

TDK-dolgozat

Dániel Zoltán Szilvágyi
24th of December, 2024.



Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Networked Systems and Services

Supporting security-by-design via automating attack tree generation from abstract system models

Scientific Students' Association Report

Author:

Dániel Zoltán Szilvági
Year III

Advisor:

Dr. Levente Buttyán,
Professor

2024.

Contents

Kivonat	i
Abstract	ii
1 Introduction	1
2 Background	3
2.1 The DOSS project	3
2.1.1 Using digital twins for security testing	4
2.2 BME and the DOSS project	4
2.2.1 The role of attack trees	5
3 Related Work	6
3.1 Threat modeling - attack trees and graphs	6
3.2 Syntax of an attack tree and graph	7
3.3 Opportunities in automation	7
3.3.1 Attack tree and graph generation	8
3.3.1.1 Model-driven approach	8
3.3.1.2 Analysis-driven approach	8
3.3.1.3 Vulnerability-driven approach	9
3.3.1.4 Our solution	9
4 Design	11
4.1 The basis of the program - the declarative model	11
4.1.1 Model Components	11
4.1.2 The static system model	12
4.1.3 Dynamic system model	12
4.1.4 General system rules	13
4.1.5 Attacker-specific rules	13
4.1.6 Potential test cases and testable subgoals	15

4.2	Components, interfaces of the program	17
4.2.1	Inputs, interfaces	17
4.2.1.1	The instructions - goal of the tree	17
4.2.1.2	The high-level model - Archimate	17
4.2.2	The core	18
4.2.2.1	Turning system model into a declarative model	18
4.2.2.2	The tree generation - Prolog	18
4.2.2.3	Reducing complexity - ASP	18
4.2.2.4	Reconstructing the tree	19
4.2.3	Results, output interfaces	19
4.2.3.1	Analyzing results - recommendations from the high-level attack tree	19
4.2.3.2	Exporting	20
5	Implementation	21
5.1	The declarative form of the system model	21
5.2	Translating graphical into logical model	21
5.3	Generating system facts from indirect relationships - ASP	22
5.4	Generating the tree - Prolog	22
5.5	Reconstructing the attack tree	24
5.6	Exporting the results - neo4j	24
6	Evaluation	25
6.1	The example system	25
6.2	The test runs	26
6.2.1	Making unavailable a sensing service	26
6.2.2	Tampering data flow	27
6.2.3	Modifying the data	28
6.3	Analyzing the results	29
6.4	Optimization results	30
7	Conclusion	31
	Acknowledgements	32
	Appendix	35

Kivonat

Az informatikai rendszerek ellenálló képességének biztosítása a kibertámadásokkal szemben fontos kérdés és nem triviális kihívás. Az egyre inkább előtérbe kerülő security-by-design megközelítés hangsúlyozza annak fontosságát, hogy a biztonsági intézkedéseket már a tervezés kezdetