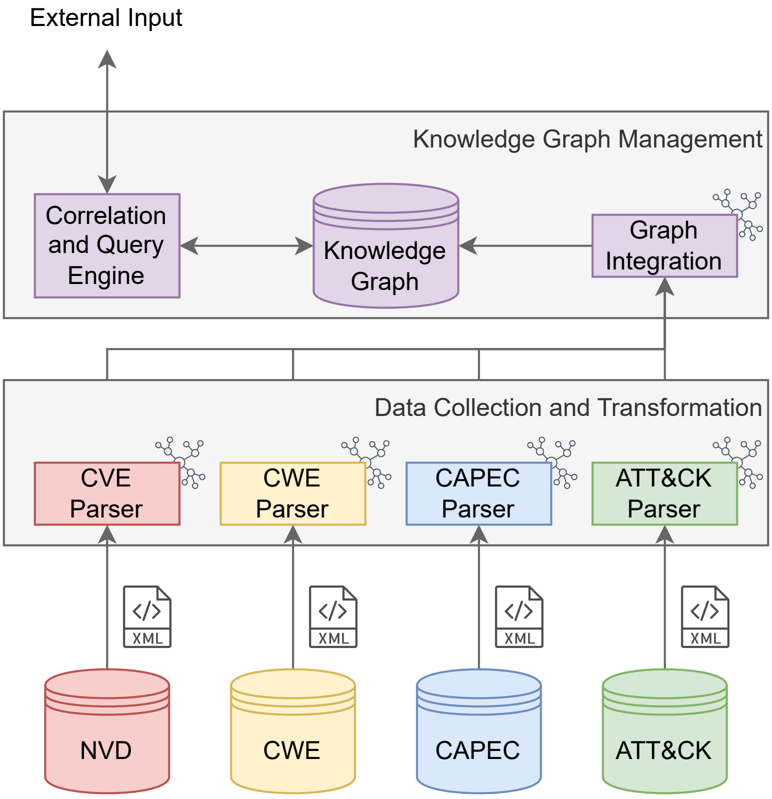


Figure 3.9. Data Collection, Integration and Correlation architecture to support the proposed knowledge graph.

to a CVE node in the knowledge graph, every CVE in NVD will need to be processed automatically.

For the purpose of this study, when considering a CPE string, just the mandatory sub-string *part:vendor:product* has been considered. This is due to a change in the CPE protocol, from version 2.2 to version 2.3 which extended the CPE string with new additional optional fields, mostly left blank by human analysts, and thus not practically helpful for our purposes.

Let us recall that in NVD, every CVE vulnerability has attached one or multiple *configuration trees* representing the possible alternative CPE configurations that may lead to a potential exploit. A configuration tree can thus be traversed from its root down to the leaves represented by CPE strings[7] and it suggests alternative configurations (i.e., those related by an OR relationship) or configurations that need to co-exist to make the exploit possible (i.e., those related by an AND relationship). Each CPE string (i.e., each leaf) in a CPE configuration tree has associated a *vulnerable* boolean value. Given a particular configuration, this value has the purpose to differentiate between actual vulnerable components (e.g., vulnerable applications, hardware or operating system), and non-vulnerable components that however act as a necessary condition for the vulnerability exploitation.

As an example, let us consider the simple CPE configuration tree in Figure 3.10. It

[7] A CPE string is obtained by the concatenation of attributes specified in Section 3.1.1.

can be seen that the application *vuln_application_name* from vendor_1 is the actual vulnerable component while the hardware component *platform_name* from vendor_2 is not vulnerable itself but it is an enabling component for *vuln_application_name* i.e., *vuln_application_name* is vulnerable only if running on *platform_name* as they are related on the configuration tree by an AND.

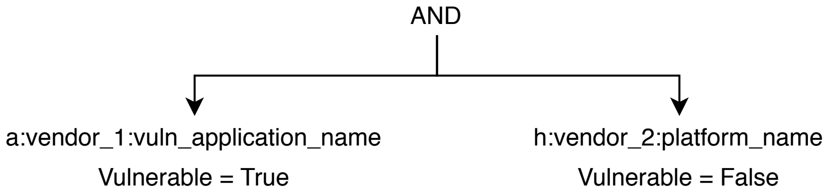


Figure 3.10. Example of a CPE configuration tree

This simple example suggests that given a configuration tree and its particular semantics, CPEs in every sub-tree must be analyzed independently. Moreover, while performing the analysis it is necessary to distinguish between strings labelled as $\mathsf{vulnerable} = \mathsf{True}$ from those labelled as $\mathsf{vulnerable} = \mathsf{False}$.

Let us remark that each CPE in the configuration tree has its own *part* attribute. Thus, given the one-to-many association between one vulnerability and related CPEs leaves, it is possible that vulnerable leaves (i.e., CPE leaves in the configuration tree having the vulnerable flag set to true) show a kind of “heterogeneity”[8] between part attributes. Said differently, vulnerable leaves of the same configuration tree may refer to different parts of the system suggesting that the vulnerability may be “vertical”.

Thus, it can be concluded that selecting the first part attribute or the most frequent one appearing in each tree or forgoing the structure and semantics of the configuration tree (i.e. “flattening” the $\mathsf{AND} - \mathsf{OR}$ tree structure) is not enough.

From the previous observation, more than three part classes have been defined to be possibly associated to a CVE. This is necessary, as different compositions of CPEs part attributes in a single CVE are usually evidence of a vertical problem in the stack (e.g., problems at the firmware level that could affect both hardware and operating system parts or in system libraries that could affect both application and operating system parts). For this reason, for any given CVE entry, the following part classes have been defined:

- Class A: all the considered CPE strings refer to the application part (i.e., their part attribute is equal to a);
- Class H: all the considered CPE strings refer to the hardware part (i.e., their part attribute is equal to h);
- Class O: all the considered CPE strings refer to the operating system part (i.e., their part attribute is equal to o);
- Class A-H: there is a mix of CPE strings, some referring to the application part and others to the hardware part (i.e., their part attribute is equal to a or to h);
- Class A-O: there is a mix of CPE strings, some referring to the application part and others to the operating system part (i.e., their part attribute is equal to a or to o);

[8] Intended as "diversity". Not to be confused with the concept of "tree heterogeneity".

- Class O-H: there is a mix of CPE strings, some referring to the operating system part and others to the hardware part (i.e., their part attribute is equal to o or to h);
- Class A-O-H or MIXED: there is a mix of CPE strings, where all the possible parts of the system are referenced (i.e., their part attribute is equal to a, o or to h).

To capture such classes, the *vulnerability part vector* vec_i has been introduced: a 3-dimensional binary vector where each entry is associated with a part attribute (i.e., the first to A, the second to O and the third and last one to H) and where every entry is equal to 1 if in the configuration there exists at least one CPE with the corresponding part, 0 otherwise. Each vector maps directly to a class.

At this point, computing a value for the attributes *vulnerable part* and *contextual part* of a CVE entails computing two *vulnerability part vectors* vec_i, one relevant to parts that are labeled as vulnerable, one relevant to parts that are labeled as not vulnerable but still necessary in order to express the vulnerability (i.e. contextual). Moreover, the algorithm that performs this calculation must necessarely take into account the semantics introduced by the configuration tree structure which links CVE to each CPE.

To this aim, Algorithm 1 is presented. This proposed algorithm correlates data in the NVD repository in order to identify the vulnerable part as well as the contextual part of a given input vulnerability.

Algorithm 2 shows the implementation of the compute_vulnerable_part_vector() function and the compute_contextual_part_vector() used in Algorithm 1 that is able to select leaf CPE strings taking into account the configuration tree structure.

The algorithm works as follows: it starts by assuming that all parts are potentially vulnerable (i.e., making the hypothesis of being in the class AOH) and that no part is potentially part of the context and then iterates over every sub-tree to find a subset of parts that are vulnerable as well as the set of parts that are part of the context in every configuration sub-tree. More in detail, the proposed algorithm:

1. Takes a sub-tree st_j (i.e. a single configuration) having as root a child of the CPE configuration tree root[9]
2. Maintains two copies of the sub-tree:
   - A vulnerable copy, where only leaf nodes (and relevant paths) labelled as vulnerable = False are pruned
   - A contextual copy, where no nodes are pruned
3. Takes all the relevant leaves and identifies the vulnerable parts for the considered configuration tree (calculated in *compute_common_parts*, line 8 and 21 of algorithm 2, as the union of all part attributes of the CPE strings that partake in the configuration tree)
4. Either:
   - Computes the intersection with the previously identified vulnerable parts to update the vector indicating the vulnerable part class

[9] A single configuration is an AND – OR tree. Multiple configurations are linked to a single CVE by OR.

Algorithm 1: Classification Algorithm with multi-part classes

```
Input: A CVEs list $L_{CVE}$ and the knowledge graph $\mathcal{G}$
Output: A list of tuple $\langle cve\_id, cve\_vulnerable\_part, cve\_contextual\_part \rangle$ where
        $cve\_id$ is the CVE identifier, $cve\_vulnerable\_part$ is the associated affected
        vulnerable part and $cve\_contextual\_part$ is the associated contextual part
$L'_{CVE} \leftarrow \emptyset$;
foreach $v_i \in L_{CVE}$ do
    $vulnerable\_vec_i \leftarrow$ compute_vulnerable_part_vector($v_i$);
    $contextual\_vec_i \leftarrow$ compute_contextual_part_vector($v_i$);
    $vulnerable\_class \leftarrow$ map_vec_to_class($vulnerable\_vec_i$);
    $contextual\_class \leftarrow$ map_vec_to_class($contextual\_vec_i$);
    $L'_{CVE} \leftarrow \{\langle v, vulnerable\_class, contextual\_class \rangle\}$;
end
return $L'_{CVE}$


function map_vec_to_class($vec$)
begin
    case $vec == [0, 0, 0]$ do
        return MIXED
    case $vec == [0, 0, 1]$ do
        return H
    case $vec_i == [0, 1, 0]$ do
        return O
    case $vec_i == [1, 0, 0]$ do
        return A
    case $vec_i == [0, 1, 1]$ do
        return OH
    case $vec_i == [1, 1, 0]$ do
        return AO
    case $vec_i == [1, 0, 1]$ do
        return AH
    case $vec_i == [1, 1, 1]$ do
        return AOH
end
```

- Computes the union with the previously identified contextual parts to update the vector indicating the contextual part class

5. Maps the vectors indicating contextual and vulnerable part classes into part classes

6. Returns the resulting *vulnerable_class* and *contextual_class* for each CVE

Algorithm 2:
compute_vulnerable_part_vector() function implementation leveraging on the intersection of vulnerable configuration trees
compute_contextual_part_vector() function implementation leveraging on the union of vulnerable and non vulnerable (i.e. contextual) configuration trees

```
function compute_vulnerable_part_vector($v_i$)
begin
    $parts \leftarrow \{A, O, H\}$;
    $T_{CPE} \leftarrow$ get_CPE_tree_from_NVD($v_i$);
    foreach $sub\text{-}tree\ st_j \in T_{CPE}$ do
        $vul\_st_j \leftarrow$ prune_not_vulnerable_branches(st_j);
        $st\_leaves_j \leftarrow$ get_leaves($vul\_st_j$);
        $part_j \leftarrow$ compute_common_parts($st\_leaves_j$);
        $parts \leftarrow parts \cap part_j$;
    end
    return $parts$
end


function compute_contextual_part_vector($v_i$)
begin
    $parts \leftarrow \emptyset$;
    $T_{CPE} \leftarrow$ get_CPE_tree_from_NVD($v_i$);
    foreach $sub\text{-}tree\ st_j \in T_{CPE}$ do
        $st\_leaves_j \leftarrow$ get_leaves($st_j$);
        $part_j \leftarrow$ compute_common_parts($st\_leaves_j$);
        $parts \leftarrow parts \cup part_j$;
    end
    return $parts$
end
```

As an example let us consider the vulnerability CVE-2022-0138 whose configuration tree is depicted in Figure 3.11. The configuration tree tells us that this vulnerability has five sub-trees as alternative exploitable and among those, four sub-trees reveals that two CPEs are needed for the exploit: one is the real vulnerable part and the other the context needed for the exploit. Using the sub-trees analysis according to function compute_vulnerable_part_vector Algorithm 2 we get $part = \emptyset$, leading to classify the vulnerability as MIXED meaning that it may affect vertically the system. This assumption is reinforced by the result of function compute_contextual_part_vector which yields $part = \{A, O, H\}$ which maps to the classification AOH, confirming that the context for such vulnerability is to be found across hardware, operating systems and applications.

Thus, in such a way, computing *contextual* and *vulnerable* platform classes for a vulnerability becomes possible through the application of a correlation engine on the proposed knowledge graph. It is however necessary to point out that deriving

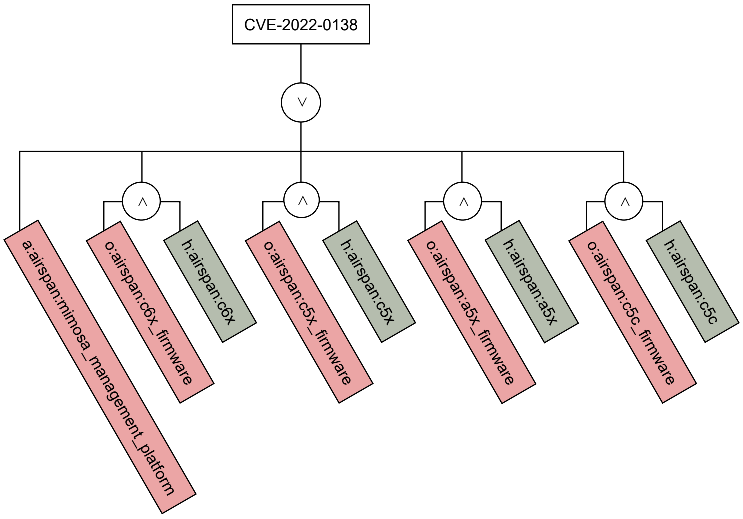


Figure 3.11. Example of the CPE configuration tree for CVE-2022-0138.

new information from a single source is not sufficient for an ontology to qualify as a *knowledge graph*, according to definition [34]. Proof of this capability is to be shown in the following section.

3.3.2 Task 2: Finding Mitigation Actions

This section will be dedicated to showing how a correlation engine may be able to extract information relevant to the cyber security assessment process, by correlating information spanning different repositories of the proposed knowledge graph.

In particular, the problem chosen to demonstrate this capability is the following:

Given the set of vulnerabilities affecting a specific host, get a set of possible mitigation actions that could be applied, ranked according to their relevance with respect to the considered vulnerability

This entails for each vulnerability v_i (i) finding the set of mitigation actions, as well as (ii) devising a metric to compute a ranking between different mitigation actions.

The first step needed to answer the question is simple given the original structure of the repositories and the available explicit cross-links. Indeed, starting from individual repositories (i.e., NVD, CWE, CAPEC and ATT&CK), any entry can be analyzed to check if it has a mitigation attribute associated. If a mitigation attribute is found, a new mitigation node can be instantiated in the knowledge graph and associated with the corresponding source node as well as specific type of mitigation, depending on where it has been found.[10] Then, for every pair $\langle v_i, m_j \rangle$ where v_i is a vulnerability which is part of the knowledge graph and m_j is a mitigation, an edge can be set up from v_i to m_j if the considered mitigation is reachable from v_i navigating existing links between different repositories. This is done through a simple exploration of the knowledge graph following the schema detailed in Figure 3.8.

[10] Mitigation actions are represented as nodes in the proposed knowledge graph, while may appear as attributes in some of the repositories. Moreover, in the proposed knowledge graph the “parent” concept of “mitigation” is specialized in different *types*, i.e. NVD,CWE.. , of mitigation, according to their origin.

Once mitigation actions have been instanced as nodes, and connected to reachable vulnerabilities as well as their node of origin in the knowledge graph, the main remaining challenge is to compute the relevance of a mitigation m_j for a vulnerability v_i.

Computing the relevance of a mitigation for a given vulnerability

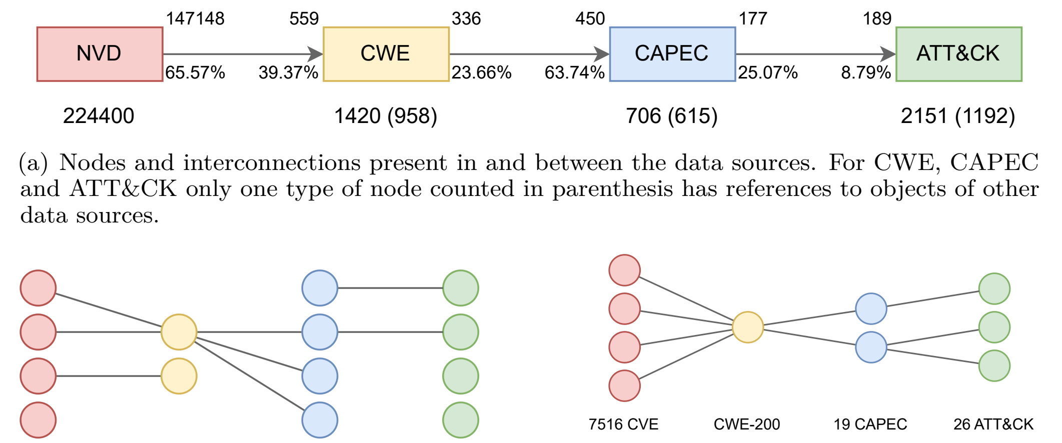
Let us recall that the *relevance* attribute of the edge $\langle vulnerability, mitigation\rangle$ has the aim to quantify how much the proposed mitigation is a good solution for the considered vulnerability.

In principle, a mitigation m_j has a good relevance for a vulnerability v_i (i.e., the relevance level is 1) if it is a patch (or a solution) directly provided by the vendor for the specific issue. If this is the case, the mitigation is typically referenced directly inside NVD. Unfortunately, most of the time NVD entries do not provide a mitigation themselves. However, vulnerabilities typically have references to CWE (which abstracts the vulnerability in a class of issue) and CWE nodes usually may have a mitigation field associated. Thus, it is possible to navigate among repositories, using explicit references, to check if there exists a known mitigation for a specific vulnerability.

The basic idea is that the "closer" the mitigation is to the vulnerability (i.e., fewer references we need to explore), the "higher" will be the relevance of the mitigation for the vulnerability.

Let us recall that, when traversing existing links from CVE to CWE to CAPEC we are generalizing and abstracting the considered issue. Thus, if a vulnerability has one mitigation available in NVD, this will have the highest possible relevance. If not, it becomes necessary to look to mitigation defined for linked CWEs (i.e., for the category of the issue where the vulnerability belongs) and in such case, a smaller relevance score is assigned due to the fact that we are considering a solution to a potential more general problem and thus not perfectly relevant to the specific issue. For this reason, in the proposed instance, it has been decided to assign a relevance level equal to 0 to CAPEC mitigations and ATT&CK mitigations. The rationale is that the link between the attack patterns, attack techniques and the considered vulnerability is not direct (i.e., it is obtained by transitively exploring links from NVD to CWE to CAPEC and from CAPEC to ATT&CK). Thus, it is not easy to assess to what extent the vulnerability is really involved in the attack pattern or in the attack techniques and consequently the relevance of the considered mitigation for the vulnerability.

This decision is supported by Figure 3.12, which highlights some topological issues that arise from the concatenation of different data sources into a single, interconnected graph structure. Most notably three things can be seen from this figure: (i) Figure 3.12(a) highlights the participation in the connections of each element of each data source. In particular it can be seen that in every case, no more than 65.57% of the elements (nodes) of a specific repository has references to any other element of another repository. (ii) Figure 3.12(b) highlights the fact that the references between elements of different repositories are made on a next-hop basis, i.e. an analyst curating a single repository only takes care of connecting an element of such repository to other elements of "subsequent" repositories (e.g. NVD analysts usually only reference CWE entries, CWE analysts usually only reference CAPEC entries and so on). No attention is given to path integrity, e.g. incoming references (what is referencing the current element) and what the referenced elements reference themselves. This leads to isolated elements, as well as "paths" on the NVD-ATT&CK chain either terminating early, or starting late. (iii) Figure 3.12(c) highlights the fact that due to the difference in magnitude between the number of elements of different



(a) Nodes and interconnections present in and between the data sources. For CWE, CAPEC and ATT&CK only one type of node counted in parenthesis has references to objects of other data sources.

(b) Example of topology derived from the interconnection of the data sources. Notice how some elements and paths may (i) remain isolated, (ii) be orphaned (either start "late" or terminate "early" in the chain).

(c) Example of topology derived from the interconnection of the data sources, centered on *CWE-200*. Notice how CWE-200 serves as a focal point for 7516 CVE, 19 CAPEC and 26 ATT&CK elements. According to how these elements are referenced, there is no guarantee that all 19 CAPEC and 26 ATT&CK are still relevant for all 7516 CVE.

Figure 3.12. Examples of topology features derived from the interconnections between data sources.

repositories, bottlenecks and choke points are created which harm the accuracy of the traversal. The figure proposes the example of *CWE-200*, which is referenced by 7516 CVE and references 19 CAPEC. Given the next-hop nature of the references between elements of different repositories, there is no guarantee that all 19 referenced CAPECs are relevant with respect to all 7516 CVE. Solving these issues remains an open research problem and a probable avenue for future works.

Thus, given an edge e of the knowledge graph between a vulnerability v_i and a mitigation m_j, the relevance level, can be defined as follows:

$$relevance(v_i, m_j) = \begin{cases} 1 & \text{if } m_j \text{ is a NVD mitigation associated to } v_i \\ \frac{1}{dist(v_i, m_j)+1} & \text{if } m_j \text{ is a CWE mitigation reachable from } v_i \\ 0 & \text{otherwise} \end{cases} \tag{3.1}$$

The following will detail how the *distance* $dist(v_i, m_j)$ between a vulnerability v_i and a CWE mitigation m_j is computed. Let us recall that CWE nodes are of three different types: views, categories and weaknesses, but only weaknesses have associated the *Proposed_Mitigation* field. Considering that a CVE may refer to a category node (and not necessarily to a weakness), it could be necessary to traverse the CWE graph $\mathcal{G}_{CWE}$ described in Section 3.1 to find a mitigation, increasing thus the distance from the considered vulnerability.

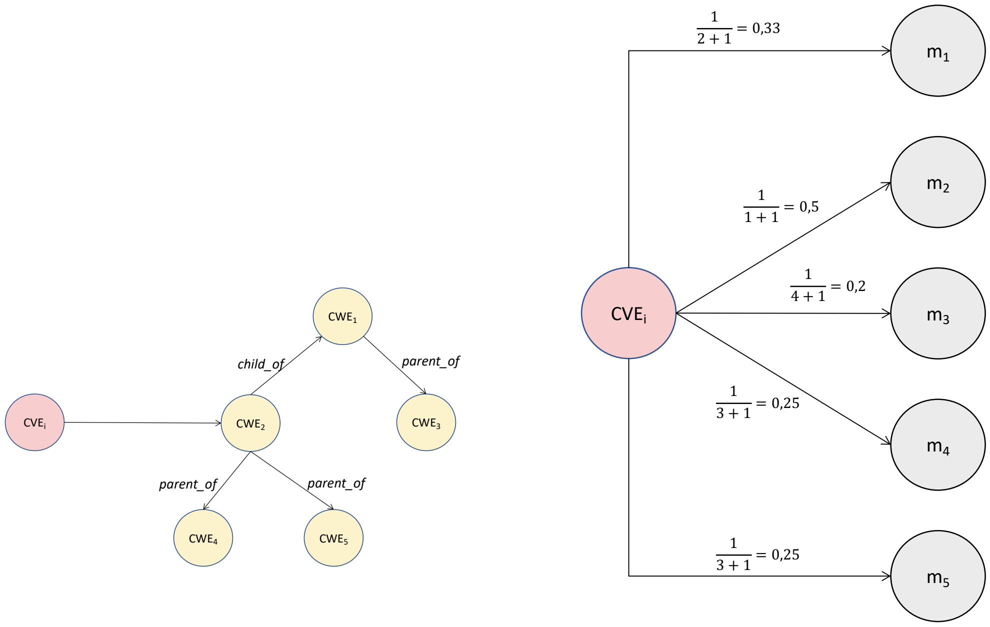
Let us note that the CWE graph $\mathcal{G}_{CWE}$ links view, category and weakness nodes using hierarchical relationships such as *child_of* (or *parent_of*). Given two nodes in the CWE graph $\mathcal{G}_{CWE}$, namely cwe_i and cwe_j, if cwe_i is *child_of* cwe_j then we have that the issue related to cwe_i is more specific than the one described by cwe_j. As a consequence, if cwe_i has a mitigation associated with it, such mitigation may not be applicable to cwe_j as cwe_i is a particular case. Conversely, if cwe_i is *parent_of* cwe_j then we have that the issue related to cwe_i is more generic than the one described by cwe_j. Thus, if cwe_i has a mitigation, this could fit also to cwe_j even if it could not be completely effective.

While computing the distance of the mitigation, the semantics of the CWE relationships need to be considered by giving different weights to traversed edges depending on the associated relationships.

In the following, a heuristic to traverse the CWE graph and compute the distance is proposed. Let v_i be the vulnerability under analysis, let m_j be the CWE mitigation linked to v_i in the knowledge graph and let cwe_j be a CWE node having m_j as an attribute. Let P be the path on the CWE graph $\mathcal{G}_{CWE}$ connecting a CWE directly referenced by v_i (namely cwe_1) with cwe_j. We can define the *distance* $dist(v_i, m_j)$ between the vulnerability v_i and the mitigation m_j as follows:

$$dist(v_i, m_j) = 1 + \sum_{e_{u,v} \in P} \alpha_{u,v}$$

where $\alpha_{u,v} = 1$ if the direct edge $e_{u,v}$ corresponds to the relationships u *is child_of* v and $\alpha_{u,v} = 2$ if the direct edge $e_{u,v}$ corresponds to the relationships u *is parent_of* v.[11]



(a) Example of connection between a vulnerability and a set of weaknesses in the CWE graph

(b) Example of connection between a vulnerability and a set of mitigation actions in the knowledge graph

Figure 3.13. Example of a portion of the knowledge graph during the instantiating process.

As an example, let us consider a vulnerability cve_i. Figure 3.13(a) shows the portion of the CWE graph $\mathcal{G}_{CWE}$ reachable directly or through multiple hops from

[11] While counter intuitive, *child of* and *parent of* relationships in the CWE repository are to be interpreted as "I am the parent of" and "I am the child of". This is clarified later in Figure 3.13(a)

cve_i according to the knowledge graph. Let us assume that cve_i has no mitigation referenced in NVD and let us assume that all CWEs depicted in Figure 3.13(a) have at least one mitigation associated.

Under these conditions, when instantiating the knowledge graph, one mitigation node will be defined for every mitigation m_i associated with a weakness cwe_i represented in the figure and for each mitigation, it is necessary to compute the relevance level in order to have a metric to rank these mitigation actions, from most to least relevant. Figure 3.13(b) shows the result of such computation. Indeed, if the mitigation m_2 associated with cwe_2 is considered, we have that its distance is 1 (i.e., cwe_2 is directly referenced by cve_i and there is no need to find a path within $\mathcal{G}_{CWE}$, thus the relevance is simply $0,5$). Conversely, considering the mitigation m_1, its distance is 2 as we need to consider the edge $\langle cwe_2, cwe_1 \rangle$ with $\alpha_{2,1} = 1$ while the distance for mitigation m_4 is 3 has as we need to consider the edge $\langle cwe_2, cwe_4 \rangle$ with $\alpha_{2,4} = 2$.

This ensures that m_3, associated to CWE_3 is the least relevant mitigation for cve_1, as CWE_3 and CWE_2 may share the same father (i.e. refer to as the same family of vulnerabilities, e.g. CWE-77 "Command Injection") but may refer to different specific sub-families of problems (e.g. CWE-78 "OS Command Injection" and CWE-624 "Executable Regular Expression Error"). This also ensures that m_3 retains at least some relevance with respect to cve_1, as it may happen that different families of vulnerabilities stemming from the same ancestor may still retain retain enough similarity so that actions taken to mitigate one may be relevant for the other.

In the same fashion, this metric ensures that a mitigation on CWE_4 and CWE_5 will always be less relevant than a mitigation on CWE_1. This is desired, as specializations of a broader vulnerability may rapidly lose relevance. As an example, all children of CWE-74 ("Injection") may be easily mitigated by the actions described in CWE-74, while mitigation actions for CWE-94 ("Code Injection") may not be relevant for CWE-79 ("Cross-site Scripting").

Thus, in such a way, computing the ranked list of mitigation actions for a vulnerability becomes possible through the application of a correlation engine on the proposed knowledge graph. Moreover, since at least two sources of information have been used during the correlation in order to produce new information, the proposed ontology can now be properly defined as a *knowledge graph* [34]. In the following section it will be shown how it is possible to extract even more information through correlation of the output of already correlated information.

3.3.3 Task 3: Finding Mitigation Action Cost of Application

This section will be dedicated to showing how a correlation engine may be able to extract information relevant to the cyber security assessment process, by correlating information spanning different repositories of the proposed knowledge graph as well as information that is derived from previous correlations.

In particular, the problem chosen to demonstrate this capability is the following:

Given the set of vulnerabilities affecting a specific host and a ranked set of possible mitigation actions that could be applied, get the cost derived from applying each mitigation action to the host

This entails devising a metric to capture the concept of *difficulty* in applying a mitigation. As per the result of Section 3.3.2, it is possible to fetch all mitigation actions m_j associated to any vulnerability v_i in the proposed knowledge graph.

Let us also recall that the *difficulty* attribute in the proposed knowledge graph should be associated to the relationship between a vulnerability v_i and a mitigation m_j. Consequently it makes sense to model the concept of *difficulty* as the composition of two factors: one related to the "base" difficulty of working with a specific vulnerability, and another one related to the "base" difficulty of applying a certain mitigation regardless of the vulnerability it is applied on.

Mitigation Actions in NVD

Concerning mitigation actions deriving directly from NVD, the format used to tag them has been analyzed, and it has been found that two labels are commonly used: (i) patch and (ii) mitigation. Mitigation actions with label patch have associated a direct reference that allows to download and install the patch for the vulnerability. Conversely, mitigation actions with label mitigation provide a reference to an advisory where the specific issue is described and alternative solutions to resolve the vulnerability are provided. Thus, it has been decided to capture this difference by assigning a base difficulty score of 0.167 to NVD mitigation actions tagged as patch and a base difficulty score of 0.333 to NVD mitigation actions tagged as mitigation (i.e., a direct and specific solution exists but it may be not so easy to apply it).[12]

Mitigation Actions in CWE

Considering mitigation actions deriving from the CWE repository, we need to recall that they are originally specified by the $Proposed\_Mitigation$ field of a CWE entry (i.e. usually a CWE Weakness), which in turn is composed of a description in natural text and two attributes: *mitigation strategies* and *phases*. It is interesting to note that for both of these attributes, the values used to label the mitigation are well-defined and part of a finite dictionary. In particular, for phases we have: (1) Architecture and Design, (2) Build and Compilation, (3) Distribution, (4) Documentation, (5) Implementation, (6) Installation, (7) Integration,(8) Manufacturing, (9) Operation, (10) Patching and Maintenance. (11) Policy, (12) Requirements, (13) System Configuration and (14) Testing, (15) Null value.[13]

Admissible values for mitigation strategies are: (1) Attack Surface Reduction, (2) Compilation or Build Hardening, (3) Enforcement by Conversion, (4) Environment Hardening, (5) Firewall, (6) Input Validation, (7) Language Selection, (8) Libraries or Frameworks, (9) Output Encoding, (10) Parameterization, (11) Refactoring, (12) Resource Limitation, (13) Sandbox or Jail and (14) Separation of Privilege, (15) Null value.[14]

Thus, a mitigation action in CWE is a tuple composed of a description in natural text, a phase attribute and a mitigation strategy attribute of which its values are selected among the listed possible options. As a consequence, the particular composition of a CWE mitigation action's phase and strategy attributes could be used to estimate its base difficulty.

To quantify the difficulty of a CWE vulnerability, a numerical score to any dimension considered needs to be assigned i.e., (i) the *mitigation strategy* and (ii) the *mitigation phase*. These scores are reported in reported in Table 3.7 and Table 3.8, and show the mapping between numerical score and mitigation phase and mitigation strategy respectively.

[12] These values are not random and derive from a mapping that will be explained when introducing the nuances of CWE mitigation actions. This choice has been done to ensure consistency and compatibility across the board.

[13] Null value or attribute not present

[14] Null value or attribute not present

Table 3.7. Mitigation Phase Mapping

Mitigation Phase	Life-cycle Phase	Numerical Score
architecture and design	(iii) Design	0.83
build and compilation	(iv) Development	0.66
distribution	(vi) Training and release transition	0.33
documentation	(vii) Operations and maintenance	0.16
implementation	(iv) Development	0.66
installation	(vi) Training and release transition	0.33
integration	(v) Testing	0.5
manufacturing	(iv) Development	0.66
operation	(vii) Operations and maintenance	0.16
patching and maintenance	(vii) Operations and maintenance	0.16
policy	(vii) Operations and maintenance	0.16
requirements	(ii) Analysis	1
system configuration	(vi) Training and release transition	0.33
testing	(v) Testing	0.5
null value		1

Concerning the mitigation phase mapping, the rationale behind the numerical values is the following: phases considered by CWE can be mapped on to the classical phases of a waterfall software development life-cycle i.e., (i) System investigation, (ii) Analysis, (iii) Design, (iv) Development, (v) Testing, (vi) Training and release transition, (vii) Operations and maintenance. Further in the development life-cycle we need to act, higher will be the difficulty of applying a mitigation. Thus, a scale between 0 and 1 has been defined (where a value close to 1 is assigned to early stages of the process and values close to 0 are assigned to the last phased of the life-cycle) and such intervals have been divided in equally-sized intervals. More in detail, the phases from (ii) to (vii) of the waterfall software development life-cycle have been considered, the interval $[0, 1]$ has been split in 6 consecutive equal intervals and the numerical score has been assigned accordingly.[15] The rationale being that it is harder to apply a mitigation action that requires a total rework of the affected platform, instead of a mitigation action that requires a partial rework or can even be applied during operation, e.g. a mitigation action that requires changes at the “Design” phase of a certain product (or groups of products) is harder to implement than a mitigation action that can be rolled out during “Operations and maintenance”.

Concerning the mapping of the mitigation strategy attribute, the description of the allowed values provided in the CWE documentation have been analyzed, and they have been classified at first according to a discrete scale (i) SEVERE, (ii) HIGH, (iii) MEDIUM, (iv) LOW. This discrete qualitative scale has then been mapped into a numerical range between 0 and 1 where values close to 0 represent strategies that are less difficult than those close to 1. As an example, applying a mitigation action that involves acting on a “firewall” and its rules is to be considered less difficult than applying a mitigation action that requires using different “libraries or frameworks”. As a general rule, mitigation phases and strategies with “null value” are always assigned to the maximum allowed score of 1. This decision has been made to enforce a worst-case analysis by default.

Thus, the base difficulty score for a CWE mitigation action can be evaluated as

$$base\_difficulty(m_j) = MPS \times MSS$$

[15] The numerical scores assigned to the patch and mitigation labels associated to NVD mitigation actions are aligned with the numerical scores of “(vii) Operations and maintenance” and “(vi) Training and release transition” respectively.

Table 3.8. Mitigation Strategy Mapping

Mitigation Strategy	Numerical Score
enforcement by conversion	1
environment hardening	0.5
firewall	0.25
input validation	1
language selection	1
libraries or frameworks	1
output encoding	1
refactoring	1
resource limitation	0.5
sandbox or jail	0.5
separation of privilege	0.25
parameterization	1
attack surface reduction	0.5
compilation or build hardening	0.75
null value	1

where MPS is the numerical score of the *phase* attribute of m_j and MSS is the numerical score of the *strategy* attribute of m_j.

Vulnerability Contribution

In order to calculate the contribution of a vulnerability v_i to the computation of the *difficulty* of applying a mitigation m_j, the result of Section 3.3.1 can be exploited. In particular, the *vulnerable part* associated to a vulnerability v_i can be used to estimate the "base" difficulty of applying any mitigation action to v_i.

Table 3.9. Vulnerable Part Mapping

Vulnerable Part Class		Numerical Score
A	(Application)	0.33
O	(Operating System)	0.66
H	(Hardware)	1
AO	(Libraries)	0.5
OH	(Firmware)	0.833
AH	(High-level Firmware)	0.833
AOH or MIXED	(Cross-layer system libraries)	1

To this aim, table 3.9 shows the numerical mapping associated to the 7 vulnerable classes that have been defined in the previous section. As usual, a scale between 0 and 1 has been used and higher values have been assigned to vulnerable parts where carrying out any action requires more "base" effort (e.g., fixing a hardware vulnerability is more difficult than fixing a vulnerability on an application. The rationale behind the proposed numerical value is the following:

- The scale has been divided in three homogeneous intervals and the three main vulnerable classes have been assigned as the boundaries of such intervals i.e., Application, Operating System and Hardware, considering that acting on the application level is generally easier than acting on the operating system and on the hardware;

- The highest complexity score has also been assigned to the classes AOH or MIXED (Cross-layer system libraries) as these require to perform fixes over all the protocol stack and we consider this as complex as fixing the hardware;

- Fixing firmware has been considered more complex than fixing an operating system or an application but somehow easier than fixing hardware and thus an intermediate value has been assigned (which is actually the mean point between the hardware complexity and the operating system complexity);
- The same rationale is applied to libraries that stand between application and operating system.

Summarizing, the *difficulty* value of applying a mitigation m_j to a linked vulnerability v_i can be estimated as follows:

$$difficulty(v_i, m_j) = VPS \times \begin{cases} 0.167 & \text{if } m_j \text{ is a NVD mitigation associated to } v_i \text{ with label } \texttt{patch} \\ 0.333 & \text{if } m_j \text{ is a NVD mitigation associated to } v_i \text{ with label } \texttt{mitigation} \\ MPS \times MSS & \text{if } m_j \text{ is a CWE mitigation} \\ 1 & \text{otherwise} \end{cases} \tag{3.2}$$

where VPS is the vulnerable part score as for Table 3.9, MPS is the mitigation phase score as for Table 3.7 and MSS is the mitigation strategy score as for Table 3.8.

Thus, in such a way, computing the cost of application of a mitigation action m_j to a vulnerability v_i becomes possible through the application of a correlation engine on the proposed knowledge graph. Moreover, this correlation hinges on the results of two other previous correlations, strengthening the claim of validity for the proposed knowledge graph according to definition [34].

3.4 Evaluation

In this section, challenges and issues related to the population and usage of the proposed knowledge graph will be discussed, given the data available in the considered repositories.

To construct and analyze the presented knowledge graph, the attention has been focused on vulnerabilities affecting Industrial Control Systems (ICSs) to mimic the analysis performed over hosts participating to a critical system. However, due to the generality of the proposed method, the same approach can be used by feeding the knowledge graph with any subset of vulnerabilities gathered by a vulnerability scanner.

In the next sections, the dataset that has been used as input for the analysis will be described, and then the results of instantiating of the knowledge graph, as described in Section 3.3, will be discussed starting from the considered dataset.

3.4.1 Dataset

In order to perform an evaluation of the proposed knowledge graph, the first step has been to instantiate a dataset gathering vulnerabilities (in the form of CVE entries extracted from the NVD database) affecting Industrial Control Systems (ICSs). The dataset includes all vulnerabilities referenced by 2553 ICS-CERT advisories published over the period $2010 - 2023$.

The Industrial Control Systems CERT (ICS-CERT)[16] is part of the Cybersecurity and Infrastructure Security Agency (CISA) and focuses on isolation and advertisement of threats to Industrial Control Systems (ICSs) environment. CISA

[16] https://www.cisa.gov/topics/industrial-control-systems

shares cybersecurity and infrastructure security knowledge and practices with its stakeholders, in order to enable better risk management within the U.S. infrastructures.

ICS-CERT makes available its *advisories* to provide timely information about current security issues, vulnerabilities, and exploits. Advisories are identified by a unique id and inside each advisory, it is possible to find a textual description of the issue, pointers to CWE and CVE as well as a risk evaluation and possible countermeasures.

The final dataset includes 6232 CVE spanning the period September 1999 - September 2023. Table 3.10 provides a summary of the CVE collected starting from the ICS-CERT advisories.

Table 3.10. CVE Dataset Summary

Year	**#CVE**
$\leq$ 2015	889
2016	327
2017	429
2018	562
2019	601
2020	829
2021	1117
2022	1028
2023	450
Total	6232

For each CVE, the related data has been retrieved from NVD (i.e., information related to CPEs and CVSS described in Figure 3.2). Then, to build an instance of the proposed knowledge graph, the whole CWE, CAPEC and ATT&CK repositories have also been downloaded. All the data gathered has been used to create our knowledge graph and it has been stored in a single graph database (using Neo4J[17]) to allow for simple integration.

3.4.2 Evaluating Task 1: Identify the vulnerable part and the contextual part of a CVE vulnerability

This section will analyze the result of applying the algorithms and methodologies presented in Section 3.3.1 when considering the subset of vulnerabilities derived from ICS-CERT advisories as input for the knowledge graph, as described in the previous section.

In particular, it will be shown:

- How vulnerabilities that derive from ICS-CERT advisories are distributed across the identified part classes (see Fig. 3.14 and Table 3.11)
- The result of the analysis of the vulnerability configuration trees leading to such a classification.

Table 3.11 and Fig. 3.14 summarize the result of the classification performed on the evaluation dataset according to the execution of Algorithm 1 using the implementation of the compute_vulnerable_part_vector() function shown in Algorithm 2.

Let us recall that given a vulnerability v_i, the sub-trees of its configuration tree may be associated with different parts (see Figure 3.11) and, in this case, the

[17] https://neo4j.com/

Table 3.11. CVE Classification into Vulnerable Classes

Vulnerable Class	#CVE
A -Application	3432
O - Operating System	2008
H - Hardware	67
AH - High-level Firmware	18
OH - Firmware	60
AO - Cross-layer System Libraries	11
Mixed - System Libraries or System Protocol	623
Empty - Without CPE information	13
Total	**6232**

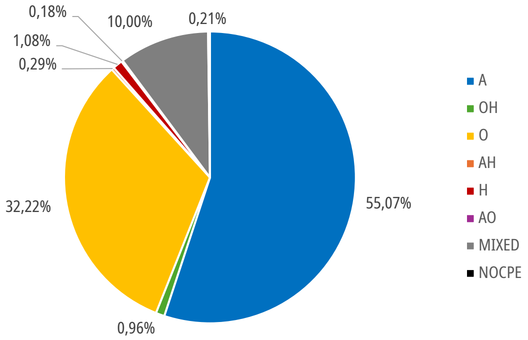


Figure 3.14. CVE Classification into Vulnerable Classes.

proposed algorithm will classify them as Mixed (about 10% of considered evaluation set). Moving forward the vulnerabilities in this class will be called *vulnerability with heterogeneous vulnerable parts.*

Thus, it is of interest to investigate the internal factors leading to this type of classification.

Analyzing the whole dataset, what has been found is that the large majority of CVE have associated vulnerable configurations trees all belonging to the same class (i.e. about 89.1% of CVE). Concerning vulnerabilities with at least two configuration trees belonging to different classes (i.e. about 10.7% of CVE), the analysis pivoted on how many different classes of configuration trees are associated to the same CVE and it has been found that in the evaluation dataset there does not exists more than four different classes of vulnerable configuration trees linked to the same CVE[18] (i.e., the "degree of heterogeneity" of the classes of trees present in a CVE is at most 4).[19]

Concerning CVEs with at least two vulnerable configuration trees belonging to different classes, it was of interest to investigate if the classes of the vulnerable configuration trees were complementary or mutually exclusive. With this aim, the intersection[20] of the classes of such configuration trees has been computed, to

[18] CVE with associated CPE configuration trees fitting in two classes are 9.3%, those fitting in three classes are 1.3% and only one CVE has different CPE trees that fit four classes.

[19] Heterogeneity intended as diversity. Not to be confused with tree heterogeneity.

[20] Let us remind that the "classes" of configuration trees derive from the combination (set-union) of the part attribute of the CPE strings partaking in the tree, e.g. a tree comprised of two "A" part CPE strings and one "O" part CPE string will have "AO" as its class. Therefore the intersection between classes of trees is defined as the class composed of the common CPE part elements of the trees (e.g. the intersection of "AO" and "OH" results in "O").

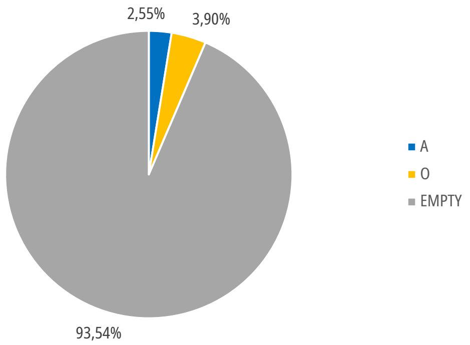


Figure 3.15. Distribution of the intersection of multiple vulnerable configurations.

understand if there is a common aspect in all the vulnerable configuration trees. The result is shown in Figure 3.15.

It is interesting to see that when evaluating the intersection, there is a majority of CVEs resulting in an empty class. This means that these CVEs have associated vulnerable configurations that are somehow complementary and that have as a main consequence the fact of affecting vertically the system. In those cases, a manual inspection has been carried out and it has led to the discovery that the main issue is very likely to be either a fault in a system library or a fault in the protocol stack of the operating system.

3.4.3 Evaluating Task 2: Finding Mitigation Actions

This section will analyze the result of applying the algorithms presented in Section 3.3.2 when considering the subset of vulnerabilities derived from ICS-CERT advisories as input for the knowledge graph, as described in the previous section.

In particular, it will be shown:

- How vulnerabilities that derive from ICS-CERT advisories are served by mitigation actions across the knowledge graph and how much are these mitigation actions relevant with respect to each vulnerability (see Fig. 3.16)
- How much are these mitigation actions actually relevant with respect to each vulnerability (see Fig. 3.17))

Figure 3.16 presents the result of the traversal of the knowledge graph in search for mitigation actions, starting from the evaluation dataset. It is interesting to notice how about 90.61% of vulnerabilities in the evaluation dataset reach at least more than one mitigation action[21]. Since this simple traversal yields more than one mitigation action for each vulnerability, analyzing the mitigation actions in order to pick only the most *relevant* mitigation actions becomes of interest.

Let us recall that due to the semantics of the tree structure of CWE, not all mitigation actions reached by a vulnerability may be relevant: some mitigation actions might be tailored only to certain specializations of the vulnerability thus becoming ineffective, while others might be too broad to be effective.

In order to account for only the most *relevant* (and thus the most effective) mitigation actions for each vulnerability, the maximum *relevance* value among all the

[21] 79.24% of vulnerabilities reach $\geq$ 10 mitigation actions, 16.96% $\geq$ 50 mitigation actions, 2.13% $\geq$ 100 mitigation actions

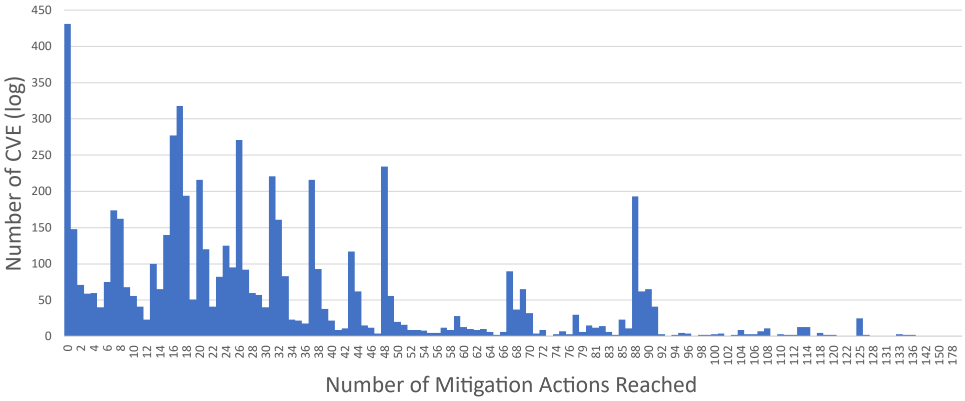


Figure 3.16. Distribution of the number of mitigation actions reached by each vulnerability.

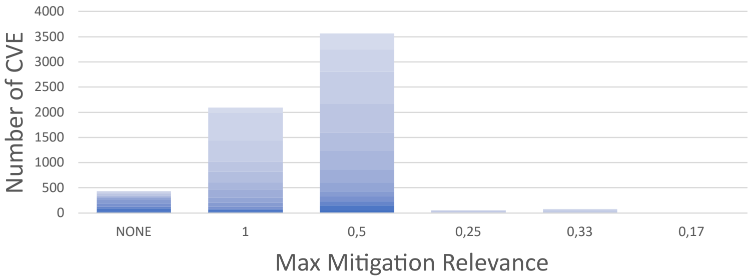
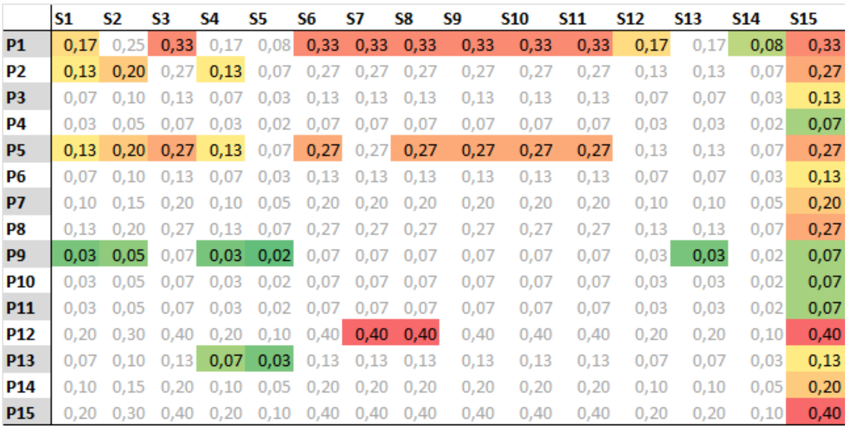


Figure 3.17. Distribution of the maximum mitigation relevance value of each vulnerability. The color scale follows the year of the CVE, ranging from 1999 (darker) to 2023 (lighter).

mitigation actions reached by each vulnerability has been studied in Figure 3.17. It is interesting to note how 6.92% of vulnerabilities reach no mitigation action, 33.58% of vulnerabilities have their closest mitigation in NVD ($relevance = 1$), and 57.25% have their closest mitigation in their associated CWE Weakness ($relevance = 0.5$). Navigation of the CWE tree is necessary only for 2.25% of vulnerabilities, which have their most *relevant* mitigation actions at *relevance* values of either 0.33, 0.25 or for a single case 0.17. The figure also highlights the temporal distribution of the vulnerabilities, with the general trend being that older vulnerabilities tend to have their most *relevant* mitigation actions at a further distance when compared to the most *relevant* mitigation actions from newer vulnerabilities (i.e. newer vulnerabilities tend to have an associated mitigation action in NVD, while older vulnerabilities tend to lack mitigation actions in NVD).

It is also interesting to notice how with every passing year, newer vulnerabilities have been more numerous than the vulnerabilities of the previous year. This growth in the quantity of newer vulnerabilities is to be accredited to the efforts made by US legislators and officials in bolstering the country's Cybersecurity posture [2].



	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11	S12	S13	S14	S15
P1	0,17	0,25	0,33	0,17	0,08	0,33	0,33	0,33	0,33	0,33	0,33	0,17	0,17	0,08	0,33
P2	0,13	0,20	0,27	0,13	0,07	0,27	0,27	0,27	0,27	0,27	0,27	0,13	0,13	0,07	0,27
P3	0,07	0,10	0,13	0,07	0,03	0,13	0,13	0,13	0,13	0,13	0,13	0,07	0,07	0,03	0,13
P4	0,03	0,05	0,07	0,03	0,02	0,07	0,07	0,07	0,07	0,07	0,07	0,03	0,03	0,02	0,07
P5	0,13	0,20	0,27	0,13	0,07	0,27	0,27	0,27	0,27	0,27	0,27	0,13	0,13	0,07	0,27
P6	0,07	0,10	0,13	0,07	0,03	0,13	0,13	0,13	0,13	0,13	0,13	0,07	0,07	0,03	0,13
P7	0,10	0,15	0,20	0,10	0,05	0,20	0,20	0,20	0,20	0,20	0,20	0,10	0,10	0,05	0,20
P8	0,13	0,20	0,27	0,13	0,07	0,27	0,27	0,27	0,27	0,27	0,27	0,13	0,13	0,07	0,27
P9	0,03	0,05	0,07	0,03	0,02	0,07	0,07	0,07	0,07	0,07	0,07	0,03	0,03	0,02	0,07
P10	0,03	0,05	0,07	0,03	0,02	0,07	0,07	0,07	0,07	0,07	0,07	0,03	0,03	0,02	0,07
P11	0,03	0,05	0,07	0,03	0,02	0,07	0,07	0,07	0,07	0,07	0,07	0,03	0,03	0,02	0,07
P12	0,20	0,30	0,40	0,20	0,10	0,40	0,40	0,40	0,40	0,40	0,40	0,20	0,20	0,10	0,40
P13	0,07	0,10	0,13	0,07	0,03	0,13	0,13	0,13	0,13	0,13	0,13	0,07	0,07	0,03	0,13
P14	0,10	0,15	0,20	0,10	0,05	0,20	0,20	0,20	0,20	0,20	0,20	0,10	0,10	0,05	0,20
P15	0,20	0,30	0,40	0,20	0,10	0,40	0,40	0,40	0,40	0,40	0,40	0,20	0,20	0,10	0,40

Figure 3.18. Distribution of mitigation difficulty into the Phases-Strategies space. Grayed out values represent a (Phase, Strategy) combination that is never present in the analyzed datasets. Mitigation Phase IDs are: (P1) Architecture and Design, (P2) Build and Compilation, (P3) Distribution, (P4) Documentation, (P5) Implementation, (P6) Installation, (P7) Integration, (P8) Manufacturing, (P9) Operation, (P10) Patching and Maintenance. (P11) Policy, (P12) Requirements, (P13) System Configuration and (P14) Testing, (P15) Null value. Mitigation strategies IDs are: (S1) Attack Surface Reduction, (S2) Compilation or Build Hardening, (S3) Enforcement by Conversion, (S4) Environment Hardening, (S5) Firewall, (S6) Input Validation, (S7) Language Selection, (S8) Libraries or Frameworks, (S9) Output Encoding, (S10) Parameterization, (S11) Refactoring, (S12) Resource Limitation, (S13) Sandbox or Jail and (S14) Separation of Privilege, (S15) Null value.

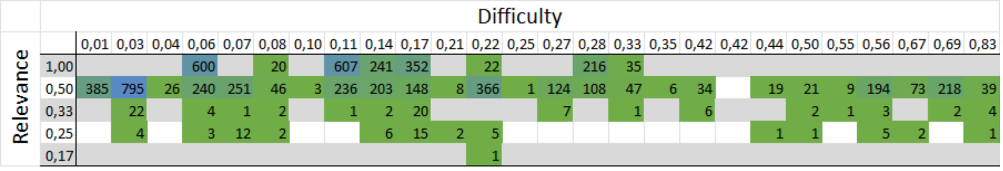
3.5 Evaluating Task 3: Finding Mitigation Action Cost of Application

This section will analyze the result of applying the algorithms and methodologies presented in Section 3.3.3 when considering the subset of vulnerabilities derived from ICS-CERT advisories as input for the knowledge graph, as described in the previous section.

In particular, it will be shown:

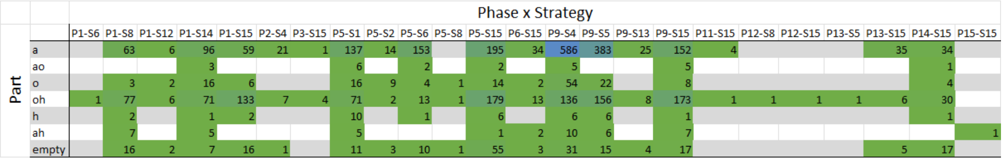
- The associated difficulty of applying each mitigation action with respect to their relevance value (see Fig. 3.19 and Fig. 3.20).

The second metric which is of interest for the evaluation of a mitigation action is its *difficulty* to be deployed to the environment. The first two elements which contribute to each mitigation action's *difficulty* score in the case of a CWE mitigation are it's *Phase* and *Strategy* scores. Figure 3.18 shows the contribution of the combination of these scores to the overall mitigation *difficulty* calculation (the values of the *phase* and *strategy* scores have been defined in Tables 3.7, 3.8 of the previous section). While theoretically there are 15 × 15 possible combinations, only 46 combinations of $\langle phase_{pi}, strategy_{sj} \rangle$ are present in the dataset, with a total of 11 different possible values. This information is encoded in the figure by leaving the corresponding cell uncolored, and having the value grayed out. The presence of colors in the cells highlights the existence of the combination of the *Phase* and *Strategy* values in at least one mitigation action of the CWE repository. The gradient of the color of each cell highlights the difficulty of each combination. From the figure it is noticeable how there is a large presence of mitigation actions that just



	Difficulty																									
Relevance	0,01	0,03	0,04	0,06	0,07	0,08	0,10	0,11	0,14	0,17	0,21	0,22	0,25	0,27	0,28	0,33	0,35	0,42	0,42	0,44	0,50	0,55	0,56	0,67	0,69	0,83
1,00				600		20		607	241	352		22			216	35										
0,50	385	795	26	240	251	46	3	236	203	148	8	366	1	124	108	47	6	34		19	21	9	194	73	218	39
0,33		22		4	1	2		1	2	20				7		1		6			2	1	3		2	4
0,25		4		3	12	2			6	15	2	5								1	1		5	2		1
0,17												1														

Figure 3.19. Distribution of CVE vulnerabilities into the "best" mitigation's Relevance-Difficulty space.



	Phase x Strategy																							
Part	P1-S6	P1-S8	P1-S12	P1-S14	P1-S15	P2-S4	P3-S15	P5-S1	P5-S2	P5-S6	P5-S8	P5-S15	P6-S15	P9-S4	P9-S5	P9-S13	P9-S15	P11-S15	P12-S8	P12-S15	P13-S5	P13-S15	P14-S15	P15-S15
a		63	6	96	59	21	1	137	14	153		195	34	586	383	25	152	4				35	34	
ao				3				6		2		2		5			5						1	
o		3	2	16	6			16	9	4	1	14	2	54	22		8						4	
oh	1	77	6	71	133	7	4	71	2	13	1	179	13	136	156	8	173	1	1	1	1	6	30	
h		2		1	2			10		1		6		6	6		1						1	
ah		7		5				5				1	2	10	6		7							1
empty		16	2	7	16	1		11	3	10	1	55	3	31	15	4	17					5	17	

Figure 3.20. Distribution of CVE vulnerabilities into the "best" mitigation's Difficulty partial score (Phase Score × Strategy Score) and the CVE's Part Score. Only mitigations with a relevance value of 0.5 have been considered.

indicate a *phase* and leave a null value for the *strategy* (S15). The *phases* which are most commonly combined with a null *strategy* are: *implementation*, *architecture and design* and *testing*. The most recurrent combinations of $\langle phase_{pi}, strategy_{sj}\rangle$ for all the CWE mitigation actions in the dataset are: $\langle implementation, null\_value\rangle$ (occurring 488 times), $\langle architecture\_and\_design, null\_value\rangle$ (occurring 365 times), $\langle implementation, input\_validation\rangle$ (occurring 190 times).

Another measure of interest for each mitigation action is its *relevance* - *difficulty* trade off. Thus, the distribution of the vulnerabilities in the dataset across the $Relevance \times Difficulty$ space, according to their most *relevant* (max relevance) and least *difficult* (min difficulty) mitigation action is studied in figure 3.19. Notice how in this case, the addition of the *Vulnerability Part Score* (VPS, as defined in table 3.9) results in more than 11 discrete difficulty values. The majority (90.83%) of the vulnerabilities reside between the rows indicating relevance 1.00 and 0.50, i.e. have their mitigation action either in NVD, or on their directly associated CWE Weakness. Only 139 (3.22%) vulnerabilities in the dataset have their best relevance mitigation score below 0.50. As per before, 431 vulnerabilities have no access to any mitigation action. On the difficulty axis, most vulnerabilities (74.17%) indicate mitigation actions with a difficulty equal or inferior to 0.25, indicating the presence of at least one mitigation action with a relatively low difficulty level.

It is important to remember that the difficulty score is also influenced by the vulnerability's *Vulnerable Part Score* (VPS). For this reason, it is of interest to also study the impact and distribution of the vulnerability's vulnerable *part* score during the mitigation *difficulty* calculation. This study is achieved by figure 3.20, which highlights the composition of the components of the *difficulty* calculation (*Part* and $Phase \times Strategy$ scores) given a set *relevance* value of 0.5. While at first glance there seems to be a relationship between certain *Part* categories and mitigation actions with certain combinations of *Phase* and *Strategy* attributes, it is important to consider that the distribution of vulnerabilities into the *Part* categories is not uniform (i.e. certain *Part* categories may collect more vulnerabilities than others).

Taking this consideration into account, after normalization, no clear indication is evident about any association between a vulnerability's *Part* class and its linked mitigation's *Phase* and *Strategy* attribute combination. However, an interesting result is obtained by studying the distribution of vulnerabilities into their selected mitigation's combination of *Phase* and *Strategy* attributes (collapse all rows into one by summing values, consider only the columns). From this study, it is evident that just the top three combinations of *Phase* and *Strategy* attributes (i.e. P9-S4, P9-S5, P5-S15)[22] are sufficient to encompass the mitigation actions selected by 1868 vulnerabilities (52.35% of the total in Figure 3.20).

[22] Operation - Environment Hardening, Operation - Firewall, Implementation - Null Value

Table 3.12. Mitigation actions found for CVE-2008-0760, scored accordingly

CVE-2008-0760

Mitigation	**Relevance**	**VP:Score**		**MP:Score**		**MS:Score**		**Difficulty**
Application Firewall	0.5	A	0.33	Operation	0.16	Firewall	0.25	0.013
Use lowest privileges	0.5	A	0.33	Operation	0.16	Environment Hardening	0.5	0.026
Use Sandbox	0.5	A	0.33	Operation	0.16	Sandbox or Jail	0.5	0.026
Store library includes outside root	0.5	A	0.33	Operation	0.16	Attack Surface Reduction	0.5	0.026
Don't use register globals on PHP	0.5	A	0.33	Implementation	0.66	Environment Hardening	0.5	0.109
Check for unreleased resources	0.25	A	0.33	Testing	0.5	Null Value	1	0.165
Assume all inputs to be malicious	0.5	A	0.33	Implementation	0.66	Input Validation	1	0.218
Decode and canonicalize input	0.5	A	0.33	Implementation	0.66	Input Validation	1	0.218
Minimal details on error messages	0.5	A	0.33	Implementation	0.66	Null Value	1	0.218
Assume all inputs to be malicious	0.25	A	0.33	Implementation	0.66	Input Validation	1	0.218
Decode and canonicalize input	0.25	A	0.33	Implementation	0.66	Input Validation	1	0.218
Assume all inputs to be malicious	0.17	A	0.33	Implementation	0.66	Input Validation	1	0.218
Decode and canonicalize input	0.17	A	0.33	Implementation	0.66	Input Validation	1	0.218
Duplicate client security checks on server	0.5	A	0.33	Architecture and Design	0.83	Null Value	1	0.270
Use library or framework that prevents the issue	0.5	A	0.33	Architecture and Design	0.83	Libraries or Frameworks	1	0.270
Create a whitelist of acceptable input objects	0.5	A	0.33	Architecture and Design	0.83	Enforcement by Conversion	1	0.270

As a last step, Table 3.12 shows a concrete example of the result of the application of all three tasks conducted on vulnerability CVE-2008-0760. This vulnerability is part of the evaluation set and identifies a "Directory traversal vulnerability" affecting a specific application[23]. Since only one configuration is present, and all vulnerable platforms of this configuration point to a specific application, its *vulnerable platform* is established as "A" (Application). This maps to a VPS of 0.33, as per Table 3.9. No *patch* or *mitigation* are available in NVD, so CWE traversal becomes necessary. CVE-2008-0760 is directly connected to CWE-22 (Path Traversal)[24]. CWE-22 is a *Weakness* type CWE with 11 Potential Mitigations associated. Thus, the 11 mitigation actions are reported in the table as having a *relevance* value of 0.5. Continuing the exploration, CWE-22 is the parent of CWE-23 (Relative Path Traversal) which has two Potential Mitigations associated. Since a $parent\_of$ relationship has been traversed, these mitigation actions are given a *relevance* score of 0.25. Another mitigation action is given a *relevance* score of 0.25, being part of CWE-22's *grandfather*, CWE-664 (Improper Control of a Resource Through its Lifetime) through CWE-668 (Exposure of Resource to Wrong Sphere). Two more mitigation actions derive from children of CWE-668, which in the CWE tree structure are effectively CWE-22's *sibling* nodes. This grants these two mitigation actions a *relevance* score of 0.17. Each mitigation action also has its own *phase* and *strategy* attributes, which map to their own MPS and MSS. With all three scores evaluated, the *difficulty* of applying each mitigation action is calculated and a ranking can be applied. As expected, the top four less *difficult* mitigation actions that can be applied to CVE-2008-0760 do not involve extensive rework of the application (the *phase* is always Operation and the *strategies* span across Firewall, Environmental Hardening, Sandbox or Jail and Attack Surface Reduction). Also as expected, *difficult* mitigation actions involve re-implementing and for the last three ranked mitigation actions completely re-designing the application. Lastly, it can be noted that less *relevant* mitigation actions tend to sit at the bottom of each given $MPS \times MSS$ *bracket*. This hints at the tendency of the *relevance* score to act as a discriminant between mitigation actions with comparable *phase* and *strategy* combinations.

3.6 Conclusion

This chapter investigated the structure of the main public repositories used by the cyber security community to collect and share information about vulnerabilities and their possible exploits i.e., NVD, CWE, CAPEC and ATT&CK. Starting from this huge amount of information managed independently by the repositories owners, the aim of this chapter has been twofold: (i) presenting and discussing such repositories taking a different perspective i.e., by considering them as one unique knowledge graph that can be traversed and analysed to answer complex queries and (ii) verifying the suitability of the knowledge graph as a reference knowledge base in the most common vulnerability analysis tasks. It has been found that the high degree of connection between concepts in different repositories supports well the two considered tasks. In particular, a mechanism leveraging the knowledge graph has been proposed to automatically correlate information across multiple repositories to speed up the process and support analysts in the vulnerability analysis process. More in detail, when identifying which is the vulnerable part of the system targeted by a specific vulnerability, it has been observed that performing the classification automatically

[23] https://nvd.nist.gov/vuln/detail/CVE-2008-0760

[24] https://cwe.mitre.org/data/definitions/22.html

may provide the following benefit to security operators as well as self protecting systems:

- It is immediately possible to relate the issue to a specific part of the system and evaluate the dependencies between the vulnerable part and the enabling context.
- Assuming that patches are available for any vulnerability, it is possible to assess the difficulty of mitigating the vulnerability also as a function of its class. Usually, patching vulnerable hardware is harder than patching an application. Thus, the classification can help in prioritizing mitigation activities.
- The classification between vulnerable and non-vulnerable configurations can be used as input for the enrichment of the CVSS score and in particular for the definition and computation of the environmental score.

Concerning the second analysis task, a strategy to identify common mitigation actions (in terms of strategies) for a set of vulnerabilities has been proposed. This information will provide the analyst with fast feedback on the possible options and will contribute to enhancing the overall level of awareness.

Let us note that this benefit derives directly from the deep knowledge of the repositories and from the formalization of the relationships that has been done in the knowledge graph. Indeed, it has been necessary to go deeper in understanding the details of the relationships that characterize data stored. As an example, CPE analysis revealed that some attributes associated with nodes could be misleading (e.g., looking just at the part attribute of the first CPE string linked to a vulnerability may lead to a wrong classification) and it highlighted the complexity of finding the real vulnerable part.

Concerning the third analysis task, a methodology to estimate the cost of applying a mitigation action on a vulnerability has been proposed. This information will provide the analyst with a base estimation needed as input to the cost-benefit estimation that is at the heart of the risk management process.

The results of the chapter represent a first step toward the design and implementation of a fully automated context-aware vulnerabilities analysis and classification platform that can be built by using all the considered repositories as a unique reference knowledge base.

Chapter 4

Enhancing the Quality of Automatically Generated Inventories

Building and maintaining inventories is a fundamental component of many security processes [24]. Among all inventories used to feed security processes, this chapter will focus on the device inventory and the vulnerability inventory. These two inventories are particularly relevant data sources that derive from a monitored environment and are used not only to feed risk management [38, 62] but also other critical processes such as Incident Management [16] and their supporting systems [85]. Since these sources constitute the primary data source for the analysis of complex situations, some of these potentially involving multiple correlation steps, their quality, in terms of completeness and accuracy, must be ensured to avoid that a propagation of false positives and false negatives through the analysis steps cascades into an extreme overestimation (or underestimation) of the real security situation.

Device and vulnerability inventories are typically built from the output of network and vulnerability scanning tools. Such scanning tools use a network-based or agent-based scanning mechanism in order to detect vulnerabilities. Network-based scanners typically rely on probing mechanisms over the network, observe hosts/devices' answers and match them to fingerprints that may not be unique across different configurations and product versions [88, 89]. While they can generally be considered non-intrusive, they allow for the possibility of introducing false positives when it comes to detecting and identifying existing configurations and associated vulnerabilities. Agent-based techniques, on the other hand, employ active agents that are installed locally on monitored hosts/devices and may be able to interact with the operating system to gain a complete understanding of the system platforms and configuration [53]. Therefore, they are still prone to false positives but tend to be more accurate and intrusive than their network-based counterpart. The availability of Agent-based techniques however is heavily dependant on the availability of an active agent that is targeted at the specific architecture that needs to be monitored. It is possible that particular environments (Industrial Control Systems or ICS for short being an example) may feature devices and architectures that are not supported by any active agent, thus restricting the possible choice of scanner only to network-based scanners.

Figure 4.1 highlights another important issue of commercially available vulnerability scanners, the fact that regardless of being agent-based or network-based, scanners use advisories and not CVE to report vulnerabilities in an ICT system.

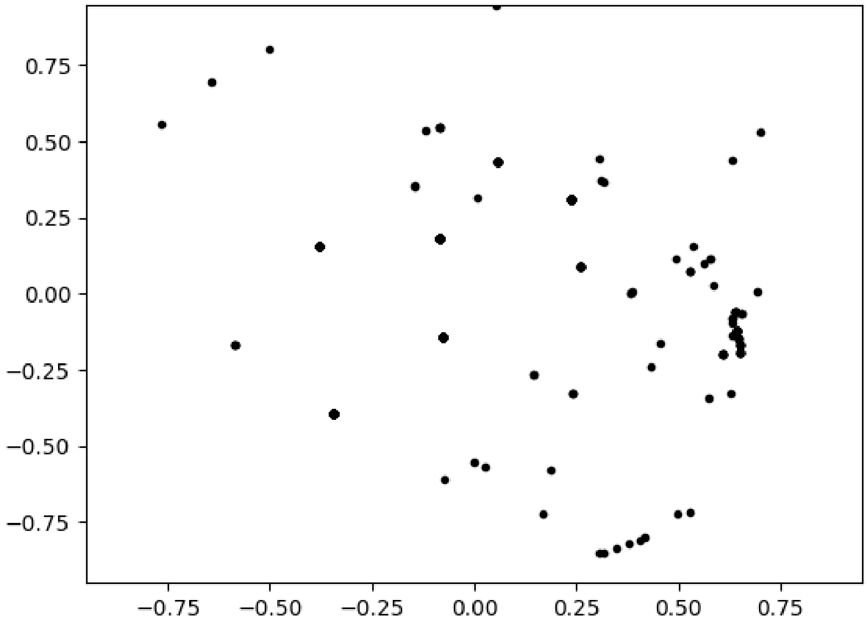


Figure 4.1. Multidimensional scaling (MDS) plot of CVE vulnerabilities aggregated by advisory *GLSA-201301-01*, distributed according to the similarity between their vulnerable vendors and products. Elements close together usually represent vulnerabilities of different versions of the same product, elements distanced from each other represent vulnerabilities of different products and vendors.

Inspecting the advisory pool of major commercially available vulnerability scanners, yields two important characteristics of advisories: (i) Advisories are aimed towards human consumption, and as such are predominantly populated by natural text, and (ii) Advisories are an aggregation of CVE vulnerabilities based on either vendor locality (e.g. *GLSA-201301-01*[1] “Mozilla Products: Multiple vulnerabilities” as represented in the figure) or based on a single vulnerability affecting multiple products. Vulnerabilities in CVE format are usually listed as references to each advisory, with the possibility of semantics being hidden in the paragraphs of natural text. As an example, *GLSA-201301-01* lists 499 CVE vulnerabilities pertaining to mozilla products. Among these, *CVE-2011-3101*[2] is listed as one of the references of this advisory, but analyzing its platform configuration yields that this vulnerability only affects google chrome on the linux operating system, both of which are not mozilla products. Even within the rest of the referenced vulnerabilities, it may happen that a host does not have all the platforms which are associated to each referenced vulnerability, namely: Mozilla Firefox, Thunderbird, SeaMonkey, NSS, GNU IceCat, and XULRunner. Thus, if left unattended, this characteristic of vulnerability scanners is another source of false positives and consequent loss of accuracy for subsequent inventories and processes that rely on these inventories.

The main focus of this chapter will be the problem of improving the quality (in

[1] https://security.gentoo.org/glsa/201301-01

[2] https://nvd.nist.gov/vuln/detail/CVE-2011-3101

terms of accuracy) of the automatically generated inventories without increasing the degree of intrusiveness in the monitored system. To achieve this aim, human knowledge will be leveraged, as it will be assumed that an analyst or a system administrator is able to validate a vulnerability by confirming or discarding the presence of the platforms that it affects. Since humans will be involved in the process, another requirement for the solution of this problem will also be to make it as efficient as possible.

This chapter provides the following contributions:

- A formalization of the problem of the quality enhancement of automatically generated inventories.
- A computational pipeline that includes several proposed algorithms for the quality enhancement of automatically generated inventories, as well as several strategies able to operate on the accuracy-scalability trade-off of attack graph-based methodologies for risk analysis. In particular the following will be proposed:
    - A methodology and an unoptimized algorithm for platform filtering.
    - An optimization for platform filtering targeted at platform versions.
    - An advancement for platform filtering that leverages the dependency between CPE strings.
    - An alternative, optimized algorithm for platform filtering that is aware of the structure of CPE strings.
    - Several analysis-aware aggregation strategies to trade accuracy for scalability in attack graph-based methodologies for risk analysis.

4.1 Formalization of the Problem

The first contribution of this chapter is a formalization of the problem of the quality enhancement of automatically generated inventories. To this aim, it is necessary to lay the groundwork, disambiguating and presenting notions which will be necessary to formalize the problem.

Problem Setting

For the purpose of this chapter, the ICT network (*monitored environment*) of a given organization is considered. The considered network is monitored by one (or multiple) *monitor(s)*. Since the analysis tasks proposed in this chapter focus on monitoring hosts/devices belonging to the network and their vulnerabilities, the only inventories that have been considered for this analysis are (i) the *Device Inventory* and (ii) the *Vulnerability Inventory*. Thus, this has lead to the decision to abstract the underlying monitored environment into only these two inventories.[3]

The *Device Inventory* $\mathcal{DI}$ is simplified into the set of pairs $\langle h_i, p_j \rangle$ where h_i represents the host/device identifier and p_j represents a specific platform (hardware, operating system or software) deployed on the device. Every host's platform p_j will

[3] Normally, during complex security processes, an environment is described by a multitude of inventories, not only by these two.

be expressed through a *Common Platform Enumeration* (CPE) string.[4] Thus, an entry in $\mathcal{DI}$ will be a pair $\langle h_i, cpe_j \rangle$.

The *Vulnerability Inventory* $\mathcal{VI}$ is simplified into the set of pairs $\langle h_i, vul_j \rangle$ where h_i represents the host/device identifier and vul_j represents a vulnerability affecting the host. In this chapter, vulnerabilities will be expressed according to the *Common Vulnerabilities and Exposures* (CVE)[5] format and a CVE identifier[6] will be used to univocally identify a vulnerability in the inventory.

Let us note that it is always possible to retrieve all the details related to a vulnerability starting from its CVE identifier, by querying publicly available external repositories such as NVD,[7] as shown in Chapter 3.

In the following sections, it will be assumed that a vulnerability scanner can interact freely with hosts/devices belonging to the monitored environment (i.e. the network connectivity can be represented by a fully connected graph and no firewalls are present in the monitored system). This is necessary to extract the required relevant information that will be used to build $\mathcal{DI}$ and $\mathcal{VI}$. In the following sections, only *non-intrusive* vulnerability scanners will be considered i.e., vulnerability scanners that are deployed in the monitored environment, that can interact with hosts/devices in the network though the use of probing mechanisms but that cannot observe the host internally. Due to their reliance on probing mechanisms, vulnerability scanners are only able to report a *raw* $\mathcal{DI}$ and $\mathcal{VI}$ i.e., a $\mathcal{DI}$ and a $\mathcal{VI}$ that are potentially affected by both false positives (pairs reported in the inventory but not really characterizing the host) and false negatives (pairs not reported in the inventory but characterizing the host). Throughout the chapter, such inventories will be denoted as as $\mathcal{DI}_{raw}$ and $\mathcal{VI}_{raw}$. Examples of tools that can be used as monitors are Nmap[8] to construct the raw device inventory $\mathcal{DI}_{raw}$ and Tenable Nessus[9] or Greenbone OpenVAS[10] to construct the raw vulnerability inventory $\mathcal{VI}_{raw}$.

Let us note that typically a vulnerability is placed in the raw vulnerability inventory as a consequence of the detection performed by one (or multiple) vulnerability scanner(s) in response to one or multiple probes. Indeed, every vulnerability scanner interacts with every host through a probing mechanism, and depending on the result of the probe, the monitor identifies possible vulnerabilities existing on the host. One or multiple probes are connected to *advisories*. Advisories are usually a collection of zero, one or many vulnerabilities which are linked to the successful result of the probe. In the occasions where the information about the advisory is available, the identifier of the advisory adv_k will also be included in the vulnerability inventory (and in particular in the vulnerability definition), to abstract the probing detection mechanism (i.e., $vul_j = \langle CVE_k, adv_k \rangle$ where CVE_k is a vulnerability identifier and adv_k is the identifier of the advisory associated with CVE_k).

Let us recall that inventories may be affected by false positives and false negatives. Thus, to measure the *quality* of an inventory $\mathcal{I}$ (being it a device inventory or a

[4] According to the NIST documentation on CPE version 2.3 [78], a well-formed CPE string is obtained by the concatenation of sub-strings (one for every defined CPE attribute) where three attributes are mandatory i.e., *part*, *vendor* and *product*. When an optional attribute is not specified, it is replaced in the CPE string with $*$. See Section 3.1 of Chapter 3.

[5] `https://www.cve.org/`

[6] A CVE identifier follows a simple syntax: `CVE-`*`YEAR-5-digits-number`*, where *`YEAR`* identifies when the identifier has been *created* and *`5-digits-number`* is a sequence number in the year. See Section 3.1 of Chapter 3.

[7] `https://nvd.nist.gov/`

[8] Nmap (“Network Mapper”) is an open-source tool for network exploration and security auditing, cfr. `https://nmap.org`

[9] `https://www.tenable.com/products/nessus/nessus-essentials`

[10] `https://www.openvas.org`

vulnerability inventory) two metrics have been introduced:

- The *completeness* of the inventory, denoted as $C(\mathcal{I})$, measures the degree of coverage of the information reported in the inventory with respect to the real host/device configuration. The completeness is computed as

$$C(\mathcal{I}) = 1 - \frac{\#false\ negatives}{\#real\ entity\ in\ the\ inventory};$$

- The *accuracy* of the inventory, denoted as $A(\mathcal{I})$, measures the degree of noise reported in the inventory with respect to the real host/device configuration. The accuracy is computed as

$$A(\mathcal{I}) = 1 - \frac{\#false\ positives}{\#real\ entity\ in\ the\ inventory}.$$

Problem Statement

It is of interest to improve the quality of the raw inventories collected from the monitored environment. In particular, starting from $\mathcal{DI}_{raw}$ and $\mathcal{VI}_{raw}$ produced directly by monitors, the goal is to compute new versions $\mathcal{DI}^{+}$ and $\mathcal{VI}^{+}$ of both the inventories with increased quality, without being intrusive in the monitored system i.e., without adding additional active monitors in the monitored environment. More formally, the proposed goal is to produce the inventories $\mathcal{DI}^{+}$ and $\mathcal{VI}^{+}$ such that:

- The quality of the produced device inventory $\mathcal{DI}^{+}$ is as good as the quality of the raw device inventory $\mathcal{DI}_{raw}$ (i.e, $C(\mathcal{DI}^{+}) \geq C(\mathcal{DI}_{raw})$ and $A(\mathcal{DI}^{+}) \geq A(\mathcal{DI}_{raw})$) and
- The quality of the produced vulnerability inventory $\mathcal{VI}^{+}$ is greater than (or equal to, in the absence of *false positives* in the raw vulnerability inventory $\mathcal{DI}_{raw}$) the quality of the raw vulnerability inventory $\mathcal{DI}_{raw}$ (i.e., $C(\mathcal{VI}^{+}) \geq C(\mathcal{VI}_{raw})$ and $A(\mathcal{VI}^{+}) > A(\mathcal{VI}_{raw})$ if $A(\mathcal{VI}_{raw}) < 1$, $A(\mathcal{VI}^{+}) = A(\mathcal{VI}_{raw}) = 1$ otherwise).

4.2 Computational Pipeline

Inventory quality plays a crucial role within the context of self-protecting systems, where the quality of the inventories which are used as input for the system directly influences the quality of the risk-based adaptation loop. Thus, this section introduces a computational pipeline that is able to achieve an improvement in the accuracy and scalability of the self-protecting system by improving the quality of the raw inventories collected from the monitored environment.

Considering the architecture of the self-protecting system shown in Figure 4.2, the proposed computational pipeline should be modular and should be integrated in the *Data Collection, Aggregation and Integration* component with the aim of: (i) enhancing the accuracy of risk estimation and (ii) improving its overall scalability. The proposed pipeline will add the following two main components to the already existing Data Collection, Aggregation and Integration component: (i) a *Vulnerability Filtering* block and (ii) a *Vulnerability Aggregation* block. According to the desired level of accuracy of the output, these new proposed components can be tuned and

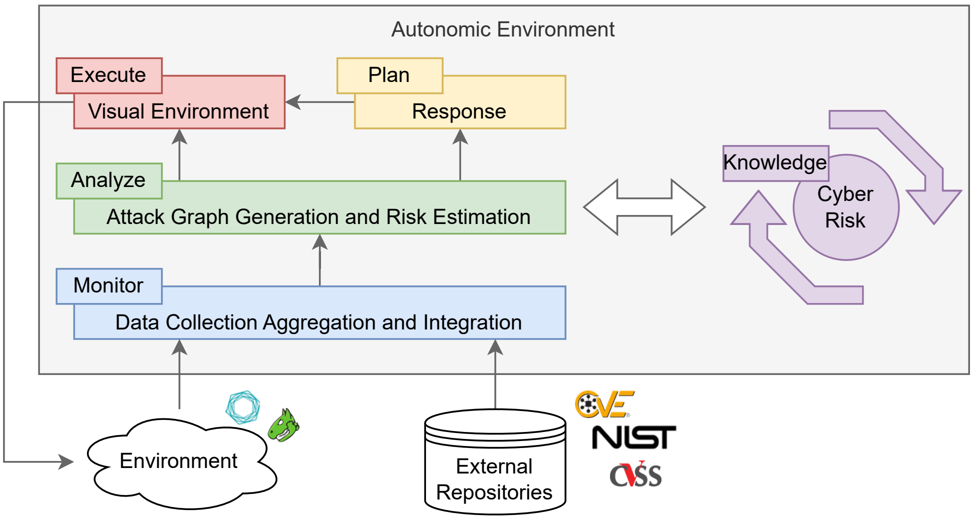


Figure 4.2. MAPE-K architecture for attack graph-based self-protecting systems.

orchestrated together with the old components of Data Collection, Aggregation and Integration to generate different flows of computation. Different flows of computation are characterized by the participation of varying new components that will affect the level of accuracy of the output. An overview of the proposed pipeline pipeline is available at Figure 4.3.

The main task of the Data Collection, Aggregation and Integration component is to gather, sanitize and format all the information needed in the subsequent steps of the MAPE-K control loop to compute the attack graph, estimate and analyze the risk in the monitored system, and finally plan mitigation actions to manage the risk. Among all the information handled by the Data Collection, Aggregation and Integration component, this section focuses on the *Vulnerability Inventory* $\mathcal{VI}$ used to generate the Attack Graph and estimate the likelihood of each vulnerability in the monitored system. In the following, the information extracted "as it is", without any additional filtering or aggregation component that works in conjunction with the Data Collection, Aggregation and Integration component will be referred as *Raw Data Collection*. The term "raw" refers to the existing widespread implementation of the data collection components that, to the best of the author's knowledge, mainly focuses on merging and integrating information coming from multiple external data sources [70, 77, 71]. This corresponds to workflow 0 in Figure 4.3 that will represent the baseline for any further analysis (i.e., the current state of the art for attack graph-based self-protection systems). Note that workflow 0 can still be used in isolation, as a fully functional implementation of the *Data Collection, Aggregation and Integration* component bypassing the other proposed pipelines.

Vulnerability Filtering Component

At the basis of the *Vulnerability Filtering* component is the notion that state of the art, probe-based vulnerability scanners[11] are usually the main source of information employed by the *Raw Data Collection* process in order to produce inventories. Thus due to their probe-based nature, it is possible for these vulnerability scanners to

[11] A vulnerability scanner is a tool that automatically identifies vulnerabilities in a network by scanning target systems and looking for known vulnerabilities and misconfigurations [64]. See Section 4.1.

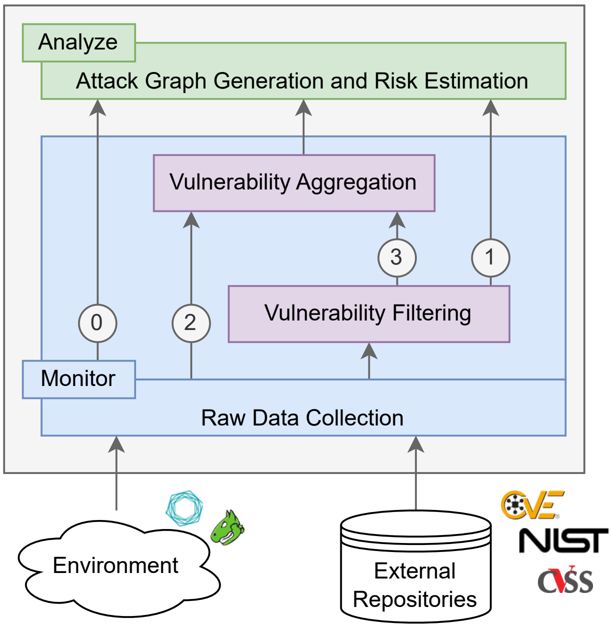


Figure 4.3. High-Level architecture of the proposed computational pipeline.

produce inaccuracies, such as *false positives*, i.e., vulnerabilities that are not affecting the system or not exploitable in the considered context. This is due to, for example: (i) misconfiguration, (ii) misinterpretation of the standard system behavior, (iii) dynamic content of systems' applications, (iv) temporary conditions on the network, and (v) lack of context [13].

To attempt to mitigate this possibility, the *Vulnerability Filtering* sub-component has been introduced in the computational pipeline in order to reduce the number of false positives deriving from the Raw data collection process i.e., vulnerabilities that are listed in the vulnerability inventory generated by the Raw Data collection algorithms.

Vulnerability Aggregation Component

False positives in the vulnerability inventory have a double disadvantage. On one side, they reduce the accuracy of the risk estimation as it is performed on a set of vulnerabilities that do not exist. On the other hand, they increase the size of the vulnerability inventory, thus negatively impacting the scalability of the subsequent attack graph-based analysis.

In order to attack the scalability issues related to attack graph based analysis, the introduction of a *Vulnerability Aggregation* sub-component has been proposed. This sub-component aims to further improve the scalability of the overall self-protecting system by reducing the size of the input processed by the attack graph generation and risk estimation module and, in particular, the vulnerabilities in the inventory $\mathcal{VI}$. This is done by aggregating vulnerabilities according to analysis-aware semantic criteria. To reach this goal, the core idea is to aggregate a set of vulnerabilities with common features (e.g., that affect the same host and have the same set of pre and post-conditions) into a *meta-vulnerability* characterized by such common features.

Let us note that multiple aggregation policies may exist depending on the feature considered. Ideally, it would be optimal to maximize the number of vulnerabilities aggregated into a meta-vulnerability (to maximize the scalability), while minimizing the potential accuracy loss.

Computational Flows

The Vulnerability Filtering and the Vulnerability Aggregation components can be orchestrated to support 3 different computational flows as shown in Figure 4.3. In particular:

CF1 involves only the vulnerability filtering (flow 1 in Figure 4.3). This flow theoretically guarantees the best accuracy, and can support a proactive security analysis with unlimited resources for attack graph computation algorithms (from both the computational and the temporal point of view).

CF2 involves only the vulnerability aggregation (flow 2 in Figure 4.3). This flow trades accuracy for scalability and is more suitable to support better real-time analysis where the computation time of attack graphs must be reduced as much as possible and the analysis is fully automated without potential delays introduced by vulnerability filtering. This flow is also particularly suitable when the accuracy of the inventories generated by the raw data collection is particularly high.

CF3 involves both the vulnerability filtering and the vulnerability aggregation components (flow 3 in Figure 4.3). This flow covers the most general case that depends on the specific context and is based on the definition of a trade-off between the accuracy and performance of the risk analysis.

4.3 An Algorithm for Vulnerability Filtering

This section will describe an initial solution aimed at increasing the quality of the automatically generated inventories starting from $\mathcal{DI}_{raw}$ and $\mathcal{VI}_{raw}$ provided by vulnerability and network scanners.[12] The presented solution could be implemented in the *Vulnerability Filtering* sub-component proposed in Section 4.2, in order to realize computational flows *CF1* and *CF3*.

The proposed solution is presented in two parts: (i) the first part of the proposed solution pivots on the assumption of having an accurate and complete device inventory $\mathcal{DI}_i$ in order to employ the CPE strings contained within to *validate* vulnerabilities in the vulnerability inventory $\mathcal{VI}_{raw}i$ with the effect of reducing the number of false positives; (ii) the second part of the proposed solution is a user-aided approach to increase the quality of a raw device inventory $\mathcal{DI}_{raw}$, in order to attempt to satisfy the requirement of having an accurate and complete device inventory $\mathcal{DI}_i$ needed for the first part of the solution.

More in detail, the second part of the proposed solution revolves around submitting a series of questions to the analyst or system administrator which leverage his/her knowledge of a given host's configuration[13] in order to resolve symmetric situations in the vulnerability inventory and, as a consequence, reduce the number of false positives in the inventory.

Part 1: Validating a raw Vulnerability Inventory $\mathcal{VI}_{raw}$ starting from an accurate and complete Device Inventory $\mathcal{DI}^*$

Algorithm 3 describes the first part of the proposed solution. It proceeds by

[12] Let us note that given vulnerability and network scanners like Nmap and Tenable Nessus or Greenbone Open Vas, constructing $\mathcal{DI}_{raw}$ and $\mathcal{VI}_{raw}$ is a simple parsing operation of the produced reports.

[13] Intended as which applications, operating system and hardware components are on a given host.

analyzing hosts one after the other and validating all the vulnerabilities present in the vulnerability inventory $\mathcal{VI}_{raw}$ before hopping to a new host (lines 2-4 and 6 in Algorithm 3). By exploiting the fact that the information that links CPE configurations to any vulnerability in CVE format is listed in NVD, starting from the raw vulnerability inventory $\mathcal{VI}_{raw}$ as defined in the previous section, it is possible to retrieve from NVD the *contextual CPEs* as well as their configurations (line 7 in Algorithm 3 i.e., CPEs and configurations that represent one or more enabling conditions for the vulnerability to exist on the host, see Section 3.1). *Contextual CPEs* can then be matched with those reported in $\mathcal{DI}^*$ (line 8 in Algorithm 3). If a match exists and atleast one configuration is satisfied, the CVE is considered as "validated" and inserted in the vulnerability inventory $\mathcal{VI}^+$ which will be then the output of the algorithm (lines 9-10 in Algorithm 3) otherwise the vulnerability is safely discarded, since no enabling factor exists within the considered host.

Algorithm 3: Part 1 of the proposed solution

```
   Input: A complete and accurate device inventory $\mathcal{DI}^*$ (expressed as a set of pairs
          $\langle h_i, CPE_j \rangle$)
   Input: A raw Vulnerability Inventory $\mathcal{VI}_{raw}$
   Output: A Vulnerability Inventory $\mathcal{VI}^+$
1  $\mathcal{VI}^+ \leftarrow \emptyset$;
2  $host\_list \leftarrow$ get_host_list($\mathcal{DI}^*$);
3  foreach $h_i \in host\_list$ do
4      $vuln\_list_i \leftarrow$ get_vulnerability_list($h_i, \mathcal{VI}_{raw}$);
5      $existing\_CPE_i \leftarrow$ get_config_from_DI($h_i, \mathcal{DI}^+$);
6      foreach $v_j, adv_j \in vuln\_list_i$ do
7          $cand\_CPE_j \leftarrow$ get_CPE_from_NVD($v_j$);
8          $valid_j \leftarrow$ match($cand\_CPE_j, existing\_CPE_i$);
9          if $valid_j = true$ then
10             $\mathcal{VI}^+ \leftarrow \mathcal{VI}^+ \cup \{\langle h_i, v_j, adv_j \rangle\}$;
11         end
12     end
13 end
14 return $\mathcal{VI}^+$;
```

The match() function in line 8, Algorithm 3, is implemented by relying on the presence of one or more configuration trees inside NVD. Indeed, given a specific vulnerability v_j in NVD, the associated CPE strings are not flat but they are rather organized in a set of alternative configurations (i.e., logically related through a OR operator). Each configuration then is represented by a AND – OR tree called *configuration tree*. A configuration tree defines which combinations of software, hardware, and operating system are necessary for the vulnerability to exist. Detailed information about NVD and CPE strings and configuration trees can be found in Chapter 3.

The main principle which has been leveraged in order to drive the validation strategy is that given a configuration tree associated with a particular CVE, in order to validate the CVE it is sufficient that a CPE in an OR condition is validated. Conversely, if a CPE in an AND condition is invalidated (i.e., flagged as non-existing) the whole AND condition is to be considered invalid with respect to the whole subtree. More in detail, line 5, Algorithm 3 retrieves the configuration trees assigned to a vulnerability and represents them as logical formulas defined by the AND – OR structure of the configuration tree having one boolean variable for each CPE string (leaf node).[14] The match() function takes such formulas, sets to true all the variables

[14] Any formula in first order logic can always be represented as an AND – OR tree and vice-versa.

corresponding to CPEs that exist in the considered device inventory $\mathcal{DI}^*$ and to false those without a match, evaluates all the logical formulas and finally returns the resulting boolean value obtained by concatenating all the formulas using an OR operator.[15]

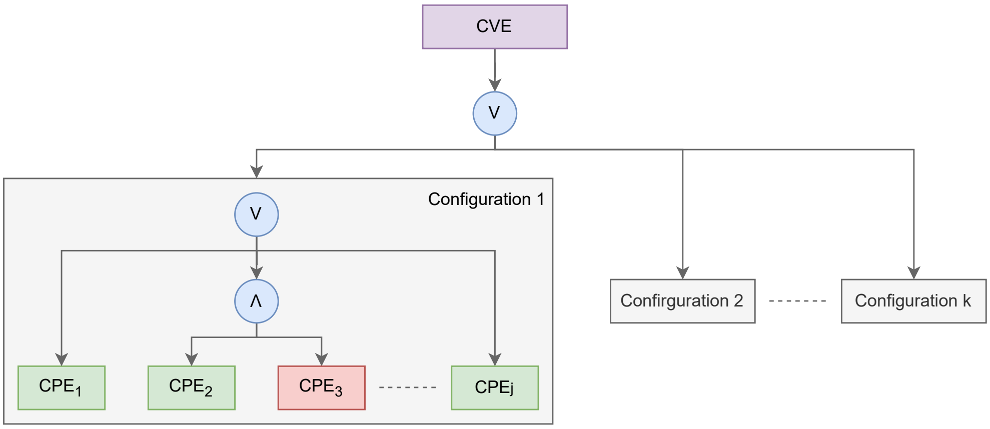


Figure 4.4. Structure of CPE configurations stored in NVD. Red and green denote vulnerable / non-vulnerable status.

As an example, let us consider the CVE reported in Figure 4.4 (Figure 3.1 of Chapter 3). To simplify the example let us assume that configuration trees from 2 to k contain a single CPE (where CPE_1^j is the first and only CPE in the configuration j). During the execution Algorithm 3, the following formula is produced

$$(CPE_1^1 \vee (CPE_2^1 \wedge CPE_3^1) \vee \cdots \vee CPE_i^1) \bigvee_{j=2}^{k} CPE_1^j$$

where every CPE_k^j is a boolean variable. The formula is evaluated as true if the specific configuration of platforms described by the formula is compatible with the considered device inventory $\mathcal{DI}^*$, i.e. if the CPE_k^j present in $\mathcal{DI}^*$ for host h_1 are sufficient to evaluate the formula as true without any doubt. Evaluating this formula as true entails the validation of the presence of the considered vulnerability on the host h_1. Conversely, evaluating this formula as false entails the validation of the absence of the considered vulnerability on the host h_1, i.e. the considered vulnerability is a false positive and should not appear in $\mathcal{VI}^+$.

Part 2: Reconstructing the minimum-sized Device Inventory to improve the accuracy of the Raw Vulnerability Inventory

The accuracy of the vulnerability inventory $\mathcal{VI}^+$ produced by Algorithm 3 is highly dependant on the accuracy of the device inventory used as input. As a consequence, if the device inventory is not accurate, the resulting vulnerability inventory $\mathcal{VI}^+$ will not be accurate as well. Furthermore, given the setting of the problem, it is not possible to further improve the quality of the vulnerability inventory without the introduction of a potentially intrusive solution, such as a local agent, in order to obtain an additional source of information. Since part of the scope of the problem is to remain as non-intrusive as possible, it is necessary to involve an analyst or a system administrator by leveraging his/her knowledge of the platform configuration of the

[15] CPE configuration trees in NVD are linked one another through a common OR parent node.

monitored environment in order to solve symmetric situations in the vulnerability inventory and, as a consequence, attempt to reduce the number of false positives in the inventory. The rationale behind this solution is that an analyst or a system administrator easily knows (or has the means to know) if a platform is really associated with a particular host. Therefore, for each host, he/she can easily validate entries included in the device inventory (thus increasing its accuracy). However, it could be possible that a device inventory may be comprised of a large number of entries, which would translate directly to a large number of questions to be subjected to the human agent. For this reason, it is necessary to submit the questions to the human agent according to a utility function that prioritizes conditions of uncertainty that allow for the validation/discarding of more vulnerabilities at the same time.

Since the questions submitted to the human agent revolve around single platforms, the utility function which is part of the proposed prioritization mechanism must leverage the configuration tree of every vulnerability (similarly to what is done in Algorithm 3) in order to rank CPE strings based on two complementary criteria: (i) the frequency and type of occurrences in the configuration trees and (ii) their recurrence in the advisories.

The rationale supporting the first criteria is that if a CPE string CPE_j appears in many vulnerabilities linked by OR operators, and if CPE_j is present in the considered host, all these vulnerabilities in which CPE_j appears will be validated through a single question submitted to the human agent. Conversely, if CPE_j is involved in several AND conditions of several vulnerabilities, and CPE_j is not present in the considered host, then it will prune entire configuration sub-trees, potentially cutting the number of elements in the Device Inventory which are still relevant for the validation process.[16] This means that the utility function must account for positive as well as negative answers from the human agent while still striving to maximize the impact of the answer on the structure of the CPE configuration trees of the vulnerabilities in the Vulnerability Inventory. In other terms, the utility function must aim to maximize its effectiveness despite not knowing apriori which answer will the human agent choose.

Fortunately, by cleverly exploiting the semantics of the tools used to obtain the information that is used to populate the inventories, there is a way to produce an estimation of the expected answer for each platform in a system. Indeed, vulnerability and network scanning tools function according to a probe-based mechanism. Intuitively, if a platform is referenced by many advisories, it is reasonable to think that the probability of that platform residing in the host is greater than the probability of that platform not belonging to the considered host. This intuition is supported by the fact that an advisory is assigned to a host only if one or more probes linked to an advisory detects the presence of possibly vulnerable platforms through certain fingerprints. As such, it is reasonable to believe that the probability of a platform being really present in a host rises if multiple probes detect it. This bias in probability can be used in conjunction with the first criteria to empower the utility function with an oracle that tries to provide a reasonable guess of the expected answer that the human agent will provide.

Lastly, each time that the human agent provides an answer, it is necessary to re-evaluate and re-prioritize the platforms in the Device Inventory that still need to be validated before asking an additional question. This is necessary due to the fact that

[16] A platform in the Device Inventory is considered relevant if it partakes in a meaningful way to any formula derived from any configuration of not-yet confirmed/discarded vulnerabilities in the Vulnerability Inventory. In simpler terms, "if a platform helps me solve a configuration which is still unsolved, then it is useful".

each platform has its own impact on the configuration trees of the vulnerabilities in the Vulnerability Inventory. For this reason it is not uncommon for many platforms to become irrelevant with respect to the task of validating vulnerabilities, through a single question.

Algorithm 4 presents the pseudo-code of the proposed solution. Similarly to the previous part, each host is analyzed independently and sequentially. For each host h_i the solution initially computes a structure defined as the *vulnerability graph* $\mathcal{G}_i = (V_i, E_i)$ of host h_i (line 5, Algorithm 4).

The set of nodes V_i that are part of the vulnerability graph $\mathcal{G}_i$ is partitioned into three sub-sets: (i) *advisory nodes* V_{adv} which represent every advisory id listed in the raw vulnerability inventory $\mathcal{VI}_{raw}$, (ii) *vulnerability nodes* V_{CVE} which represent every CVE listed in the raw vulnerability inventory $\mathcal{VI}_{raw}$ and (iii) *platform nodes* V_{CPE} which represent every CPE string retrieved starting from a CVE in the raw vulnerability inventory $\mathcal{VI}_{raw}$. Thus, $V_i = V_{adv} \cup V_{CVE} \cup V_{CPE}$.

The set of edges E_i that are part of the vulnerability graph $\mathcal{G}_i$ is also partitioned into two sub-sets: (i) $E_{(CVE,CPE)}$ represents the set of edges linking a vulnerability with its associated CPEs and (ii) $E_{adv,CVE}$ represents the set of edges linking an advisory to all its associated vulnerabilities. Thus, $E_i = E_{(CVE,CPE)} \cup E_{adv,CVE}$.

An example of a vulnerability graph for a host h_i is shown in Figure 4.5.

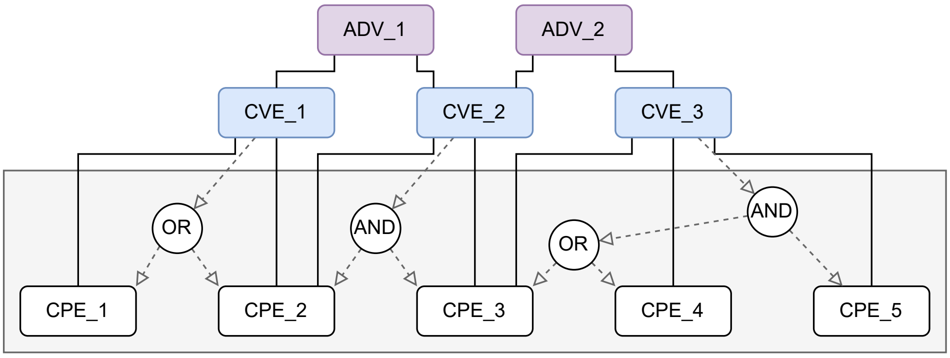


Figure 4.5. Schema of an exemplary $\mathcal{G}_i$.

The vulnerability graph G_i is initially computed starting from the raw vulnerability inventory and is refined and used throughout the validation process. The proposed algorithm then proceeds to assign scores to every edge in $E_{(CVE,CPE)}$ to identify the "most relevant" CPE strings with respect to a specific vulnerability. This is achieved considering the configuration tree linking a CVE with its associated CPE strings in isolation These scores are only local with respect to each vulnerability, so an aggregation step is required to have a score that can be used on each host. For this reason, these "local" scores are then aggregated across multiple vulnerabilities in order to achieve a single comprehensive score to each and every CPE string in a host, that is able to take into account the contribution of a platform across multiple vulnerabilities (lines 12 and 15, Algorithm 4). Using the calculated score as a prioritization metric, a question is selected and raised to the human agent. In particular, the proposed algorithm starts from the CPE string with the highest score and proposes it to the user expecting a boolean answer in return (lines 17 and 18, Algorithm 4).

If the existence of the proposed CPE string is confirmed by the human agent, it is inserted in the device inventory $\mathcal{DI}^+$ and a comprehensive evaluation is triggered in order to understand if this answer contributed to the validation of one or more vulnerabilities. This is achieved using the same technique presented in Algorithm 3, by evaluating the logical formulas associated with configuration trees for every connected vulnerability (lines 23-35, Algorithm 4). Differently from the previous part, not all CPE strings may have been assigned a value yet. For this reason, formulas that derive from configuration trees of vulnerabilities may still be undetermined. This allows three outcomes to derive from the matching function: (i) the matching function returns with a true value, meaning that the vulnerability is validated through already validated CPE strings, and as a result it is inserted in the vulnerability inventory $\mathcal{VI}^+$, (ii) the matching function returns with a false value, meaning that the vulnerability is invalidated by already discarded CPE strings, and as a result it is discarded and not included in $\mathcal{VI}^+$ and (iii) the formula cannot be evaluated yet, meaning that with the current validated and discarded CPE strings, it is not yet possible to state if the vulnerability exists or not, and that there is the need to continue the validation process.

As a reminder, each time a CPE string or a CVE vulnerability is validated or discarded, the vulnerability graph $\mathcal{G}_i$ is updated and thus, in the next iteration of the algorithm, scores and prioritizations will be updated considering the insight gained by the interaction with the human agent.

Computing the Prioritization Scores

The previous section introduced Algorithm 4 which aims to enhance the quality of an automatically generated Vulnerability Inventory $\mathcal{VI}_{raw}$ through the strategic employment of a human agent. More in detail, it has been shown that by validating or discarding platforms contained in the Device Inventory $\mathcal{DI}_{raw}$ it is possible as a consequence to validate or discard vulnerabilities from a Vulnerability Inventory $\mathcal{VI}_{raw}$ in order to refine it into a possibly more accurate $\mathcal{VI}^+$, which should contain less false positives than $\mathcal{VI}_{raw}$.

This section will go on detail on the prioritization score used to order the sequence of platforms to be validated by the human agent. More in particular, this section will show how given an instance of the vulnerability graph $\mathcal{G}_i$ for a given host h_i, is is possible to compute the scores associated with edges in $E_{(CVE,CPE)}$ (i.e., the assign_score() function) and how to aggregate the scores (i.e., the aggregate_score() function) in order to have a single prioritization score to rank platforms on a host.

For a given host h_i the assign_score() function takes as input the vulnerability graph $\mathcal{G}_i$ and computes for each edge $e_{v,p} \in E_{(CVE,CPE)}$ a numerical score representing the *utility* $W_{v,p}$[17] of validating or discarding CPE p. The concept of *utility* is highly tied to the potentiality of CPE p to validate or discard the existence of the vulnerability v in h_i, and is formalized as:

$$W_{v,p} = \alpha_p \times TS_{v,p} + (1 - \alpha_p) \times FS_{v,p}$$

where

- $TS_{v,p}$ represents the utility of a CPE p with respect to its potential to validate the CVE v when a positive answer from the human agent is expected (e.g., when a positive answer will allow to fully validate the vulnerability without any additional question)

[17] Abusing the notation it is possible to refer to $W_{v,p}$ as the entry of the $W_{(CVE,CPE)}[]$ array used in Algorithm 4.

Algorithm 4: Vulnerability Filtering

```
Input: A raw Device Inventory $\mathcal{DI}_{raw}$
Input: A raw Vulnerability Inventory $\mathcal{VI}_{raw}$
Output: A Device Inventory $\mathcal{DI}^{+}$
Output: A Vulnerability Inventory $\mathcal{VI}^{+}$
$\mathcal{VI}^{+} \leftarrow \emptyset$;
$\mathcal{DI}^{+} \leftarrow \emptyset$;
$host\_list \leftarrow$ get_host_list($\mathcal{VI}_{raw}$);
foreach $h_i \in host\_list$ do
    $\mathcal{G}_i(V_i, E_i) \leftarrow$ compute_vulnerability_graph($\mathcal{VI}_{raw}$);
    $W_{(CVE,CPE)}[\,] \leftarrow [0]^{|E_{(CVE,CPE)}|}$;
    $W_{CPE}[\,] \leftarrow [0]^{|V_{CPE}|}$;
    $validated\_CPE[\,] \leftarrow [\perp]^{|V_{CPE}|}$;
    $CVE\_to\_validate \leftarrow V_{CVE_i}$;
    while $CVE\_to\_validate \neq \emptyset$ do
        foreach $e_x \in E_{(CVE,CPE)}$ do
            $W_{(CVE,CPE)}[e_x] \leftarrow$ assign_score($\mathcal{G}_i$);
        end
        foreach $cpe_x \in V_{CPE}$ do
            $W_{CPE}[cpe_x] \leftarrow$ aggregate_score($W_{(CVE,CPE)}[\,]$);
        end
        $CPE_{max} \leftarrow$ get_highest($W_{CPE}[\,]$);
        $validated\_CPE[CPE_{max}] \leftarrow$ ask_for_user_validation($CPE_{max}$) ;
        if $validated\_CPE[CPE_{max}] =$ true then
            $\mathcal{DI}^{+} \leftarrow \mathcal{DI}^{+} \cup \{\langle h_i, CPE_{max} \rangle\}$;
            $\mathcal{G}_i \leftarrow$ remove_validated($\{CPE\_max\}$);
        end
        foreach $v_j \mid \exists \langle v_j, CPE_{max} \rangle \in E_{(CVE,CPE)}$ do
            $cond\_CPE_j \leftarrow$ get_CPE_from_NVD($v_j$);
            $validated_j \leftarrow$ match($cond\_CPE_j, validated\_CPE$);
            if $validated_j = true$ then
                $\mathcal{VI}^{+} \leftarrow \mathcal{VI}^{+} \cup \{\langle h_i, v_j, \perp \rangle\}$;
                $CVE\_to\_validate \leftarrow CVE\_to\_validate \setminus \{v_j\}$;
                $\mathcal{G}_i \leftarrow$ remove_validated($\{v_j\}$);
            end
            else if $validated_j =$ false then
                $CVE\_to\_validate \leftarrow CVE\_to\_validate \setminus \{v_j\}$;
                $\mathcal{G}_i \leftarrow$ remove_validated($\{v_j\}$);
            end
        end
    end
end
return $\mathcal{VI}^{+}, \mathcal{DI}^{+}$
```

- $FS_{v,p}$ represents the utility of a CPE p with respect to its potential to validate the CVE v when a negative answer from the human agent is expected (e.g., when a negative answer will allow to fully validate the vulnerability without any additional question)
- α_p represents the probability that the CPE p is really in the host given the collected advisories (i.e. the probability of a positive answer from the human agent)

Informally $TS_{v,p}$ and $FS_{v,p}$ are the answer to the question "*What is the potential of CPE p to resolve CVE v if the expected answer is positive/negative respectively?*"

For the computation of $TS_{v,p}$ and $FS_{v,p}$, it is necessary to analyze the CPE configuration trees associated with the vulnerability v. In particular it is necessary to interpret the all the configuration trees associated to vulnerability v as a formula in first order logic, in which CPE strings (leaf nodes) are interpreted as boolean variables.[18] A true/false assignment to a variable maps to the corresponding platform p (CPE string) residing or not in the considered host h_i. The second necessary step is a normalization passage in which the formula is converted in Disjunctive Normal Form to mimic an $\mathsf{OR}-\mathsf{AND}$ tree. This step is necessary as the logic formula expressed in DNF highlights the dependency of each platform with respect to other platforms, as well as providing a standardized form to apply any subsequent computational process to.

At last, the true score $TS_{v,p}$ is computed as:

$$TS_{v,p} = \mathsf{max}_i\Big(\frac{1}{|AND_i|}\Big) \times \frac{1}{|OR|}$$

where $|AND_i|$ is the number of $CPEs$ related by the i-th AND operator in the configuration tree and $|OR|$ is the number of alternative configurations in the configuration tree.

Similarly, the false score $FS_{v,p}$ is computed as:

$$FS_{v,p} = \Big(\sum_i |AND_i|\Big) \times \frac{1}{\mathsf{max}_i(|AND_i|) \times |OR|}$$

The coefficient α_p is used to estimate the expectation of a possible positive or negative answer from the human agent. The estimation of this value depends on the number of advisories that include at least one vulnerability which is in turn linked to a considered CPE string p. Notice that α_p is a coefficient that is global across the considered host h_i, therefore it does not depend on any specific vulnerability v. The rationale supporting this coefficient is that given a host h_i, multiple advisories assigned to h_i, reporting vulnerabilities that insist on the same CPE string p, should increase the possibility of CPE string p being present in the host h_i, as per the probe-based mechanism attached to the advisories. In simpler terms, "If multiple leads point to the same conclusion, maybe that conclusion holds some truth". Thus, the α_p coefficient can be computed as:

$$\alpha_p = \frac{deg_{adv}(p)}{max(deg_{adv})}$$

[18] Configuration trees of a same vulnerability are related one another by OR. This would make the structure containing all the configuration trees of the same vulnerability a $\mathsf{OR}-\mathsf{AND}-\mathsf{OR}$ tree.

where $deg_{adv}(p)$ is the number of paths in $\mathcal{G}_i$ ($E_{adv,CVE}$ and $E_{CVE,CPE}$) starting from any advisory adv and ending in p, and $max(deg_{adv})$ is the maximum number of paths in $\mathcal{G}_i$ starting from any advisory adv and ending in any platform CPE (i.e. $max(deg_{adv}(cpe)) \; \forall \; cpe \; \in \; \mathcal{G}_i)$.

Lastly, the aggregate_score() function implements a simple aggregation for every CPE of the computed utility scores over the edges. Thus, W_p is computed as:

$$W_p = \sum_{e_{v,p} \in E_{(CVE,CPE)}} W_{v,p}.$$

Thus it is possible to implement this proposed algorithm as part of the computational pipeline in an effort to bolster the accuracy Data Collection, Aggregation and Integration of a self-protecting system. A comprehensive evaluation and case study of the proposed solution will be analyzed later on, in Chapter 5.

4.4 Optimizing Platform Versions

During the development and validation of the solution proposed in the previous section, it has been noticed that the size of the vulnerability graph $\mathcal{G}_{\rangle}$ greatly increases when also considering the *version* attribute of CPE strings in addition to the mandatory attributes *part*, *product* and *vendor*. This is due to the fact that vulnerabilities may span over several versions of a product before a patch or an effective mitigation is finally released. Alas, it is possible that a vulnerability introduced in a version of a product may not be immediately found by analysts: it is possible that in the time between the introduction and the discovery of the vulnerability several versions of the same product have already been released, all equally vulnerable.

A possible solution to the increase in size of $\mathcal{G}_{\rangle}$ can be found by optimizing the encoding of contiguous CPE version attributes in the platform configuration tree of a CVE vulnerability. This solution is not new, as many vulnerability repositories, such as NVD, encode contiguous product versions as product version intervals, thus setting the version attribute of all affected CPE strings to the generic ANY (*) value and attaching metadata to the CPE string to encode the affected version intervals.

The reliance on metadata is necessary since version intervals are not part of the NIST documentation on CPE version 2.3 [78]. Since the solution proposed in the previous section has been built following the NIST documentation, a modification has to be made in order to optimize the proposed solution with respect to version intervals. The general principle behind this modification is not to change the proposed solution's algorithm or heuristic, but to act on $\mathcal{G}_{\rangle}$. In particular, since the proposed solution's algorithm and heuristic relies heavily on a platform's occurrence and frequency, it would make sense to (i) incorporate version intervals in $\mathcal{G}_{\rangle}$ as well as (ii) transforming each single version interval of a product into possibly multiple version intervals which are able to highlight eventual version overlap.

The intuition is that simply encoding version intervals in the vulnerability graph $\mathcal{G}_{\rangle}$ might not be enough, as the algorithm categorizes two partially *overlapping* version intervals as two separate entities. For this reason, in order for the proposed algorithm to effectively recognize the presence of an overlap, it may be necessary to explicitly highlight the presence of an overlap by splitting each version range into (i) version ranges that overlap completely with other version ranges, i.e. the versions contained by the version range are a subset of any other overlapping version range, and (ii) version ranges that never overlap with any other version range, i.e. intersection of this version range with any other version range yields $\varnothing$. This is necessary, because in order to be effective, the proposed solution's heuristic must be able to find recurring CPE strings within the platform configuration trees of vulnerabilities. With this intuition in mind, the next logical step would be to manipulate the version intervals in order to increase this recurrence. More formally, *overlapping* version intervals should be broken up into the minimum number of version "segments" possible, so that no version segment should terminate or start in the middle of another version segment. Thus, given any two version segments A and B for the same product p pertaining to the same host h_i, the set-intersection $A \cap B$ must result in either $\varnothing$ if $A \neq B$ or either A or B if $A \equiv B$.

This modification to $\mathcal{G}_{\rangle}$ can be achieved via two subsequent steps:

- **The first step** covers the encoding of the version intervals in the version attribute of the CPE strings of $\mathcal{G}_{\rangle}$. This is necessary since $\mathcal{G}_{\rangle}$ does not allow the presence of metadata for a given CPE string. The non destructiveness of this operation is guaranteed by the fact that in NVD, a CVE string's version attribute is set to ANY ($*$) by convention each time a version interval is attached to such CVE string.

- **The second step** covers the discretization of each version interval belonging to platform configurations of vulnerabilities in a given host. Mathematically as mentioned before, the desired outcome of this operation is that given any two version segments A and B for the same product p pertaining to the same host h_i, the set-intersection $A \cap B$ must result in either $\varnothing$ if $A \neq B$ or either A or B if $A \equiv B$. This is achieved trough the identification of *boundary versions* for each version interval of each product pertaining to a given host (i.e. versions that act as the outermost boundaries of the version interval.

More in detail, the discretization of a version interval into one or multiple version segments is highly dependant on the existence of a version boundary that falls in the middle (i.e. "cuts") of another version interval of the same product. As an example, if for a given product p there exist two version intervals $I_1 = (\geq 1, \leq 3)$ and $I_2 = (\geq 1, \leq 2)$, it is necessary to slice $I_1 = (\geq 1, \leq 3)$ into segments $I_1' = (\geq 1, \leq 2)$ and $I_1'' = (> 2, \leq 3)$ since one of the boundaries of I_2 (2) falls within interval I_1. This ensures that (i) no other interval intersects I_1'', and (ii) I_1' intersects I_2 and $I_1' \equiv I_2$. Notice also that in this example, sice no other boundary value cuts in the middle of I_2, there is no need to split I_2 into segments. Another interesting detail is that the boundary 2 has been assigned to I_1' and not I_1''. This operation is not trivial, and it will be explained in detail below, as it is dependant on all the version intervals that partake in p for host h_i.

Algorithm 5 implements the second step of the proposed optimization. In particular for each CPE p_j already present in $\mathcal{G}_{\rangle}$, the proposed algorithm retrieves its version boundaries and checks if a collision exists with other version intervals of the same product (Line 6). If a collision exists, the algorithm proceeds to split the interval identified by p_j into two or three version segments (s_list), and substitutes each occurrence of the colliding version interval p_j in $\mathcal{G}_{\rangle}{}^{opt}$ with the two or three version segments identified by s_list (Lines 6-9). The process continues until no collision[19] can be found on $\mathcal{G}_{\rangle}{}^{opt}$ (Line 2).

Assigning the Boundary Value

To recall the example that has been presented in the previous section, if a boundary

[19] Either all version segments and intervals for a product coincide, or their intersection is null.

Algorithm 5: Platform Version Optimization

```
Input: A Vulnerability Graph $\mathcal{G}_{\rangle}$
Output: An optimized Vulnerability Graph ${\mathcal{G}_{\rangle}}^{opt}$
${\mathcal{G}_{\rangle}}^{opt} \leftarrow \mathcal{G}_{\rangle}$;
while $has\_colliding\_version\_intervals({\mathcal{G}_{\rangle}}^{opt})$ do
    $V_{CPE} \leftarrow get\_CPE\_nodes({\mathcal{G}_{\rangle}}^{opt})$;
    foreach $p_j \in V_{CPE}$ do
        $r\_bound, l\_bound \leftarrow get\_interval\_bounds(p_j)$;
        if $product\_has\_colliding\_version\_intervals(p_j, r\_bound, l\_bound)$ then
            $c\_rng, c\_val \leftarrow get\_colliding\_interval\_and\_value(p_j, r\_bound, l\_bound)$;
            $s\_list \leftarrow segments\_from\_interval\_and\_collision(p_j, c\_rng, c\_val)$;
            $update\_graph({\mathcal{G}_{\rangle}}^{opt}, p_j, s\_list)$;
        end
    end
end
return ${\mathcal{G}_{\rangle}}^{opt}$
```

value of a version interval I_2 falls within a version interval I_1, version interval I_1 must be split into two version segments I_1' and I_1''. If the boundary value of a version interval is itself part of the version interval (operators $=, \geq, \leq$), deciding its assignment to one of the newly created version segments is not a trivial task. Version segments of the same platform p, under the same host h_i must satisfy the mathematical property of either not having anything in common with each other, or being completely overlapping with respect to other segments, i.e. given any two version segments A and B for the same product p pertaining to the same host h_i, the set-intersection $A \cap B$ must result in either $\varnothing$ if $A \neq B$ or either A or B if $A \equiv B$. This raises two possible outcomes for a boundary value that falls within a version segment, either (i) the version boundary value is included in one of the two segments that it generates (I_1' or I_1'' with $\leq$ or $\geq$ operators), or (ii) the version boundary value is not included in any of the two segments that it generates, and instead spawns a third segment comprising only itself (I_1''' with $=$ operator).

In particular, the logic implemented in the $segments\_from\_interval\_and\_collision()$ function of Algorithm 5 adheres to the following logic:

Given the two segments A and B divided by the boundary value *bound* (i.e. A:$(.., < \textit{ or } \leq \textit{bound})$, B:$(> \textit{ or } \geq \textit{bound}, ..)$), the boundary value *bound* is to be either

$$\begin{cases} \text{included in } A : (.., \leq \textit{bound}) & \text{if no other interval includes } \textit{bound} \text{ with operator } \geq, < \text{ or } = \\ \text{included in } B : (\geq \textit{bound}, ..) & \text{if no other interval includes } \textit{bound} \text{ with operator } \leq, > \text{ or } = \\ \text{included in } C : (= \textit{bound}) & \text{otherwise} \end{cases} \tag{4.1}$$

where A and B are the two segments derived from the split of a larger version interval by *bound*, and C is a third, potentially new, segment which only contains *bound*.

This modification on $\mathcal{G}_{\rangle}$ preserves the proposed solution's ability to correctly function as well as integrates an important optimization with respect to the increased size of $\mathcal{G}_{\rangle}$ without having to trade any accuracy. And while version attributes are optional according to the NIST documentation on CPE strings, and encoding them in G_i comes at a cost in terms of complexity, it is necessary to stress the importance of this decision and of its consequences on the accuracy of the resulting analysis.

For this reason the vulnerability graph G_i^{NoVe} will be also introduced, defined as G_i with its CPE strings truncated to only consider the mandatory attributes *part*, *product* and *vendor*. This graph will be used during the validation on the case study (Chapter 5) to allow to accurately compare the impact in terms of accuracy and scalability of such decision with respect to the filtering process.

4.4.1 Example of Application

The following section will detail an example of the application of the proposed optimization given an instance of a generic product in $\mathcal{G}_\rangle$. The proposed example is also visually described in Fig. 4.6. In particular:

Given three CPE strings in $\mathcal{G}_\rangle$ pertaining to a single product, and given their attached metadata retrieved from NVD

1. *cpe:2.3:a:vendor:product:1.0.0:*:*:*:*:*:*:**
2. *cpe:2.3:a:vendor:product:*:*:*:*:*:*:*:** with $version \geq$ 1.0.2
3. *cpe:2.3:a:vendor:product:*:*:*:*:*:*:*:** with $version \geq$ 0.9.0 & $version \leq$ 1.1.0

According to Algorithm 5, the following *boundary versions* are identified for the given product

- the boundary 1.0.0 from CPE_1 ($version =$ 1.0.0)
- the boundary 1.0.2 from CPE_2 ($version \geq$ 1.0.2)
- the boundary 0.9.0 and 1.1.0 from CPE_3 ($version \geq$ 0.9.0 and $version \leq$ 1.1.0)

Among these

- the boundary 1.0.0 from CPE_1 collides with CPE_3 (1.0.0 between 0.9.0 and 1.1.0)
- the boundary 1.0.2 from CPE_2 collides with CPE_3 (1.0.2 between 0.9.0 and 1.1.0)
- the boundary 1.1.0 from CPE_3 collides with CPE_2 (1.0.0 between 0.9.0 and 1.1.0)
- the boundary 0.9.0 from CPE_3 does not collide with any CPE

The proposed algorithm would then proceed to

- Leave CPE_1, as-it-is, since it identifies precisely one version
- Segment CPE_2 into two CPE strings:
  1. a CPE string CPE_2' for the interval [1.0.2, 1.1.0]
  2. a CPE string CPE_2'' for the open interval]1.1.0, $+\infty$[

  This is necessary since version 1.1.0 of CPE_3 falls within the version interval of CPE_2 The boundary 1.1.0 is assigned to CPE_2' since also CPE_3 includes 1.1.0 as a boundary.
- Segment CPE_3 into four CPE strings:
  1. a CPE string CPE_3' for the interval [0.9.0, 1.0.0[
  2. a CPE string CPE_3'' for version 1.0.0
  3. a CPE string CPE_3''' for the interval]1.0.0, 1.0.2[
  4. a CPE string CPE_3'''' for the interval [1.0.2, 1.1.0]

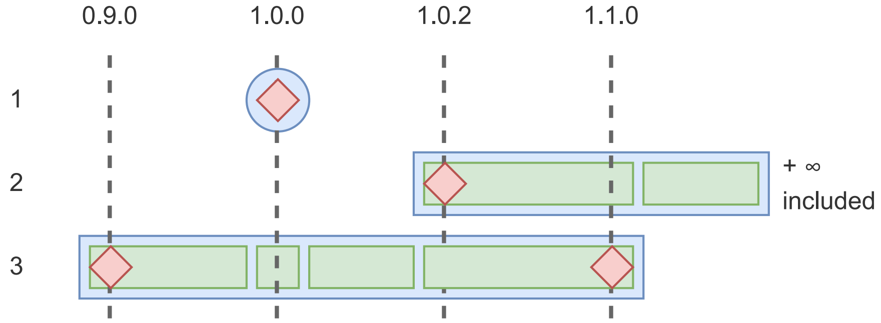


Figure 4.6. Example of the version splicing performed during 4.4.1. The three CPE strings in input are represented on the Y axis, while the boundary versions are represented on the X axis. Blue indicates the versions and version intervals in input, green indicates the versions and version intervals in output for each CPE string. Red indicates the boundary values identified by each CPE in input.

> This is necessary since 1.0.0 and 1.0.2 fall within the version interval of CPE_3 Moreover, since 1.0.0 appears in CPE_1 as a unique version, it must not be included in CPE_3' or CPE_3'''. Thus, the creation of CPE_3'' is necessary Also, since 1.0.2 is already included in the interval defined by CPE_2', it is necessarily included in CPE_3'''' to allow for CPE_2' and CPE_3'''' to overlap completely.

Notice that the proposed optimization may transform each single CPE string into potentially several new CPE strings, depending on the number of required version segments. It however guarantees that the resulting CPE strings either identify one single version, or identify version intervals which either overlap completely with other version intervals or never overlap at all with any other version interval for the considered product. This enables the heuristic described in the previous section to act on potentially smaller, discrete version intervals, and to more easily identify and select the most plausible versions or version intervals,[20] effectively exploiting the added complexity.

4.5 Considering Platform Dependency

This section aims to deliver an advancement with respect to the solution proposed in Section 4.3 (Vulnerability Filtering). The focus of this new proposed advancement lies in considering a new tree-based model to track similarities between attributes of vulnerable platforms in CPE format.

Differently from what has been proposed before, this advancement operates on the structure of a CPE string itself. More in detail, until now across this chapter a CPE string has been handled as a monolithic entity, while in reality a CPE string is composed by a series of attributes. The intuition behind this proposal is that validating or discarding a platform in CPE string on a host h_i entails the validation or invalidation of its attributes. Following this principle, validating or discarding a platform that shares most of its attributes with another, may also validate or discard the other platform as a result of the validation or invalidation of the attributes that compose such platform.

[20] Given a flat OR configuration in which all CPE strings bear the same weight, in this example the filtering algorithm and the heuristic would prioritize CPE strings pertaining to versions 1.0.0 and [1.0.2, 1.1.0] before others, as they are the most frequent.

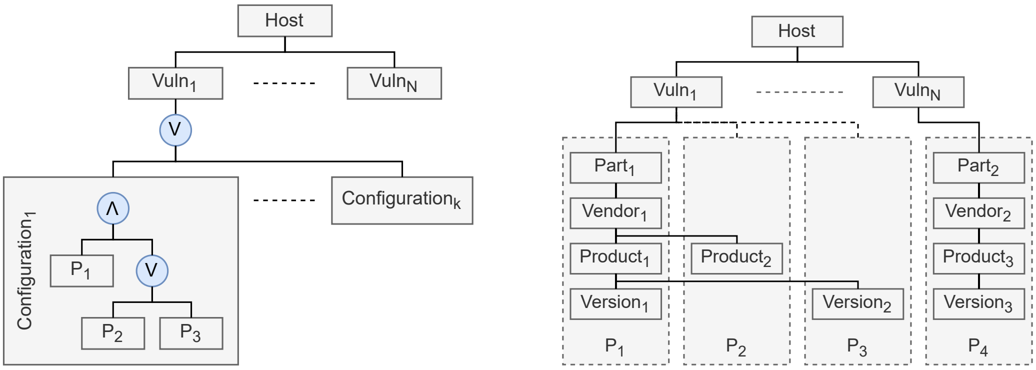
As a bonus, since this proposed optimization operates on the structure of the CPE string itself, it is possible to combine it together with the optimization of the version attribute of CPE strings described in Section 4.4 (Version Optimization), as the two are orthogonal.

The proposed advancement of the methodology described in Algorithm 4 of Section 4.3 (Vulnerability Filtering) expands the vulnerability graph $\mathcal{G}_i$ by implementing a dependency model able to potentially validate multiple platforms as a result of a single interaction with the human agent.

The main intuition behind this new dependency model is to represent the platforms in the vulnerability graph $\mathcal{G}_i$ as a tree-shaped graph having the host h_i as root. This is achieved by the computation of a *vulnerability tree* $\mathcal{T}_i = (V_i^T, E_i^T)$ for each host h_i in the vulnerability inventory $\mathcal{VI}$.

The set of nodes V_i^T of the vulnerability tree is partitioned into three sub-sets: (i) *root nodes* V_r^T which collects the host h_i currently under analysis, (ii) *vulnerability nodes* V_v^T which includes a node for every vulnerability v listed in the vulnerability inventory $\mathcal{VI}$, (iii) *platform nodes* $V_{CPE\_part}^T$, $V_{CPE\_vendor}^T$, $V_{CPE\_product}^T$, $V_{CPE\_version}^T$ including a node respectively for every part, vendor, product and version attribute appearing in each platform associated to any vulnerability in the inventory $\mathcal{VI}$. Consequently, $V_i^T = V_r^T \cup V_v^T \cup V_{CPE\_part}^T \cup V_{CPE\_vendor}^T \cup V_{CPE\_product}^T \cup V_{CPE\_version}^T$.

The set of edges E_i^T is partitioned into the following sub-sets: (i) $E_{h_i,v}^T$ is the set of edges linking a host h_i to its associated vulnerability v, (ii) $E_{v,p}^T$ is the set of edges linking a vulnerability v with its associated platform p, and (iii) E_p^T is the set of edges derived from a platform p and connecting (through a path) a part node to a vendor node, to a product node, and finally to a version node. Since they form a sub-tree in $\mathcal{T}_i$, we refer to these edges as *platform sub-tree*, (grey-shaped sub-trees in Fig. 4.7(b)). Consequently, $E_i^T = E_p^T \cup E_{p,v}^T \cup E_{h_i,v}^T$.



(a) Example of the vulnerability graph G_i described in Section 4.3.

(b) Example of the proposed vulnerability tree $\mathcal{T}_i$.

Figure 4.7. Comparison between the (a) vulnerability graph $\mathcal{G}_i$ described in Section 4.3 (Vulnerability Filtering) and (b) the proposed vulnerability tree $\mathcal{T}_i$ to model the dependencies between attributes of platforms.

Figure 4.7 visually highlights the differences between the vulnerability graph $\mathcal{G}_i$ described in Section 4.3 (Vulnerability Filtering) and the proposed vulnerability tree $\mathcal{T}_i$. The main difference is that the existing model (Fig 4.7(a)) focuses on the applicability of vulnerabilities but does not take advantage of the attributes of platforms represented in CPE format.

The proposed tree model (Fig 4.7(b)) highlights commonalities between different platforms of the same host, such as different versions of the same product and other products of the same vendor. As a consequence, the proposed tree model also enables the validation or invalidation of common elements.

As an example, considering the setting described in Figure 4.7, it can be immediately seen that by validating the presence of P_1, only $Product_2$ will be needed to validate P_2, and only $Version_2$ is needed to validate P_3. This is due to the fact that validating P_1 entails validating $Part_1$, $Vendor_1$, $Product_1$ and $Version_1$. P_2 shares the same $Part_1$ and $Vendor_1$ with P_1, its only difference being $Product_2$. Similarly, P_3 shares the same $Part_1$, $Vendor_1$ and $Product_1$ with P_1, its only difference being $Version_2$. Conversely, invalidating the presence of P_2 also invalidates the presence of P_1 and P_3, since all three depend on $Part_1$ and $Vendor_1$, i.e. it would not make sense to look for versions of products of which their vendor is not even part of the considered host. This example highlights the main property of the proposed tree model: platform attributes that haven't been validated yet but align with any invalidated attribute in the platform sub-tree are automatically discarded without requiring additional human input. Conversely, if all attributes of a platform that have not been validated yet match with validated attributes in the structure, the platform sub-tree can be safely validated without further human input.

This advancement can be easily implemented in Algorithm 4 of Section 4.3 (Vulnerability Filtering) just after the ask_for_user_validation() function (line 18 of the same algorithm.). The human agent's response should trigger a full traversal of $\mathcal{T}_i$ in order to validate or discard platforms which have been indirectly affected by the human agent's response with regards to the chosen platform and its attributes. Thus it is possible to achieve another advancement for the solution proposed in Section 4.3 (Vulnerability Filtering), which is also compatible with previous advancements and optimizations, such as the one described in Section 4.4 (Version Optimization).

4.6 A Platform Structure-Aware Algorithm

This section will describe an alternative approach with respect to the solution proposed in Section 4.3 (Vulnerability Filtering). As per the advancements proposed in the previous sections, this new proposed approach aims to attack the complexity generated by considering the optional *version* attribute of CPE strings in addition to the mandatory attributes *part*, *product* and *vendor*. The intuition behind this approach is similar to the intuition of Section 4.5 (Platform Dependency), for which a platform using the CPE format is not just a monolithic entity, but is composed by a series of attributes that are associated one to the other not only through juxtaposition, but also through meaningful semantic relationships. Simple examples of the semantic relationships between attributes of a CPE string can be found between contiguous couples of attributes such as *product* and *vendor*, or *product* and *version*: it is not intuitive to disassociate a product to its vendor, as it is not intuitive to disassociate a version to a specific product. Indeed, the presence of semantic relationships between attributes of a CPE string is also not a coincidence, as it is a design choice of the CPE standard in order to allow for comparison of CPE strings, as per NIST IR-7696 [78].

Focusing on the semantic relationships between attributes of platforms opens up a new venue to explore in order to achieve a new approach to attack the problem.

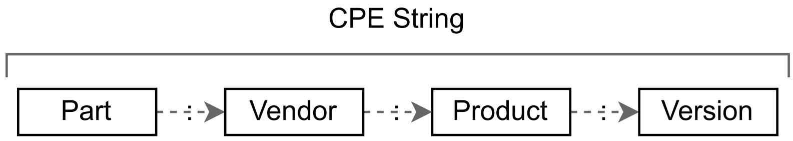


Figure 4.8. CPE string interpretation of attributes *part*, *vendor*, *product* and *version* as a unary tree.

As stared in the beginning of the section, the main intuition behind this new approach is that a CPE string is not a monolithic entity, but can be represented as an unary tree in which each level represents an attribute, as exemplified by Figure 4.8. For this reason, a new approach which is aware of the dependency model between attributes of CPE strings can be proposed as an alternative to the general approach proposed in Section 4.3 (Vulnerability Filtering). In particular, invalidating an attribute of a CPE string has the benefit of discarding multiple CPE strings (platforms) which depend on the single discarded attribute, at the cost of possibly requiring multiple steps for validation since a single CPE string (platform) under this model requires all its attributes to be validated.

In order to allow this new approach to reason on the dependency model between attributes of CPE strings, a new structure has to be proposed, capable of correctly representing the dependency model between attributes of multiple CPE strings related to a single host. This could be achieved by a tree structure, in which the root identifies the host, and nodes represent attributes of the CPE strings that partake in the host.

More formally:

Definition 1 (*Host-Platform Graph* $\mathcal{HPG}_i$ for a host h_i)

Let h_i be a host of the monitored environment and let CPE_i be the set of CPEs strings $s_j = part : vendor : product : version$ such that $\langle h_i, s_j \rangle \in \mathcal{DI}_{raw}$.

Let the set of nodes V_i^{HP} that are part of the host-platform graph $\mathcal{HPG}_i$ be

- $V^{host} = h_i$ represents the host upon which the graph is built.
- $V_i^{part} = \{part_k | \ \exists s_j =$ "part$_k$:-:-:-" $\in CPE_i\}$ represents the set of distinct *part* attributes existing in the device inventory pertaining to host h_i.
- $V_i^{vendor} = \{vendor_k | \ \exists s_j =$"x:vendor$_k$:-:-"$\in CPE_i\}$ represents the set of distinct *vendor* attributes existing in the device inventory pertaining to host h_i.
- $V_i^{product} = \{product_k | \ \exists s_j =$"x:y:$product_k$:-" $\in CPE_i\}$ represents the set of distinct *product* attributes existing in the device inventory pertaining to host h_i.
- $V_i^{version} = \{version_k | \ \exists s_j =$"x:y:z:$version_k$" $\in CPE_i\}$ represents the set of distinct *versions* attributes existing in the device inventory pertaining to host h_i.

Let the set of edges E_i^{HP} that are part of the host-platform graph $\mathcal{HPG}_i$ be

- $E_i^{(host,part)} = \langle h_i, part_k \rangle$ represents the edges between the host node and the *part* nodes.
- $E_i^{(part,vendor)} = \langle part_k, vendor_z \rangle$ represents the edges between each *part* nodes and each *vendor* nodes, as they appear in the CPEs of the device inventory for h_i.
- $E_i^{(vendor,product)} = \langle vendor_k, product_z \rangle$ represents the edges between each *vendor* nodes and each *product* nodes, as they appear in the CPEs of the device inventory for h_i.
- $E_i^{(product,version)} = \langle product_k, version_z \rangle$ represents the edges between each *product* nodes and each *versions* nodes, as they appear in the CPEs of the device inventory for h_i.

The *host-platform graph* for h_i is a graph $\mathcal{HPG}_i = (V_i^{HP}, E_i^{HP})$ where

- $V_i^{HP} = V_i^{host} \cup V_i^{part} \cup V_i^{vendor} \cup V_i^{product} \cup V_i^{version}$.
- $E_i^{HP} = E_i^{(host,part)} \cup E_i^{(part,vendor)} \cup E_i^{(vendor,product)} \cup E_i^{(product,version)}$.

An example of a host-platform graph $\mathcal{HPG}_i$ for host h_i is shown in Figure 4.9.

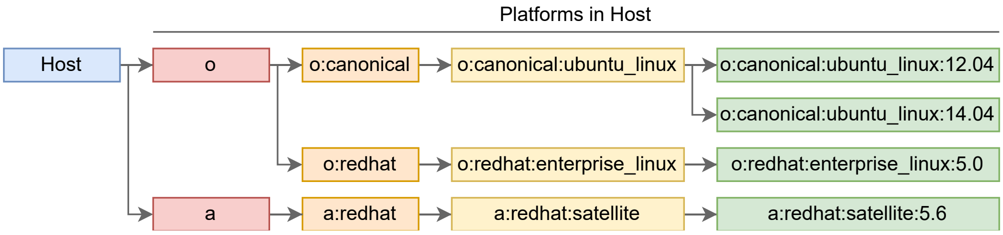

Platforms in Host
Host
o
o:canonical
o:canonical:ubuntu_linux
o:canonical:ubuntu_linux:12.04
o:canonical:ubuntu_linux:14.04
o:redhat
o:redhat:enterprise_linux
o:redhat:enterprise_linux:5.0
a
a:redhat
a:redhat:satellite
a:redhat:satellite:5.6

Figure 4.9. Host-Platform Graph $\mathcal{HPG}_i$ for host h_i. The colors highlight the concept of “levels” within the graph. Each level originates from a different truncation of a CPE strings.

As the figure suggests, the structure of the host-platform graph $\mathcal{HPG}_i$ classifies nodes in levels and each level is associated with an attribute of the CPE string (e.g. part, vendor, product, version). Moreover, nodes of one level are linked only to nodes of other levels: no edge exists between nodes of the same level (e.g. no edges exist between two “vendor” nodes). Thus, the host-platform graph $\mathcal{HPG}_i$ has the capacity to highlight relationships between multiple CPE strings through their participation in common sub-trees, e.g. versions of the same product sharing the same product, vendor and part parent nodes. This in turn means that operating on intermediate nodes of $\mathcal{HPG}_i$ could potentially amplify the number of items validated or discarded during a single interaction with a human agent. Indeed, given the setting described in Figure 4.9, two versions (and platforms) share the same part, vendor and product (o, *canonical* and *ubuntu_linux*). Confirming the absence of any of these three attributes from h_i automatically also confirms the absence of any platform (identified by versions 12.04 and 14.04 of the ubuntu linux operating system) which shares such attributes.

The same figure also highlights an important constraint that is necessary to correctly formulate and utilize the host-platform graph. As stated before, according to documentation, semantically each attribute of a CPE string is tied to the previous attribute of the same string. In order to enforce this semantic dependence in a simple manner, a tree structure must be enforced on $\mathcal{HPG}_i$ by design, i.e. no node in V_i^{HP} shall have more than one parent node. This is not granted by default by the CPE standard, since as an example in the figure the CPE vendor attribute “redhat” appears both as a software (a) vendor as well as a operating system (o) vendor. Indeed it is possible for the same vendor to be associated to different parts and even for the same product to be associated to multiple vendors inside the CPE dictionary. This may be due to the possibility of acquisition of a product by a different vendor, as well as re-branding efforts conducted by the vendor. This phenomenon is not limited to the $\langle part, vendor\rangle$ or $\langle vendor, product\rangle$ attribute tuples, but is also possible for some other attributes due to their generality, e.g. a value of 1.0.0 may be frequently used in the *version* attribute, but each usage is semantically associated to potentially different products.[21]

In order to actuate this enforcement, *truncated* CPE strings are used as identifiers for nodes in V_i^{HP}. For the purpose of this work, the truncation of a CPE string p on an attribute x is defined as the sub-string p' composed of the sequence of contiguous CPE attributes starting from *part*, up to and including x. As an example, given *a:apache:http_server:2.0.0*, truncating this string on *vendor* yields *a:apache*. Under this convention, the type of the node represents the last attribute to be included in its identifier, e.g. given the same CPE string *a:apache:http_server:2.0.0*, a V_i^{vendor} node built on the given CPE shall be identified by the CPE’s *part* and *vendor* attributes *a:apache*, while a $V_i^{product}$ node built on the same CPE shall be identified by the CPE’s *part*, *vendor* and *product* attributes *a:apache:http_server*. Thus, this convention on the generation of identifiers for nodes in V_i^{HP} is still able to represent the semantic dependency between adjacent attributes in a CPE string, as well as enforce a tree-structure on $\mathcal{HPG}_i$. The only exception for this convention is V_i^{host}, which retains its unique identifier that maps to a host h_i. In other words, with the exception of V_i^{host}, the graph $\mathcal{HPG}_i$ is to be generated as a *Trie* (or *prefix tree*) in which each value associated to an edge is the value of an attribute in CPE, and each *key* (depth-first path on the tree) identifies a CPE string pertaining to the host h_i, according to the device inventory $\mathcal{DI}_{raw}$.

To validate or invalidate the presence of a vulnerability and improve the quality of the vulnerability inventory, it is also important to reason over the relationships between CPEs and CVEs. To this aim, a second graph is introduced, namely the *truncated vulnerability graph* $\mathcal{G}_i^{tx}$ that keeps track of the relationships between platforms, vulnerabilities and the contribution of each platform to validate or discard a given vulnerability.

This graph is built starting from the platform configuration ($\mathsf{OR}-\mathsf{AND}$) tree associated with a vulnerability v_j from NVD. However, differently from a complete configuration tree, its leaf nodes are not complete CPE strings but *truncations* of CPE strings. As a reminder, the truncation of a CPE string p on an attribute x has been defined as the sub-string p' composed of the sequence of contiguous CPE attributes starting from *part*, up to and including x.[22]

[21] This problem seems to be absent in the vulnerability tree $\mathcal{T}_i$ of Section 4.5, since experimentally it has been observed that platforms within a single vulnerability retain semantic and chronological affinity, i.e. no re-branding of a platform, nor other semantically relevant conflicts. Despite this, the proposed enforcement can be still carried out also on $\mathcal{T}_i$ to ensure semantical congruity.

[22] As an example, given *a:apache:http_server:2.0.0*, truncating this string on *vendor* yields *a:apache*.

More formally

Definition 2 (*Truncated Vulnerability Graph* $\mathcal{G}_i^{tx}$ for a host h_i)

Let h_i be a host of the monitored environment. The *truncated vulnerability graph* for host h_i is the graph $\mathcal{G}_i^{tx} = (V_i^{tx}, E_i^{tx})$ where

The set of nodes V_i^{tx} is partitioned into three sub-sets:

- *Advisory nodes* V_{adv} including a node for every advisory id listed in the raw vulnerability inventory $\mathcal{VI}_{raw}$, pertinent to host h_i.
- *Vulnerability nodes* V_{CVE} including a node for every CVE listed in the raw vulnerability inventory $\mathcal{VI}_{raw}$, present in host h_i.
- *Platform nodes* V_{CPE}^{tx} including a node for every CPE retrieved starting from a CVE in the raw vulnerability inventory $\mathcal{VI}_{raw}$, truncated at its attribute x, pertinent to host h_i.

The set of edges E_i^{tx} is partitioned into two sub-sets:

- $E_i^{(CVE,CPE^{tx})}$ is the set of edges linking a vulnerability with its associated CPEs (or truncations of thereof).
- $E_i^{adv,CVE}$ linking an advisory to all its associated vulnerabilities, within the context of host h_i.

An example of a truncated vulnerability graph for a host h_i is shown in Figure 4.10.

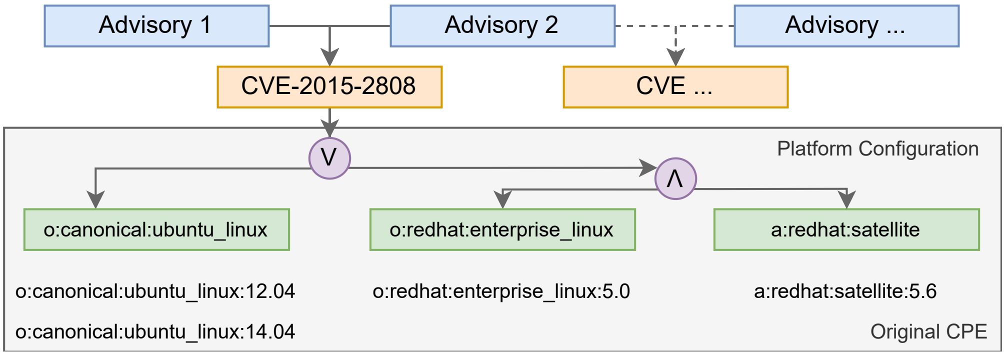


Figure 4.10. Truncated graph $\mathcal{G}_i^{tx}$ with $x = \mathit{product}$ for host h_i.

Thus, the truncated vulnerability graph for a host h_i retains the $\mathsf{OR} - \mathsf{AND}$ structure of relevant configuration trees, but is populated by leaf nodes that only hold a subset of the attributes of the original CPE strings, depending on their "truncation level" x. As a reminder, truncated CPE strings have also been used as identifiers for nodes in $\mathcal{HPG}$, and therefore, given a host h_i and a truncation level x, leaf nodes in $\mathcal{G}_i^{tx}$ correspond to the nodes found in the xth "level" of $\mathcal{HPG}_i$.[23]

These new structures allow for a new platform structure-aware algorithm that operates on $\mathcal{HPG}$ in order to produce new, accurate vulnerability and device inventories $\mathcal{VI}^+$ and $\mathcal{DI}^+$, starting from the "raw" inventories $\mathcal{VI}_{raw}$ and $\mathcal{DI}_{raw}$ to

[23] Levels of $\mathcal{HPG}_i$ spawn from a BFS exploration starting from the host node in $\mathcal{HPG}_i$.

be proposed. In order to guide the exploration of $\mathcal{HPG}$ performed by this new algorithm, several complementary new algorithms must also be designed to function as heuristics.

An algorithm for the host-platform graph $\mathcal{HPG}_i$

This section will describe the new proposed algorithm that is able to make use of the host-platform graph ($\mathcal{HPG}_i$)'s structural properties in order to enhance the quality of automatically generated inventories.

Intuitively, given a host h_i, the new algorithm should perform an exploration of $\mathcal{HPG}_i$, starting from its root (i.e. V^{host}), and progressively descend through its nodes. At each step of the exploration the currently explored node should be submitted to a human agent for validation. Since nodes in $\mathcal{HPG}_i$ are identified by truncations of CPE strings (i.e. sets of contiguous attributes), this exploration allows for platform strings expressed in CPE format to be validated or discarded progressively, one attribute at a time. As a consequence, this also allows the algorithm to discard entire sub-trees (thus multiple CPEs that share the same parent attributes) in the event that an intermediate node were to be discarded. The order of the exploration that determines which nodes are submitted to the human agent necessitates its own complementary algorithms. These algorithms which will be detailed later in the section, must be able to produce a cost function that is able to rank edges in $\mathcal{HPG}_i$ according to various proposed metrics and desiderata.

Algorithm 6: $\mathcal{HPG}$ exploration

```
Input: A raw Device Inventory $\mathcal{DI}_{raw}$
Input: A raw Vulnerability Inventory $\mathcal{VI}_{raw}$
Output: A Device Inventory $\mathcal{DI}^+$
Output: A Vulnerability Inventory $\mathcal{VI}^+$
$\mathcal{VI}^+ \leftarrow \emptyset$;
$\mathcal{DI}^+ \leftarrow \emptyset$;
$host\_list \leftarrow$ get_host_list($\mathcal{VI}_{raw}$);
foreach $h_i \in host\_list$ do
    $\mathcal{HPG}_i \leftarrow build\_hpg(\mathcal{DI}_{raw})$;
    foreach $x \in [part, vendor, product, version]$ do
        $\mathcal{G}_i^t[x] \leftarrow build\_trunc\_graph(\mathcal{VI}_{raw})$;
    end
    $CVE\_to\_validate \leftarrow get\_cve(\mathcal{VI}_{raw}, h_i)$;
    while $CVE\_to\_validate \neq \emptyset$ do
        $\mathcal{HPG}_i \leftarrow score\_hpg(\mathcal{G}_i^{tx}[])$;
        $n \leftarrow select\_node(\mathcal{HPG}_i)$;
        $result \leftarrow validate\_node(n)$;
        if $result = valid$ then
            $\mathcal{DI}^+ \leftarrow \mathcal{DI}^+ \cup n$;
        end
        else if $result = discard$ then
            $\mathcal{HPG}_i \leftarrow remove\_subtree(n)$;
        end
        $CVE\_to\_validate, \mathcal{VI}^+, \mathcal{G}_i^t[] \leftarrow check\_CVE\_conditions(\mathcal{DI}^+, \mathcal{VI}_{raw}, h_i)$;
        $\mathcal{HPG}_i \leftarrow prune\_useless(CVE\_to\_validate)$;
    end
end
return $\mathcal{VI}^+, \mathcal{DI}^+$
```

Algorithm 6 presents the pseudo-code for a solution which uses $\mathcal{HPG}$ in order to produce the improved inventories $\mathcal{VI}^+$ and $\mathcal{DI}^+$. For each host h_i, its host-platform graph $\mathcal{HPG}_i$ is computed as well as its associated truncated vulnerability graphs, one for each "level" of $\mathcal{HPG}_i$. Then, until all CVEs of h_i are either validated or discarded, the algorithm assigns scores to the edges of $\mathcal{HPG}_i$ using a cost function based on the truncated vulnerability graphs, and performs an exploration of $\mathcal{HPG}_i$. Each time a node which represents the value of a CPE attribute is selected by the exploration, it is selected to be validated or discarded from h_i. If the node is validated, it is included in $\mathcal{DI}^+$ and the exploration continues. If the node is discarded, its whole sub-tree is discarded (line 17) and removed from $\mathcal{HPG}_i$. On each cycle, CVEs's configuration trees are checked to determine if a CVE has been validated (and included in $\mathcal{VI}^+$) or discarded. Furthermore, nodes that aren't useful anymore for the validation of the remaining CVEs are pruned from $\mathcal{HPG}_i$ (lines 20-21).

Guiding the exploration of the host-platform graph $\mathcal{HPG}_i$

The exploration of $\mathcal{HPG}$ is guided by costs on nodes of $\mathcal{HPG}$ assigned by cost functions that reason on truncated vulnerability graphs $\mathcal{G}_i^{tx}$ (line 11). In particular, four different cost functions are proposed:

- **R - Random** in which the edges of $\mathcal{HPG}$ are assigned a random cost.
- **P - Platform-based** which uses the $\mathsf{OR}-\mathsf{AND}$ structure of $\mathcal{G}_i^{tx}$ to assign a cost to the edges of $\mathcal{HPG}$.
- **V - Vulnerability-based** which uses the severity of the vulnerabilities of $\mathcal{G}_i^{tx}$ to assign a cost to the edges of $\mathcal{HPG}$.
- **VP - Vulnerability-Platform-based** which uses the severity of the vulnerabilities of $\mathcal{G}_i^{tx}$ to assign a cost to the edges of $\mathcal{HPG}$, as well as the $\mathsf{OR}-\mathsf{AND}$ structure of $\mathcal{G}_i^{tx}$ to discriminate between forks of equal cost.

Platform-based cost function (P)

The platform-based cost function leverages the structure of each truncated vulnerability graph to compute the "potential of each CPE attribute to resolve one or multiple CVEs in h_i". This is achieved by leveraging the structure of $\mathcal{G}_i^{tx}$ analyzing each attribute's number and type of occurrences, as well as analyzing each attribute's recurrence in the advisories. The rationale is that if a CPE attribute is linked to many CVEs under OR conditions and it is present in the considered host h_i, it will contribute to the validation of all of them with a single action, while if it is involved in AND conditions and it is not present in the considered host h_i, then it will discard entire sub-trees from $\mathcal{HPG}_i$, possibly also discarding multiple CPEs from h_i. In addition, if a CPE attribute (i.e. a part, a vendor, a product or a version) is referenced by multiple advisories, the probability that it is really present increases.

These two analysis serve as the basis for the platform-based cost function P, able to compute the costs of traversing the edges $E(x-1,x)$ of $\mathcal{HPG}_i$, using $\mathcal{G}_i^{tx}$, for each "level" x of $\mathcal{HPG}_i$. This cost function can be formally defined as:

$$CFP(n) = 1 - (\alpha_n \times TS_n + (1-\alpha_n) \times FS_n)$$

where

- TS_n represents the potential of a CPE attribute n to validate a CVE, assuming that n is in h_i (positive answer).
- FS_n represents the potential of a CPE attribute n to validate a CVE, assuming that n is not in h_i (negative answer).

- α_n represents the probability of CPE attribute n being in host h_i given the collected advisories.

TS_n and FS_n are computed according to the $\mathsf{OR}-\mathsf{AND}$ structure of $\mathcal{G}_i^{tx}$ associated to host h_i. In particular, given a list of $\mathsf{OR}-\mathsf{AND}$ configurations within $\mathcal{G}_i^{tx}$, one for each vulnerability v of host h_i, we define as

$$TS_n = max_j\Big(\frac{1}{|AND_j|}\Big) \times \frac{1}{|OR|}$$

$$FS_n = \Big(\sum_j |AND_j|\Big) \times \frac{1}{\mathsf{max}_j(|AND_j|)} \times \frac{1}{|OR|}$$

where $|AND_j|$ is the number of CPE attributes related by the j-th AND operator in $\mathcal{G}_i^{tx}$ and $|OR|$ is the number of alternative configurations in $\mathcal{G}_i^{tx}$.

α is used to predict the expected answer from the validator, and is based on the number of advisories that point to the given CPE attribute. Its rationale being that if multiple probe-based advisories report vulnerabilities that insist on the same CPE attribute, that attribute is likely to be present in host h_i. More formally,

$$\alpha_n = \frac{deg_{adv}(n)}{V_n}$$

where $deg_{adv}(n)$ is the number of edges in $\mathcal{G}_i^{tx}$ ending in n and $|V_n|$ is the overall number of CPE attributes in $\mathcal{G}_i^{tx}$.

Vulnerability-based cost function (V)

Given an attribute n, node of $\mathcal{HPG}_i$, the vulnerability-based cost function assigns to each incoming edge $E(n-1,n)$ in $\mathcal{HPG}_i$ the $max_v(1-severity)$ of every vulnerability v that is linked to n in $\mathcal{G}_i^{tx}$. In the event of multiple edges having the same cost during the exploration of $\mathcal{HPG}_i$, a random selection is performed.

Vulnerability-Platform-based cost function (VP)

Given an attribute n, node of $\mathcal{HPG}_i$, the vulnerability-platform-based cost function assigns to each incoming edge $E(n-1,n)$ in $\mathcal{HPG}_i$ the $max_v(1-severity)$ of every vulnerability v that is linked to n in $\mathcal{G}_i^{tx}$. In the event of multiple edges having the same cost during the exploration of $\mathcal{HPG}_i$, the cost function computes the CFP of these edges and uses this secondary cost value as a discriminant.

Random-based cost function (R)

This cost function is trivial, as it assigns a random $[0-1]$ cost to every edge in $\mathcal{HPG}_i$ without any reasoning on semantics. However it is interesting to be kept as a baseline for the evaluation which will be carried out in Section 5.5 of the next chapter.

Thus, through the introduction of Algorithm 6 which operates on the *host-platform graph* $\mathcal{HPG}_i$, it is possible to utilize the full extent of the semantic relationships that tie together the attributes of a CPE string, as per NIST IR-7696 [78], in order to attack the complexity in the process of improving the quality of automatically generated inventories.

4.7 Strategies for Aggregation

The following section will describe a solution aimed at trading the quality of the *Analysis* sub-process, part of the vulnerability assessment process, in favour of increased scalability. The proposed methodology aims to further improve the scalability of the overall self-protecting system by reducing the size of the input processed by attack graph-based analysis techniques. More specifically, by aggregating the vulnerabilities in the inventory $\mathcal{VI}$ over analysis-aware semantic criteria, it is possible to produce an aggregated vulnerability inventory $\mathcal{VI}^{ag}$. Using the aggregated vulnerability inventory $\mathcal{VI}^{ag}$ as an input to the risk analysis affects the generation of the attack graph and subsequent risk estimation, resulting in a trade-off between accuracy and scalability of the risk analysis. Furthermore, the proposed methodology could be implemented in the *Vulnerability Aggregation* sub-component of the computational pipeline proposed in Section 4.2 in order to realize computational flows *CF2* and *CF3*.

Attack Graph-based methodologies allow an accurate estimation of risk in ICT networks even when considering multi-step attacks and complex network architectures. Attack graph-based systems for risk estimation, however, experience poor scalability which derives from the exponential number of attack paths that need to be considered during a risk analysis, especially when employed in a self-protecting system. This is due to the fact that an attack graph must model all the possible sequences of attack (i.e. paths) that an attacker may carry out once inside an ICT environment. And while theoretically, every sequence of attack (i.e. sequence of vulnerabilities used during an attack) is required in order to obtain the best accuracy during the risk estimation, usually only a sub-set of attributes of each vulnerability is actually considered in order to perform the risk estimation, depending on the desired analysis. This leads to the claim that multiple vulnerabilities in a host h_i may be aggregated into classes of "equivalence", according to their dependencies and the finite possible values of their risk metrics [44]. This aggregation would reduce the size of the vulnerability inventory, and in-turn reduce the computational effort needed during attack graph-based analysis.

To further elaborate, the core idea behind the proposed methodology is to aggregate a set of vulnerabilities with common features (e.g. that affect the same host and have the same set of pre and post-conditions) into a *meta-vulnerability* characterized by such common features. Ideally, it would be optimal to maximize the number of vulnerabilities aggregated into a meta-vulnerability (to maximize the scalability), while minimizing the potential accuracy loss.

It is important to note that different aggregation strategies exist depending on the desired set of attributes to be considered as keys for the purposes of the aggregation. The selection of the desired set of attributes should depend on the desired analysis as well as the desired amount of trade-off between a faster but less accurate analysis and an accurate but slower analysis. During the course of this section, three aggregation strategies that take into account different aspects of the attack graph-based risk model will be considered, namely:

- **VA** (Vulnerability-based Aggregation): For each host h_i, the strategy aggregates the vulnerabilities that have the same values for *all* their attributes. In different words, two vulnerabilities having the same attribute values are aggregated into a meta-vulnerability which inherits the same attribute values. Otherwise, no aggregation is performed. As a consequence, each meta-vulnerability is a representation of the unique combinations of vulnerability attribute values of each host.

- **RA** (Risk-based Aggregation): For each host h_i, the strategy aggregates the vulnerabilities that have the same values of the attributes used in a generic risk evaluation. The resulting meta-vulnerability inherits the same values for the attributes used during a generic risk assessment and computes the maximum value between aggregated vulnerabilities for all the other attributes.
- **HA** (Host-based Aggregation): For each host h_i, the strategy aggregates all the vulnerabilities independently from their attributes. The resulting meta-vulnerability computes the maximum value of the attributes among the aggregated vulnerabilities. As a consequence, each host h_i has a single meta-vulnerability.

These three aggregation strategies represent the two extremities of the spectrum (HA granting the fastest possible analysis at the cost of accuracy and VA being the most accurate aggregation strategy at the cost of scalability), as well as a middle ground that is centered around an exemplary set of metrics used during a generic instance of risk estimation.

Since the aggregation strategies may rely only on sub-sets of each aggregated vulnerability's attributes, it is possible that a single meta-vulnerability may aggregate vulnerabilities with different (qualitative) values for attributes and thus different severity (numerical) scores and sub-scores. Since the attributes of a vulnerability and the derived scores are necessary for subsequent analysis processes, it is critical to handle the possibility of aggregating different vulnerabilities with different values for their attributes. In order to address this critical issue, the calculation of a synthetic set of attributes is proposed, for each meta-vulnerability. This calculation must follow the "worst-case" analysis criteria. For this reason, each and every attribute of each and every vulnerability must be compared in order to synthesize a new set of attributes, composed of all the "worst-case" values of each attribute of each vulnerability.[24] While the "worst-case" metric calculation process for each meta-vulnerability addresses the needs of the subsequent analysis processes, synthesizing new values for the attributes of a meta-vulnerability from the attributes of each and every aggregated vulnerability may harm the accuracy of the subsequent analysis. This is due to the fact that the new values for the attributes, synthesized by applying a "worst-case" criteria on the attributes of each and every aggregated vulnerability, may not appear in any vulnerability aggregated by the meta-vulnerability. It is important to be aware of this inaccuracy, which is not present in more stringent, less performing aggregation strategies (VA), but becomes evident in more performing, less stringent aggregation strategies (RA, HA). This creates a trade-off between the accuracy or the risk estimation and the scalability of the attack graph-based analysis, which leads to the necessity to make use of the most fitting aggregation strategy given the context.

Applying a "worst-case" criterion to the attributes of a meta-vulnerability (i.e. choosing the maximum value for each attribute across the aggregated vulnerabilities) leads to the estimation of higher risk values. The consequence of this approach is that meta-vulnerabilities tend to provide an overestimation of the risk, thus leading to a conservative risk-analysis which coincides with an upper bound in the risk estimation. This overestimation is proportional to the accuracy of each aggregation

[24] Using CVSS as an example for vulnerability attributes, given CVSS v3.1 vector strings AV:P/AC:H/PR:H/UI:R/S:U/C:H/I:H/A:H and AV:N/AC:L/PR:N/UI:N/S:C/C:N/I:N/A:N, the result of the "worst-case" synthesis is AV:N/AC:L/PR:N/UI:N/S:C/C:H/I:H/A:H. As it will be explained later, note that the synthesized CVSS vector string may not be equal to any of the original CVSS strings, i.e. it can happen that no original vulnerability may have the synthesized CVSS string.

strategy, as stricter aggregation strategies such as VA and RA result in less risk overestimation and a more accurate analysis, at the cost of generating a larger Attack Graph. In contrast, the HA strategy provides the highest possible overestimation of risk, and therefore the lowest possible accuracy. The benefit of the HA strategy is that the resulting Attack Graph will have the smallest possible size before loosing critical pieces of information (if even one meta-vulnerability is removed from the result of HA, a host h_i will be completely severed from other hosts in the resulting Attack Graph), and as such will severely limit the exponential explosion in size of traditional Attack Graphs.

As an example, a host h_i will be considered, with the vulnerabilities v_1, v_2, and v_3 reported in Table 4.1. Each vulnerability v_x has metrics m_a, m_b, and m_c, where only m_a and m_b are used for risk estimation.

Table 4.1. Example of aggregation strategies for the vulnerabilities of a host h_i.

	Risk Metrics		
	m_a	m_b	m_c
v_1	1.0	1.0	0.5
v_2	1.0	0.5	1.0
v_3	1.0	0.5	0.5

The three considered aggregation strategies generate the following meta-vulnerabilities:

- **VA**: performs no aggregation, since the metric values of the three vulnerabilities are mutually different. Therefore this strategy will produce three meta-vulnerabilities which will have the same values for metrics as v_1, v_2 and v_3, respectively.

- **RA**: produces two meta-vulnerabilities: $u^*_{RA,1}$ and $u^*_{RA,2}$. $u^*_{RA,1}$ aggregates vulnerabilities v_2 and v_3 with metrics $\{m_a : 1.0, m_b : 0.5, m_c : 1.0\}$, while vulnerability v_1 is not aggregated and included in $u^*_{RA,2}$. This is because v_1 has a different value for the risk metric m_b, which is necessary for the risk estimation.

- **HA**: aggregates all the vulnerabilities of host h_i into a single meta-vulnerability $u^*_{HA,1}$ with metrics $\{m_a : 1.0, m_b : 1.0, m_c : 1.0\}$.

Thus it is possible to implement several aggregation strategies that substitute vulnerabilities in $\mathcal{VI}$ with meta-vulnerabilities that aggregate multiple vulnerabilities. The accuracy of the aggregation performed by the meta-vulnerabilities depends on the aggregation strategy used and the semantic-aware criteria that have been used to perform the aggregation. The result is an aggregated vulnerability inventory $\mathcal{VI}^{ag}$ which is able to trade an Attack Graph-based risk analysis' accuracy in favour of increased scalability. The magnitude of the trade-off between accuracy and scalability can be tuned according to the desired analysis, by deploying different aggregation strategies.

4.8 Conclusion

This chapter introduced the problem of enhancing the quality of automatically generated inventories used during the cyber risk management process. In particular, vulnerability and network scanners monitor the network environment in order to produce a Vulnerability Inventory $\mathcal{VI}$ which collects and lists all the vulnerabilities residing in each host in the network, and a Device Inventory $\mathcal{DI}$ which lists all the platform configurations of hosts in the network. Starting from these two inventories, the aim of this chapter has been twofold: (i) the introduction and formalization of the problem of the quality of the data collected in the inventories by vulnerability and network scanners, and (ii) the proposal of a computational pipeline and architecture which is able to provide solutions that aim to mitigate the aforementioned problem.

It has been found that despite the problem of the quality of the data collected in the inventories by vulnerability and network scanners exists, it is possible to: (i) leverage the knowledge of a human agent in order to increase the accuracy of such inventories by filtering out false positives, and (ii) trade the accuracy of the analysis in favour of an increased scalability of the process, if so desired.

In particular, probe-based network and vulnerability scanners, although unintrusive, may falsely detect platform configurations linked to active vulnerabilities, therefore potentially introducing false positives in the inventory. The chapter proposes a filtering process capable of reducing the number of false positives in the inventories that involves submitting queries to a human agent in order to validate or invalidate the presence of platforms on a host. Platforms on a host which have been validated or invalidated can then be used in conjunction with the applicability conditions of each vulnerability in order to validate or invalidate each vulnerability. In addition to this process, this chapter also proposes an aggregation methodology able to perform a trade-off between the accuracy of the risk analysis and its scalability in Attack Graph-based methodologies. This methodology is able to aggregate vulnerabilities on each host, using analysis-aware strategies which are configurable by the user in order to obtain the desired level of trade-off between accuracy and scalability of the Attack Graph-based risk analysis.

The two proposed processes have been integrated into a modular architecture to be implemented during the Data Collection Aggregation and Integration process of a state-of-the-art autonomic system built upon the MAPE-K architecture. This new modular architecture is able to support four computational flows:

- **CF0**: Performing no pre-processing on the vulnerability inventory $\mathcal{VI}_{raw}$
- **CF1**: Filtering vulnerabilities from the vulnerability inventory $\mathcal{VI}_{raw}$ in order to produce a new vulnerability inventory $\mathcal{VI}^{+}$
- **CF2**: Aggregating vulnerabilities from the vulnerability inventory $\mathcal{VI}_{raw}$ into meta-vulnerabilities in order to produce a new vulnerability inventory $\mathcal{VI}_{ag}$
- **CF3**: Filtering vulnerabilities from the vulnerability inventory $\mathcal{VI}_{raw}$, then aggregating the resulting filtered vulnerabilities into meta-vulnerabilities in order to produce a new vulnerability inventory $\mathcal{VI}^{+}_{ag}$

It is important to note that the proposed processes and architecture derive from an in-depth study of current state of the art vulnerability and network scanning techniques, in conjunction with some of the repositories described in Chapter 3.

The results of the chapter represent a first step toward the design and implementation of a solution capable of performing fully automated cyber risk assessment by refining and resolving problems that affect existing paradigms and methodologies adopted by state of the art self-protecting systems.

Chapter 5 will introduce a case study and will analyze the results of a comprehensive experimental evaluation of each solution described in this chapter.

Chapter 5

Analyzing the Accuracy-Scalability Trade-off in Attack Graph-Based Self-Protecting Systems - A Case Study

Building and maintaining inventories is a fundamental component of many security processes [24]. Among all inventories used to feed security processes, Chapter 4 focused on the device inventory and the vulnerability inventory. These two inventories are particularly relevant data sources that derive from a monitored environment and are used not only to feed risk management [38, 62] but also other critical processes such as Incident Management [16] and their supporting systems [85]. Since these sources constitute the primary data source for the analysis of complex situations, some of these potentially involving multiple correlation steps, their quality, in terms of completeness and accuracy, must be ensured to avoid that a propagation of false positives and false negatives through the analysis steps cascades into an extreme overestimation (or underestimation) of the real security situation.

This chapter will perform a thorough evaluation of the effects of the methodologies and algorithms described in the previous chapter (Chapter 4) on the device and vulnerability inventories, as well as on the subsequent analysis phase of the cyber risk management process. In particular, this chapter will analyze how the filtering and aggregation sub-components of the proposed computational pipeline affects subsequent risk analysis based on the attack graph model [38, 93], a graph-based model of the potential attack steps in a network.

It is important to note that while risk analysis and more broadly cyber risk management processes strive towards the best possible accuracy, this contrasts with the performance of the autonomous system. Indeed, attack graphs are very powerful and potentially accurate models, but they suffer from scalability issues forcing to opt for the computation of an *approximated* attack graph [68, 70, 43]. This introduces a certain degree of uncertainty in the risk estimation, where accuracy is traded off for scalability. The situation gets even worse if we consider that state-of-the-art attack graph-based self-protection systems feed the control loop with input data that is not completely accurate due to false positives affecting the monitoring probes. This results in a cumulative effect of accuracy loss, affecting first the data collection phase and then the attack graph and risk computation phase.

Thus, the main focus of this chapter will be the evaluation of the algorithms and methodologies developed to: (i) attack the problem of improving the quality (in terms of accuracy) of the automatically generated inventories without increasing the degree of intrusiveness in the monitored system, and (ii) achieve a trade-off between accuracy and scalability in attack graph-based analysis methodologies, through the use of semantic-aware aggregation.

To achieve a thorough evaluation, a methodology to produce several statistically relevant testing environments will be proposed, which will allow to achieve a comparable, statistically relevant and congruous result across all the proposed methodologies and algorithms. Having a real testing environment will also be necessary not only in order to evaluate the single components, but also to carry out the evaluation of the whole proposed computational pipeline.

This chapter provides the following contributions:

- A methodology to produce a statistically relevant testing environment on-demand using virtualization.
- A comprehensive evaluation of the methodologies and techniques proposed in Chapter 4 using real data coming from a case study of a network composed of eight real hosts, in particular the following methodologies will be evaluated:
  - An unoptimized algorithm for platform filtering.
  - An optimization for platform filtering targeted at platform versions.
  - An advancement for platform filtering that leverages the dependency between CPE strings.
  - An alternative, optimized algorithm for platform filtering that is aware of the structure of CPE strings.
- A comprehensive evaluation of the computational pipeline proposed in Section 4.2 aimed at the quality enhancement of automatically generated inventories in state-of-the-art self-protecting systems. In particular the evaluation of the pipeline will comprehend:
  - A thorough evaluation of how the filtering component affects the structure of the resulting attack graph.
  - The impact of the accuracy-scalability trade-off operated by several analysis-aware aggregation strategies on the attack graph's structure, generation and analysis.

5.1 Environment

This section will describe the environment of the proposed case study, as well as provide a methodology to construct network environments which are able to achieve statistically relevant case studies in order to evaluate cyber risk analysis processes. In particular, the application of the proposed methodology has been necessary in order to create a testing environment to perform a thorough evaluation of the computational pipeline and of the methodologies introduced in Chapter 4. Such proposed methodology is able to achieve statistical relevance while guaranteeing the reproducibility property of the results. Moreover, the proposed methodology exposes and allows to operate by design on detailed information about the platform composition of each host of the constructed network environment. The proposed methodology also grants the possibility to further probe the network environment using commercially available vulnerability and network scanners through virtualization.

While solutions such as cyber ranges that are able to provide reproducibility as well as access to in depth information about each host and network configuration exist, these solutions are usually proprietary, and thus require complex (and costly) set-up processes. This is due to the fact that cyber ranges are complex systems which allow for full scale simulation and operation of a mock network environment. As a result of integrating technologies and solutions that are able to accommodate an extensive array of tasks, from training cybersecurity specialists to prototyping entire network architectures before they are implemented in a real scenario, cyber ranges prove themselves to be too complex and costly to be feasible for applications that may require only a fraction of their potential, such as the application proposed in this chapter. This motivates the interest in developing a methodology that is able to obtain a "lightweight" reproducible and statistically relevant network environment, able to support a sub-set of the operations typically carried out on a real environment (e.g. threat hunting, risk estimation).

Thus, the proposed methodology has been designed to construct, instantiate and collect snapshots of a network environment in order to perform subsequent tasks such as risk estimation. More in particular, the proposed methodology has been designed to deterministically produce a synthetic testing environment from a set of known platforms, while maintaining *variety* and *plausibility* of host configurations, thus guaranteeing *reproducibility*. In order for the synthetic network environment to be queried by tools and methodologies used regularly by security professionals in real scenarios, virtualization has been used to instantiate each synthetically generated host configuration into a virtual appliance. In this particular case, in order to perform the evaluation of the methodologies introduced in Chapter 4, virtualization of the generated network environment has allowed for commercially available vulnerability and network scanners such as Tenable Nessus, Greenbone OpenVas and Nmap, to be used against each instance of host configuration, thus mimicking to a degree the result of a vulnerability and network scan in a physical testing environment.

The proposed methodology is shown in Figure 5.1, and can be decomposed into a stack of four sequential layers.

- **The first layer** of the stack (**store**) is composed of a collection of procedures necessary to automatically install operating systems and applications. Examples of these procedures include shell scripts for applications to be installed on linux based operating systems.

- **The second layer** of the stack (**assembler**) is tasked with assembling a single prototype of a host. Each prototype is defined by a *type* as well as a *plausible* configuration of an operating system and various applications, starting from a *desiderata*. More particularly, this layer is tasked with deterministically selecting the procedures to install one operating system and multiple compatible applications from the "store" layer. Once the procedures have been selected, this layer will compile them into one single procedure, which will constitute the prototype of a host. Both *plausibility* and partly *variety* are handled at this layer: *Plausibility* is enforced by matching the desired type of the prototype host with the compatible types of operating systems and applications collected by the "store" layer, as well as modulating the quantity of applications to be included in a single configuration of a prototype of a host. *Variety* is partly enforced by deterministically selecting only a sub-set of "plausible" applications to be included in each prototype's configuration. This allows different prototypes to be potentially assembled with different applications.

- **The third layer** (**network composer**) is tasked with selecting the high level composition of the prototypes that will compose the network environment. More particularly, this layer handles the *variety* of the network environment by deterministically selecting the number of hosts to be generated, the amount and distribution of the prototypes of hosts to be included in the network environment, and lastly the "desiderata" of each prototype.[1]
- **The fourth layer** (**orchestrator**) is tasked with handling the instantiation of the network environment into a virtualization engine in order to be accessed by vulnerability and network scanners.

It is important to note that while it would be technically possible to simulate complex network architectures and configurations (e.g. firewalls, VPNs) using common virtualization engines, the current iteration of the proposed methodology constructs a simple mesh network, assuming a worst-case scenario, i.e. it is assumed that the attacker (exemplified by vulnerability and network scanners) has full knowledge and reachability over the network. This is operatively implemented by having all the virtualized network environment reside on the same subnet, and by ensuring routing between any two hosts of the network environment.[2] Support for the generation of complex network topologies and architectures is a topic for future work.

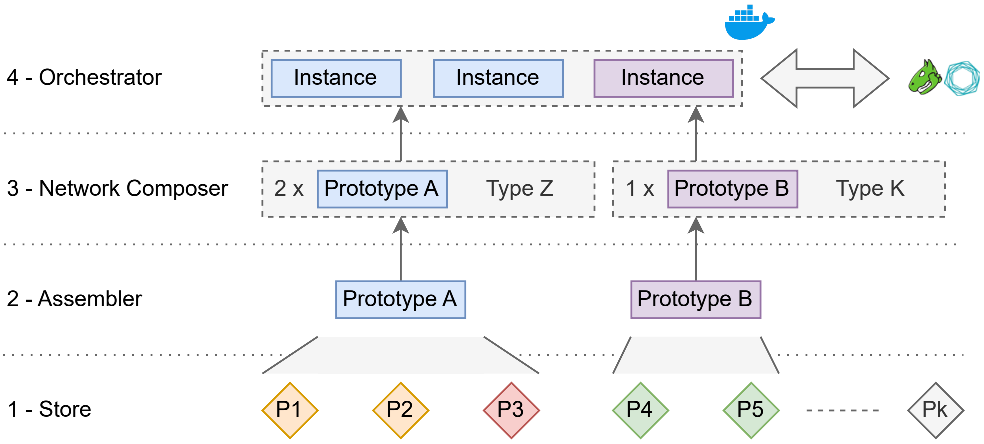


Figure 5.1. Architecture of the stack of the proposed methodology.

5.1.1 Generating a Case Study

In order to generate a case study to perform the evaluation of the methodologies introduced in Chapter 4, an implementation of this methodology has been realized using python. Docker has been chosen as the virtualization engine to support the

[1] Note that if the desired amount of applications is equal or greater than the amount of available applications for that host type, the second layer will not affect the variety of the network environment, since the selected subset will coincide with the full set of compatible applications.

[2] Custom routing rules are injected into each host by the *network composer*. All commercially available virtualization engines support the emulation of local networks and routing to various degrees of accuracy.

"orchestrator" layer. This implementation currently includes a total of 24 applications and their dependencies, four different types of hosts, various versions of two different Linux operating systems,[3] two different vulnerability scanners[4] and one network scanner.[5] The four different categorizations (types) of hosts and applications that have been implemented for this case study are the following:

- **Office Hosts**
  This category of hosts has been configured to include platforms typically present in an office environment. As such, these applications are mostly local and not exposed to the network. An example of such platforms includes web browsers, mail clients and word processors.

- **Server Hosts**
  This category of hosts has been configured to include platforms which are typically present in server environments. These services are prevalently exposed to the network (through port binding). Examples of such platforms include database management systems, web servers and other network based services (ftp, ssh, samba).

- **Development Hosts**
  This category of hosts has been configured to include tools, frameworks and libraries usually found in development environments. Such applications are usually part of very complex stacks, which support compilation and interpretation of code. Examples of such platforms include compilers, frameworks and code interpreters.

- **Generic Hosts**
  The last category of hosts has been configured to include platforms regardless of their scope. This category contains the broadest spectrum of platforms, encompassing local and network services, as well as frameworks and stacks.

In order to generate a sound network environment to use as a case study, two instances of each host category have been created, one for each operating system included in the realized implementation, Ubuntu Linux and Debian Linux. The resulting environment is summarized in Table 5.1.

Table 5.1. Host OS configuration

Host	Type	OS	OS Version	
H0	Office	Ubuntu Linux	22.10	Kinetic
H1	Office	Debian Linux	10	Buster
H2	Server	Ubuntu Linux	18.04.6 LTS	Bionic
H3	Server	Debian Linux	10	Buster
H4	Development	Ubuntu Linux	15.10	Wily
H5	Development	Debian Linux	11	Bullseye
H6	Generic	Ubuntu Linux	15.04	Vivid
H7	Generic	Debian Linux	11	Bullseye

The network vulnerability scanners that have been included in this case study are Tenable's Nessus and Greenbone OpenVas. By merging the output of the two vulnerability scanners, a third synthetic vulnerability scanner has also been obtained

[3] Two versions of debian linux, 17 versions of ubuntu linux.

[4] Greenbone Openvas and Tenable Nessus.

[5] Nmap.

and included in the case study. During the experimental evaluation, it has been observed that this third synthetic vulnerability scanner has the capability to detect more vulnerabilities than each real scanner in isolation. This is due to the fact that the vulnerability scanners Tenable's Nessus and Greenbone OpenVas operate using two different proprietary advisory and vulnerability feeds. This in turn leads to the observation that the vulnerability scanners cover different sets of CVE vulnerabilities. In particular, it has been observed that Tenable's Nessus covers 75486 CVE (33.64% of NVD) over 162428 advisories, while Greenbone OpenVas covers 16311 CVE (7.27% of NVD) over 3433 advisories. Moreover, both Tenable's Nessus and Greenbone Openvas cover the same 16301 CVE. This means that Tenable's Nessus covers 59185 "exclusive"[6] CVE, while Greenbone OpenVas covers 10 "exclusive" CVE.

The difference in CVE coverage, as well as the almost-inclusion of the CVE of one vulnerability scanner by the other can be explained by the fact that the two solutions use different advisory and vulnerability feeds. Tenable's Nessus uses proprietary advisories, aimed at businesses and security professionals. Greenbone offers two advisory feeds: a paid solution aimed towards large businesses and a community solution aimed at security professionals. Due to the fact that Openvas (i.e. the scanner used in this test case) only supports community feeds, the amount of advisories and their coverage is affected.

The preliminary results of a complete scan of the generated environment from each network vulnerability scanner can be summarized by Table 5.2.

Table 5.2. Vulnerability Scanner Results

Host	Scanner	#Advisory	#CVE
H0	nessus	38	7
	openvas	2	2
	merged	40	7
H1	nessus	48	14
	openvas	3	4
	merged	51	16
H2	nessus	83	8
	openvas	5	9
	merged	88	15
H3	nessus	82	3
	openvas	5	9
	merged	87	10
H4	nessus	46	2
	openvas	50	218
	merged	96	218
H5	nessus	44	2
	openvas	2	2
	merged	46	3
H6	nessus	122	111
	openvas	59	237
	merged	181	341
H7	nessus	102	13
	openvas	4	7
	merged	106	19

It is important to notice that, even if Tenable's Nessus's advisories cover almost every CVE vulnerability covered by Greenbone's Openvas's advisories, the differences in their probing mechanisms may lead to different detections (as evident from Table 5.2). These detections have been manually analyzed, and no prevalence of false positives have been found in one or the other vulnerability scanner, i.e. the two

[6] "Exclusive" intended as not covered by any other considered vulnerability scanner.

vulnerability scanners have the same chance of being right or wrong, no vulnerability scanner is expected to be more “faulty” (less accurate) than the other. Therefore, in order to achieve the best possible *completeness* of the automatically generated vulnerability inventory $\mathcal{VI}$ it is necessary to consider the “merged” vulnerability scanner as the baseline for further analysis. As a reminder, the completeness of an inventory $\mathcal{I}$ has been defined during the problem setting in Section 4.1 of Chapter 4 as

$$C(\mathcal{I}) = 1 - \frac{\#false\ negatives}{\#real\ entity\ in\ the\ inventory};$$

Therefore the result from both vulnerability scanners is needed to ensure the least amount possible of *false negatives*, or conversely to ensure that all the vulnerabilities in each host h_i actually appear into the vulnerability inventory $\mathcal{VI}$. It must be noted that there is always the probability that even merging the output of multiple vulnerability scanners may not be enough to correctly identify every vulnerability residing within a host h_i. This can be attributed to different factors, examples of which are the probing-mechanism itself (i.e. if it can’t be probed, it can’t be detected) as well as the existence of zero-day vulnerabilities, i.e. vulnerabilities which are not widely known and for which proper detection methods have not been implemented yet.

It is also interesting to note how in this case study hosts using Ubuntu linux as their operating system have on average more advisories and vulnerabilities (CVE) than hosts using Debian linux. This is due to the availability of older versions of the Ubuntu linux operating system in the Docker ecosystem, as opposed to Debian linux, which only makes available its two most recent versions (Debian 10 and 11). This is in turn reflected on the package manager’s ability to fetch up-to-date and vulnerability free applications, thus explaining the bias in the number of advisories and vulnerabilities found by the network vulnerability scanners between the two operating systems.

A mitigation for this quirk has been attempted by forcing the linux package manager to fetch older versions of applications, but during preliminary tests this mitigation has been discarded, as the advantage gained didn’t seem to justify the added complexity.[7]

As a last step before running the evaluation, a “raw” vulnerability inventory $\mathcal{VI}_{raw}$ and the device inventory $\mathcal{DI}_{raw}$ has been constructed for each host h_i using the output of the “merged” vulnerability scanner. In particular, the vulnerability inventory $\mathcal{VI}_{raw}$ of a given host h_i has been constructed by collecting the vulnerabilities in CVE format referenced by the advisories assigned by the “merged” vulnerability scanner to host h_i. Using a similar logic, the device inventory $\mathcal{DI}_{raw}$ of a given host h_i has been constructed by collecting the platforms in CPE format related to the vulnerabilities collected in the vulnerability inventory $\mathcal{VI}_{raw}$ for host h_i. This ensures that the device inventory $\mathcal{DI}_{raw}$ covers every platform referenced by any vulnerability in the vulnerability inventory $\mathcal{VI}_{raw}$, for a given host h_i. The ground truth $\mathcal{DI}_{true}$ has been constructed using tools provided by the Docker virtualization engine,[8] in conjunction with manual inspection. Similarly, The ground truth $\mathcal{VI}_{true}$ has been constructed by verifying applicability criteria using $\mathcal{DI}_{true}$ as input, as well as through manual inspection. Due to the difficulty involved in their detection, *false negatives* have not been factored in the ground truth $\mathcal{VI}_{true}$. Thus, the *completeness*

[7] Broken dependencies, tar files behind servers that are no longer online, lack of documentation for older versions of software, all on top of the “usual” headaches.

[8] See “docker sbom” instruction.

$C(\mathcal{I})$, with $\mathcal{I}$ being every proposed variation of $\mathcal{VI}$ and $\mathcal{DI}$ is always assumed to be equal to 1 for the remainder of this case study. Detecting and handling *false negatives* in the inventories and analyzing their impact on subsequent cyber risk management processes is interesting ground for future work.

5.2 Vulnerability Filtering

Section 4.3 proposed a methodology aimed at increasing the quality of the automatically generated inventories starting from "raw" vulnerability and device inventories $\mathcal{VI}_{raw}$ and $\mathcal{DI}_{raw}$ provided by vulnerability and network scanners.

To evaluate the proposed methodology and its prioritization approach, several other prioritization approaches with different levels of complexity have been implemented and tested using network and vulnerability inventories relating to the environment of the case study described in Section 5.1.

In particular, the following five strategies have been implemented and analyzed:

- **RN** - RaNdom
  This strategy entails the random sorting of the platforms without any further enhancement.

- **SR** - Smart Random
  This strategy still uses the random sorting of the platforms. However, it evaluates if a platform is still needed to validate at least one vulnerability. If not, the strategy skips the platform.

- **VS** - Vulnerability Severity
  This strategy assumes that vulnerabilities are sorted according to a defined criteria. The strategy ensures that only platforms of the highest ranking vulnerabilities are chosen first, in a random order. Once the highest ranking vulnerabilities have been validated or discarded, the strategy moves on to the platforms pertaining to the next highest ranking vulnerabilities. By all means and purposes, this strategy advances on a per-vulnerability basis. As per the Smart Random strategy, this strategy also evaluates if a platform is still needed to validate the considered vulnerability. In the proposed case study, vulnerabilities are sorted in descending order with respect to their CVSS Base Score.

- **PP** - Platform Prioritization
  This strategy applies the platform sorting using Algorithm 4 described in Section 4.3. This strategy prioritizes platforms that have the highest potential to contribute to the validation or invalidation of a vulnerability.

- **VP** - Vulnerability severity then Platform prioritization
  This strategy sorts the vulnerability as described in Vulnerability Severity, then ranks platforms pertaining to vulnerabilities of the same severity using the platform prioritization strategy described in Section 4.3, considering only the CPE topology of the vulnerabilities of equal severity.

It should be noted that none of the strategies are completely deterministic: in *RN* and *SR* the platforms follow a random order (i.e. no sorting is applied), *VS* performs a sorting on the vulnerabilities according to their scoring, and then uses a random order for the platforms pertaining to vulnerabilities with the same scoring, and *PP* and *VP* may choose at random if multiple platforms have the same score.

For this reasons, 50 instances have been generated for each strategy in order to compare statistically relevant results.

An important note is that vulnerability repositories such as NVD, usually represent contiguous product versions as version ranges through the use of metadata associated to a CPE string. This notation is not part of NIST's standard for CPE, and therefore should not be considered at this time. Therefore for the purposes of this evaluation it has been necessary to expand the instances of platform version ranges into single versions of platforms. For this case study, 301 CVE vulnerabilities (82.69% of the total 364 CVEs in the case study) have at least one platform version range associated. In the case of NVD, NIST provides a public API able to return the list of discrete versions contained by a version range. In total, 408 platform version ranges (27.06% of the total 1508 CPEs in the case study) have been unpacked into 5710 platforms for the considered case study.

Note that this is the intended workflow when operating with NIST's NVD repository. In the next section an optimization will be discussed to address potential inefficiencies of this approach.

Table 5.3. Result of the complete validation of all hosts of the case study.

Host		Vulnerabilities - CVE		Platforms - CPE	
	Total	7		96	
H0	Confirmed	2	(28.6%)	2	(2.1%)
	Discarded	5	(71.4%)	94	(97.9%)
	Total	16		1013	
H1	Confirmed	8	(50.0%)	3	(0.3%)
	Discarded	8	(50.0%)	1010	(99.7%)
	Total	15		1216	
H2	Confirmed	6	(40.0%)	5	(0.4%)
	Discarded	9	(60.0%)	1211	(99.6%)
	Total	10		1136	
H3	Confirmed	1	(10.0%)	1	(0.1%)
	Discarded	9	(90.0%)	1135	(99.9%)
	Total	218		2108	
H4	Confirmed	211	(96.8%)	10	(0.5%)
	Discarded	7	(3.2%)	2098	(99.5%)
	Total	3		34	
H5	Confirmed	1	(33.3%)	1	(2.9%)
	Discarded	2	(66.7%)	33	(97.1%)
	Total	341		6048	
H6	Confirmed	321	(94.1%)	31	(0.5%)
	Discarded	20	(5.9%)	6017	(99.5%)
	Total	19		1604	
H7	Confirmed	5	(26.3%)	3	(0.2%)
	Discarded	14	(73.7%)	1601	(99.8%)

Table 5.3 shows the result of the *complete validation* of every host in the case study described in Section 5.1. Within the context of this chapter, it is opportune to define the *complete validation* of a host h_i as the validation (i.e. filtering process) of both its “raw” vulnerability inventory $\mathcal{VI}_{raw}$ as well as its “raw” device inventory $\mathcal{DI}_{raw}$, derived from the output of the vulnerability scanners. As a reminder, hosts

H0 and H1 represent *Office* hosts, H2 and H3 represent *Server* hosts, H4 and H5 represent *Development* hosts and H6 and H7 represent *Generic* hosts.

For host H0 out of a total of 7 detected vulnerabilities, only 2 (28.6%) are actually confirmed by the proposed methodology, and the remaining 5 (71.4%) are false positives. Indeed only 2 (2.1%) platforms out of the 96 referenced by the vulnerabilities of H0 actually reside in the host. The proposed methodology confirms 8 (50.0%) vulnerabilities on host H1, out of a total of 16 detected vulnerabilities, as the remaining 8 (50.0%) are false positives. Indeed only 3 (0.3%) platforms out of the 1013 referenced by the vulnerabilities of H1 actually reside in the host. For host H2 out of a total of 15 detected vulnerabilities, only 6 (40.0%) are actually confirmed by the proposed methodology, and the remaining 9 (60.0%) are false positives. Indeed only 5 (0.4%) platforms out of the 1216 referenced by the vulnerabilities of H2 actually reside in the host. For host H3 out of a total of 10 detected vulnerabilities, only 1 (10.0%) is actually confirmed by the proposed methodology, and the remaining 9 (90.0%) are false positives. Indeed only 1 (0.1%) platform out of the 1136 referenced by the vulnerabilities of H3 actually reside in the host. The proposed methodology confirms 211 (96.8%) vulnerabilities on host H4 out of a total of 218 detected vulnerabilities, as the remaining 7 (3.2%) are false positives. Surprisingly only 10 (0.5%) platforms out of the 2108 referenced by the vulnerabilities of H4 actually reside in the host. For host H5 out of a total of 3 detected vulnerabilities, only 1 (33.3%) is actually confirmed by the proposed methodology, and the remaining 2 (66.7%) are false positives. Indeed only 1 (2.9%) platform out of the 34 referenced by the vulnerabilities of H5 actually reside in the host. For host H6 out of a total of 341 detected vulnerabilities, 321 (94.1%) are actually confirmed by the proposed methodology, and the remaining 20 (5.9%) are false positives. Indeed only 31 (0.5%) platforms out of the 6048 referenced by the vulnerabilities of H6 actually reside in the host. Lastly, the proposed methodology confirms only 5 (26.3%) vulnerabilities for host H7 out of a total of 19 detected vulnerabilities, as the remaining 14 (73.7%) are false positives. Indeed only 3 (0.2%) platforms out of the 1604 referenced by the vulnerabilities of H7 actually reside in the host.

The fist striking observation is that the number of confirmed platforms associated with the vulnerabilities detected by vulnerability scanners on a given host h_i is very low ($\leq 2.9\%$). This phenomenon has also been observed to have no apparent correlation with the percentage of false positives (i.e. discarded vulnerabilities) identified by the proposed methodology. Manual inspection of the vulnerabilities and their associated platforms has found that many vulnerabilities cover wide technology stacks, e.g. protocols or system libraries embedded in applications as well as operating systems, and as such, may cover a wide range of platforms. An example of this is found in H4, for which several vulnerabilities related to a version of Java Development Kit (JDK) also affect other 2107 platforms, which refer to platforms integrating the vulnerable component in their architecture, as well as operating systems that ship with the vulnerable component as part of their system stack. To provide another example, manual inspection has found that vulnerabilities detected on host H3 are associated to 312 different platforms which identify various, mutually exclusive, operating systems and firmware, as well as 42 different platforms which identify various hardware components which are incompatible with each other.

The second observation is that there is no apparent correlation between the precision of a scanner and the considered host. As an example, hosts H0, H2, H5 and H7 hover around the same 60% false positive rate, even tough they belong to different categories of hosts (thus having different categories of platforms). To give another example, despite the fact that H1, H3, H4 and H6 have been equipped with older versions of platforms and operating systems, their ratio of false positives swings from one extreme (90.0%) to the other (3.2%).

It is important to remind that all proposed strategies are capable by design of reaching a complete vulnerability validation state. The difference between each proposed strategy is in the ordering of the platforms to be validated, thus it becomes interesting to analyze the number of steps needed for each strategy to converge to the result shown in Table 5.3.

Table 5.4. Results of the application of the strategies on the case study

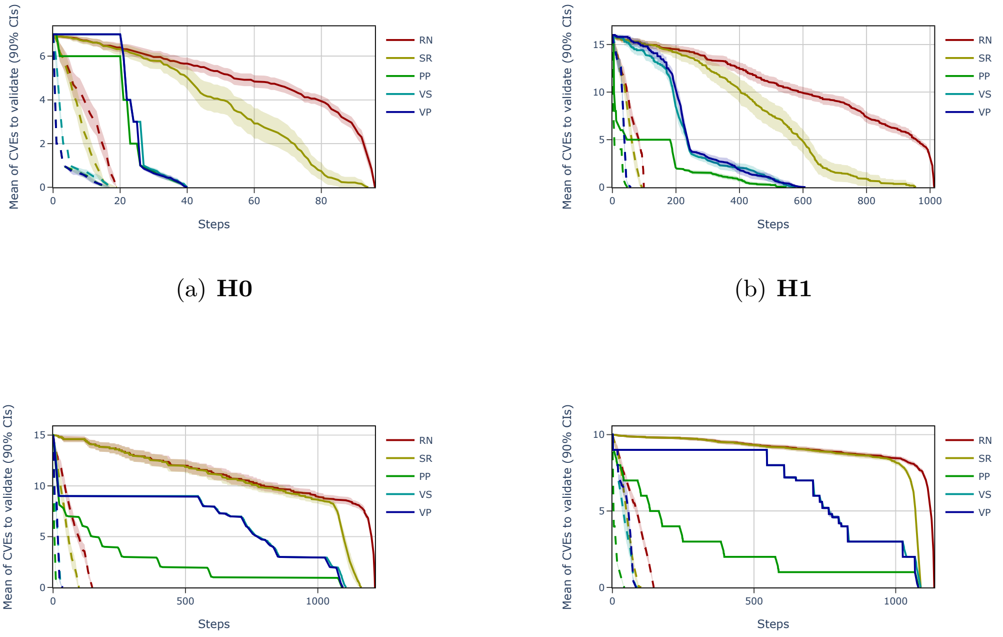
Host	Metric	Strategies				
		RN	SR	PP	VS	VP
H0	*Ci1*	94.62	67.50	34.68	35.26	34.48
	Mean	94.02	63.38	33.68	34.32	33.40
	Ci0	93.42	59.26	32.68	33.38	32.32
H1	*Ci1*	1009.18	614.45	389.60	428.04	434.56
	Mean	1008.14	577.16	366.86	401.36	409.96
	Ci0	1007.10	539.87	344.12	374.68	385.36
H2	*Ci1*	1215.87	1122.71	1084.39	1095.75	1086.23
	Mean	1215.74	1117.04	1083.28	1093.80	1084.88
	Ci0	1215.61	1111.37	1082.17	1091.85	1083.53
H3	*Ci1*	1135.96	1082.47	1076.32	1081.85	1074.74
	Mean	1135.88	1081.30	1075.44	1080.76	1073.94
	Ci0	1135.80	1080.13	1074.56	1079.67	1073.14
H4	*Ci1*	2084.62	1020.15	590.88	782.74	637.39
	Mean	2078.58	953.26	543.08	735.36	577.08
	Ci0	2072.54	886.37	495.28	687.98	516.77
H5	*Ci1*	34.54	29.59	28.91	28.57	29.30
	Mean	34.32	28.72	27.96	27.62	28.36
	Ci0	34.10	27.85	27.01	26.67	27.42
H6	*Ci1*	6035.24	3063.92	991.11	1653.58	1304.73
	Mean	6030.84	2918.10	960.66	1610.24	1262.62
	Ci0	6026.44	2772.28	930.21	1566.90	1220.51
H7	*Ci1*	1603.93	1469.23	1321.78	1335.11	1329.46
	Mean	1603.84	1452.56	1320.96	1333.92	1328.06
	Ci0	1603.75	1435.89	1320.14	1332.73	1326.66

Table 5.4 shows the mean of the number of steps to validate all the vulnerabilities (90% Confidence Intervals CIs) in every host of the case study described in Section 5.1, for each strategy. As a reminder, hosts H0 and H1 represent *Office* hosts, H2 and H3 represent *Server* hosts, H4 and H5 represent *Development* hosts and H6 and H7 represent *Generic* hosts.

For host H0, all strategies except *RN* and *SR* obtain similar results requiring about ≈ 34 steps to validate all the vulnerabilities, while *RN* and *SR* necessitate ≈ 94 and ≈ 63 steps respectively. In the case of host H1, vulnerability-based strategies *VS* and *VP* require about ≈ 405 steps to terminate, *PP* requires ≈ 367 steps, *RN* requires ≈ 1008 steps, and *SR* necessitates ≈ 577 steps. Host H2 follows a similar trend to H0, in which all strategies except *RN* and *SR* require about ≈ 1087 steps to terminate, while *RN* and *SR* necessitate ≈ 1216 and ≈ 1117 steps respectively. All strategies of host H3 with the exception of *RN* require ≈ 1077 steps to terminate, while RN necessitates ≈ 1136 steps. In host H4, platform-driven strategies PP and VP require ≈ 550 steps to terminate, VS requires ≈ 735 steps, SR requires ≈ 953 steps and RN requires ≈ 2079 steps. For host H5, all strategies except RN require ≈ 28 steps to terminate, while RN necessitates ≈ 34 steps. Host H6 follows a similar trend to H1, for which vulnerability-based strategies VS and VP require about ≈ 1400 steps to terminate, PP requires ≈ 960 steps, RN requires ≈ 6031 steps, and SR necessitates ≈ 2918 steps. Lastly, host H7 requires ≈ 1330 steps for all strategies except RN and SR, which necessitate ≈ 1604 and ≈ 1453 steps respectively.

Notice that the difference in size with respect to the number of steps is proportional with the number of CVE present in each host, e.g. H6 has both the most detected CVE vulnerabilities and steps required for termination among all other hosts. Conversely, H5 has the least amount of detected CVE vulnerabilities as well as the least amount of steps required for termination among all other hosts. This is due to the fact that platform configurations of different CVEs may not overlap completely, i.e. each CVE may be enabled by a different set of platforms. This phenomenon has been experimentally observed to be more present in the case of CVEs linked by different advisories, as it is usual for an advisory to reference either (i) a single issue spanning multiple products or (ii) multiple issues regarding a single or a family of products.

At a first glance, a meaningful difference between random (RN and SR) and



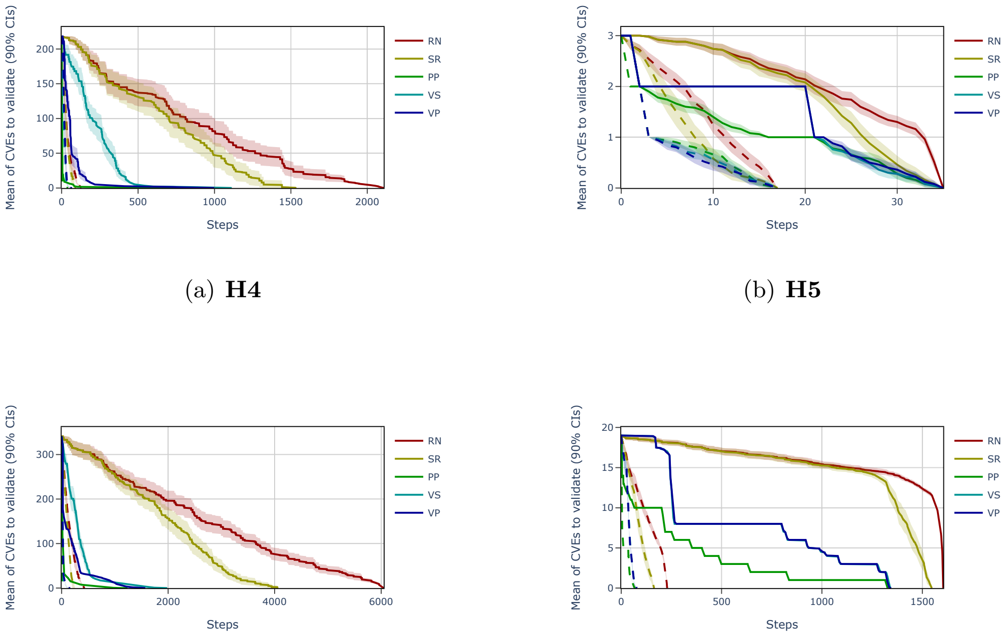
(a) **H0**

(b) **H1**

(c) **H2**

(d) **H3**

Figure 5.2. Mean of the CVEs that still have to be validated after each step of the prioritization strategies. H0 and H1 represent *Office* hosts, H2 and H3 represent *Server* hosts. Dashed lines represent validations carried out by using G_i^{NoVe} as input, thus truncating the CPE string at the *product* attribute and ignoring versions.



(a) **H4**

(b) **H5**

(c) **H6**

(d) **H7**

Figure 5.2. Mean of the CVEs that still have to be validated after each step of the prioritization strategies. H4 and H5 represent *Development* hosts, H6 and H7 represent *Generic* hosts. Dashed lines represent validations carried out by using G_i^{NoVe} as input, thus truncating the CPE string at the *product* attribute and ignoring versions.

semantic driven strategies (*PP*, *VS* and *VP*) can be immediately noticed for hosts H1, H4, H6 and H7. Hosts H0, H2, H3 and H5 do not apparently exhibit any difference between strategies. This is due to (i) H0 and H5 having few vulnerabilities insisting on few, distinct platforms, (ii) H2 and H3 having vulnerabilities having large OR configurations of distinct platforms which are incompatible with their respective hosts, and (iii) the fact that the result shown in the table is the final state of the application of the proposed strategies, i.e. the average number of steps necessary to validate or discard all vulnerabilities in $\mathcal{VI}_{raw}$. Since these reasons contribute to the lack of conclusive evidence over the differences between the different strategies applied to these hosts, it is interesting to analyze the strategies' evolution over the number of steps. Thus, Figure 5.2 has been plotted to visually analyze the average number of CVE vulnerabilities that still have to be validated after x steps of each strategy.

The first result that can be noticed by analyzing the figure is that despite the steps taken to completely validate or discard each vulnerability are comparable for hosts H0, H2, H3 and H5, as shown in Table 5.7, "random" approaches *RN* and *SR* present the worst evolution with respect to every other considered strategy. This is easily noticed in the figure and confirmed in Table 5.5, for which the number of vulnerabilities still left to validate does not decrease significantly (below 50%) until 76.7% and 68.3% of the total elapsed steps, for *RN* and *SR* respectively.[9] This

[9] 86.0% and 76.6% for ≥ 66%, 95.2% and 87.5% for ≥ 90%

is expected, since these approaches do not consider any semantics while deciding the order of the platforms to submit to the human agent, and therefore proposed platforms may not be significantly impactful with respect to vulnerabilities in the inventory. Also, as expected, strategy *RN* performs consistently worse than strategy *SR* since *RN* does not remove "non-useful" platforms from the ordering.

With respect to "vulnerability-driven" approaches *VS* and *VP*, the evolution of the proposed strategies across all the considered hosts highlights a limitation, as they are characterized by a staircase-like trend. Manual inspection of the results has determined the cause to be the presence of many OR branches in the CPE configuration trees of certain CVEs. Examples of CVEs which have been found to be blocking for "vulnerability-driven" approaches are: *CVE-2015-2808* which contains 194 unique CPE strings across 53 configuration trees, the biggest of which is an OR of 60 CPEs, *CVE-2007-1858* which contains a single configuration tree which is a single OR of 43 CPE strings and *CVE-2022-23305* which contains 171 unique CPE strings organized in 5 configuration trees, the biggest of which is an OR of 130 elements. In these platform configurations, it is typical that only one (or none) of the CPEs in the OR is really present in the host, and its identification requires multiple steps. For this reason, strategies that validate vulnerabilities sequentially are impacted significantly by this type of vulnerability. Another interesting finding is that most of the time, the two approaches *VS* and *VP* share the same evolution. This is due to the fact that these approaches order the vulnerabilities first, and then perform ordering of the related platforms. Also in this case manual inspection of the results has been carried out, and it has been found that vulnerabilities with the same CVSS base score (i.e. ranked equally by the vulnerability driven sorting) rarely share platforms. This in turn means that the secondary ordering strategy included in *VP* never comes into play, defaulting to a random ordering, which is exactly what *VS* does.

The proposed "platform-semantic-driven" approach *PP* has consistently the best evolution over the number of steps, across all the hosts of the case-study. This is expected, since *PP* has been designed to prioritize the platforms that have the highest potential to impact a vulnerability, whether by validating or discarding it, as per Algorithm 4 of Section 4.3. In particular as shown in Table 5.5, *PP* in average validates 50% of the vulnerabilities after just 14.7% of the total steps required to perform a complete validation across all the hosts, while its closest competitors *VS* and *VP* use significantly more steps (41.2% and 38.6% respectively) to achieve the same result. The same trend can be observed for the validation of the last 33% and 10% of vulnerabilities, for which *PP* consistently outperforms the other considered strategies.[10]

Table 5.5. Mean percentage of steps required for each strategy to achieve respectively $\geq 50\%$, $\geq 66\%$ and $\geq 90\%$ accuracy of the vulnerability inventory $\mathcal{VI}$ (i.e. validation progress), across every host.

Strategy	**CVE validated**		
	50%	66%	90%
RN	76.7%	86.0%	95.2%
SR	68.3%	76.6%	87.5%
PP	14.7%	23.3%	47.6%
VS	41.2%	52.4%	74.5%
VP	38.6%	49.4%	70.2%

[10] *PP* achieves 66% and 90% validation within an average of 23.3% and 47.6% of the total amount of steps required for termination, respectively.

The reason for *PP*'s comparable performance with respect to other ordering strategies in Table 5.4 is due to the presence of CVEs with many OR branches in the CPE configuration trees which do not share CPEs with other vulnerabilities. This heavily degrades the performance of Algorithm 4 (Section 4.3, implemented by strategy *PP*), which has been designed to exploit platform frequency and configuration tree structure. Thus it is inevitable that in the absence of platform frequency and in the presence of a "flat" OR configuration tree structure, the performance of the proposed algorithm degrades. As an example, between steps 600 and 1200, *PP* on *H2* takes about 500 steps to discard *CVE-2013-2566*, a vulnerability with 552 platforms across 3 different platform configuration trees, the biggest of which is an OR tree of 487 different platforms. Since this vulnerability is not present in *H2*, any strategy has to discard all the platforms in every flat OR configuration in order to discard the vulnerability.

5.2.1 Comments on Complexity and Accuracy

An important notion also highlighted by Figure 5.2 is the difference in the number of steps needed to bring each strategy to conclusion using either the vulnerability graph G_i (identified by contiguous lines) or G_i^{NoVe} (the vulnerability graph that ignores CPE versions, identified by dashed lines) as input. This difference is substantial, as it can be easily seen that in each and every case, every strategy that uses the vulnerability graph G_i as input (i.e. G_i considers the CPE version attribute) will always require significantly more steps to terminate with respect to each and every strategy that uses G_i^{NoVe} as input. This is mainly due to the difference in size between the two vulnerability graphs, G_i^{NoVe} and G_i. Indeed, as a reference, the number of distinct CPE strings that are part of G_i amount to from 105.88% up to 1546.09% those of G_i^{NoVe}, depending on the host,[11] while the overall size of G_i is from 90.00% to 866.67% larger than G_i^{NoVe}.

The increase in complexity brought by the version attribute of CPE strings, while costly for the number of steps needed to bring closure, is fundamental to guarantee a meaningful *accuracy* of the inventories produced by the methodology described in Section 4.3.

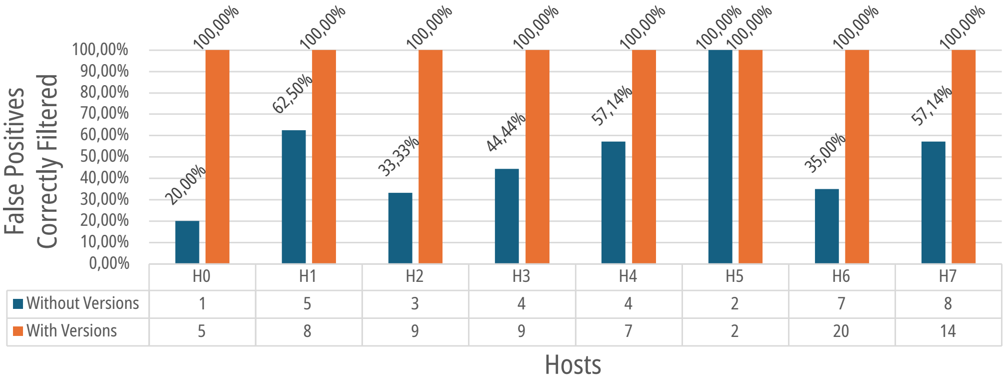
As a reminder, Section 4.1 introduced the notion of *accuracy* of an inventory I as

$$A(\mathcal{I}) = 1 - \frac{\#false\ positives}{\#real\ entity\ in\ the\ inventory}.$$

Assuming perfect accuracy of the human agent, as well as the correctness of the applicability clauses (CPE configuration trees) of each vulnerability in CVE format, the proposed methodology will always achieve *max* $A(\mathcal{I}^+)$ with $\mathcal{I}^+$ being the result of the application of the methodology on the automatically generated device and vulnerability inventories $\mathcal{DI}$ and $\mathcal{VI}$ (i.e. the methodology will correctly identify and exclude false positives from both the device and vulnerability inventories). However, it has been observed experimentally that using the vulnerability graph G_i which includes the version attribute of CPE strings consistently achieves inventories with a greater or equal accuracy ($A(\mathcal{I})$ with $\mathcal{I}$ being $\mathcal{DI}$ and $\mathcal{VI}$) if compared to the result obtained excluding the version attribute and using the vulnerability graph G_i^{NoVe}.

Indeed, Figure 5.3 highlights that forgoing version attributes from CPE strings will almost always result in a sizeable introduction of false positives (i.e. vulnerabilities that are confirmed as compatible by the proposed filtering methodology, but are not truly present in the analyzed host) in the final vulnerability inventory $\mathcal{VI}^+$.

[11] Vulnerability graphs G_i and G_i^{NoVe} are always related to a single host h_i.



	H0	H1	H2	H3	H4	H5	H6	H7
■ Without Versions	1	5	3	4	4	2	7	8
■ With Versions	5	8	9	9	7	2	20	14

Figure 5.3. False positives eliminated from the vulnerability inventories after applying Algorithm 4 of Section 4.3, using G_i^{NoVe} and G_i as input.

This is due to the fact that while some versions of a product may be vulnerable, not all versions of a product are. This fact is further aggravated by how many commercially available vulnerability scanners function. As stated during the formalization of the problem, in Section 4.1, many commercially available vulnerability and network scanners rely on probe-based mechanisms in order to identify the presence of vulnerabilities and platforms on a system. It is highly possible that a probe may not be able to accurately discern between two similar but different versions of a product, one potentially vulnerable, one certainly not. This in turn may lead the vulnerability scanner into detecting a different version of a product, possibly introducing false positives in the resulting inventories $\mathcal{VI}_{raw}$ and $\mathcal{DI}_{raw}$. If versions are excluded from the vulnerability graph G_i, the proposed methodology will not recognize the error and will carry on the false positives into the final inventories $\mathcal{VI}^+$ and $\mathcal{DI}^+$. For this exact reason, it becomes necessary to include the optional version attribute of CPE strings in the analysis and thus, it becomes necessary to use the vulnerability graph G_i instead of G_i^{NoVe}, as G_i has the capability to identify and discern between different versions of products, albeit at the cost of increasing the number of steps required for termination. This increased cost is necessary in order to allow the methodology described in Section 4.3 to aim for the best possible accuracy in pruning the number of false positives introduced by the vulnerability scanners' probing based mechanisms.

In conclusion, assuming perfect accuracy of the human agent, as well as the correctness of the applicability clauses (CPE configuration trees) of each vulnerability in CVE format, every ordering strategy implemented over the methodology proposed in Section 4.3 has the potential to improve the accuracy of the vulnerability inventory $\mathcal{VI}_{raw}$ after each step, and given enough steps, has the capability to consistently achieve a perfect accuracy value ($A(\mathcal{VI}^+) = 1$) with respect to the ground truth and the assumptions that govern this case study. This satisfies the claim advanced during the formalization of the problem, in Section 4.1 that given a filtering process, the accuracy of the resulting inventories must be greater or equal than the accuracy of the inventories in input to the process. More formally, the evaluation described in this section has shown that the methodology described in Section 4.3 has the capability to achieve $A(\mathcal{VI}^+) > A(\mathcal{VI}_{raw})$ if $A(\mathcal{VI}_{raw}) < 1$ (i.e. for H0, H1, H2, H3, H4, H6, H7) or $A(\mathcal{VI}^+) = A(\mathcal{VI}_{raw}) = 1$ in the case of H5. Similarly to $\mathcal{VI}$, the methodology is also capable of increasing the accuracy of the device inventory $\mathcal{DI}$, since the proposed methodology confirms or discards one or more platforms from

$\mathcal{DI}$ at each step, thus satisfying $A(\mathcal{DI}^+) \geq A(\mathcal{DI}_{raw})$.

5.3 Optimizing Platform Versions

The previous section provided an experimental evaluation of the methodology aimed at reducing the number of false positives in a vulnerability inventory $\mathcal{VI}_{raw}$. The result of this evaluation exposed a curious behaviour common to all the strategies used during this case study, as it is apparent that for half of the considered case study no strategy terminates significantly "far" from others. And while this is true, interestingly, the different strategies do show different evolutions over the number of steps, with "platform-based" strategy *PP* being consistently the best performing strategy (i.e. first to consistently validate 90% of $\mathcal{VI}_{raw}$) and "random-based" strategies *RN* and *SR* being the worst.

It must be noted, however, that the number of steps required both for termination as well as to reach at least 90% accuracy on $\mathcal{VI}^+$ is very high, with hosts H1, H2, H3 H4, H6 and H7 requiring more than 500 steps on average to validate every vulnerability in $\mathcal{VI}_{raw}$, regardless of the platform ordering strategy used. This is in part a consequence of the expansion of the 408 platform version ranges (27.06% of the total 1508 CPEs) into 5710 platforms for the considered case study. And since the new 5710 platforms increase the original CPE set size of the whole environment by nearly 379%, it becomes of interest to study an approach able to optimize platform version ranges.

Thus, Section 4.4 (Version Optimization) proposes an optimization of the methodology described in Section 4.3 (Vulnerability Filtering), aimed at handling the additional complexity introduced by the inclusion of the optional *version* attribute of a CPE string in the vulnerability graph $\mathcal{G}_i$.

Indeed, as seen in the previous section, the complexity of the vulnerability graph $\mathcal{G}_i$ increases up to 866.67% with respect to a vulnerability graph in which the optional *version* attribute of a CPE string is not considered. The previous section has also shown why it is not advisable to forgo the optional *version* attribute of a CPE string, as doing so may considerably affect the proposed methodology's ability to identify and remove the false positives from "raw" vulnerability and device inventories $\mathcal{VI}_{raw}$ and $\mathcal{DI}_{raw}$ provided by vulnerability and network scanners.

Thus, the additional complexity is required in order to retain accuracy. In order to attempt to reduce this necessary complexity, a two-part optimization has been proposed as a necessary mean to achieve the maximum possible accuracy of the filtering process while trying to counter the increase in complexity which may result from doing so. The proposed optimization achieves the following: (i) it encodes platform version ranges into $\mathcal{G}_i$, and (ii) it divides platform version ranges into smaller platform version ranges in order to facilitate the proposed methodology's platform prioritization mechanics described by Algorithm 4 (strategy *PP*) from Section 4.3 (Vulnerability Filtering).

The evaluation of the proposed optimization and its effects on the proposed methodology has been carried out consistently with the evaluation of the proposed methodology in the previous section (Section 5.2, Vulnerability Filtering). Thus, the two parts of the proposed optimization have been applied to all five platform prioritization strategies (*RN*, *SR*, *PP*, *VS* and *VP*), which have been implemented on inventories derived from the environment described in Section 5.1 (Hosts H0-H7).

As per the previous case, it should be noted that none of the strategies are completely deterministic: in *RN* and *SR* the platforms follow a random order (i.e. no sorting is applied), *VS* performs a sorting on the vulnerabilities according to their scoring, and then uses a random order for the platforms pertaining to vulnerabilities with the same scoring, and *PP* and *VP* may choose at random if multiple platforms have the same score. For this reasons, 50 instances have been generated for each strategy in order to compare statistically relevant results.

Table 5.6. Result of the complete validation of all hosts of the case study.

Host		Vulnerabilities - CVE		Platforms - CPE	
H0	Total	7		27	
	Confirmed	2	(28.6%)	2	(7.4%)
	Discarded	5	(71.4%)	25	(92.6%)
H1	Total	16		198	
	Confirmed	8	(50.0%)	3	(1.5%)
	Discarded	8	(50.0%)	195	(98.5%)
H2	Total	15		333	
	Confirmed	6	(40.0%)	5	(1.5%)
	Discarded	9	(60.0%)	328	(98.5%)
H3	Total	10		321	
	Confirmed	1	(10.0%)	1	(0.3%)
	Discarded	9	(90.0%)	320	(99.7%)
H4	Total	218		461	
	Confirmed	211	(96.8%)	10	(2.2%)
	Discarded	7	(3.2%)	451	(97.8%)
H5	Total	3		17	
	Confirmed	1	(33.3%)	1	(5.9%)
	Discarded	2	(66.7%)	16	(94.1%)
H6	Total	341		1369	
	Confirmed	321	(94.1%)	31	(2.3%)
	Discarded	20	(5.9%)	1338	(97.7%)
H7	Total	19		464	
	Confirmed	5	(26.3%)	3	(0.6%)
	Discarded	14	(73.7%)	461	(99.3%)

Table 5.6 shows the result of the *complete validation* of every host in the case study described in Section 5.1. Within the context of this chapter, it is opportune to refresh the definition of the *complete validation* of a host h_i as the validation (i.e. filtering process) of both its “raw” vulnerability inventory $\mathcal{VI}_{raw}$ as well as its “raw” device inventory $\mathcal{DI}_{raw}$, derived from the output of the vulnerability scanners. As a reminder, hosts H0 and H1 represent *Office* hosts, H2 and H3 represent *Server* hosts, H4 and H5 represent *Development* hosts and H6 and H7 represent *Generic* hosts.

For host H0 out of a total of 7 detected vulnerabilities, only 2 (28.6%) are actually confirmed by the proposed methodology, and the remaining 5 (71.4%) are false positives. Indeed only 2 (7.4%) platforms out of the 27 referenced by the vulnerabilities of H0 actually reside in the host. The proposed methodology confirms 8 (50.0%) vulnerabilities on host H1, out of a total of 16 detected vulnerabilities, as the remaining 8 (50.0%) are false positives. Indeed only 3 (1.5%) platforms out of the 198 referenced by the vulnerabilities of H1 actually reside in the host. For host H2 out of a total of 15 detected vulnerabilities, only 6 (40.0%) are actually confirmed by the proposed methodology, and the remaining 9 (60.0%) are false positives. Indeed only 5 (1.5%) platforms out of the 333 referenced by the vulnerabilities of H2 actually reside in the host. For host H3 out of a total of 10 detected vulnerabilities, only 1 (10.0%) is actually confirmed by the proposed methodology, and the remaining 9 (90.0%) are false positives. Indeed only 1 (0.3%) platform out of the 461 referenced by the vulnerabilities of H3 actually reside in the host. The proposed methodology confirms 211 (96.8%) vulnerabilities on host H4 out of a total of 218 detected vulnerabilities, as the remaining 7 (3.2%) are false positives. Surprisingly only 10 (2.2%) platforms out of the 461 referenced by the vulnerabilities of H4 actually reside in the host. For host H5 out of a total of 3 detected vulnerabilities, only 1 (33.3%) is actually confirmed by the proposed methodology, and the remaining 2 (66.7%) are false positives. Indeed only 1 (5.9%) platform out of the 17 referenced by the vulnerabilities of H5 actually reside in the host. For host H6 out of a total of 341 detected vulnerabilities, 321 (94.1%) are actually confirmed by the proposed methodology, and the remaining 20 (5.9%) are false positives. Indeed only 31 (2.3%) platforms out of the 1369 referenced by the vulnerabilities of H6 actually reside in the host. Lastly, the proposed methodology confirms only 5 (26.3%) vulnerabilities for host H7 out of a total of 19 detected vulnerabilities, as the remaining 14 (73.7%) are false positives. Indeed only 3 (0.6%) platforms out of the 464 referenced by the vulnerabilities of H7 actually reside in the host.

The fist striking observation is that the number of confirmed and discarded vulnerabilities remains the same for the optimized and for the non optimized environment (Section 5.2, Vulnerability Filtering). This is expected, since the optimization proposed in this section aims to reduce the number of steps required for termination, by operating on the platforms. This is also expected, because in order to be effective the proposed optimization should not harm the accuracy of the overall process.

The second observation that can be made is that the number of confirmed platforms associated with the vulnerabilities detected by vulnerability scanners on a given host h_i is very low ($\leq$ 7.4%) in absolute terms, but is definitively higher than the values reported in the non optimized environment ($\leq$ 2.9%) as expected from an overall reduction in the number of platforms in the device inventory $\mathcal{DI}_{raw}$. As per the non optimized environment, this phenomenon has also been observed to have no apparent correlation with the percentage of false positives (i.e. discarded vulnerabilities) identified by the proposed methodology. Manual inspection of the vulnerabilities and their associated platforms has found that many vulnerabilities cover wide technology stacks, e.g. protocols or system libraries embedded in applications as well as operating systems, and as such, may cover a wide range of platforms. An example of this is found in H4, for which several vulnerabilities related to a version of Java Development Kit (JDK) also affect other 481 platforms (2107 in the non optimized environment), which refer to platforms integrating the vulnerable component in their architecture, as well as operating systems that ship with the vulnerable component as part of their system stack. To provide another example, manual inspection has found that vulnerabilities detected on host H3 are associated to 75 different platforms which identify various operating systems and firmware, as well as 42 different platforms which identify various hardware components which are incompatible with each other.

The third observation is that there is no apparent correlation between the precision of a scanner and the considered host. As an example, hosts H0, H2, H5 and H7 hover around the same 60% false positive rate, even tough they belong to different categories of hosts (thus having different categories of platforms). To give another example, despite the fact that H1, H3, H4 and H6 have been equipped with older versions of platforms and operating systems, their ratio of false positives swings from one extreme (90.0%) to the other (3.2%).

It is important to remind that all proposed strategies are capable by design of reaching a complete vulnerability validation state. The difference between each vulnerability is in the ordering of the platforms to be validated, thus it becomes interesting to analyze the number of steps needed for each strategy to converge to the result shown in Table 5.6.

Table 5.7. Results of the application of the strategies on the case study

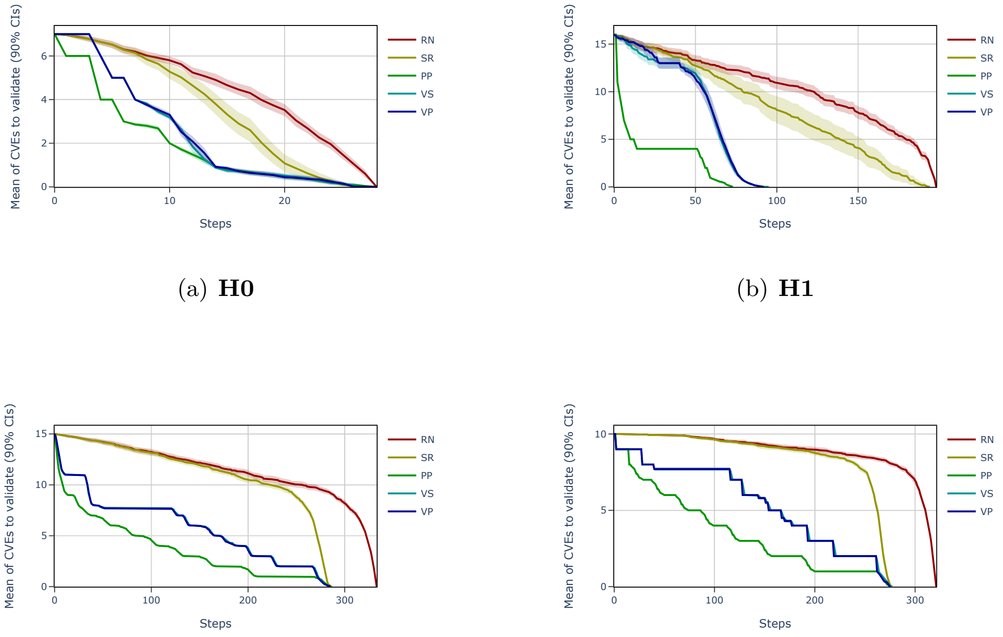
Host	Metric	Strategies				
		RN	SR	PP	VS	VP
H0	*Ci1*	27.36	20.45	21.33	21.17	21.27
	Mean	27.06	19.52	20.26	20.08	20.24
	Ci0	26.76	18.59	19.19	18.99	19.21
H1	*Ci1*	196.51	143.04	67.13	81.59	81.42
	Mean	195.88	133.48	66.12	79.98	79.54
	Ci0	195.25	123.92	65.11	78.37	77.66
H2	*Ci1*	332.47	280.47	278.62	278.22	277.66
	Mean	332.20	279.48	277.60	277.30	276.64
	Ci0	331.93	278.49	276.58	276.38	275.62
H3	*Ci1*	320.65	271.08	269.91	270.79	269.60
	Mean	320.42	270.24	269.16	269.92	268.70
	Ci0	320.19	269.40	269.05	268.41	267.80
H4	*Ci1*	454.62	214.79	119.69	142.77	136.94
	Mean	453.16	201.00	115.50	139.54	132.02
	Ci0	451.70	187.21	111.31	136.31	127.10
H5	*Ci1*	15.02	11.26	11.60	11.44	10.69
	Mean	14.42	10.34	10.72	10.46	9.82
	Ci0	13.82	9.42	9.98	9.48	8.95
H6	*Ci1*	1366.00	821.37	559.76	673.55	668.30
	Mean	1365.24	801.62	553.42	661.04	656.92
	Ci0	1364.48	781.87	547.08	648.53	645.54
H7	*Ci1*	463.61	369.58	313.71	314.20	314.91
	Mean	463.36	362.98	312.76	313.26	314.04
	Ci0	463.11	356.38	311.81	312.32	313.17

Table 5.7 shows the mean of the number of steps to validate all the vulnerabilities (90% Confidence Intervals CIs) in every host of the case study described in Section 5.1, for each strategy. This result has been achieved by encoding platform version ranges into the CPE string, and forgoing their expansion into discrete versions, as opposed by the processing applied in the previous section. Notice that this constitutes only part (i) of the proposed two-part optimization. Part (ii) will be discussed later in the section. As a reminder, hosts H0 and H1 represent *Office* hosts, H2 and H3 represent *Server* hosts, H4 and H5 represent *Development* hosts and H6 and H7 represent *Generic* hosts.

For host H0, all strategies except *RN* obtain similar results requiring about ≈ 21 steps to validate all the vulnerabilities, while *RN* necessitates ≈ 27 steps. Similarly, for host H1, vulnerability-based strategies require about ≈ 81 steps to terminate, *RN* requires ≈ 196 steps, *SR* requires ≈ 133 steps, and *PP* requires ≈ 66 steps. Host H2 follows a trend similar to H0 in which all strategies except *RN* require about ≈ 278 steps to terminate, while *RN* necessitates ≈ 332 steps. All strategies of host H3 with the exception of *RN* require ≈ 270 steps to terminate, while *RN* necessitates ≈ 320 steps. In host H4, vulnerability-based strategies require about ≈ 135 steps to terminate, *RN* requires ≈ 453 steps, *SR* requires ≈ 201 steps, and *PP* requires ≈ 115 steps. For host H5, all strategies except *RN* require ≈ 10 steps to terminate, while *RN* necessitates ≈ 14 steps. Host H6 follows a similar trend to H1 and H4 in which vulnerability-based strategies require about ≈ 657 steps to terminate, *RN* requires ≈ 1365 steps, *SR* requires ≈ 801 steps, and *PP* requires ≈ 553 steps. Lastly, host H7 requires ≈ 313 steps for all strategies except *RN* and *SR*, which necessitate ≈ 463 and ≈ 363 steps respectively.

Notice that the difference in size with respect to the number of steps is still proportional with the number of CVE present in each host, as per the result of the previous section, e.g. H6 has both the most detected CVE vulnerabilities and steps required for termination among all other hosts. Conversely, H5 has the least amount of detected CVE vulnerabilities as well as the least amount of steps required for termination among all other hosts. This is due to the fact that platform configurations of different CVEs may not overlap completely, i.e. each CVE may be enabled by a different set of platforms. This phenomenon has been experimentally observed to be more present in the case of CVEs linked by different advisories, as it is usual for an advisory to reference either (i) a single issue spanning multiple products or (ii) multiple issues regarding a single or a family of products.

At a first glance, Table 5.7 demonstrates empirically that encoding platform



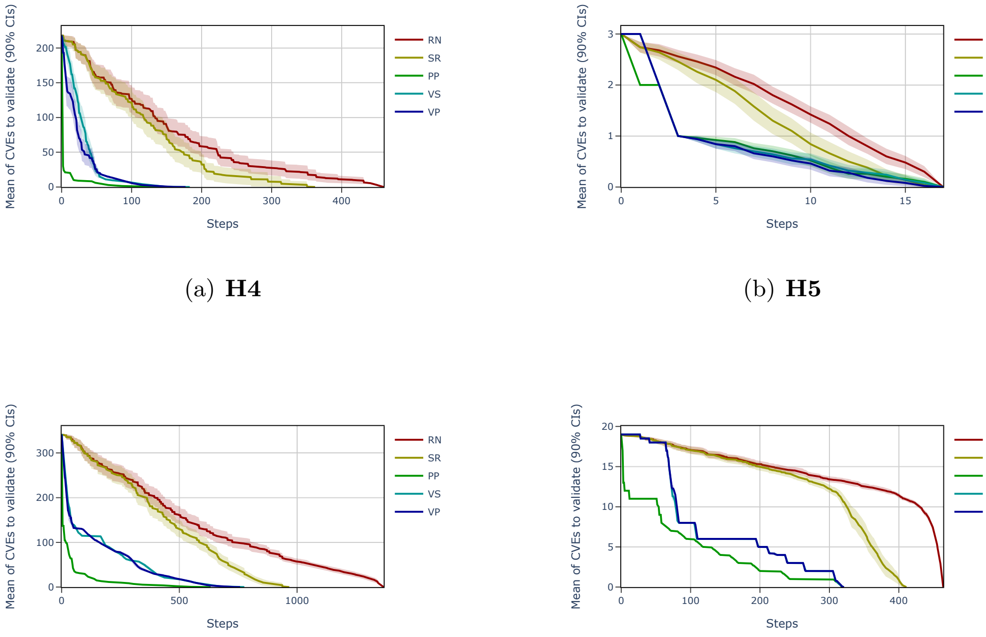
(a) **H0**

(b) **H1**

(c) **H2**

(d) **H3**

Figure 5.4. Mean of the CVEs that still have to be validated after each step of the prioritization strategies. H0 and H1 represent *Office* hosts, H2 and H3 represent *Server* hosts.



(a) **H4**

(b) **H5**

(c) **H6**

(d) **H7**

Figure 5.4. Mean of the CVEs that still have to be validated after each step of the prioritization strategies. H4 and H5 represent *Development* hosts, H6 and H7 represent *Generic* hosts.

version ranges, usually expressed as metadata to be unpacked into discrete platforms, yields a significant performance benefit during the filtering process described in Section 4.3 (Vulnerability Filtering). Indeed, with just this first part of the proposed optimization, only H6 requires more than 500 steps for its vulnerability inventory $\mathcal{VI}_{raw}$ to be completely validated (i.e. to reach an accuracy of $= 1$), while all other hosts only require a fraction of the steps to achieve the same level of accuracy. In particular, the results described by Table 5.7 are on average 69.60% better than those of Table 5.4 (non optimized environment),[12] i.e. it takes an average of 69.60% less steps to achieve the same result. On average, hosts H1, H4 and H7 achieve a mean reduction of the number of steps required for termination of $\geq 75\%$, hosts H2 and H3 achieve a mean reduction of the number of steps required for termination of $\geq 70\%$ and $< 75\%$, and only H0, H5 and H6 achieve a mean reduction in the number of steps required for termination of $\leq 65\%$. In particular, the average, minimum and maximum reduction of the number of steps needed for completion for each strategy is

- 72.61% on average for strategy *RN*, with a maximum of 80.57% on H1 and a minimum of 57.98% on H5.
- 73.31% on average for strategy *SR*, with a maximum of 78.91% on H4 and a minimum of 64.00% on H5.

[12] 81.98% at maximum, 39.40% at minimum.

- 66.28% on average for strategy *PP*, with a maximum of 81.98% on H1 and a minimum of 39.85% on H0.
- 68.73% on average for strategy *VS*, with a maximum of 81.02% on H4 and a minimum of 41.49% on H0.
- 67.04% on average for strategy *VP*, with a maximum of 80.60% on H1 and a minimum of 39.40% on H0.

This is expected, as the amount of steps reduced by this optimization is highly dependant on the distribution of the vulnerabilities that are enabled by platforms expressed as version ranges, across the hosts.[13]

As per the previous section, a meaningful difference between random (*RN* and *SR*) and semantic driven strategies (*PP*, *VS* and *VP*) continues to be present for hosts H1, H4, H6 and H7. Hosts H0, H2, H3 and H5 do not apparently exhibit any difference between strategies. This is due to (i) H0 and H5 having few vulnerabilities insisting on few, distinct platforms, (ii) H2 and H3 having vulnerabilities having large OR configurations of distinct platforms which are incompatible with their respective hosts, and (iii) the fact that the result shown in the table is the final state of the application of the proposed strategies, i.e. the average number of steps necessary to validate or discard all vulnerabilities in $\mathcal{VI}_{raw}$. Since these reasons contribute to the lack of conclusive evidence over the differences between the different strategies applied to these hosts, it is interesting to analyze the strategies' evolution over the number of steps. Thus, Figure 5.4 has been plotted to visually analyze the average number of CVE vulnerabilities that still have to be validated after x steps of each strategy.

The considerations made for Figure 5.2 in the previous section still stand, as despite the steps taken to completely validate or discard each vulnerability are comparable for hosts H0, H2, H3 and H5, as shown in Table 5.7, “random” approaches *RN* and *SR* continue to present the worst evolution with respect to every other considered strategy. This is easily noticed in Figure 5.4 and confirmed in Table 5.8, for which the number of vulnerabilities still left to validate does not decrease significantly (below 50%) until 69.3% and 61.6% of the total elapsed steps, for *RN* and *SR* respectively.[14] This is expected, since these approaches do not consider any semantics while deciding the order of the platforms to submit to the human agent, and therefore proposed platforms may not be significantly impactful with respect to vulnerabilities in the inventory. Also, as expected, strategy *RN* performs consistently worse than strategy *SR* since *RN* does not remove “non-useful” platforms from the ordering.

With respect to “vulnerability-driven” approaches *VS* and *VP*, the evolution of the proposed strategies across all the considered hosts highlights a limitation, as they are characterized by a staircase-like trend. Manual inspection of the results has determined the cause to be the presence of many OR branches in the CPE configuration trees of certain CVEs. Examples of CVEs which have been found to be blocking for “vulnerability-driven” approaches using the proposed optimization are: *CVE-2015-2808* which contains 146 unique CPE strings[15] across 53 configuration trees, the biggest of which is an OR of 26 CPEs, *CVE-2007-1858* which contains a single configuration tree which is a single OR of 43 CPE strings and *CVE-2022-23305*

[13] H0 and H5 being the minimum at respectively 52.23% and 62.23%, only having a single platform containing a version range.

[14] 78.8% and 71.2% for $\geq 66\%$, 93.7% and 85.0% for $\geq 90\%$.

[15] This number includes platform version ranges encoded in CPE strings.

which contains 42 unique CPE strings organized in 5 configuration trees, the biggest of which is an OR of 37 elements. In these platform configurations, it is typical that only one (or none) of the CPEs in the OR is really present in the host, and its identification requires multiple steps. For this reason, strategies that validate vulnerabilities sequentially are impacted significantly by this type of vulnerability. Another interesting finding is that most of the time, the two approaches *VS* and *VP* share the same evolution. This is due to the fact that these approaches order the vulnerabilities first, and then perform ordering of the related platforms. Also in this case manual inspection of the results has been carried out, and it has been found that vulnerabilities with the same CVSS base score rarely share platforms. This in turn means that the secondary ordering strategy included in *VP* never comes into play, defaulting to a random ordering, which is exactly what *VS* does.

The proposed "platform-semantic-driven" approach *PP* continues to consistently have the best evolution over the number of steps, across all the hosts of the case-study. This is expected, since *PP* has been designed to prioritize the platforms that have the highest potential to impact a vulnerability, whether by validating or discarding it, as per Algorithm 4 of Section 4.3 (Vulnerability Filtering). In particular as shown in Table 5.8, *PP* in average validates 50% of the vulnerabilities after just 12.6% of the total steps required to perform a complete validation across all the hosts, while its closest competitors *VS* and *VP* use significantly more steps (32.1% for both) to achieve the same result. The same trend can be observed for the validation of the last 33% and 10% of vulnerabilities, for which *PP* consistently outperforms the other considered strategies.[16]

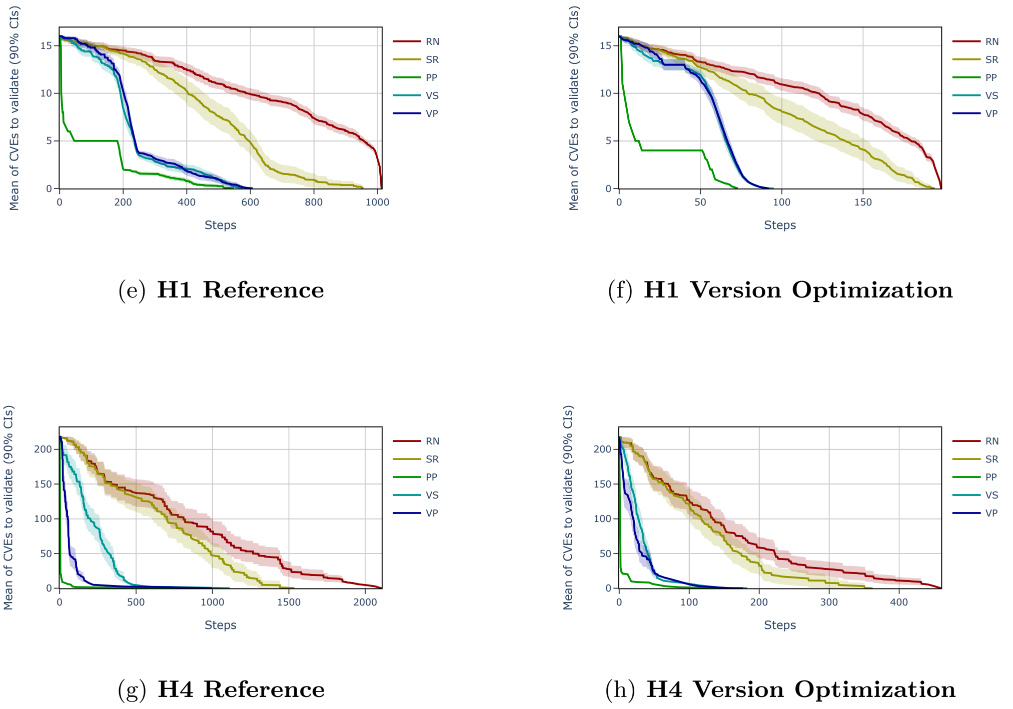
Table 5.8. Mean percentage of steps required for each strategy to achieve respectively $\geq 50\%$, $\geq 66\%$ and $\geq 90\%$ accuracy of the vulnerability inventory $\mathcal{VI}$ (i.e. validation progress), across every host.

Strategy	CVE validated		
	50%	66%	90%
RN	69.3%	78.8%	93.7%
SR	61.6%	71.2%	85.0%
PP	12.6%	22.5%	54.5%
VS	32.1%	42.5%	72.4%
VP	32.1%	41.6%	71.9%

As per the previous case, the reason for *PP*'s comparable performance with respect to other ordering strategies in Table 5.7 for hosts H0, H2, H3 and H5 is due to the presence of CVEs with many OR branches in the CPE configuration trees which do not share CPEs with other vulnerabilities. Note that this issue persists even after deploying the proposed optimization (i.e. with contiguous platform version ranges "compressed" into a single platform). This heavily degrades the performance of Algorithm 4 (Section 4.3 - Vulnerability Filtering, implemented by strategy *PP*), which has been designed to exploit platform frequency and configuration tree structure. Thus it is inevitable that in the absence of platform frequency and in the presence of a "flat" OR configuration tree structure, the performance of the proposed algorithm degrades. As an example, after about 200 steps, *PP* on H2 takes about 90 steps to discard *CVE-2015-2808*, a vulnerability with 146 platforms across 53 different platform configuration trees, the biggest of which is an OR tree of 26 different platforms. Since this vulnerability is not present in H2, any strategy has to discard all the platforms in every flat OR configuration in order to discard

[16] *PP* achieves $\geq 66\%$ and $\geq 90\%$ accuracy within an average of 22.9% and 54.5% of the total amount of steps required for termination, respectively.

the vulnerability. While the issue persists, deploying the proposed optimization reduces its magnitude by potentially reducing the number of elements present in every flat OR configuration, thus reducing the number of steps needed to validate or invalidate each configuration. Moreover, compacting contiguous platform versions into platform version ranges has the possibility to reduce the number of platforms that vulnerabilities have in common, allowing the proposed algorithm to be more incisive. An example of this can be found during the early steps on H6, where *PP* manages to validate 171 vulnerabilities out of a total of 341 (50.15%) with one single step. This is due to the fact that these 171 vulnerabilities (e.g. *CVE-2018-2938*, *CVE-2017-10356* and *CVE-2019-2958*) all share the same enabling platform (*a:oracle:jdk:1.7.0*), present in H6.



(e) **H1 Reference**

(f) **H1 Version Optimization**

(g) **H4 Reference**

(h) **H4 Version Optimization**

Figure 5.5. Mean of the CVEs that still have to be validated after each step of the prioritization strategies. Comparison between the results of Section 5.2 (Vulnerability Filtering) and Section 5.3 (Version Optimization). H1 represents an *Office* host, H4 represents a *Development* host.

Figure 5.5 compares the evolution of the average number of CVE vulnerabilities that still have to be validated after x steps of each platform ordering strategy for hosts H1 and H4. While the reduction in the number of steps required for termination is apparent, the evolution of the different strategies with respect to the number of steps remains similar with respect to the references of Section 5.2 (Vulnerability Filtering). This as already hinted before, is a consequence of the distribution of the vulnerabilities that are enabled by platforms expressed as version ranges, across the hosts, and is expected, as the amount of steps reduced by this optimization is highly dependant on this distribution.

5.3.1 Optimizing platform versions for strategy *PP*

The second part of the proposed optimization attempts to enhance the performance of the proposed methodology by dividing platform version ranges into smaller, non overlapping platform version ranges. This is done in order to facilitate the proposed methodology's prioritization Algorithm 4 described in Section 4.3 (Vulnerability Filtering), which relies on frequency and distribution of platforms in each platform configuration tree of vulnerabilities in a given host h_i.

However, splicing platform version ranges into multiple platforms has the side-effect of increasing the size of the vulnerability graph $\mathcal{G}_i$, as already observed during the comparison between $\mathcal{G}_i$ and $\mathcal{G}_i^{NoVe}$ (i.e. vulnerability graph with or without the optional platform version attribute). In particular, applying this optimization in the considered test case increases the size of the vulnerability graph $\mathcal{G}_i$ by 8.28%, on average (max 29.31%, min 0.00%). Thus, it is expected that this part of the optimization will only benefit strategy PP and not the other strategies.

For the proposed test case, part 2 of the proposed optimization achieves a reduction in the number of steps needed for termination of at best 16.92% on H6 and at worst −24.41% on H7, with a mean of −1.50% for strategy PP. If all strategies are considered, the values become slightly worse, with a maximum of 19.57% on H6 and a minimum of −27.62% on H7, with a mean of −4.85% Moreover, a similar trend can be observed by studying the evolution of strategy PP over the number of steps, which produces a vulnerability inventory of at least $\geq 50\%$ accuracy after 13.9% of the total steps required for termination, $\geq 66.0\%$ after 21.6% steps and $\geq 90\%$ after 56.7% steps.

While this result is not bad considering the increase in size of the vulnerability graph $\mathcal{G}_i$ by 8.28% on average, manual analysis of the execution traces together with the considerations made in this section lead to the conclusion that this second part of the proposed two-step optimization is highly situational and should be applied contextually, and that further work is needed to be able to fully exploit its advantages. Ideally, a possible venue for future work would be the analysis of an orchestration mechanism that, depending on the structure and statistical properties of the platforms residing in a host is able to intelligently apply the second step of the optimization on a per-host basis. This possibility has already been demonstrated with hosts H5 and H7, for which an orchestration system able to allow or block the application of the second step of the proposed optimization would benefit both, H5 being benefited by allowing the second step of the optimization, and H7 being benefited by not applying this second step. A hint on which this venue could be developed is given by the composition of H5 and H7, as platforms version ranges in H7 create a significant amount of "new" platforms version ranges than H5, causing a 29.31% increase in size of the vulnerability graph $\mathcal{G}_i$. Another hint on which this venue could be developed is given is the distribution of the platforms across the vulnerabilities of each host. Indeed, in different other case studies this second step of the optimization has yielded a reduction of the number of steps required by strategy PP to achieve termination by more than 15%. Manual inspection of these environments has found a high degree of overlap of large platform version intervals, across a sizeable amount of vulnerabilities (i.e. many isolated "small" version intervals and one "large" central version interval shared by every vulnerability). Such feature is not present in the proposed case study, and for this reason, the second step of the optimization performs globally worse.

5.4 Considering Platform Dependency

Section 4.5 (Platform Dependency) delivered another advancement to the methodology described in Section 4.3 (Vulnerability Filtering) by considering the dependencies between different CPE strings used to identify platforms on a host. Indeed, this advancement operates on the structure of a CPE string itself, which is not to be considered as a monolithic entity anymore, but as a collection of attributes.

Thus, the main intuition behind this advancement is that the validation or invalidation of a CPE string on a host h_i leads to a validation or invalidation of the string's attributes on h_i. As an example, validating the presence of a specific version of a product on h_i, should also entail the validation of the presence of the same product on h_i. Conversely, discarding a product from a host h_i should also entail the absence from h_i of every other platform which identify specific versions of the same product.

To this aim, Section 4.5 (Platform Dependency) presents an advancement which is orthogonal to the process and optimizations presented previously, capable of considering the *dependency* between different platforms and potentially capable of increasing the effectiveness of the answer provided by the human agent by allowing such answer to validate or invalidate more than one platform at the same time.

The evaluation of the proposed advancement and its effects on the proposed methodology has been carried out consistently with the evaluation of the proposed methodology in Section 5.2 (Vulnerability Filtering). Thus, the proposed advancement has been applied to all five prioritization strategies (*RN*, *SR*, *PP*, *VS* and *VP*), which have been implemented on inventories derived from the environment described in Section 5.1 (hosts H0-H7).

It should be noted that none of the strategies are completely deterministic: in *RN* and *SR* the platforms follow a random order (i.e. no sorting is applied), *VS* performs a sorting on the vulnerabilities according to their scoring, and then uses a random order for the platforms pertaining to vulnerabilities with the same scoring, and *PP* and *VP* may choose at random if multiple platforms have the same score. For this reasons, 50 instances have been generated for each strategy in order to compare statistically relevant results.

As per the previous optimization, the final result of the validation described by Table 5.3 of Section 5.2 (Vulnerability Filtering) still stands. Note that this new advancement is orthogonal to the optimization proposed in Section 5.3 (Version Optimization), thus if the two are applied at the same time, the final result will be described by Table 5.6. This is due to the fact that this proposed advancement affects the number of steps required by each strategy to converge to the same final state, but doesn't fundamentally change the input data or significative portions of the algorithm.

Moreover, this advancement will only affect hosts that have platforms that describe the same product at different levels of granularity, e.g. specific version vs general product. To go in detail, a *compatible group* of platforms must be defined as the group of platforms identifying different versions of the same product, together with a platform that identifies the product in a generic fashion (i.e. no version is specified in the CPE string). Following this definition, an example of a compatible group of platforms is a generic *o:apple:macos:-* platform string, identifying a generic instance of the *macos* operating system made by *apple*, grouped together with other platform strings which identify different versions of the same operating system.

Thus, the following occurrences of compatible groups in the dataset derived from Section 5.3 (Version Optimization) can be found: H0 has 2 compatible groups composed of 3.0 platforms on average, H1 has no compatible groups, H2 has 4 compatible groups composed of 3.0 platforms on average, H3 has 3 compatible groups composed of 2.3 platforms on average, H4 has 15 compatible groups composed of 3.5 platforms on average, H5 has no compatible groups, H6 has 21 compatible groups composed of 3.6 platforms on average, H7 has 3 compatible groups composed of 2.3 platforms on average. This entails the fact that if the advancement proposed in this section is used in conjunction with the optimization proposed in Section 5.3 (Version Optimization), this advancement will have no effect on hosts H1 and H5 since no products are referenced with different levels of granularity (i.e. no product is referenced both with and without a specific version).

Conversely, if the optimization proposed in Section 5.3 (Version Optimization) is not applied, H0 has 2 compatible groups composed of 30.0 platforms on average, H1 has 8 compatible groups composed of 63.4 platforms on average, H2 has 9 compatible groups composed of 89.6 platforms on average, H3 has 8 compatible groups composed of 93.4 platforms on average, H4 has 18 compatible groups composed of 92.4 platforms on average, H5 has no compatible groups, H6 has 37 compatible groups composed of 61.6 platforms on average, H7 has 15 compatible groups composed of 59.9 platforms on average. Under these circumstances, the proposed advancement will still have no effect on H5, but it will gain effect on H1. It is also interesting to notice that the proposed advancement has the potential to be more effective in this scenario, since on average more groups are found, and each group is composed of more platforms on average, with respect to the already-optimized scenario.

5.4.1 Application to a non-optimized environment (Section 5.2)

Table 5.9 shows the mean of the number of steps to validate all the vulnerabilities (90% Confidence Intervals CIs) in every host of the case study described in Section 5.1, for each strategy, without any prior optimization pass. Given this setting, a correct reference to compare this result would be Table 5.4 of Section 5.2 (Vulnerability Filtering). As a reminder, hosts H0 and H1 represent *Office* hosts, H2 and H3 represent *Server* hosts, H4 and H5 represent *Development* hosts and H6 and H7 represent *Generic* hosts.

For host H0, all strategies except *RN* obtain similar results requiring about ≈ 34 steps to validate all the vulnerabilities, while *RN* necessitates ≈ 94 steps. Similarly, for host H1, vulnerability-based strategies require about ≈ 360 steps to terminate, *RN* requires ≈ 1008 steps, *SR* requires ≈ 537 steps, and *PP* requires ≈ 313 steps. Host H2 follows a trend similar to H0 in which all strategies except *RN* and *SR* require about ≈ 1085 steps to terminate, while *RN* and *SR* necessitate ≈ 1216 and ≈ 1117 steps respectively. All strategies of host H3 with the exception of *RN* require ≈ 1080 steps to terminate, while *RN* necessitates ≈ 1136 steps. In host H4, platform-based strategies *PP* and *VP* require about ≈ 470 steps to terminate, *RN* requires ≈ 1760 steps, *SR* requires ≈ 852 steps, and *VS* requires ≈ 682 steps. For host H5, all strategies except *RN* require ≈ 28 steps to terminate, while *RN* necessitates ≈ 34 steps. Host H6 follows a similar trend to H1 and H4 in which vulnerability-based strategies require about ≈ 1400 steps to terminate, *RN* requires ≈ 5871 steps, *SR* requires ≈ 2836 steps, and *PP* requires ≈ 961 steps. Lastly, host H7 requires ≈ 1320 steps for all strategies except *RN* and *SR*, which necessitate ≈ 1604 and ≈ 1453 steps respectively.

Compared to the result obtained in Table 5.4 of Section 5.2 (Vulnerability Filtering), it can be noticed that the proposed advancement has little effect on some of the results, while on others it has a decent impact. In particular, it can be noticed that while the only hosts affected are H0, H1, H2, H3, H4, H6 and H7,

Table 5.9. Results of the application of the strategies on the case study, without prior optimization

Host	Metric	Strategies				
		RN	SR	PP	VS	VP
H0	*Ci1*	94.62	67.50	34.68	34.89	34.48
	Mean	94.02	63.38	33.68	34.08	33.40
	Ci0	93.42	59.26	32.68	33.27	32.32
H1	*Ci1*	1008.64	573.01	332.24	383.86	375.39
	Mean	1007.56	537.18	313.14	363.84	357.30
	Ci0	1006.48	501.35	294.04	343.82	339.21
H2	*Ci1*	1215.87	1122.71	1084.39	1093.25	1086.23
	Mean	1215.74	1117.04	1083.28	1091.32	1084.88
	Ci0	1215.61	1111.37	1082.17	1089.39	1083.53
H3	*Ci1*	1135.96	1082.47	1075.53	1081.85	1074.74
	Mean	1135.88	1081.30	1074.68	1080.76	1073.94
	Ci0	1135.80	1080.13	1073.83	1079.67	1073.14
H4	*Ci1*	1801.18	918.49	513.57	724.19	532.80
	Mean	1759.96	851.82	466.64	682.26	485.54
	Ci0	1718.74	785.15	419.71	640.33	438.28
H5	*Ci1*	34.54	29.59	28.91	28.57	29.30
	Mean	34.32	28.72	27.96	27.62	28.36
	Ci0	34.10	27.85	27.01	27.01	27.42
H6	*Ci1*	5893.93	2972.83	991.11	1571.84	1286.33
	Mean	5871.48	2835.78	960.66	1516.94	1240.40
	Ci0	5849.03	2698.73	930.21	1462.04	1194.47
H7	*Ci1*	1603.93	1469.23	1321.78	1334.79	1329.46
	Mean	1603.84	1452.56	1320.96	1333.14	1328.06
	Ci0	1603.75	1435.89	1320.14	1331.49	1326.66

some strategies of H1, and H4 achieve a reduction in the number of steps required to terminate of $\geq 10\%$. Meanwhile, H0, H2, H3 and H7 achieve a reduction of $\leq 1\%$ (but still $\geq 0\%$). This is expected since only these hosts have platforms that are compatible with this advancement. Another property that can be noticed is that this advancement benefits the most vulnerability-based strategies *VS* and *VP* and random-based strategies *RN* and *SR*, with the average, minimum and maximum reduction of the number of steps needed for completion being

- 2.25% on average for strategy *RN*, with a maximum of 15.33% on H4 and a minimum of 0.00% on H0, H2, H3, H5 and H7.
- 2.55% on average for strategy *SR*, with a maximum of 10.64% on H4 and a minimum of 0.00% on H0, H2, H3, H5 and H7.
- 3.60% on average for strategy *PP*, with a maximum of 14.64% on H1 and a minimum of 0.00% on H0, H2, H5, H6 and H7.
- 2.92% on average for strategy *VS*, with a maximum of 9.35% on H1 and a minimum of 0.00% on H3 and H5.
- 3.81% on average for strategy *VP*, with a maximum of 15.86% on H4 and a minimum of 0.00% on H0, H2, H3, H5 and H7.

This is also to be expected, since the distribution of the platforms compatible with this advancement is not uniform across the vulnerabilities (CVE) of each considered host h_i.

5.4.2 Application to an already-optimized environment (Section 5.3)

Table 5.10. Results of the application of the strategies on the case study, after prior optimization

Host	Metric	Strategies				
		RN	SR	PP	VS	VP
H0	*Ci1*	27.36	20.45	21.33	19.81	21.27
	Mean	27.06	19.52	20.26	18.68	20.24
	Ci0	26.76	18.59	19.19	17.55	19.21
H1	*Ci1*	196.51	143.04	67.13	81.59	81.42
	Mean	195.88	133.48	66.12	79.98	79.54
	Ci0	195.25	123.92	65.11	78.37	77.66
H2	*Ci1*	331.69	279.73	278.62	278.22	277.66
	Mean	331.40	278.74	277.60	277.30	276.64
	Ci0	331.11	277.75	276.58	276.38	275.62
H3	*Ci1*	319.86	269.63	269.91	270.79	269.60
	Mean	319.63	268.79	269.16	269.92	268.70
	Ci0	319.40	267.95	268.41	269.05	267.80
H4	*Ci1*	454.62	214.79	119.69	143.08	136.94
	Mean	453.16	201.00	115.50	138.38	132.02
	Ci0	451.70	187.21	111.31	133.68	127.10
H5	*Ci1*	15.02	11.26	11.60	11.44	10.69
	Mean	14.42	10.34	10.72	10.46	9.82
	Ci0	13.82	9.42	9.84	9.48	8.95
H6	*Ci1*	1366.00	821.37	559.76	673.55	668.30
	Mean	1365.24	801.62	553.42	661.04	656.92
	Ci0	1364.48	781.87	547.08	648.53	645.54
H7	*Ci1*	463.61	369.58	313.71	314.20	314.91
	Mean	463.36	362.98	312.76	313.26	314.04
	Ci0	463.11	356.38	311.81	312.32	313.17

Table 5.10 shows the mean of the number of steps to validate all the vulnerabilities (90% Confidence Intervals CIs) in every host of the case study described in Section 5.1, for each strategy, after applying the optimization described in Section 4.4 (Version Optimization). Given this setting, a correct reference to compare this result would be Table 5.7 of Section 5.3 (Version Optimization). As a reminder, hosts H0 and H1 represent *Office* hosts, H2 and H3 represent *Server* hosts, H4 and H5 represent *Development* hosts and H6 and H7 represent *Generic* hosts.

For host H0, all strategies except *RN* obtain similar results requiring about ≈ 21 steps to validate all the vulnerabilities, while *RN* necessitates ≈ 27 steps. Similarly, for host H1, vulnerability-based strategies require about ≈ 81 steps to terminate, *RN* requires ≈ 196 steps, *SR* requires ≈ 133 steps, and *PP* requires ≈ 66 steps. Host H2 follows a trend similar to H0 in which all strategies except *RN* require about ≈ 278 steps to terminate, while *RN* necessitates ≈ 332 steps. All strategies of host H3 with the exception of *RN* require ≈ 270 steps to terminate, while *RN* necessitates ≈ 320 steps. In host H4, vulnerability-based strategies require about ≈ 135 steps to terminate, *RN* requires ≈ 453 steps, *SR* requires ≈ 201 steps, and *PP* requires ≈ 115 steps. For host H5, all strategies except *RN* require ≈ 10 steps to terminate, while *RN* necessitates ≈ 14 steps. Host H6 follows a similar trend to H1 and H4 in which vulnerability-based strategies require about ≈ 657 steps to terminate, *RN* requires ≈ 1365 steps, *SR* requires ≈ 801 steps, and *PP* requires ≈ 553 steps. Lastly, host H7 requires ≈ 313 steps for all strategies except *RN* and *SR*, which necessitate ≈ 463 and ≈ 363 steps respectively.

Compared to the result obtained in Table 5.7 of Section 5.3 (Version Optimization), it can be noticed that the proposed advancement has little effect on the results. As expected, it can be noticed that the only hosts affected are H0, H1, H2, H3, H4, H6 and H7. This is expected since only these hosts have platforms that are compatible with this advancement. Another property that can be noticed is that this advancement benefits only vulnerability-based strategy *VS* and random-based strategies *RN* and *SR*, with the average, minimum and maximum reduction of the number of steps needed for completion being

- 0.06% on average for strategy *RN*, with a maximum of 0.25% on H3 and a minimum of 0.00% across all hosts except H2 and H3.
- 0.10% on average for strategy *SR*, with a maximum of 0.54% on H3 and a minimum of 0.00% across all hosts except H2 and H3.
- 0.00% on average for strategy *PP*, with a maximum and a minimum of 0.00% across all hosts.
- 0.98% on average for strategy *VS*, with a maximum of 6.97% on H0 and a minimum of 0.00% on H2, H3, H5, H6 and H7.
- 0.00% on average for strategy *VP*, with a maximum and a minimum of 0.00% on every host.

This is also to be expected, since the distribution of the platforms compatible with this advancement is not uniform across the vulnerabilities (CVE) of each considered host h_i. This is also the reason why most strategies are not affected by this advancement, in particular random-based strategies achieve very little improvement due to their random nature coupled with the small number of platforms that are actually compatible with this advancement. Similarly, *PP* and *VP* do not achieve any improvement since the order of the selection of the CPE strings is tightly bound by a mathematical formula, with little or no room for variation between different executions. Indeed, if a "generic" platform (i.e. the platform with the version attribute set to either NONE - or ANY *) is selected after all corresponding platforms which relate to single (or to a range of) versions, this advancement will have no effect. This poses as a limitation of this approach, and is addressed by the proposal in Section 4.6 (Structure-Aware).

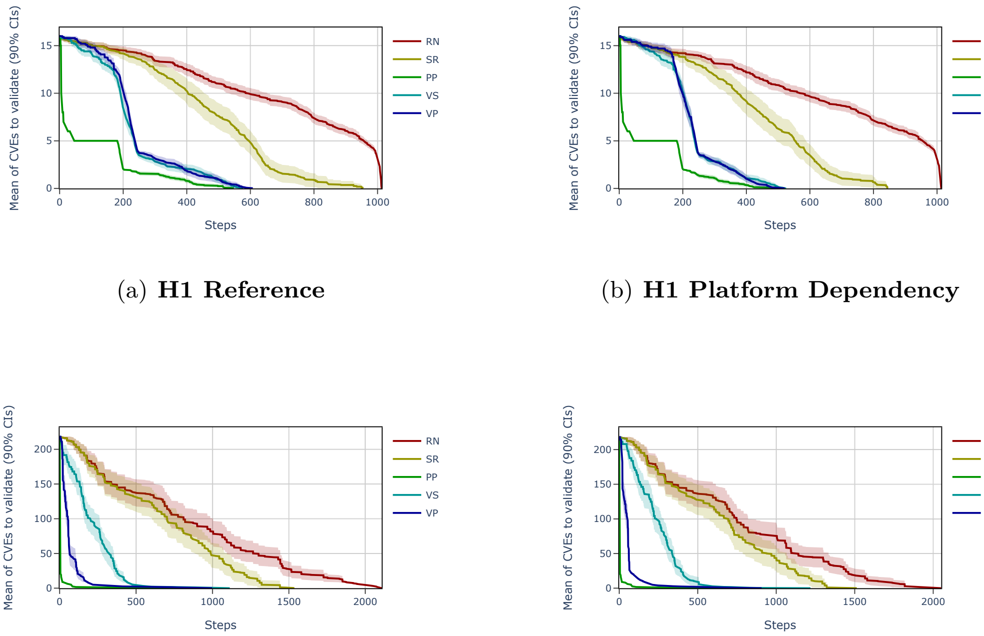
5.4.3 Considerations on the evolution of the non-optimized environment

The considerations made for Figure 5.2 in Section 5.2 (Vulnerability Filtering) still stand, as despite the steps taken to completely validate or discard each vulnerability are comparable for hosts H0, H1, H2, H3 and H5, as shown in Table 5.9, "random" approaches *RN* and *SR* continue to present the worst evolution with respect to every other considered strategy. If confronted against Table 5.5 of Section 5.2 (Vulnerability Filtering), Table 5.11 confirms the same trend, with the only difference of achieving minor increments in convergence speed for vulnerability-based strategies *VS* and *VP* as well as random-based strategies *RN* and *SR*, as expected from the results of Table 5.9.

Table 5.11. Mean percentage of steps required for each strategy to achieve respectively $\geq 50\%$, $\geq 66\%$ and $\geq 90\%$ accuracy of the vulnerability inventory $\mathcal{VI}$ (i.e. validation progress), across every host.

Strategy	**CVE validated**		
	50%	66%	90%
RN	76.5%	85.8%	94.5%
SR	68.3%	76.7%	87.7%
PP	14.8%	23.3%	48.4%
VS	42.3%	53.0%	73.0%
VP	39.4%	50.1%	70.7%

Figure 5.6 compares the evolution of the average number of CVE vulnerabilities that still have to be validated after x steps of each platform ordering strategy for hosts H1 and H4.



(a) **H1 Reference**

(b) **H1 Platform Dependency**

(c) **H4 Reference**

(d) **H4 Platform Dependency**

Figure 5.6. Mean of the CVEs that still have to be validated after each step of the prioritization strategies. Comparison between the results of Section 5.2 (Vulnerability Filtering) and Section 5.4 (Platform Dependency). H1 represents an *Office* host, H4 represents a *Development* host.

In this case, there is no apparent difference between the number of steps required for termination as well as the evolution of the different strategies with respect to the number of steps, when compared against the references of Section 5.2 (Vulnerability Filtering). This is expected, since as discussed before, the distribution of the platforms compatible with this advancement is not uniform across the vulnerabilities (CVE) of each considered host h_i. Indeed, if during the execution of the validation of the vulnerabilities of a host h_i, a "generic" platform (i.e. the platform with the version attribute set to either NONE - or ANY *) is selected after all corresponding platforms which relate to single (or to a range of) versions, this advancement will have no effect. This poses as a limitation of this approach, and is addressed by the proposal in Section 4.6 (Structure-Aware).

5.4.4 Considerations on the evolution of the already-optimized environment

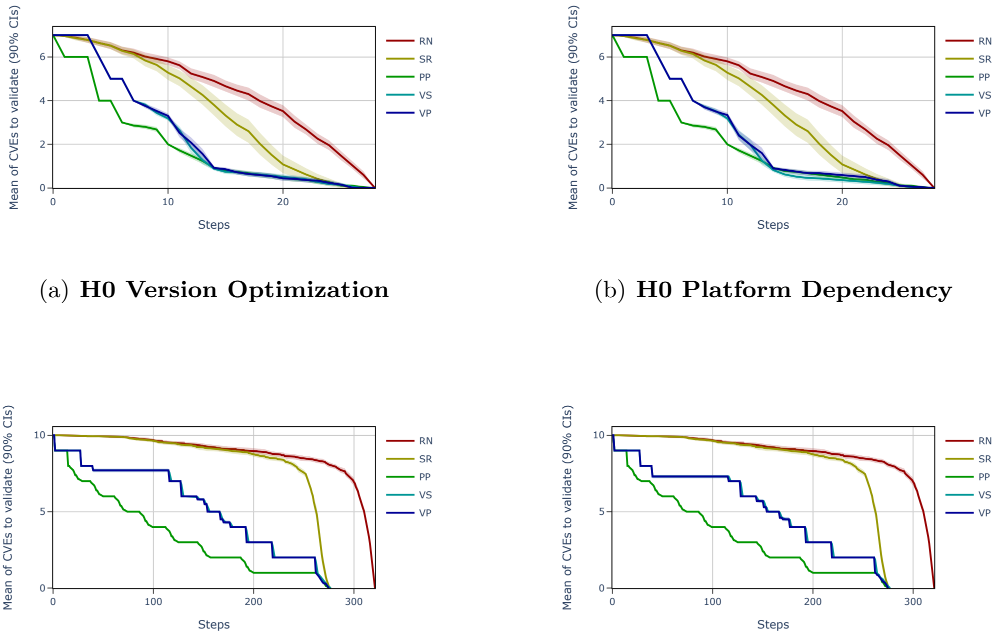
The considerations made for Figure 5.4 in the previous section (Version Optimization) still stand, as despite the steps taken to completely validate or discard each vulnerability are comparable for hosts H0, H2, H3 and H5, as shown in Table 5.10, "random" approaches *RN* and *SR* continue to present the worst evolution with respect to every other considered strategy. If confronted against Table 5.8 of Section 5.3 (Version Optimization), Table 5.12 confirms the same trend, with the only difference of achieving minor increments in convergence speed for vulnerability-based strategies *VS* and *VP* as well as random-based strategies *RN* and *SR*, as expected from the results of Table 5.10.

Table 5.12. Mean percentage of steps required for each strategy to achieve respectively $\geq 50\%$, $\geq 66\%$ and $\geq 90\%$ accuracy of the vulnerability inventory $\mathcal{VI}$ (i.e. validation progress), across every host.

Strategy	**CVE validated**		
	50%	66%	90%
RN	69.3%	78.8%	93.6%
SR	61.6%	71.2%	85.0%
PP	12.6%	22.5%	54.5%
VS	28.9%	42.2%	71.8%
VP	28.8%	41.6%	71.9%

Figure 5.7 compares the evolution of the average number of CVE vulnerabilities that still have to be validated after x steps of each platform ordering strategy for hosts H0 and H3.

Also in this case, there is no apparent difference between the number of steps required for termination as well as the evolution of the different strategies with respect to the number of steps, when compared against the references of Section 5.3 (Version Optimization). This is expected, since as discussed before, the distribution of the platforms compatible with this advancement is not uniform across the vulnerabilities (CVE) of each considered host h_i. Indeed, if during the execution of the validation of the vulnerabilities of a host h_i, a "generic" platform (i.e. the platform with the version attribute set to either NONE - or ANY *) is selected after all corresponding platforms which relate to single (or to a range of) versions, this advancement will have no effect. This poses as a limitation of this approach, and is addressed by the proposal in Section 4.6 (Structure-Aware).



(a) **H0 Version Optimization**

(b) **H0 Platform Dependency**

(c) **H3 Version Optimization**

(d) **H3 Platform Dependency**

Figure 5.7. Mean of the CVEs that still have to be validated after each step of the prioritization strategies. Comparison between the results of Section 5.3 (Version Optimization) and Section 5.4 (Platform Dependency). H0 represents an *Office* host, H3 represents a *Server* host.

5.5 A Platform Structure-Aware Algorithm

Section 4.5 (Platform Dependency) provided an interesting advancement which considers platforms expressed as CPE strings not as a monolithic entity, but as a collection of attributes. Section 4.6 (Structure-Aware) takes this approach further, by designing a new algorithm in order to fully exploit the formal relationships between the attributes of a CPE string.

The main idea behind the proposal is to approach the validation of vulnerabilities by validating one platform attribute at a time. While this increases the number of steps needed to validate a single platform (since all its attributes need to be validated one at a time), this approach has the possibility of discarding entire sets of platforms that share a subset of attributes early on, and more effectively than the solution advanced in the last section (e.g. if a vendor is not present in the system, any platform which derives from that vendor is automatically discarded). To achieve this, the proposed approach constructs two graph structures: the Host-Platform Graph ($\mathcal{HPG}_i$), and the truncated vulnerability graph $\mathcal{G}_i^{tx}$, and then performs operations on these graphs that lead to an increase the quality of the automatically generated inventories $\mathcal{VI}_{raw}$ and $\mathcal{DI}_{raw}$ provided by vulnerability and network scanners. In particular, Algorithm 6 of Section 4.6 (Structure-Aware) presents the pseudo-code for this new approach, which pivots around the exploration of the host-platform graph $\mathcal{HPG}_i$. Since each node of $\mathcal{HPG}_i$ maps to a CPE attribute, exploring and validating these nodes has the capacity to discard entire sets (sub-trees) of CPE which share common attributes. The exploration of $\mathcal{HPG}_i$ is guided by four proposed cost functions (R, P, V and VP) which reason on $\mathcal{G}_i^{tx}$.

The evaluation of the newly proposed approach has been carried out consistently with the evaluation of all the previous methodologies (Section 5.2, Vulnerability Filtering), optimizations (Section 5.3, Version Optimization) and approaches (Section 5.4, Platform Dependency) to ensure comparability. Thus, the newly proposed approach has been applied using all four proposed cost functions, which have been implemented on inventories derived from the environment described in Section 5.1 (Hosts H0-H7). It should be noted that each cost function is comparable to one or multiple strategies from previous sections, in particular: (i) R is comparable to RN and SR, (ii) P is comparable to PP, (iii) V is comparable to VS, and (iv) VP is comparable to VP.

It should also be noted that none of the explorations of $\mathcal{HPG}$ driven by the cost functions are completely deterministic: in R the platforms follow a random order (i.e. no sorting is applied) and thus lead to a random exploration of the host-platform graph, V performs a sorting on the vulnerabilities according to their scoring, and then applies the same score to all the platforms pertaining to vulnerabilities with the same scoring and thus may lead to a random branching choice during the exploration of the graph, and P and VP may lead to a random branching choice if multiple platforms have the same score. For this reasons, 50 instances have been generated for each exploration of the graph driven by each cost function in order to compare statistically relevant results.

As per the previous sections, the final result of the validation described by Table 5.3 of Section 5.2 (Vulnerability Filtering) still stands. Note that this new approach can also be carried out on the modified vulnerability and device inventories proposed in Section 5.3 (i.e. by compressing contiguous product versions as product version ranges), thus in such case the final result will be described by Table 5.6 of Section 5.3 (Version Optimization). This is due to the fact that the host-platform graph $\mathcal{HPG}$ derives from the CPE strings of the device inventory, thus if these inventories are pre-processed the resulting host-platform graph will change according to the result of the pre-processing.

5.5.1 Application to a non-optimized environment (Section 5.2)

Table 5.13 shows the mean of the number of steps to validate all the vulnerabilities (90% Confidence Intervals CIs) in every host of the case study described in Section 5.1, for each cost function, without any prior optimization pass. Given this setting, a correct reference to compare this result would be Table 5.4 of Section 5.2 (Vulnerability Filtering). As a reminder, hosts H0 and H1 represent *Office* hosts, H2 and H3 represent *Server* hosts, H4 and H5 represent *Development* hosts and H6 and H7 represent *Generic* hosts.

For host H0, random-based cost function R requires about ≈ 33 steps to validate all the vulnerabilities, while platform-based cost functions P and VP necessitate ≈ 34 steps, and lastly V necessitates ≈ 28 steps. For host H1, R requires about ≈ 147 steps to validate all the vulnerabilities, while P and V necessitate ≈ 142 steps, and lastly VP necessitates ≈ 93 steps. Host H2 follows a trend similar to H0 in which random-based cost function R requires about ≈ 49 steps to validate all the vulnerabilities, while platform-based cost functions P and VP necessitate ≈ 46 steps, and lastly V necessitates ≈ 44 steps. All cost functions of host H3 require ≈ 36 steps to terminate. In host H4, R requires about ≈ 115 steps to validate all the vulnerabilities, while P and V necessitate ≈ 77 steps, and lastly VP necessitates

Table 5.13. Results of the application of the strategies on the case study, without prior optimization

Host	Metric	Cost Functions			
		R	P	V	VP
H0	*Ci1*	33.67	34.64	30.23	34.4
	Mean	33.24	34.52	28.28	34.27
	Ci0	32.81	34.40	26.33	34.15
H1	*Ci1*	166.99	161.82	172.81	118.04
	Mean	147.48	142.62	142.84	92.92
	Ci0	127.97	123.42	112.87	67.81
H2	*Ci1*	49.93	48.38	45.58	47.57
	Mean	48.68	47.44	44.18	46.12
	Ci0	47.43	46.50	42.78	44.68
H3	*Ci1*	36.06	36.62	35.92	36.66
	Mean	35.74	36.50	35.74	36.52
	Ci0	35.42	36.38	35.56	36.39
H4	*Ci1*	120.29	77.92	83.06	98.25
	Mean	114.56	75.32	79.10	94.60
	Ci0	108.83	72.72	75.14	90.95
H5	*Ci1*	12.08	12.70	12.20	12.76
	Mean	11.62	12.58	12.12	12.62
	Ci0	11.16	12.46	12.04	12.49
H6	*Ci1*	1199.13	746.35	543.51	1002.74
	Mean	1146.94	716.72	490.00	958.33
	Ci0	1094.75	687.09	436.49	913.91
H7	*Ci1*	60.78	51.77	59.43	54.02
	Mean	58.98	51.66	57.70	53.02
	Ci0	57.18	51.55	55.97	52.03

$\approx$ 95 steps. For host H5, all cost functions require $\approx$ 12 steps to terminate. Host H6 requires about $\approx$ 1147 steps to validate all the vulnerabilities for random-based cost function R, while P necessitates $\approx$ 717 steps, V necessitates $\approx$ 490 steps, and lastly VP necessitates $\approx$ 958 steps. Lastly, in host H7 random-based cost function R requires about $\approx$ 59 steps to validate all the vulnerabilities, while platform-based cost functions P and VP necessitate $\approx$ 52 steps, and lastly V necessitates $\approx$ 58 steps.

Compared to the result obtained in Table 5.4 of Section 5.2 (Vulnerability Filtering), it can be immediately noticed that this new approach reduces significantly the overall mean number of steps needed for termination, across all comparable strategies. In particular, it can be noticed that while not all the hosts are always affected positively by this new approach (e.g. all cost functions on H6 achieve -19.39% at minimum, but 46.22% on average), the average reduction in the mean number of steps required to terminate across all hosts and comparable strategies is still significant, at 72.32%. The few, negative or neutral results are expected for hosts with few vulnerabilities and platforms (H6 as discussed before, as well as H0 which achieves 0.15% at minimum, 25.52% on average across all cost functions) considering that the proposed host-platform graph structure $\mathcal{HPG}$ requires the human agent to validate multiple nodes (i.e. attributes of a platform) in order to validate a single platform. Conversely, the real power of $\mathcal{HPG}$ is its ability to discard multiple platforms with one single interaction with the human agent. Indeed, by taking a look at the number of false positives found in $\mathcal{VI}$ and $\mathcal{DI}$ (summarized in Table 5.3 of Section 5.2, Vulnerability Filtering) it can be immediately noticed how the hosts in which this new approach obtains the best results, are also the same hosts which have the highest amount of platforms discarded during the validation process, with the best average reduction in the mean number of steps required to validate all vulnerabilities across all cost functions being seen in hosts H2, H3 and H7, amounting to $\geq 95\%$. All three hosts have a *false positive* ratio among their platforms of $\geq 99.6\%$, the highest of all other hosts in the considered case study. Conversely, the worst performing hosts H0 and H6 which sit at a mean reduction in the number of required steps of 25.52% and 46.22% respectively have a *false positive* ratio among their platforms of 97.9% and 97.1% respectively, the lowest of the proposed case study. This property is also at the base for another observation that can be made on these experimental results, as an inspection of the results reveals that this new approach benefits the most vulnerability-based cost functions V and VP and random-based cost function R, with the average, minimum and maximum reduction of the number of steps needed for completion being

- 85.83% on average for cost function R when compared against RN, with a maximum of 96.79% on H3 and a minimum of 63.28% on H5.
- 77.31% on average for cost function R when compared against SR, with a maximum of 96.69% on H3 and a minimum of 47.55% on H0.
- 58.35% on average for cost function P when compared against PP, with a maximum of 96.68% on H3 and a minimum of -19.39% on H6.
- 68.42% on average for cost function V when compared against VS, with a maximum of 96.62% on H3 and a minimum of 0.15% on H0.
- 71.69% on average for cost function VP when compared against VP, with a maximum of 96.67% on H3 and a minimum of 15.33% on H0.

Manual inspection of the execution traces has shown evidence that the random-based cost function R has the best reduction of the number of required steps on average (average reduction of steps of 85.83% when compared against RN, and 77.31% when compared against SR across all hosts), while vulnerability-based cost functions come second, at 68.42% for V when compared against VS and 71.69% for VP when compared against VP, across all hosts. When compared against PP, P has the least benefit from this proposed approach, reaching a still respectable 58.35% reduction on the number of steps required to terminate on average.

5.5.2 Application to an already-optimized environment (Section 5.3)

Table 5.14 shows the mean of the number of steps to validate all the vulnerabilities (90% Confidence Intervals CIs) in every host of the case study described in Section 5.1, for each strategy, after applying the optimization described in Section 4.4 (Version Optimization). Given this setting, a correct reference to compare this result would be Table 5.7 of Section 5.3 (Version Optimization). As a reminder, hosts H0 and H1 represent *Office* hosts, H2 and H3 represent *Server* hosts, H4 and H5 represent *Development* hosts and H6 and H7 represent *Generic* hosts.

For host H0, random-based cost function R requires about ≈ 19 steps to validate all the vulnerabilities, while platform-based cost functions P and VP necessitate

Table 5.14. Results of the application of the cost functions on the case study, after prior optimization

Host	Metric	Cost Functions			
		R	P	V	VP
H0	*Ci1*	19.83	20.27	18.74	20.72
	Mean	19.46	20.12	18.16	20.57
	Ci0	19.09	19.97	17.58	20.43
H1	*Ci1*	33.76	30.79	29.17	29.33
	Mean	33.22	30.68	28.32	28.52
	Ci0	32.68	30.57	27.47	27.72
H2	*Ci1*	42.85	43.22	42.28	43.83
	Mean	42.52	43.06	41.80	43.68
	Ci0	42.19	42.90	41.32	43.53
H3	*Ci1*	35.96	36.81	36.19	36.78
	Mean	35.64	36.70	36.02	36.66
	Ci0	35.32	36.59	35.85	36.54
H4	*Ci1*	106.89	71.27	72.28	92.22
	Mean	102.26	69.42	69.02	90.10
	Ci0	97.63	67.57	65.76	87.98
H5	*Ci1*	12.3	12.72	12.39	12.73
	Mean	11.96	12.60	12.28	12.60
	Ci0	11.62	12.48	12.17	12.47
H6	*Ci1*	388.44	325.10	267.14	385.88
	Mean	378.62	324.06	257.10	379.62
	Ci0	368.80	323.02	247.06	373.37
H7	*Ci1*	52.74	51.93	48.89	50.88
	Mean	52.08	51.84	48.08	50.60
	Ci0	51.42	51.75	47.27	50.32

$\approx$ 20 steps, and lastly V necessitates $\approx$ 18 steps. For host H1, R requires about $\approx$ 33 steps to validate all the vulnerabilities, while vulnerability-based cost functions V and VP necessitate $\approx$ 28 steps, and lastly P necessitates $\approx$ 31 steps. Host H2 requires about $\approx$ 43 steps to validate all the vulnerabilities using random-based cost function R, while platform-based cost functions P and VP necessitate $\approx$ 43 steps, and lastly V necessitates $\approx$ 42 steps. All cost functions of host H3 require $\approx$ 36 steps to terminate. In host H4, R requires about $\approx$ 92 steps to validate all the vulnerabilities, while cost functions P and V necessitate $\approx$ 64 steps, and lastly VP necessitates $\approx$ 86 steps. For host H5, all cost functions require $\approx$ 12 steps to terminate. Host H6 requires about $\approx$ 359 steps to validate all the vulnerabilities for random-based cost function R, while P necessitates $\approx$ 322 steps, V necessitates $\approx$ 232 steps, and lastly VP necessitates $\approx$ 366 steps. Lastly, in host H7 all cost functions except V require about $\approx$ 51 steps to validate all the vulnerabilities, while V necessitates $\approx$ 46 steps.

Compared to the result obtained in Table 5.7 of Section 5.3 (Version Optimization), it can be immediately noticed that this new approach reduces significantly the overall mean number of steps needed for termination, across all comparable strategies, consistently with the result on the environment without any other prior optimization. Differently from the previous result, applying the proposed approach on top of an already optimized environment yields diminishing returns with respect to applying the same approach on a non optimized environment. In particular, it can be noticed that while not all the hosts are affected positively by this new approach (e.g. all cost functions on H5 achieve a reduction of -25.05% at minimum, -11.24% on average), the average reduction in the mean number of steps required to terminate across all hosts and comparable strategies is still significant, at 52.38%. The few, negative or neutral results are expected for hosts with few vulnerabilities and platforms (H5 has already been discussed, while cost functions on H0 achieve -2.44% at minimum, 7.55% on average) considering that the proposed host-platform graph structure $\mathcal{HPG}$ requires the human agent to validate multiple nodes (i.e. attributes of a platform) in order to validate a single platform. Conversely, the real power of $\mathcal{HPG}$ is its ability to discard multiple platforms with one single interaction with the human agent. Indeed, by taking a look at Table 5.6 of Section 5.3 (Version Optimization), it can be immediately noticed how the hosts in which this new approach obtains the best results, are also the same hosts which have the highest amount of platforms discarded during the validation process, with the best average results across all cost functions being seen in hosts H2, H3 and H7, amounting to $\geq 85.13\%$. All three hosts have a *false positive* ratio among their platforms of $\geq 98.5\%$, the highest of all other hosts in the considered case study. Conversely, the worst performing hosts H0 and H5 which sit at a reduction in the number of required steps of 7.55% and -12.02% respectively have a false positive ratio among their platforms of 92.6% and 94.1% respectively, the lowest of the proposed case study. This property is also at the base for another observation that can be made on these experimental results, as an inspection of the results reveals that this new approach continues to benefit the most vulnerability-based cost functions V and VP and random-based cost function R even on-top of an already optimized environment, with the average, minimum and maximum reduction of the number of steps needed for completion being

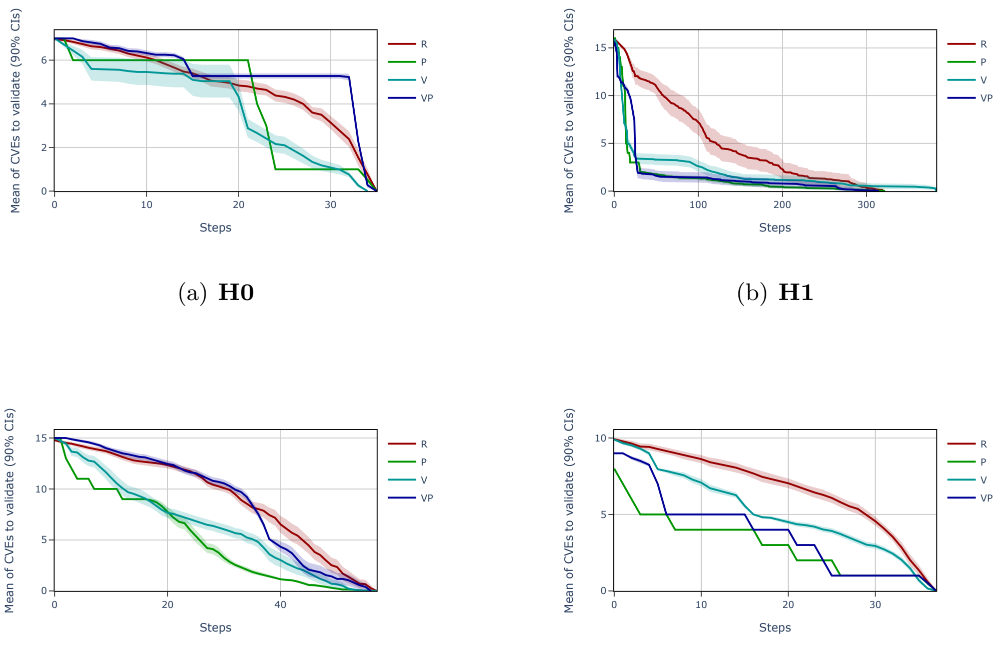
- 68.49% on average for cost function R when compared against RN, with a maximum of 88.81% on H7 and a minimum of 12.62% on H5.
- 52.36% on average for cost function R when compared against SR, with a maximum of 86.81% on H3 and a minimum of -15.67% on H5.
- 42.50% on average for cost function P when compared against PP, with a maximum of 86.76% on H3 and a minimum of -11.57% on H5.
- 46.74% on average for cost function V when compared against VS, with a maximum of 86.42% on H3 and a minimum of -20.46% on H5.
- 51.80% on average for cost function VP when compared against VP, with a maximum of 86.59% on H3 and a minimum of -25.05% on H5.

Manual inspection of the execution traces has shown evidence that random-based cost function R is the most impacted on average (average reduction of steps of 68.49% when compared to RN, and 52.36% when compared against SR across all hosts), while vulnerability-based cost functions come second, at 46.74% for V when compared against VS and 51.80% for VP when compared against VP, across all hosts. When compared against PP, P has the least benefit from this proposed approach, reaching a still respectable 42.50% reduction on the number of steps required to terminate on average.

5.5.3 Considerations on the evolution of the non-optimized environment

A first glance at the mean number of steps required for termination (Table 5.13) reveals that a consistent difference between random (R) and semantic driven cost functions (P, V and VP) can be immediately noticed only for hosts H4 and H6. Hosts H2, H3, H5 and H7 do not apparently exhibit any difference between different cost functions, and hosts H0 and H1 present alternating differences between the cost functions. This is due to (i) H0 and H5 having few vulnerabilities insisting on few, distinct platforms, (ii) H2 and H3 having vulnerabilities with large OR configurations of distinct platforms which are incompatible with their respective hosts, (iii) H1 and H7 having a large number of mutually-exclusive versions referred to few products (e.g. in H7 each product has on average 10 versions), and (iv) the fact that the result shown in the table is the final state of the application of the proposed approach, i.e. the average number of steps necessary to validate or discard all vulnerabilities in $\mathcal{VI}_{raw}$. These findings make it interesting to analyze the approach's evolution over the number of steps. Thus, Figure 5.8 has been plotted to visually analyze the average number of CVE vulnerabilities that still have to be validated after x steps of each exploration driven by each cost function.

The first result that can be noticed by analyzing the figure is that despite the fact that the number of steps taken to completely validate or discard each vulnerability are comparable for hosts H2, H3, H5 and H7, as shown in Table 5.13, "random-



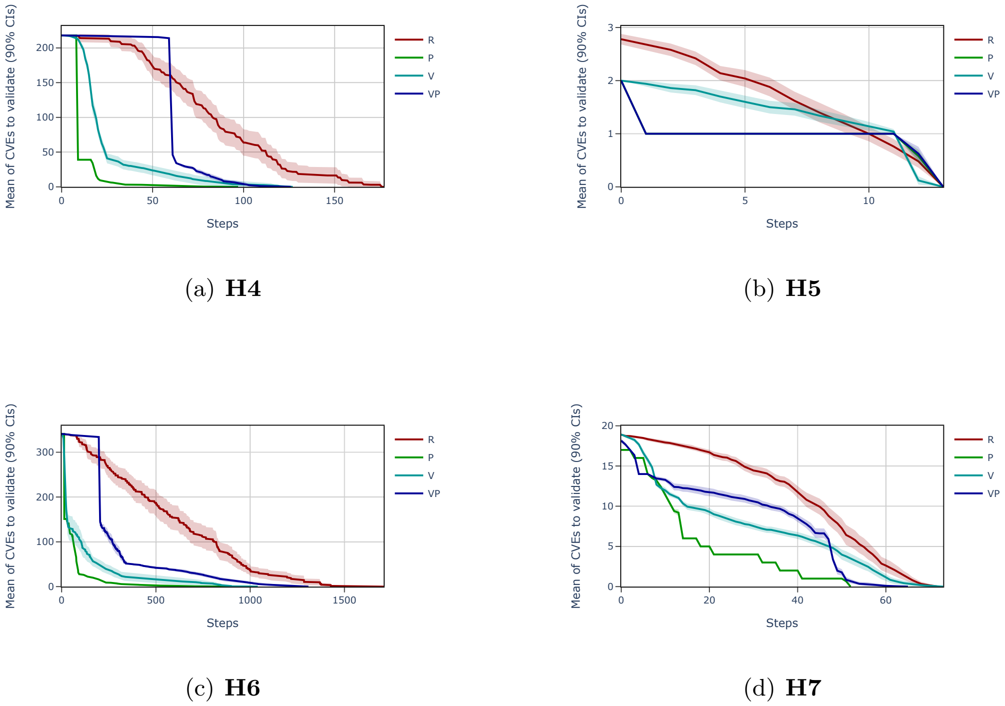
(a) **H0**

(b) **H1**

(c) **H2**

(d) **H3**

Figure 5.8. Mean of the CVEs that still have to be validated after each step of the exploration of $\mathcal{HPG}$ driven by each cost function. H0 and H1 represent *Office* hosts, H2 and H3 represent *Server* hosts.



(a) **H4**

(b) **H5**

(c) **H6**

(d) **H7**

Figure 5.8. Mean of the CVEs that still have to be validated after each step of the exploration of $\mathcal{HPG}$ driven by each cost function. H4 and H5 represent *Development* hosts, H6 and H7 represent *Generic* hosts.

based" approach represented by the cost function R continue to present the worst evolution with respect to every other considered cost function, even if by a smaller margin than the previous cases. This is easily noticed in the figure and confirmed in Table 5.15, for which the number of vulnerabilities still left to validate does not decrease significantly (below 50%) until 58.6% of the total elapsed steps, for R.[17] This is expected, since "random" approaches do not consider any semantics while guiding the exploration of $\mathcal{HPG}$, i.e. deciding the order of the platforms to submit to the human agent, and therefore proposed platforms may not be significantly impactful with respect to the vulnerabilities to be validated.

The staircase-like trend originally found on "vulnerability-driven" strategies VS and VP is now found on "platform-driven" cost functions P and VP. This is not a limitation per se, but a characteristic of how these cost functions guide the exploration of the graph. In particular it can be seen that each "plateau" is followed by a sharp drop (e.g. H4 ad H6 discard ≈ 150 vulnerabilities in a single step each) in the number of vulnerabilities left to validate or discard. Manual inspection of the execution traces reveals that this behaviour is a symptom of a breadth-first exploration of the host-platform graph $\mathcal{HPG}$, aggravated by the presence of many OR branches in the CPE configuration trees of certain CVEs. Indeed P and VP are either capable of guiding the exploration towards a platform which is present in the given host h_i and that validates (i.e. solves at least one configuration of) many

[17] 68.3% for $\geq 66\%$, 83.0% for $\geq 90\%$.

Table 5.15. Mean percentage of steps required for each exploration of $\mathcal{HPG}$ driven by each cost function to achieve respectively $\geq 50\%$, $\geq 66\%$ and $\geq 90\%$ accuracy of the vulnerability inventory $\mathcal{VI}$ (i.e. validation progress), across every host.

Cost Function	CVE validated		
	50%	66%	90%
R	58.6%	68.3%	83.0%
P	19.7%	37.5%	56.1%
V	34.8%	47.1%	65.4%
VP	39.4%	55.0%	68.1%

vulnerabilities of h_i within the first few steps, or tend to spend a lot of time "floating" on the surface, discarding all the platforms which are part of the OR branches of all the configurations of each vulnerability down to the last one, before cascading down. This is reflected by Table 5.15, for which the number of vulnerabilities still left to validate does not decrease significantly (below 50%) until 19.7% and 39.4% of the total elapsed steps, for P and VP respectively. Interestingly, it only takes 56.1% and 68.1% of the total elapsed steps for P and VP respectively to validate or discard $\geq 90\%$ of all the vulnerabilities in a given host.[18]

Lastly, the vulnerability-based cost function V sits in between random-based and vulnerability-based cost functions, with a normal tendency to decline over time, without any quirks (no staircases, no asymptotic evolution). This is reflected in the fact that under this cost function, the number of vulnerabilities still left to validate does not decrease significantly (below 50%) until 34.8%,[19] similarly to VP.

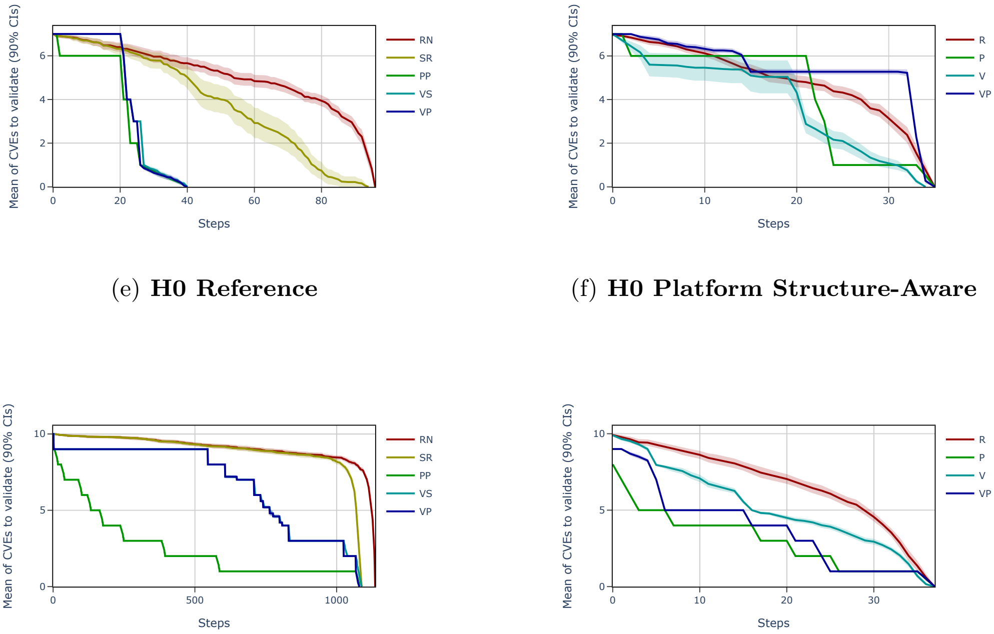
Figure 5.9 compares the evolution of the average number of CVE vulnerabilities that still have to be validated after x steps of each platform ordering strategy for hosts H0 and H3. In this case, the reduction in the number of steps required for termination as well as the evolution of the different strategies with respect to the number of steps are apparent, and are very different if compared against the references of Section 5.2 (Vulnerability Filtering). This is due to the fact that the advancement proposed in this section takes on a completely different approach on the issue by leveraging the Host-Platform Graph $\mathcal{HPG}$ in order to enhance the quality of the considered inventories. Some similarities can still be observed between the two graphs, however. Most notably: (i) all strategies seem to terminate approximately in the same "spot" for both methodologies in H3 and (ii) a staircase-like trend persists in cost function P and strategy PP as well as VP for both considered hosts. Moreover, this feature which is also present in strategy VS is most notably not present anymore in cost function V. As already stated before, this is due to the fact that while the underlying data (i.e. platforms in CPE format) remains the same, the advancement proposed in this section takes a different approach with respect to the reference methodology proposed in Section 5.2 (Vulnerability Filtering).

5.5.4 Considerations on the evolution of the already-optimized environment

A first glance at the mean number of steps required for termination (Table 5.14) reveals that a consistent difference between random (R) and semantic driven cost functions (P, V and VP) can be immediately noticed only for hosts H1 and H4. Hosts H2, H3 and H5 exhibit small, alternating differences between different cost functions, and hosts H0, H6 and H7 present alternating differences between the

[18] 37.5% and 55.0% respectively for $\geq 66\%$.

[19] 47.1% for $\geq 66\%$, 65.4% for $\geq 90\%$.



(e) **H0 Reference**

(f) **H0 Platform Structure-Aware**

(g) **H3 Reference**

(h) **H3 Platform Structure-Aware**

Figure 5.9. Mean of the CVEs that still have to be validated after each step of the prioritization strategies. Comparison between the results of Section 5.2 (Vulnerability Filtering) and Section 5.5 (Structure-Aware). H0 represents an *Office* host, H3 represents a *Server* host. Keep in mind that cost function R is comparable to strategies RN and SR, cost function P is comparable to strategy PP, cost function V is comparable to strategy VS and cost function VP is comparable to strategy VP (The colors match).

cost functions. This is due to (i) H0 and H5 having few vulnerabilities insisting on few, distinct platforms, (ii) H2 and H3 having vulnerabilities with large OR configurations of distinct platforms which are incompatible with their respective hosts, (iii) H1 and H7 having a multiple mutually-exclusive versions or version ranges referred to few products (e.g. in H7 each product has on average 2 versions), and (iv) the fact that the result shown in the table is the final state of the application of the proposed approach, i.e. the average number of steps necessary to validate or discard all vulnerabilities in $\mathcal{VI}_{raw}$. These findings make it interesting to analyze the approach's evolution over the number of steps. While these issues appear also in the case of a non-optimized environment, the reduction in number of platforms due to the version optimization has an effect on the magnitude of these issues. As an example, the cost function V gains efficacy on H7, in H1 all semantic-driven cost functions gain efficacy when compared to R and for H6, cost function VP looses efficacy due to the platform compression caused by encoding version ranges increasing the effectiveness of R. Thus, Figure 5.10 has been plotted to visually analyze the average number of CVE vulnerabilities that still have to be validated after x steps of each exploration driven by each cost function.

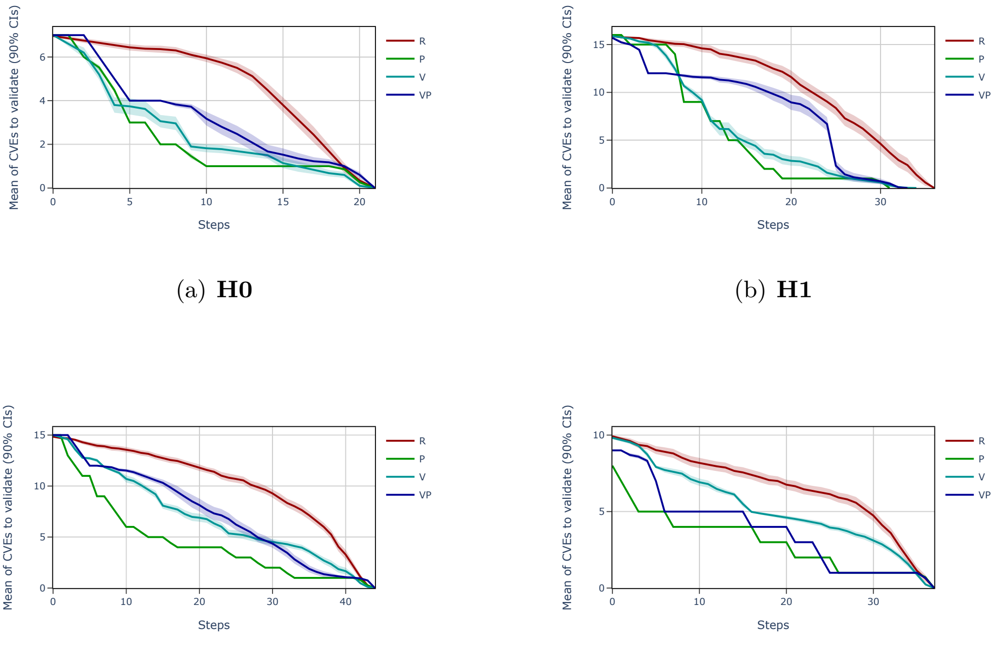
The first result that can be noticed by analyzing the figure is that despite the fact that no cost function appears to always minimize the number of steps taken

Table 5.16. Mean percentage of steps required for each exploration of $\mathcal{HPG}$ driven by each cost function to achieve respectively $\geq 50\%$, $\geq 66\%$ and $\geq 90\%$ accuracy of the vulnerability inventory $\mathcal{VI}$ (i.e. validation progress), across every host.

Cost Function	**CVE validated**		
	50%	66%	90%
R	70.0%	80.2%	91.7%
P	16.2%	37.7%	65.0%
V	35.8%	48.4%	75.6%
VP	45.4%	66.4%	83.1%

to completely validate or discard each vulnerability for hosts H2, H3 and H5, as shown in Table 5.14, the proposed "random-based" approach represented by the cost function R continues to present the worst evolution with respect to every other considered cost function, even if by a smaller margin than the previous cases. This is easily noticed in the figure and confirmed in Table 5.16, for which the number of vulnerabilities still left to validate does not decrease significantly (below 50%) until 70.0% of the total elapsed steps, for R.[20] This is expected, since "random" approaches do not consider any semantics while guiding the exploration of $\mathcal{HPG}$, i.e. deciding the order of the platforms to submit to the human agent, and therefore proposed platforms may not be significantly impactful with respect to the vulnerabilities to

[20] 80.2% for $\geq 66\%$, 91.7% for $\geq 90\%$.



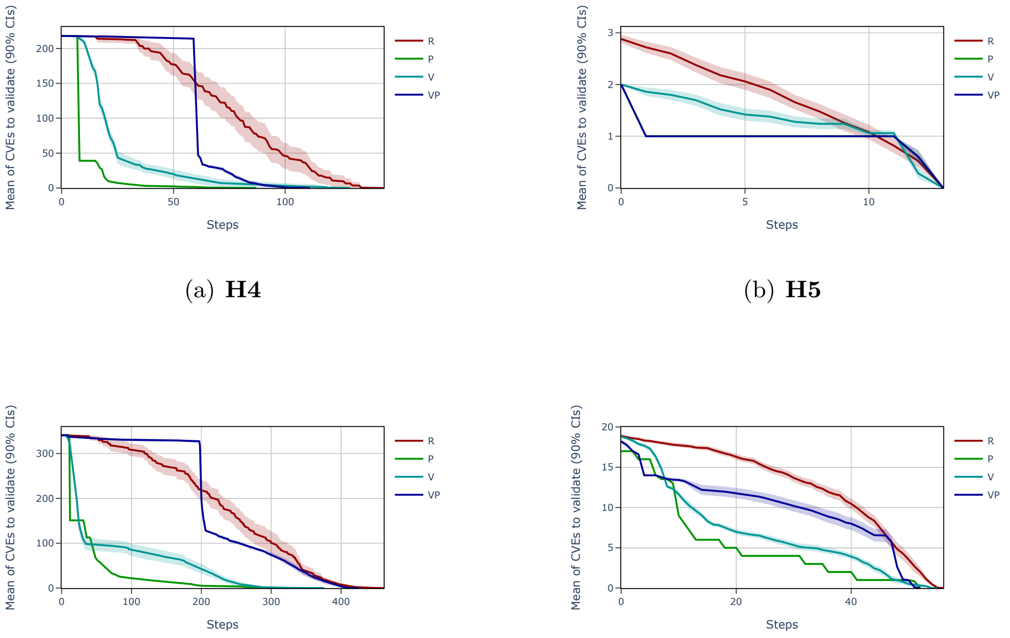
(a) **H0**

(b) **H1**

(c) **H2**

(d) **H3**

Figure 5.10. Mean of the CVEs that still have to be validated after each step of the exploration of $\mathcal{HPG}$ driven by each cost function. H0 and H1 represent *Office* hosts, H2 and H3 represent *Server* hosts.



(a) **H4**

(b) **H5**

(c) **H6**

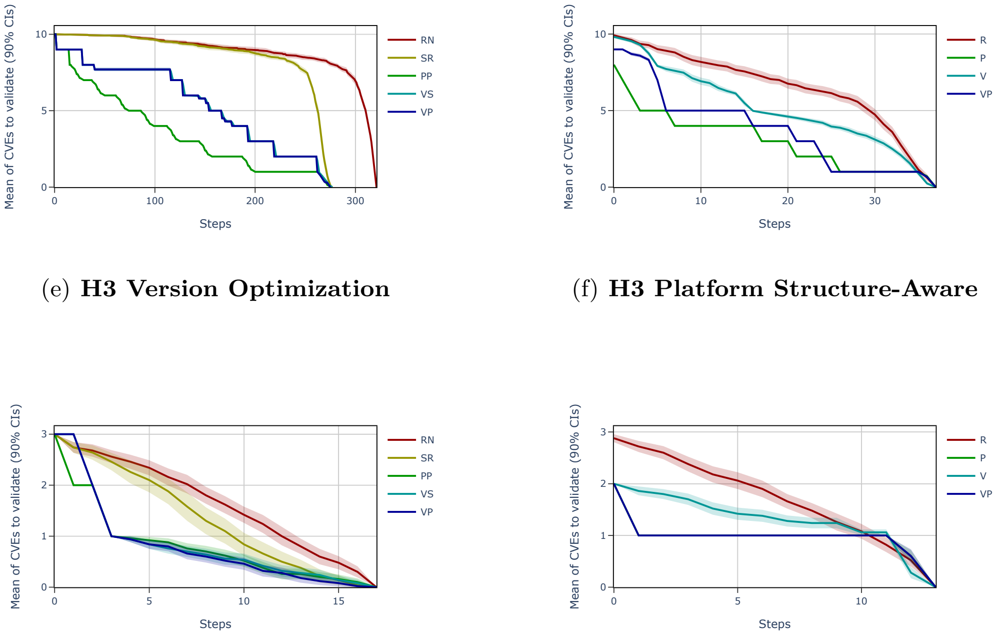
(d) **H7**

Figure 5.10. Mean of the CVEs that still have to be validated after each step of the exploration of $\mathcal{HPG}$ driven by each cost function. H4 and H5 represent *Development* hosts, H6 and H7 represent *Generic* hosts.

be validated.

The staircase-like trend originally found on "vulnerability-driven" strategies *VS* and *VP* is now found on "platform-driven" cost functions *P* and *VP*. This is not a limitation per se, but a characteristic of how these cost functions guide the exploration of the graph. In particular it can be seen that each "plateau" is followed by a sharp drop (e.g. H4 ad H6 discard ≈ 150 vulnerabilities in a single step each) in the number of vulnerabilities left to validate or discard. Manual inspection of the execution traces reveals that this behaviour is a symptom of a breadth-first exploration of the host-platform graph $\mathcal{HPG}$, aggravated by the presence of many OR branches in the CPE configuration trees of certain CVEs. Indeed *P* and *VP* are either capable of guiding the exploration towards a platform which is present in the given host h_i and that validates (i.e. solves at least one configuration of) many vulnerabilities of h_i within the first few steps, or tend to spend a lot of time "floating" on the surface, discarding all the platforms which are part of the OR branches of all the configurations of each vulnerability down to the last one, before cascading down. This is reflected by Table 5.16, for which the number of vulnerabilities still left to validate does not decrease significantly (below 50%) until 16.2% and 45.4% of the total elapsed steps, for *P* and *VP* respectively. Interestingly, in the already-optimized case *P* and *VP* diverge significantly in their evolution, requiring 65.0% and 83.1% of the total elapsed steps for *P* and *VP* respectively to validate or discard ≥ 90% of all the vulnerabilities in a given host.[21] Manual inspection of the execution traces has revealed that the cause is due to *VP* enforcing a *vulnerability-first* approach, for which the severity of each vulnerability is a stronger factor in the ordering of the platforms than the OR−AND structure of the configuration trees. This in turn means that the exploration guided by *VP* will not focus on the most impactful platforms first (which is exactly what *P* does), but will select first platforms associated to "severe" CVEs, and then only among those select the most impactful.

Lastly, the vulnerability-based cost function *V* sits in between random-based and vulnerability-based cost functions, with a normal tendency to decline over time, without any quirks (no staircases, no asymptotic evolution). This is reflected in the fact that under this cost function, the number of vulnerabilities still left to validate does not decrease significantly (below 50%) until 35.8%[22] steps.



(e) **H3 Version Optimization**

(f) **H3 Platform Structure-Aware**

(g) **H5 Version Optimization**

(h) **H5 Platform Structure-Aware**

Figure 5.11. Mean of the CVEs that still have to be validated after each step of the prioritization strategies. Comparison between the results of Section 5.3 (Version Optimization) and Section 5.5 (Structure-Aware). H3 represents a *Server* host, H5 represents a *Development* host.

Figure 5.11 compares the evolution of the average number of CVE vulnerabilities that still have to be validated after x steps of each platform ordering strategy for hosts H3 and H5. Also in this case, the reduction in the number of steps required for termination as well as the evolution of the different strategies with respect to the number of steps are apparent, and are very different if compared against the references of Section 5.3 (Version Optimization). This is due to the fact that the advancement proposed in this section takes on a completely different approach on the

[21] 37.7% and 66.4% respectively for $\geq$ 66%.

[22] 48.4% for $\geq$ 66%, 75.6% for $\geq$ 90%.

issue by leveraging the Host-Platform Graph $\mathcal{HPG}$ in order to enhance the quality of the considered inventories. Some similarities can still be observed between the two graphs, however. Most notably: (i) all strategies seem to terminate approximately in the same "spot" for both methodologies and (ii) a staircase-like trend persists in cost function P and strategy PP as well as VP. Moreover, this feature which is also present in strategy VS is most notably not present anymore in cost function V. As already stated before, this is due to the fact that while the underlying data (i.e. platforms in CPE format) remains the same, the advancement proposed in this section takes a different approach with respect to the base methodology proposed in Section 5.2 (Vulnerability Filtering).

5.6 Experimental Evaluation of Computational Pipeline

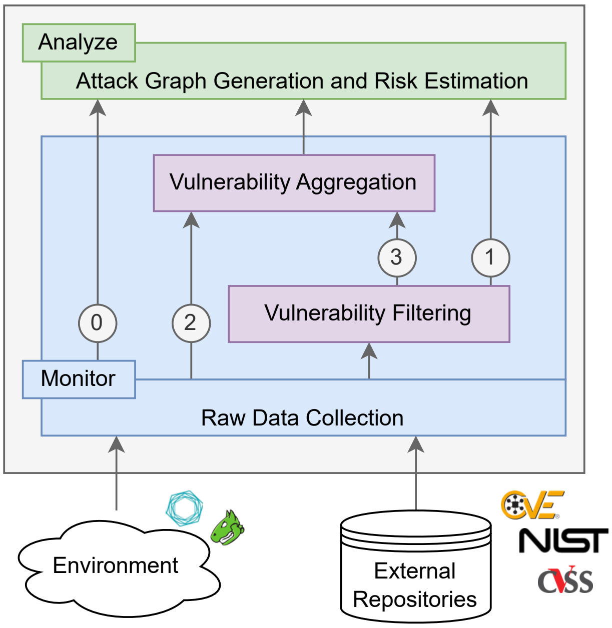


Figure 5.12. High-Level architecture of the proposed computational pipeline.

Section 4.2 introduced a computational pipeline (Figure 5.12) to be integrated in the data collection aggregation and integration component of a self protecting system. The first aim of this computational pipeline is to enhance the quality of automatically generated inventories which usually derive from vulnerability scanners monitoring the environment, through the application of a filtering module which implements the methodologies, strategies and approaches described in Sections 4.3 to 4.6. The second aim of this computational pipeline is to allow the analyst to trade-off accuracy in favour of scalability during the attack graph-based risk analysis through the use of an aggregation module, which implements the methodology described in Section 4.7.

Thus, this section will present the experimental evaluation of the proposed computational pipeline, as well as its vulnerability filtering and aggregation components to assess their benefit and limitations when included in the self-protecting system. This section will also describe the additional experimental settings used to perform the evaluation of the computational pipeline, as well as an evaluation of the results regarding scalability and risk estimation.

5.6.1 Setting Configuration

In order to perform the validation of the whole computational pipeline, several network environments have been produced using the environment described previously in Section 5.1 as a base.

To test the scalability, the base environment has been expanded in order to achieve adequate complexity in order to perform a meaningful analysis of the scalability of the analysis processes. In particular, several different LAN networks (i.e., networks with full reachability between hosts) of different sizes have been considered, where multiple replicas of the 8 hosts described in Section 5.1 (H0-H7) have been included. Such topology has been considered for different motivations: (i) a LAN setting is the common setup used for the validation of the state of the art attack graph algorithms [50, 87] and (ii) LANs represent a common building block of a larger enterprise network and are currently the parts of the network that have the largest impact on scalability.

In addition, a complex network environment has been constructed using an inspection of the network of the department of Computer, Control and Management Engineering at Sapienza university as a base, which has been carried out with network mapping software and manual inspection. After an anonymization and sanitization pass, the results of this inspection revealed that the network of the department taken as a reference is composed of around 250 devices arranged in 5 LANs connected through a router in a star topology. In order to represent these 250 devices, part of the environment described previously in Section 5.1 has been used. More precisely, the 8 hosts of the previously generated environment have been arranged and duplicated within the new complex network environment in order to mimic the real network devices. The arrangement of the hosts in the network LANs has been achieved through each hosts's type (e.g. "server" hosts have been assigned to the DMZ). This has allowed full visibility over the filtered (i.e. ground truth) and unfiltered vulnerabilities of these hosts in the network. Table 5.17 summarizes the configurations of the hosts and the number of (false and true) platforms and vulnerabilities identified by the scanners.

The results of the proposed computational pipeline have been compared with the current state of the art for attack graph-based self-protecting systems [38]. In particular, the state of the art implementation [19] of an attack graph has been used as a baseline reference (SoA in the figures).[23] This reference attack graph has been fed with the vulnerability inventory manually constructed as ground truth to compare the benefit of the proposed approach with respect to a solution that is not biased from initial false positives (i.e., a good scenario has been considered for the state of the art).

The evaluation of the whole pipeline has been conducted on a Linux server with an Intel(R) Xeon(R) Gold 6248 CPU 2.50GHz and 256 GB of memory.

[23] The Attack Graph follows the NETSPA model, in which edges encode the vulnerabilities which an attacker can take to "jump" from one privilege@host to another.

Table 5.17. Composition of the experimental environment. A true positive (TP) is an element identified by the scanner and validated manually in the ground truth while a false positive (FP) is an element reported by the scanner but not present in the system and in the ground truth.

Host	**OS**	**Type**	**LAN Patecipation**	**Platforms**		**Vulnerabilities**	
				TP	FP	TP	FP
H0	Ubuntu Linux 22.10	office	ADMIN, INF, GST, AUTO	2	94	2	5
H1	Debian Linux 10	office	ADMIN, INF, GST, AUTO	3	1010	8	8
H2	Ubuntu Linux 18.04.6 LTS	server	DMZ	5	1211	6	9
H3	Debian Linux 10	server	DMZ	1	1135	1	9
H4	Ubuntu Linux 15.10	development	INF, GST, AUTO	10	2098	211	7
H5	Debian Linux 11	development	INF, GST, AUTO	1	33	1	2
H6	Ubuntu Linux 15.04	generic	ADMIN, INF, GST, AUTO	31	6017	321	20
H7	Debian Linux 11	generic	ADMIN, INF, GST, AUTO	3	1601	5	14

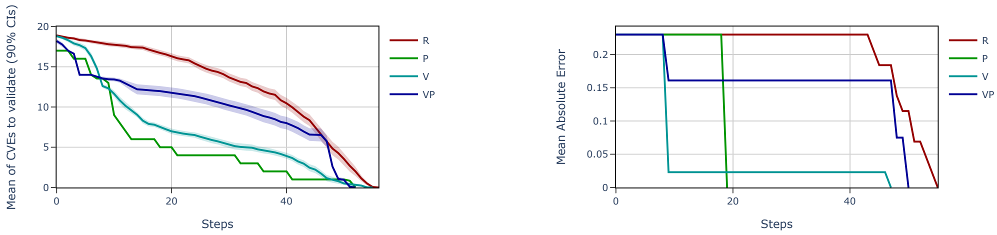
5.6.2 Vulnerability Filtering Component Evaluation

The first step of the pipeline (as described in Figure 5.12) is vulnerability filtering, whose goal is the improvement of the accuracy of risk estimation by filtering out false positives coming from vulnerability scanners.

To perform an evaluation of this component, it is necessary to examine how different filtering approaches affect the results of the cyber risk analysis and how they affect the attack graph size in a mesh network composed by the 8 hosts (i.e., a configuration resembling the network of a small company with heterogeneous hosts).

In this configuration, the the state of the art (SoA) reference attack graph contains 629 vulnerabilities as well as 74 false positive vulnerabilities detected by the scanners. By computing the risk of a vulnerability as its CVSS base score, these false positives directly impact the risk estimation for four of the eight hosts (H1, H2, H4 and H6 are unaffected) in the environment with a Mean Absolute Error (MAE) of the risk estimation of 0.38 on average (i.e., the value of the risk of a target differs from its risk in the state of the art reference attack graph by 0.38 on average), with a maximum of 0.59 on H3 and a minimum of 0.17 on H0. This result stresses the importance of filtering false positives from vulnerability scans due to their negative impact on cyber risk evaluation. This result has been obtained by computing the risk value of a host h_i as the highest risk value (i.e. CVSS base score) among all the vulnerabilities pertaining to host h_i, which maps to a simple worst-case scenario. And while using the CVSS base score may be a simplistic computation of a risk value, if more elements are factored in the risk calculation it is possible for the Mean Absolute Error of the risk estimation to increase even beyond the one presented in this case study.

Since this early result highlights that the filtering process is beneficial for the risk estimation in four of the eight hosts present (i.e. without any filtering process the risk estimation performed on these hosts return results which are distant from the truth) it becomes interesting to study the evolution of the Mean Absolute Error of the risk estimation at each step of the filtering process.



(a) **H7 Mean of CVEs to validate**

(b) **H7 MAE**

Figure 5.13. Comparison between the mean of the CVEs that still have to be validated after each step of the exploration of $\mathcal{HPG}$ driven by each cost function (5.13(a)) and the variation between the risk evaluated in partially filtered attack graphs and in the ground truth across elapsed filtering steps (5.13(b)). This figure considers host H7, which represents a *Generic* host.

Figure 5.13 shows the Mean Absolute Error of the worst-case risk estimation performed on H7 after each step of the filtering process using the methodology described in Section 4.6 (Structure-Aware). For the evaluation of the proposed computational pipeline, the structure-aware methodology (Section 4.6) has been chosen in conjunction with the platform vulnerability optimization (Section 4.4) due to its performance in the proposed scenario with respect to other proposed methodologies.

The first relevant aspect to point out is that the false positive vulnerabilities in H7 (14 CVE) cause an average error in the risk estimation of 0.23% (i.e. MAE = 0.23 at step 0, without any filtering applied). In addition, it can be noticed that platform-based approach P is the quickest approach to achieve a MAE of 0, only needing 19 steps to do so. Meanwhile all other approaches V, VP and R necessitate respectively 47, 50 and 55 steps to reach a MAE of 0 (i.e. achieve the same risk estimation as with the ground truth). The second relevant aspect that can be derived from the graph is that the MAE has a staircase-like evolution, regardless of the adopted approach. Manual inspection of the execution traces has revealed that this is a bias that derives from the source of the CVSS data used to evaluate the risk during this case study, NVD. Indeed, elements of the CVSS vector string in NVD are scored according to a well defined, qualitative scale. This in turn limits the numerical values assigned to each CVSS attribute to few well known values. And since these values are then factored in the CVSS base score formula, the possible values of the CVSS base score, used in this case study as the measure of risk of a vulnerability, are limited, non continuous and well defined. Thus, this non continuity is the main factor in the staircase-like trend of the MAE in Figure 5.13(b). By also comparing the evolution of the MAE with the mean of CVEs which are still left to validate across all filtering approaches for H7, it can be noticed that there exists an association between the drops in the “staircases” of Figure 5.13(b) (MAE) and the “plunges” taken in Figure 5.13(a) (Mean CVE). This is expected, since decreasing the number of CVEs which are still to be validated (i.e. validating or discarding vulnerabilities) in turn has the possibility to decrease both the number false positive vulnerabilities of H7, as well as the deviation from the ground truth (MAE). Indeed, the sharp drop shown by filtering approach P between steps 10 and 20 of Figure 5.13(a) is reflected by the MAE reaching zero at step 19 in Figure 5.13(b). Similarly, the sharp drop in the number of CVEs left to validate around step 50 for VP in Figure 5.13(a) is reflected by the MAE reaching zero at step 50 in Figure 5.13(b). The same can be observed for filtering approach V, in which a sharp drop during the early stages of validation can be seen for both the MAE and the number of CVEs to be validated around step 10 (MAE drops below 0.05).

Another direct effect of the elimination of false positive vulnerabilities from the hosts of the network is the reduction of the size of the attack graph, in addition to the improvement of the accuracy of risk analysis. To measure the attack graph size reduction, Figure 5.14 has been plotted to show how each step of the different filtering approaches affects the number of edges of the attack graphs built using the partially filtered inventories (i.e. filtering is halted upon reaching x steps).

All filtering approaches reduce the number of edges from 5032 to 4440, corresponding to a space reduction of 11.76% of the attack graph on the proposed 8-host network. The first result that can be noticed by analyzing the figure is that the convergence rate to the ground truth is comparable for both the number of edges (Figure 5.14) and the attack graph risk accuracy (Figure 5.13). More in detail, it can be clearly noticed that filtering approaches P and V display the best overall performance, with a sharp drop in the number of edges of the attack graph during the first 50 steps, and a relatively flat curve to prune the remaining 6.36% and 9.33% edges derived from false positive vulnerabilities on the 8 hosts, respectively.

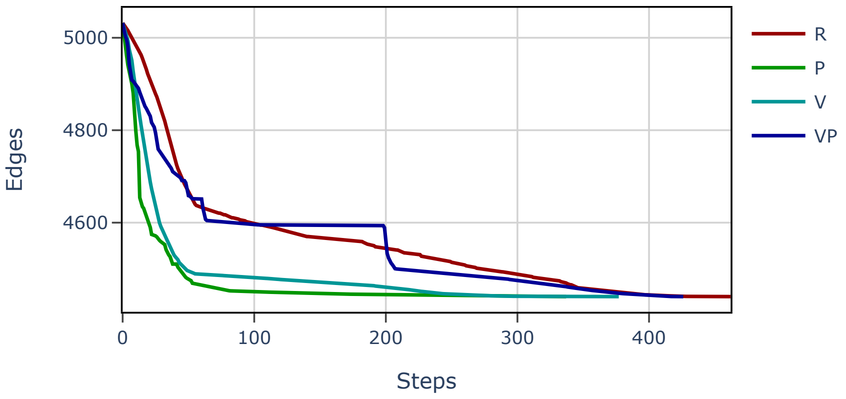


Figure 5.14. Number of edges in partially filtered attack graphs for elapsed filtering steps according to the different approaches. In this figure, each step affects all hosts of the environment concurrently, e.g. step 1 in the x-axis of the figure refers to the application of the fist step of multiple filtering approaches on all hosts of the network concurrently. The Attack Graph follows the NETSPA model, in which edges encode the vulnerabilities which an attacker can take to "jump" from one privilege@host to another (privileges are either: none, user or root, hosts are H0-H7).

Filtering approach *VP* shows initial good performance which then stalls in a flat staircase-like evolution which crosses *R* multiple times. *R* shows the worst overall performance, requiring about 462 steps to fully terminate. These are expected results indicating that the reduction of the attack graph edges follows the evolution of the accuracy improvement over the steps of the filtering approaches. The two results are however not equivalent, as the filtering methodology contributes to the elimination of the edges of the attack graph due to false positive vulnerabilities even while not directly contributing to the reduction in the error of the risk estimation of a host, i.e. eliminating multiple elements from the attack graph may not affect the risk estimation. This is apparent when considering the heavy staircase-like trend of Figure 5.13 with the smoother trend displayed by this figure.

To wrap up, the analysis performed in this section showed the advantages of filtering out false positives from the vulnerabilities in the inventories, in the proposed cyber risk analysis pipeline. Without any filtering methodology, the current state of the art solution for analyzing cyber risk is 38% less accurate on average and the attack graph has 11.76% more edges. Among the different approaches, *P* and *V* have the fastest convergence to the ground truth, resulting in a more accurate cyber risk analysis and reduction of the attack graph size, given a fixed budget of steps. To provide a comprehensive analysis of the advantages of the entire pipeline, the next section will detail the analysis of the effects of the different aggregation strategies.

5.6.3 Vulnerability Aggregation Component Evaluation

The vulnerability aggregation component described in Section 4.7 aims to reduce the size of the vulnerability inventory used to generate attack graphs, thus improving the scalability of the attack graphs in terms of size and complexity over time. This reduction in size, however, has the side effect of propagating information loss to cyber risk analysis processes which rely on the generated attack graph. Therefore, it is crucial to investigate the trade-off between the benefits of scalability and the loss of accuracy of the risk analysis across the three proposed aggregation strategies.

To this aim, Table 5.18 shows the reduction of the number of edges and the impact on the risk analysis (both in terms of the number of affected targets and Mean Absolute Error - MAE of the risk estimation) for the attack graphs obtained with the different aggregation strategies with respect to the state of the art reference attack graph derived from a mesh network of the 8 hosts.

Table 5.18. Attack graph risk accuracy and percentage of edge reduction for the different aggregation strategies according to the state of the art reference attack graph.

Aggregation Strategy	**Affected Targets**	**Risk MAE**	**AG edges reduction**
VA	0%	0	68%
RA	12.5%	0.04	87%
HA	87.5%	0.44	99%

Table 5.18 underlines a great advantage of vulnerability aggregation: a reduction between 68% and 99% of the attack graph size. Indeed, while the state of the art reference attack graph has 5032 edges, they are reduced to 1610 for Vulnerability-based Aggregation (VA), 654 for Risk-based Aggregation (RA), and 64 for Host-based Aggregation (HA). However, the analysis confirms the side effect that aggregation strategies may have on the accuracy of the cyber risk analysis. The most conservative aggregation (VA) does not impact the estimation of the risk value because it aggregates only vulnerabilities that have the same CVSS attributes.[24] Similarly, the impact of the RA strategy is small (12.5% of the hosts, MAE of 0.04) because it aggregates vulnerabilities according to features used in the risk model to define impact and likelihood of an exploit.[25] Conversely, the HA strategy results in 4 cases (50% of the total) in which the risk variation is more than 0.5. This worse accuracy is because this strategy aggregates the vulnerabilities independently from their attribute values. Thus VA, RA and HA present three possible aggregation strategies to be deployed by the aggregation sub-component of the proposed computational pipeline.

In order to correctly analyze the trade-off between scalability and accuracy, it is essential to delve deeper into the impact of the aggregation sub-component on the scalability of attack graphs. To achieve this objective, an additional analysis of the time and space complexity of attack graphs generated using aggregated vulnerability inventories is performed.

The scalability evaluation is performed by increasing the number of hosts in the network up to 700, while still using a mesh topology. The additional hosts were obtained by adding replicas of the 8 hosts detailed in Section 5.1 to the network.

Figure 5.15 shows the attack graph generation time (in seconds) for the different synthetic environments. The generation time is measured in seconds, and it includes

[24] In the risk model used in this case study, risk is equal to a CVE's CVSS base score.

[25] Only some CVSS attributes are used for the aggregation.

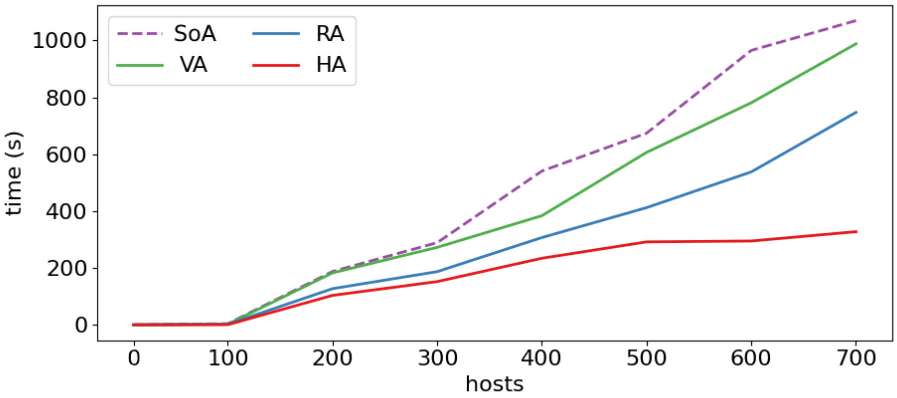


Figure 5.15. Generation time (in seconds) of the attack graphs according to the different aggregation strategies. The generation time includes the vulnerability aggregation time.

the time necessary to perform aggregation of the vulnerability inventory. The trends highlight the attack graph generation speed-up when adopting aggregation strategies, especially for large-size networks.

On average, there is a reduction of 8%, 33%, and 78% of the generation time for Vulnerability-based (VA), Risk-based (RA), and Host-based (HA) aggregation with respect to the state of the art reference attack graph, (SoA, dashed line in Figure 5.15). It is worth noting that the generation time is considerably increased when there is no aggregation (state of the art reference attack graph) and for strategies VA and RA, while it increases linearly for strategy HA. It is a result that one may expect because strategy HA keeps one vulnerability per network host, forcing by design a linear complexity in the network size.

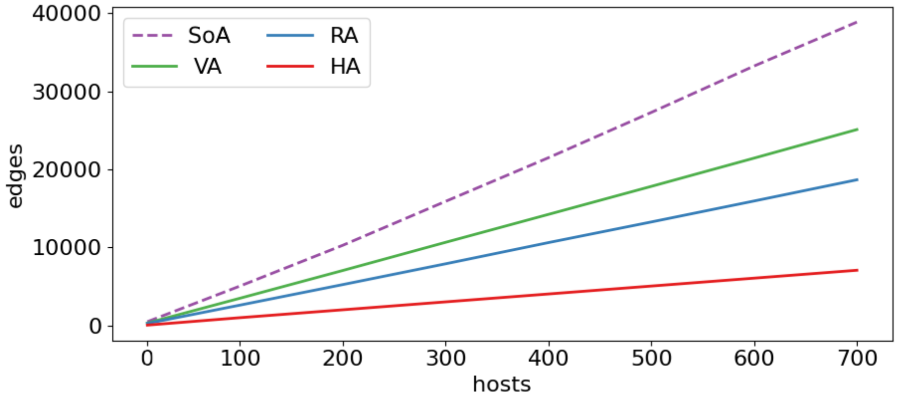


Figure 5.16. Number of edges of the attack graphs according to the different aggregation strategies. Only "useful" edges have been considered, i.e. edges between hosts, as well as edges between equal or increasing privileges of the same host.

This trend is confirmed by the space analysis reported in Figure 5.16, which summarizes the number of edges of the attack graphs for the number of hosts for each aggregation strategy. The number of edges is reduced by 40%, 55%, and 75% when adopting VA, RA, and HA strategies, respectively.

Thus, these two analyses (Figures 5.15-5.16) give a clear indication that adopting aggregation strategies provides a significant reduction of the attack graph structure (space, from 40% to 75%) as well as faster generation times (from 45% to 95%).

Concerning the analysis of the accuracy-scalability trade-off of large networks, it is important to remark that computing the attack paths is computationally expensive when performed in non-aggregated networks, and it is still an open problem in attack graph-based analysis [96, 107].

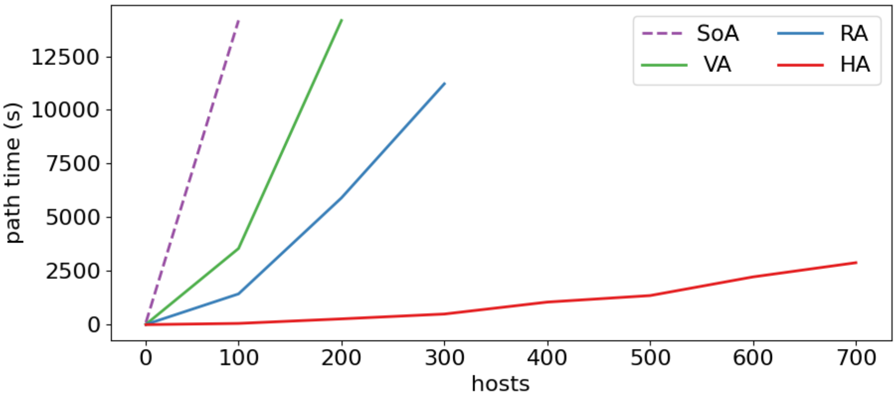


Figure 5.17. Attack path computation time (in seconds) for the different aggregation strategies.

Figure 5.17 reports the time (in seconds) required for the computation of the attack paths for 3 attack targets, randomly chosen among the hosts of the network, for the different network sizes. Since attack graphs are used to prevent and respond to cyber-attacks promptly, it has been assumed that when the attack paths computation needs more than 4 hours to be completed (14400 seconds), then the problem is intractable. When there is no aggregation strategy (state of the art reference attack graph, dashed line in Figure 5.17), the only size for which risk analysis can be computed in a tractable amount of time is the network with 8 hosts, for which the scalability-accuracy trade-off has been reported in the previous subsection. For networks with more than 100 hosts, the problem becomes intractable. In contrast, given the high reduction of attack graph size, it is still possible to compute attack paths for cyber risk analysis in tractable time when adopting aggregation strategies. In particular, the problem is tractable to up to 200 hosts when considering the VA strategy, for which the computation requires almost 4 hours. The RA strategy allows attack path computation of up to 300 hosts with a computation time of almost 3 hours. In contrast, the HA strategy scales up to 700 hosts for whom less than 1 hour is needed to compute attack paths.

Thus, aggregation strategies enable attack path computation for network sizes for which the problem would be intractable. In particular, with respect to attack graph scalability, the HA strategy outperforms all the other strategies, being linear to the network size.

Given the impossibility of computing the attack paths for networks with more than 100 hosts using the state of the art reference attack graph, the information loss of the aggregation strategies has been evaluated by analyzing the reachability accuracy for all the networks from 10 to 700 hosts. More in detail, given each pair of $\langle a_s, a_t \rangle$ where a_s is an attack source and a_t an attack target in state of the art reference attack graph, let an *s-t path* be an attack path from a_s to a_t. Then, for the attack graph obtained by applying aggregation strategies, the following elements have been evaluated:

1. True Positives (TP) are the s-t paths in the state of the art reference attack

graph and the aggregated attack graph.

2. False Positives (FP) are the s-t paths not in the state of the art reference attack graph but in the aggregated attack graph. It could happen because there may be new meta-vulnerabilities that do not exist in the state of the art reference attack graph.

3. False Negatives (FN) are the s-t paths in the state of the art reference attack graph but not in the aggregated attack graph. It could happen because the new meta-vulnerabilities may alter the attack graph topology, leading to missing paths.

4. True Negatives (TN) are the s-t paths neither in the state of the art reference attack graph nor the aggregated attack graph.

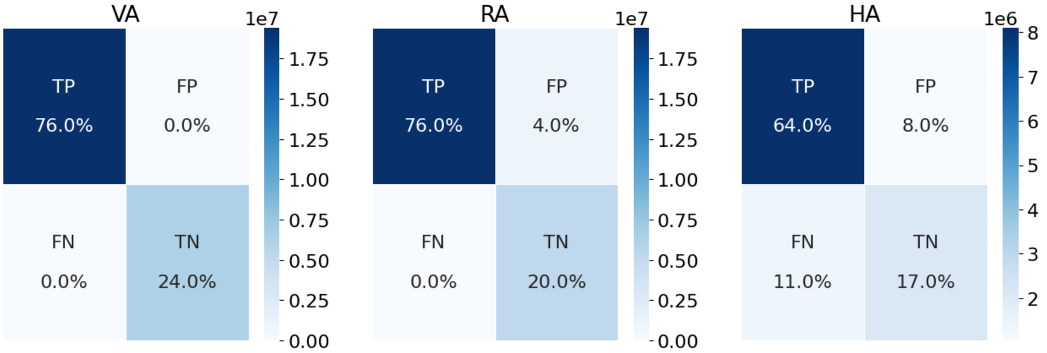


Figure 5.18. Confusion matrices for each aggregation strategy (VA=Vulnerability-based Aggregation, RA=Risk-based Aggregation, HA=Host-based Aggregation).

Figure 5.18 shows the confusion matrices of the reachable paths according to the different aggregation strategies. This figure underlines that the VA strategy has accuracy 1, as expected. Indeed, it only aggregates when all the features[26] of each vulnerability are the same, thus avoiding the generation of meta-vulnerabilities that do not exist in the state of the art reference attack graph. In contrast, RA and HA strategies have 0.96 and 0.81 accuracy, respectively. This is a result given from the presence of meta-vulnerabilities that may alter the original attack graph topology in which some paths may be wrongly present (at most 8% of the total number of paths) and others missing (11% of the paths only for HA strategy).[27] An interesting aspect regarding the RA strategy is that it does not contain false negatives, and for this reason, it misses no paths of the state of the art reference attack graph. Consequently, such a strategy is conservative because a lower accuracy may be caused only by additional paths that do not exist in the original attack graph.

This experimental evaluation analyzed the accuracy-scalability trade-off of the proposed pipeline's vulnerability filtering and aggregation components. It showed that the presence of false positive vulnerabilities from automatic scanners may project their lower accuracy in the risk analysis pipeline, thus resulting in inaccurate risk

[26] In the proposed case study, the attributes of the CVSS string.

[27] As a reminder, RA and HA generate new "meta-vulnerabilities" that inherit the worst features (i.e. CVSS attributes) from all aggregated vulnerabilities.

analyses. In contrast, aggregating the vulnerability inventories improves the attack graph scalability, allowing path computation in networks whose size is intractable with standard approaches (state of the art reference attack graph). On the other hand, a slight degradation of accuracy in the risk evaluation is the price to pay for the improved scalability. The degradation of the attack graph risk accuracy is mitigated by tolerating from 10% to 20% of uncertainty in the risk calculation.

5.6.4 Evaluation on a realistic network topology

As a last step, this section will show the practicality of the proposed solution by deploying the pipeline over a network having the same topology as the network of the department of Computer, Control and Management Engineering at Sapienza university. In particular, this complex network is composed of 5 LANs of different sizes (3 LANs one for each research area of the department, 1 administrative LAN and 1 DMZ) and each LAN is populated with virtual hosts consistent with the LAN scope as described in Table 5.17. The results are reported in Table 5.19 and are in line with previous results. In particular, it is interesting to note that the percentage of targets with different risk values is slightly higher for the RA strategy and slightly lower for the HA strategy. The reason for this difference may be attributed to the fact that the attack target has been selected in the DMZ, which contains 5 servers, which are less than all the 8 targets in the small LAN environment. This is also the reason for the slightly higher MAE of the risk, which, however, is still low (0.06 for RA, and 0.23 for HA). Considering the trade-off between the scalability parameters (the computation time) and the accuracy errors (the Risk MAE) it appears very advantageous to opt for the aggregation strategy HA because it has an MAE of only 0.23 but performs risk analysis in less than 20 minutes. In the case in which such an MAE cannot be tolerated, the RA strategy provides a much smaller MAE (0.06), requiring about 60 minutes more to compute all the attack paths.

Table 5.19. Attack graph risk accuracy, percentage of edge reduction, and path computation time for the different aggregation strategies in the real environment.

Aggregation Strategy	Affected Targets	Risk MAE	AG edges reduction	computation time
VA	0%	0	7%	≈ 200 min
RA	33%	0.06	25%	≈ 80 min
HA	66%	0.23	46%	≈ 20 min

5.7 Conclusion

This chapter performed a thorough evaluation on a real case study of the computational pipeline which aims to enhance the quality of information derived from the monitored environment as well as offer the possibility to trade accuracy in favour of the scalability of the attack graph-based risk analysis in self-protecting system. This pipeline integrates methodologies and algorithms aimed at attacking the problem of enhancing the quality of automatically generated inventories used during the cyber risk management process through a *filtering* sub-component, as well several aggregation strategies implementing an accuracy-scalability trade-off in state-of-the-art attack graph-based analysis methodologies through an *aggregation* sub-component.

The creation of this computational pipeline is motivated by the fact that while risk analysis and more broadly cyber risk management processes strive towards the best possible accuracy, this contrasts with the performance of attack graph-based methodologies. Indeed state-of-the-art attack graph-based methodologies are very powerful and potentially accurate models, but suffer from scalability issues forcing to opt for the computation of *approximated* attack graphs [68, 70, 43]. This introduces a certain degree of uncertainty in the risk estimation, where accuracy is traded off for scalability. The situation gets even worse if we consider that state-of-the-art attack graph-based self-protection systems feed the control loop with input data that are not completely accurate due to false positives affecting the monitoring probes. This results in a cumulative effect of accuracy loss, affecting first the data collection phase and then the attack graph and risk computation phase.

Thus, the main focus of this chapter has been the evaluation of the capability of algorithms and methodologies proposed in Chapter 4 with respect to: (i) attacking the problem of improving the quality (in terms of accuracy) of the automatically generated inventories without increasing the degree of intrusiveness in the monitored system, and (ii) achieving a trade-off between accuracy and scalability in attack graph-based analysis methodologies, through the use of semantic-aware aggregation.

To achieve a thorough evaluation, a methodology to produce several statistically relevant testing environments has been proposed and leveraged, in order to achieve a comparable, statistically relevant and congruous result across all the proposed methodologies and algorithms. Having a real testing environment has also been necessary not only in order to evaluate the single components, but also to carry out the evaluation of the whole proposed computational pipeline.

Lastly, the proposed algorithms and methodologies have been integrated into a modular architecture (computational pipeline) to be implemented during the Data Collection Aggregation and Integration process of a self protecting system. This computational pipeline has then been evaluated using a real scenario, derived from the network environment of the department of Computer, Control and Management Engineering at Sapienza university.

The result of the evaluation has shown that applying such computational pipeline yields good results both in terms of accuracy, with 74 false positive vulnerabilities being eliminated from the 8-host case study described in Section 5.1 which in turn amounts to a 11.76% space reduction of the resulting attack graph (from 5032 edges down to 4440), as well as when considering scalability of the attack graph-based risk analysis, with aggregation strategies capable of reducing the resulting attack graph size (i.e. number of edges) from 68% up to 99% with respect to a "good" state of the art (i.e. no false positives present) at the cost of at most 0.44 MAE of the resulting risk estimation.

It is important to note that the proposed processes, algorithms and computational pipeline derive from an in-depth study of current state of the art vulnerability and network scanning techniques, in conjunction with some of the repositories described in Chapter 3.

The results of the chapter represent a confirmation of the work done in the previous chapter, as well as a concretization through an actual case study of a first step toward the design and implementation of a solution capable of performing fully automated cyber risk assessment by refining and resolving problems that affect existing paradigms and methodologies adopted by state of the art self-protecting systems.

Chapter 6

Conclusions

This thesis represents a first step towards a solution able to enhance the quality of information supporting the cyber risk management process in self-protecting systems.

The motivation for this thesis lies in how state of the art Cyber Risk Management processes and methodologies expect and utilize information. As shown in the introduction to this thesis (Chapter 1), the Cyber Risk Management process relies on multiple sources of information, some of which derive from the monitored environment, some of which are stored in external repositories. The availability and the quality of these sources of information plays a critical role during Cyber Risk Management, directly influencing the quality in terms of accuracy and completeness of the related processes. This is especially relevant for ICT systems designed around self-protection (i.e. self-protecting systems), which is currently a desired property of many modern ICT systems as it enriches its features with the ability to detect and react to security threats at run-time. Recently, several solutions leveraging the attack graph model have been proposed to design and implement such self-protecting systems. While such systems take a first step towards effective self-protection, they do not consider: (i) the possibility of having non complete information in the external repositories, (ii) the possibility of having non accurate information in the inventories derived from the environment, and (iii) the limitations in terms of accuracy-scalability trade-off imposed by the usage of the attack graph model.

In order to study and address these problems, this thesis provided the following major contributions:[1]

- Chapter 3 presented a study of the external publicly available vulnerability repositories, in order to understand their structure, their semantics and how all these repositories can be integrated in a unified structure, able to provide the cyber risk management process with complete, accurate information.
- Chapter 4 proposed several methodologies to be integrated in a computational pipeline able to enhance the accuracy of the inventories derived from the environment by reducing the number of false positives contained within, as well as explicitly addressing and instrumenting the accuracy-scalability trade-off imposed by the attack graph model.

[1] A repository with the implementation and further documentation of the proposed methodologies is available at https://github.com/Marcvs101/enhancing-cyber-risk-management

- Chapter 5 provided a comprehensive evaluation of the methodologies which participate in the computational pipeline proposed in Chapter 4. The evaluation has been carried out on a case study as well as on a real scenario, derived from the network environment of the department of Computer, Control and Management Engineering at Sapienza university.

The following sections will present achievements of each contribution in detail.

6.1 Chapter 3 - Integrating Sources of Data to Support Automatic Correlation

Chapter 3 investigated the structure of the main public repositories used by the cyber security community to collect and share information about vulnerabilities and their possible exploits i.e., NVD, CWE, CAPEC and ATT&CK. Starting from this huge amount of information managed independently by the repositories owners, the aim of this chapter has been twofold: (i) presenting and discussing such repositories taking a different perspective i.e., by considering them as one unique knowledge graph that can be traversed and analysed to answer complex queries and (ii) verifying the suitability of the knowledge graph as a reference knowledge base in the most common vulnerability analysis tasks. It has been found that the high degree of connection between concepts in different repositories supports well the two considered tasks. In particular, a mechanism leveraging the knowledge graph has been proposed to automatically correlate information across multiple repositories to speed up the process and support analysts in the vulnerability analysis process. More in detail, when identifying which is the vulnerable part of the system targeted by a specific vulnerability, it has been observed that performing the classification automatically may provide the following benefit to security operators as well as self protecting systems:

- It is immediately possible to relate the issue to a specific part of the system and evaluate the dependencies between the vulnerable part and the enabling context.
- Assuming that patches are available for any vulnerability, it is possible to assess the difficulty of mitigating the vulnerability also as a function of its class. Usually, patching vulnerable hardware is harder than patching an application. Thus, the classification can help in prioritizing mitigation activities.
- The classification between vulnerable and non-vulnerable configurations can be used as input for the enrichment of the CVSS score and in particular for the definition and computation of the environmental score.

Concerning the second analysis task, a strategy to identify common mitigation actions (in terms of strategies) for a set of vulnerabilities has been proposed. This information will provide the analyst with fast feedback on the possible options and will contribute to enhancing the overall level of awareness.

Let us note that this benefit derives directly from the deep knowledge of the repositories and from the formalization of the relationships that has been done in the knowledge graph. Indeed, it has been necessary to go deeper in understanding the details of the relationships that characterize data stored. As an example, CPE analysis revealed that some attributes associated with nodes could be misleading (e.g., looking just at the part attribute of the first CPE string linked to a vulnerability may lead to a wrong classification) and it highlighted the complexity of finding the real vulnerable part.

Concerning the third analysis task, a methodology to estimate the cost of applying a mitigation action on a vulnerability has been proposed. This information will provide the analyst with a base estimation needed as input to the cost-benefit estimation that is at the heart of the risk management process.

The results of the chapter represent a first step toward the design and implementation of a fully automated context-aware vulnerabilities analysis and classification platform that can be built by using all the considered repositories as a unique reference knowledge base.

6.2 Chapter 4 - Enhancing the Quality of Automatically Generated Inventories

Chapter 4 introduced the problem of enhancing the quality of automatically generated inventories used during the cyber risk management process. In particular, vulnerability and network scanners monitor the network environment in order to produce a Vulnerability Inventory $\mathcal{VI}$ which collects and lists all the vulnerabilities residing in each host in the network, and a Device Inventory $\mathcal{DI}$ which lists all the platform configurations of hosts in the network. Starting from these two inventories, the aim of this chapter has been twofold: (i) the introduction and formalization of the problem of the quality of the data collected in the inventories by vulnerability and network scanners, and (ii) the proposal of a computational pipeline and architecture which is able to provide solutions that aim to mitigate the aforementioned problem.

It has been found that despite the problem of the quality of the data collected in the inventories by vulnerability and network scanners exists, it is possible to: (i) leverage the knowledge of a human agent in order to increase the accuracy of such inventories by filtering out false positives, and (ii) trade the accuracy of the analysis in favour of an increased scalability of the process, if so desired.

In particular, probe-based network and vulnerability scanners, although unintrusive, may falsely detect platform configurations linked to active vulnerabilities, therefore potentially introducing false positives in the inventory. The chapter proposed a filtering process capable of reducing the number of false positives in the inventories that involves submitting queries to a human agent in order to validate or invalidate the presence of platforms on a host. Platforms on a host which have been validated or invalidated can then be used in conjunction with the applicability conditions of each vulnerability in order to validate or invalidate each vulnerability. In addition to this process, this chapter also proposed an aggregation methodology able to perform a trade-off between the accuracy of the risk analysis and its scalability in Attack Graph-based methodologies. This methodology is able to aggregate vulnerabilities on each host, using analysis-aware strategies which are configurable by the user in order to obtain the desired level of trade-off between accuracy and scalability of the Attack Graph-based risk analysis.

The two proposed processes have been integrated into a modular architecture to be implemented during the Data Collection Aggregation and Integration process of a state-of-the-art autonomic system built upon the MAPE-K architecture. This new modular architecture is able to support four computational flows:

- **CF0**: Performing no pre-processing on the vulnerability inventory $\mathcal{VI}_{raw}$

- **CF1**: Filtering vulnerabilities from the vulnerability inventory $\mathcal{VI}_{raw}$ in order to produce a new vulnerability inventory $\mathcal{VI}^{+}$
- **CF2**: Aggregating vulnerabilities from the vulnerability inventory $\mathcal{VI}_{raw}$ into meta-vulnerabilities in order to produce a new vulnerability inventory $\mathcal{VI}_{ag}$
- **CF3**: Filtering vulnerabilities from the vulnerability inventory $\mathcal{VI}_{raw}$, then aggregating the resulting filtered vulnerabilities into meta-vulnerabilities in order to produce a new vulnerability inventory $\mathcal{VI}_{ag}^{+}$

It is important to note that the proposed processes and architecture derive from an in-depth study of current state of the art vulnerability and network scanning techniques, in conjunction with some of the repositories described in Chapter 3.

The results of the chapter represent a first step toward the design and implementation of a solution capable of performing fully automated cyber risk assessment by refining and resolving problems that affect existing paradigms and methodologies adopted by state of the art self-protecting systems.

6.3 Chapter 5 - Analyzing the Accuracy-Scalability Trade-off in Attack Graph-Based Self-Protecting Systems - A Case Study

Chapter 5 performed a thorough evaluation on a real case study of the computational pipeline introduced in Chapter 4, which aims to enhance the quality of information derived from the monitored environment as well as offer the possibility to trade accuracy in favour of the scalability of the attack graph-based risk analysis in self-protecting system. This pipeline integrates methodologies and algorithms aimed at attacking the problem of enhancing the quality of automatically generated inventories used during the cyber risk management process through a *filtering* sub-component, as well several aggregation strategies implementing an accuracy-scalability trade-off in state-of-the-art attack graph-based analysis methodologies through an *aggregation* sub-component.

The creation of this computational pipeline is motivated by the fact that while risk analysis and more broadly cyber risk management processes strive towards the best possible accuracy, this contrasts with the performance of attack graph-based methodologies. Indeed state-of-the-art attack graph-based methodologies are very powerful and potentially accurate models, but suffer from scalability issues forcing to opt for the computation of *approximated* attack graphs [68, 70, 43]. This introduces a certain degree of uncertainty in the risk estimation, where accuracy is traded off for scalability. The situation gets even worse if we consider that state-of-the-art attack graph-based self-protection systems feed the control loop with input data that are not completely accurate due to false positives affecting the monitoring probes. This results in a cumulative effect of accuracy loss, affecting first the data collection phase and then the attack graph and risk computation phase.

Thus, the main focus of this chapter has been the evaluation of the capability of algorithms and methodologies proposed in Chapter 4 with respect to: (i) attacking the problem of improving the quality (in terms of accuracy) of the automatically generated inventories without increasing the degree of intrusiveness in the monitored system, and (ii) achieving a trade-off between accuracy and scalability in attack graph-based analysis methodologies, through the use of semantic-aware aggregation.

To achieve a thorough evaluation, a methodology to produce several statistically relevant testing environments has been proposed and leveraged, in order to achieve a comparable, statistically relevant and congruous result across all the proposed methodologies and algorithms. Having a real testing environment has also been necessary not only in order to evaluate the single components, but also to carry out the evaluation of the whole proposed computational pipeline.

Lastly, the proposed algorithms and methodologies have been integrated into a modular architecture (computational pipeline) to be implemented during the Data Collection Aggregation and Integration process of a self protecting system. This computational pipeline has then been evaluated using a real scenario, derived from the network environment of the department of Computer, Control and Management Engineering at Sapienza university.

The result of the evaluation has shown that applying such computational pipeline yields good results both in terms of accuracy, with 74 false positive vulnerabilities being eliminated from the 8-host case study described in Section 5.1 which in turn amounts to a 11.76% space reduction of the resulting attack graph (from 5032 edges down to 4440), as well as when considering scalability of the attack graph-based risk analysis, with aggregation strategies capable of reducing the resulting attack graph size (i.e. number of edges) from 68% up to 99% with respect to a “good” state of the art (i.e. no false positives present) at the cost of at most 0.44 MAE of the resulting risk estimation.

The results of the chapter represent a confirmation of the work done in Chapter 4, as well as a concretization through an actual case study of a first step toward the design and implementation of a solution capable of performing fully automated cyber risk assessment by refining and resolving problems that affect existing paradigms and methodologies adopted by state of the art self-protecting systems.

Bibliography

[1] AS/NZS 4360:2004 :: Standards New Zealand. URL: https://www.standards.govt.nz/shop/asnzs-43602004/.

[2] The Comprehensive National Cybersecurity Initiative. URL: https://obamawhitehouse.archives.gov/node/233086.

[3] CVSS v2 Complete Documentation. URL: https://www.first.org/cvss/v2/guide.

[4] CVSS v3.0 Specification Document. URL: https://www.first.org/cvss/v3.0/specification-document.

[5] CVSS v3.1 Specification Document. URL: https://www.first.org/cvss/v3.1/specification-document.

[6] FIRST - Improving Security Together. URL: https://www.first.org/.

[7] ISO - ISO 31000 — Risk management, December 2021. URL: https://www.iso.org/iso-31000-risk-management.html.

[8] PANOPTESEC, October 2022. URL: https://web.archive.org/web/20221006232252/https://www.panoptesec.eu/.

[9] 14:00-17:00. ISO/IEC Guide 73:2002. URL: https://www.iso.org/standard/34998.html.

[10] Habtamu Abie and Ilangko Balasingham. Risk-Based Adaptive Security for Smart IoT in eHealth. In *Proceedings of the 7th International Conference on Body Area Networks*, Oslo, Norway, 2012. ACM. URL: http://eudl.eu/doi/10.4108/icst.bodynets.2012.250235, doi:10.4108/icst.bodynets.2012.250235.

[11] E. Aghaei, W. Shadid, and E. Al-Shaer. Threatzoom: Cve2cwe using hierarchical neural network, 2020. arXiv:2009.11501.

[12] M. U. Aksu, K. Bicakci, M. H. Dilek, A. M. Ozbayoglu, and E. I. Tatli. Automated generation of attack graphs using nvd. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, CODASPY '18, page 135–142, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3176258.3176339.

[13] Suliman Alazmi and Daniel Conte De Leon. A systematic literature review on the characteristics and effectiveness of web application vulnerability scanners. *IEEE Access*, 10:33200–33219, 2022. doi:10.1109/ACCESS.2022.3161522.

[14] Ross Anderson. *Security engineering: a guide to building dependable distributed systems*. John Wiley & Sons, 2020.

[15] Marco Angelini, Graziano Blasilli, Tiziana Catarci, Simone Lenti, and Giuseppe Santucci. Vulnus: Visual Vulnerability Analysis for Network Security. *IEEE Transactions on Visualization and Computer Graphics*, 25(1):183–192, January 2019. doi:10.1109/TVCG.2018.2865028.

[16] Marco Angelini, Silvia Bonomi, Simone Lenti, Giuseppe Santucci, and Stefano Taggi. MAD: A Visual Analytics Solution for Multi-step Cyber Attacks Detection. *Journal of Computer Languages*, 52:10–24, June 2019. doi:10.1016/j.cola.2018.12.007.

[17] ANSSI. EBIOS Risk Manager. https://www.ssi.gouv.fr/en/guide/ebios-risk-manager-the-method/.

[18] Paolo Arcaini, Elvinia Riccobene, and Patrizia Scandurra. Modeling and analyzing mape-k feedback loops for self-adaptation. In *2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 13–23, 2015. doi:10.1109/SEAMS.2015.10.

[19] Michael Lyle Artz. *NetSPA : a Network Security Planning Architecture*. Thesis, Massachusetts Institute of Technology, 2002. Accepted: 2006-03-24T18:01:15Z Journal Abbreviation: Network Security Planning Architecture. URL: https://dspace.mit.edu/handle/1721.1/29899.

[20] Karl Johan Åström and Richard M Murray. *Feedback systems: an introduction for scientists and engineers*. Princeton university press, 2021.

[21] Ahmad W. Atamli and Andrew Martin. Threat-Based Security Analysis for the Internet of Things. In *2014 International Workshop on Secure Internet of Things*, pages 35–43, Wroclaw, Poland, September 2014. IEEE. URL: http://ieeexplore.ieee.org/document/7058906/, doi:10.1109/SIoT.2014.10.

[22] Sachin Babar, Parikshit Mahalle, Antonietta Stango, Neeli Prasad, and Ramjee Prasad. Proposed Security Model and Threat Taxonomy for the Internet of Things (IoT). In Natarajan Meghanathan, Selma Boumerdassi, Nabendu Chaki, and Dhinaharan Nagamalai, editors, *Recent Trends in Network Security and Applications*, volume 89, pages 420–429. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. URL: http://link.springer.com/10.1007/978-3-642-14478-3_42, doi:10.1007/978-3-642-14478-3$_4$2.

[23] S. Barnum and A. Sethi. Attack Patterns as a Knowledge Resource for Building Secure Software. *undefined*, 2007.

[24] Matthew Barrett. Framework for improving critical infrastructure cybersecurity version 1.1, 2018-04-16 2018. doi:10.6028/NIST.CSWP.04162018.

[25] Silvia Bonomi, Marco Cuoci, and Simone Lenti. A Semi-automatic Approach for Enhancing the Quality of Automatically Generated Inventories. In *2023 IEEE International Conference on Cyber Security and Resilience (CSR)*, pages 307–314, July 2023. URL: https://ieeexplore.ieee.org/document/10225003, doi:10.1109/CSR57506.2023.10225003.

[26] Brant Cheikes, David Waltermire, and Karen Scarfone. Common Platform Enumeration: Naming Specification Version 2.3 https://csrc.nist.gov/publications/detail/nistir/7695/final. Technical Report NIST Internal or Interagency Report (NISTIR) 7695, National Institute of Standards and Technology, August 2011. URL: https://csrc.nist.gov/publications/detail/nistir/7695/final, doi:10.6028/NIST.IR.7695.

[27] Qian Chen and Sherif Abdelwahed. Towards realizing self-protecting scada systems. In *Proceedings of the 9th Annual Cyber and Information Security Research Conference*, CISR '14, page 105–108, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2602087.2602113.

[28] CLUSIF. MEHARI (MEthod for Harmonized Analysis of RIsk). http://meharipedia.x10host.com/wp/.

[29] MITRE Corporation. CAPEC - Common Attack Pattern Enumeration and Classification. URL: https://capec.mitre.org/.

[30] MITRE Corporation. Cve - common vulnerability and exposure. URL: https://cve.mitre.org/index.html.

[31] MITRE Corporation. CWE - Common Weakness Enumeration. URL: https://cwe.mitre.org/index.html.

[32] Michael J. Covington and Rush Carskadden. Threat implications of the Internet of Things. In *2013 5th International Conference on Cyber Conflict (CYCON 2013)*, pages 1–12, June 2013. ISSN: 2325-5366.

[33] CVSS Special Interest Group (SIG). CVSS: Common Vulnerability Scoring System. URL: https://www.first.org/cvss/.

[34] Lisa Ehrlinger and Wolfram Wöß. Towards a Definition of Knowledge Graphs.

[35] Colin English, Sotirios Terzis, and Paddy Nixon. Towards self-protecting ubiquitous systems: monitoring trust-based interactions. *Personal and Ubiquitous Computing*, 10:50–54, 2006.

[36] N. S. Evans, A. Benameur, and M. C. Elder. Large-scale evaluation of a vulnerability analysis framework. In Chris Kanich and Patrick Lardieri, editors, *7th Workshop on Cyber Security Experimentation and Test, CSET '14, San Diego, CA, USA, August 18, 2014*. USENIX Association, 2014. URL: https://www.usenix.org/conference/cset14/workshop-program/presentation/benameur.

[37] K. Fu and J. Blum. Inside risks controlling for cybersecurity risks of medical device software. *Communications of the ACM*, 56(10), 2013. doi:DOI:10.1145/2508701.

[38] G. Gonzalez-Granadillo, S. Dubus, A. Motzek, J. Garcia-Alfaro, E. Alvarez, M. Merialdo, S. Papillon, and H. Debar. Dynamic risk management response system to handle cyber threats. *Future Generation Computer Systems*, 83:535–552, 2018. URL: https://www.sciencedirect.com/science/article/pii/S0167739X17311433, doi:10.1016/j.future.2017.05.043.

[39] Gustavo Granadillo, Ender Alvarez, Alexander Motzek, Matteo Merialdo, Joaquin Garcia-Alfaro, and Hervé Debar. Towards an Automated Dynamic Risk Management Response System (DRMRS). November 2016. doi:10.1007/978-3-319-47560-8_3.

[40] G. Grieco, G. L. Grinblat, L. Uzal, S. Rawat, J. Feist, and L. Mounier. Toward large-scale vulnerability discovery using machine learning. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, CODASPY '16, page 85–96, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2857705.2857720.

[41] E. Hemberg, J. Kelly, M. Shlapentokh-Rothman, B. Reinstadler, K. Xu, N. Rutar, and U. O'Reilly. Linking threat tactics, techniques, and patterns with defensive weaknesses, vulnerabilities and affected platform configurations for cyber hunting. arXiv:2010.00533.

[42] Elike Hodo, Xavier Bellekens, Andrew Hamilton, Pierre-Louis Dubouilh, Ephraim Iorkyase, Christos Tachtatzis, and Robert Atkinson. Threat analysis of IoT networks using artificial neural network intrusion detection system. In *2016 International Symposium on Networks, Computers and Communications (ISNCC)*, pages 1–6, Yasmine Hammamet, Tunisia, May 2016. IEEE. URL: `http://ieeexplore.ieee.org/document/7746067/`, doi:10.1109/ISNCC.2016.7746067.

[43] Hao Hu, Jing Liu, Yuchen Zhang, Yuling Liu, Xiaoyu Xu, and Jinglei Tan. Attack scenario reconstruction approach using attack graph and alert data mining. *Journal of Information Security and Applications*, 54:102522, 2020.

[44] Nasif Imtiaz, Seaver Thorn, and Laurie Williams. A comparative study of vulnerability reporting by software composition analysis tools. In *Proceedings of the 15th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, ESEM '21, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3475716.3475769.

[45] Kyle Ingols, Richard Lippmann, and Keith Piwowarski. Practical Attack Graph Generation for Network Defense. In *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, pages 121–130, December 2006. ISSN: 1063-9527. doi:10.1109/ACSAC.2006.39.

[46] Jeff Williams. OWASP Risk Rating Methodology. `https://owasp.org/www-community/OWASP_Risk_Rating_Methodology`.

[47] K. Kanakogi, H. Washizaki, Y. Fukazawa, S. Ogata, T. Okubo, T. Kato, H. Kanuka, A. Hazeyama, and N. Yoshioka. Tracing capec attack patterns from cve vulnerability information using natural language processing technique. *Proc. of 54th Hawaii International Conference on System Sciences (HICSS)*, 1 2021.

[48] Georgios Kavallieratos and Sokratis Katsikas. Attack Path Analysis for Cyber Physical Systems. In Sokratis Katsikas, Frédéric Cuppens, Nora Cuppens, Costas Lambrinoudakis, Christos Kalloniatis, John Mylopoulos, Annie Antón, Stefanos Gritzalis, Weizhi Meng, and Steven Furnell, editors, *Computer Security*, Lecture Notes in Computer Science, pages 19–33, Cham, 2020. Springer International Publishing. doi:10.1007/978-3-030-64330-0_2.

[49] Kerem Kaynar. A taxonomy for attack graph generation and usage in network security. *Journal of Information Security and Applications*, 29:27–56, August 2016. URL: https://www.sciencedirect.com/science/article/pii/S2214212616300011, doi:10.1016/j.jisa.2016.02.001.

[50] Kerem Kaynar and Fikret Sivrikaya. Distributed Attack Graph Generation. *IEEE Transactions on Dependable and Secure Computing*, 13(5):519–532, September 2016. Conference Name: IEEE Transactions on Dependable and Secure Computing. doi:10.1109/TDSC.2015.2423682.

[51] J.O. Kephart and D.M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003. doi:10.1109/MC.2003.1160055.

[52] Elmar Kiesling, Andreas Ekelhart, Kabul Kurniawan, and Fajar Ekaputra. The SEPSES Knowledge Graph: An Integrated Resource for Cybersecurity. In Chiara Ghidini, Olaf Hartig, Maria Maleshkova, Vojtěch Svátek, Isabel Cruz, Aidan Hogan, Jie Song, Maxime Lefrançois, and Fabien Gandon, editors, *The Semantic Web – ISWC 2019*, Lecture Notes in Computer Science, pages 198–214, Cham, 2019. Springer International Publishing. doi:10.1007/978-3-030-30796-$7_1$3.

[53] Yusuf Kocaman, Serkan Gönen, Mehmet Ali Barişkan, Gökçe Karacayilmaz, and Ercan Nurcan Yilmaz. A novel approach to continuous CVE analysis on enterprise operating systems for system vulnerability assessment. *International Journal of Information Technology*, 14(3):1433–1443, May 2022. doi:10.1007/s41870-021-00840-6.

[54] Paul Kocher, Joshua Jaffe, Benjamin Jun, Carter Laren, and Nate Lawson. Self-protecting digital content. *CRI Content Security Research Initiative, Tech. Rep*, 2003.

[55] I. V. Krsul. *Software vulnerability analysis*. Purdue University West Lafayette, IN, 1998.

[56] Dat Tien Le. Quality trade-offs in self-protecting system. In *2015 IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops*, pages 152–156. IEEE, 2015.

[57] Ming Li, Peter Hawrylak, and John Hale. Concurrency Strategies for Attack Graph Generation. In *2019 2nd International Conference on Data Intelligence and Security (ICDIS)*, pages 174–179, June 2019. doi:10.1109/ICDIS.2019.00033.

[58] Ming Li, Peter J. Hawrylak, and John Hale. Implementing an Attack Graph Generator in CUDA. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 730–738, May 2020. doi:10.1109/IPDPSW50202.2020.00128.

[59] Qiang Li, Jinke Song, Dawei Tan, Haining Wang, and Jiqiang Liu. Pdgraph: A large-scale empirical study on project dependency of security vulnerabilities. In *51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2021, Taipei, Taiwan, June 21-24, 2021*, pages 161–173. IEEE, 2021. doi:10.1109/DSN48987.2021.00031.

[60] Teng Li, Ya Jiang, Chi Lin, Mohammad Obaidat, Yulong Shen, and Jianfeng Ma. DeepAG: Attack Graph Construction and Threats Prediction with Bi-directional Deep Learning. *IEEE Transactions on Dependable and Secure Computing*, pages 1–1, 2022. Conference Name: IEEE Transactions on Dependable and Secure Computing. doi:10.1109/TDSC.2022.3143551.

[61] Gaoqi Liang, Steven R. Weller, Fengji Luo, Junhua Zhao, and Zhao Yang Dong. Distributed blockchain-based data protection framework for modern power systems against cyber attacks. *IEEE Transactions on Smart Grid*, 10(3):3162–3173, 2019. doi:10.1109/TSG.2018.2819663.

[62] Sabina Magalini, Daniele Gui, Pasquale Mari, Matteo Merialdo, Emanoulis Spanakis, Vangelis Sakkalis, Fabio Rizzoni, Alessandra Casaroli, and Silvia Bonomi. Cyberthreats to hospitals: Panacea, a toolkit for people-centric cybersecurity. *Journal of Strategic Innovation and Sustainability*, 16(3), Aug. 2021. URL: `https://articlegateway.com/index.php/JSIS/article/view/4449`, doi:10.33423/jsis.v16i3.4449.

[63] Lorenzo Fernández Maimó, Ángel Luis Perales Gómez, Félix J García Clemente, Manuel Gil Pérez, and Gregorio Martínez Pérez. A self-adaptive deep learning-based system for anomaly detection in 5g networks. *Ieee Access*, 6:7700–7712, 2018.

[64] Yuma Makino and Vitaly Klyuev. Evaluation of web vulnerability scanners. In *2015 IEEE 8th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*, volume 1, pages 399–402, 2015. doi:10.1109/IDAACS.2015.7340766.

[65] André Martin, Tiaraju Smaneoto, Tobias Dietze, Andrey Brito, and Christof Fetzer. User-constraint and self-adaptive fault tolerance for event stream processing systems. In *45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2015, Rio de Janeiro, Brazil, June 22-25, 2015*, pages 462–473. IEEE Computer Society, 2015. doi:10.1109/DSN.2015.56.

[66] P. Mell, K. Scarfone, and S. Romanosky. Common vulnerability scoring system. *IEEE Security & Privacy*, 4(6):85–89, 2006.

[67] Daniel Minoli, Kazem Sohraby, and Jacob Kouns. IoT security (IoTSec) considerations, requirements, and architectures. In *2017 14th IEEE Annual Consumer Communications & Networking Conference (CCNC)*, pages 1006–1007, Las Vegas, NV, USA, January 2017. IEEE. URL: `http://ieeexplore.ieee.org/document/7983271/`, doi:10.1109/CCNC.2017.7983271.

[68] Oussama Mjihil, Dijiang Huang, and Abdelkrim Haqiq. Improving Attack Graph Scalability for the Cloud Through SDN-Based Decomposition and Parallel Processing. In Essaid Sabir, Ana García Armada, Mounir Ghogho, and Mérouane Debbah, editors, *Ubiquitous Networking*, Lecture Notes in Computer Science, pages 193–205, Cham, 2017. Springer International Publishing. doi:10.1007/978-3-319-68179-$5_1$7.

[69] Martin Monperrus. Automatic software repair: A bibliography. *ACM Computing Surveys (CSUR)*, 51(1):1–24, 2018.

[70] Stephen Moskal, Shanchieh Jay Yang, and Michael E Kuhl. Extracting and evaluating similar and unique cyber attack strategies from intrusion alerts. In *2018 IEEE international conference on intelligence and security informatics (ISI)*, pages 49–54. IEEE, 2018.

[71] Azqa Nadeem, Sicco Verwer, Stephen Moskal, and Shanchieh Jay Yang. Alert-Driven Attack Graph Generation Using S-PDFA. *IEEE Transactions on Dependable and Secure Computing*, 19(2):731–746, March 2022. Conference Name: IEEE Transactions on Dependable and Secure Computing. doi:10.1109/TDSC.2021.3117348.

[72] National Institute of Standards and Technology. Framework for Improving Critical Infrastructure Cybersecurity, Version 1.1. Technical Report NIST CSWP 04162018, National Institute of Standards and Technology, Gaithersburg, MD, April 2018. URL: http://nvlpubs.nist.gov/nistpubs/CSWP/NIST.CSWP.04162018.pdf, doi:10.6028/NIST.CSWP.04162018.

[73] Peng Ning, Dingbang Xu, Christopher G Healey, and Robert St Amant. Building attack scenarios through integration of complementary alert correlation method. In *NDSS*, volume 4, pages 97–111, 2004.

[74] NIS Directive. Directive (eu) 2016/1148 of the european parliament and of the council of 6 july 2016 concerning measures for a high common level of security of network and information systems across the union. *OJ L*, 194(19.7):2016, 2016.

[75] NIST National Institute of Standards and Technology. National vulnerability database (nvd), 2021. URL: https://nvd.nist.gov/.

[76] Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. Backstabber's knife collection: A review of open source software supply chain attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 23–43. Springer, 2020.

[77] Alessandro Palma and Silvia Bonomi. A workflow for distributed and resilient attack graph generation. In *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks-Supplemental Volume (DSN-S)*, pages 185–187. IEEE, 2023.

[78] M. Parmelee, H. Booth, D. Waltermire, and K. Scarfone. Common Platform Enumeration: Name Matching Specification Version 2.3. Technical Report NIST Internal or Interagency Report (NISTIR) 7696, National Institute of Standards and Technology, August 2011. URL: https://csrc.nist.gov/publications/detail/nistir/7696/final, doi:10.6028/NIST.IR.7696.

[79] Irdin Pekaric, Raffaela Groner, Thomas Witte, Jubril Gbolahan Adigun, Alexander Raschke, Michael Felderer, and Matthias Tichy. A systematic review on security and safety of self-adaptive systems. *Journal of Systems and Software*, 203:111716, 2023. URL: https://www.sciencedirect.com/science/article/pii/S0164121223001115, doi:10.1016/j.jss.2023.111716.

[80] Marcus Pendleton, Richard Garcia-Lebron, Jin-Hee Cho, and Shouhuai Xu. A survey on systems security metrics. *ACM Comput. Surv.*, 49(4), dec 2016. doi:10.1145/3005714.

[81] Vung Pham and Tommy Dang. CVExplorer: Multidimensional Visualization for Common Vulnerabilities and Exposures. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 1296–1301, December 2018. doi:10.1109/BigData.2018.8622092.

[82] Francisco Pozo, Guillermo Rodríguez-Navas, and Hans Hansson. Self-healing protocol: Repairing schedules online after link failures in time-triggered networks. In *51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2021, Taipei, Taiwan, June 21-24, 2021*, pages 129–140. IEEE, 2021. doi:10.1109/DSN48987.2021.00028.

[83] Grant Purdy. ISO 31000:2009-Setting a New Standard for Risk Management. *Risk analysis : an official publication of the Society for Risk Analysis*, 30:881–6, June 2010. doi:10.1111/j.1539-6924.2010.01442.x.

[84] P. Radanliev, D. De Roure, S. Cannady, R.M. Montalvo, R. Nicolescu, and M. Huth. Economic impact of IoT cyber risk - analysing past and present to predict the future developments in IoT risk analysis and IoT cyber insurance. In *Living in the Internet of Things: Cybersecurity of the IoT - 2018*, pages 3 (9 pp.)–3 (9 pp.), London, UK, 2018. Institution of Engineering and Technology. URL: `https://digital-library.theiet.org/content/conferences/10.1049/cp.2018.0003`, doi:10.1049/cp.2018.0003.

[85] Steven Lamarr Reynolds, Tobias Mertz, Steven Arzt, and Jörn Kohlhammer. User-Centered Design of Visualizations for Software Vulnerability Reports. In *2021 IEEE Symposium on Visualization for Cyber Security (VizSec)*, pages 68–78, October 2021. doi:10.1109/VizSec53666.2021.00013.

[86] Ernesto Rosario Russo, Andrea Di Sorbo, Corrado A. Visaggio, and Gerardo Canfora. Summarizing vulnerabilities' descriptions to support experts during vulnerability assessment activities. *Journal of Systems and Software*, 156:84–99, October 2019. URL: `https://www.sciencedirect.com/science/article/pii/S016412121930130X`, doi:10.1016/j.jss.2019.06.001.

[87] Abdulhakim Sabur, Ankur Chowdhary, Dijiang Huang, and Adel Alshamrani. Toward scalable graph-based security analysis for cloud networks. *Computer Networks*, 206:108795, April 2022. URL: `https://www.sciencedirect.com/science/article/pii/S1389128622000251`, doi:10.1016/j.comnet.2022.108795.

[88] K A Scarfone and P M Mell. Guide to Intrusion Detection and Prevention Systems (IDPS). Technical Report NIST SP 800-94, National Institute of Standards and Technology, Gaithersburg, MD, 2007. Edition: 0. URL: `https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-94.pdf`, doi:10.6028/NIST.SP.800-94.

[89] K A Scarfone, M P Souppaya, A Cody, and A D Orebaugh. Technical guide to information security testing and assessment. Technical Report NIST SP 800-115, National Institute of Standards and Technology, Gaithersburg, MD, 2008. Edition: 0. URL: `https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-115.pdf`, doi:10.6028/NIST.SP.800-115.

[90] L. Shen. The nist cybersecurity framework: Overview and potential impacts. *Scitech Lawyer*, 10(4):16, 2014.

[91] Adam Shostack. *Threat modeling: Designing for security.* John Wiley & Sons, 2014.

[92] I.M. Skierka. The governance of safety and security risks in connected healthcare. In *Living in the Internet of Things: Cybersecurity of the IoT - 2018*, pages 2 (12 pp.)–2 (12 pp.), London, UK, 2018. Institution of Engineering and Technology. URL: `https://digital-library.theiet.org/content/conferences/10.1049/cp.2018.0002`, doi:10.1049/cp.2018.0002.

[93] Emmanouil G. Spanakis, Silvia Bonomi, Stelios Sfakianakis, Giuseppe Santucci, Simone Lenti, Mara Sorella, Florin Dragos Tanasache, Alessia Palleschi, Claudio Ciccotelli, Vangelis Sakkalis, and Sabina Magalini. Cyber-attacks and threats for healthcare - a multi-layer thread analysis. In *42nd Annual International Conference of the IEEE Engineering in Medicine & Biology Society, EMBC 2020, Montreal, QC, Canada, July 20-24, 2020*, pages 5705–5708. IEEE, 2020. doi:10.1109/EMBC44109.2020.9176698.

[94] B. E. Strom, A. Applebaum, D. P. Miller, K. C. Nickels, A. G. Pennington, and C. B. Thomas. *MITRE ATT&CK: Design and Philosophy. White Paper*, 2020.

[95] John D Strunk, Garth R Goodson, Michael L Scheinholtz, Craig AN Soules, and Gregory R Ganger. Self-securing storage: Protecting data in compromised systems. In *OSDI*, pages 165–180, 2000.

[96] Wei Sun, Qianmu Li, Pengchuan Wang, and Jun Hou. Heuristic Network Security Risk Assessment Based on Attack Graph. In Mohammad R. Khosravi, Qiang He, and Haipeng Dai, editors, *Cloud Computing*, Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, pages 181–194, Cham, 2022. Springer International Publishing. doi:10.1007/978-3-030-99191-3$_1$4.

[97] Romilla Syed. Cybersecurity vulnerability management: A conceptual ontology and cyber intelligence alert system. *Information & Management*, 57(6):103334, September 2020. doi:10.1016/j.im.2020.103334.

[98] Daniel Tovarňák, Lukáš Sadlek, and Pavel Čeleda. Graph-Based CPE Matching for Identification of Vulnerable Asset Configurations. In *2021 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, pages 986–991, May 2021. ISSN: 1573-0077.

[99] Roman Ushakov, Elena Doynikova, Evgenia Novikova, and Igor Kotenko. CPE and CVE based Technique for Software Security Risk Assessment. In *2021 11th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*, volume 1, pages 353–356, September 2021. ISSN: 2770-4254. doi:10.1109/IDAACS53288.2021.9660968.

[100] David Waltermire, Paul Cichonski, and Karen Scarfone. Common Platform Enumeration: Applicability Language Specification Version 2.3. Technical Report NIST Internal or Interagency Report (NISTIR) 7698, National Institute of Standards and Technology, August 2011. URL: `https://csrc.nist.gov/pubs/ir/7698/final`, doi:10.6028/NIST.IR.7698.

[101] J. Webb and D. Hume. Campus IoT Collaboration and Governance using the NIST Cybersecurity Framework. In *Living in the Internet of Things: Cybersecurity of the IoT - 2018*, pages 25 (7 pp.)–25 (7 pp.), London, UK, 2018. Institution of Engineering and Technology. URL: https://digital-library.theiet.org/content/conferences/10.1049/cp.2018.0025, doi:10.1049/cp.2018.0025.

[102] Danny Weyns, Sam Malek, and Jesper Andersson. Forms: Unifying reference model for formal specification of distributed self-adaptive systems. *ACM Trans. Auton. Adapt. Syst.*, 7(1), may 2012. doi:10.1145/2168260.2168268.

[103] Emil Wåreus and Martin Hell. Automated CPE Labeling of CVE Summaries with Machine Learning. In Clémentine Maurice, Leyla Bilge, Gianluca Stringhini, and Nuno Neves, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, Lecture Notes in Computer Science, pages 3–22, Cham, 2020. Springer International Publishing. doi:10.1007/978-3-030-52683-2_1.

[104] Veneta Yosifova, Antoniya Tasheva, and Roumen Trifonov. Predicting Vulnerability Type in Common Vulnerabilities and Exposures (CVE) Database with Machine Learning Classifiers. In *2021 12th National Conference with International Participation (ELECTRONICA)*, pages 1–6, May 2021. doi:10.1109/ELECTRONICA52725.2021.9513723.

[105] Eric Yuan, Naeem Esfahani, and Sam Malek. A systematic survey of self-protecting software systems. *ACM Trans. Auton. Adapt. Syst.*, 8(4), jan 2014. doi:10.1145/2555611.

[106] Eric Yuan, Sam Malek, Bradley Schmerl, David Garlan, and Jeff Gennari. Architecture-based self-protecting software systems. In *Proceedings of the 9th International ACM Sigsoft Conference on Quality of Software Architectures*, QoSA '13, page 33–42, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2465478.2465479.

[107] Kengo Zenitani. Attack graph analysis: an explanatory guide. *Computers & Security*, page 103081, 2022.

[108] Fan Zhang, Pengfei Ju, Mengqiu Pan, Dawei Zhang, Yao Huang, Guoliang Li, and Xiaogang Li. Self-healing mechanisms in smart protective coatings: A review. *Corrosion Science*, 144:74–88, 2018.

[109] S. Zhang, X. Ou, and D. Caragea. Predicting cyber risks through national vulnerability database. *Information Security Journal: A Global Perspective*, 24(4-6):194–206, 2015.

[110] Zhi-Kai Zhang, Michael Cheng Yi Cho, Chia-Wei Wang, Chia-Wei Hsu, Chong-Kuan Chen, and Shiuhpyng Shieh. IoT Security: Ongoing Challenges and Research Opportunities. In *2014 IEEE 7th International Conference on Service-Oriented Computing and Applications*, pages 230–234, Matsue, Japan, November 2014. IEEE. URL: http://ieeexplore.ieee.org/document/6978614/, doi:10.1109/SOCA.2014.58.

[111] Zilong Zhao, Sophie Cerf, Robert Birke, Bogdan Robu, Sara Bouchenak, Sonia Ben Mokhtar, and Lydia Y. Chen. Robust anomaly detection on unreliable data. In *49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2019, Portland, OR, USA, June 24-27, 2019*, pages 630–637. IEEE, 2019. doi:10.1109/DSN.2019.00068.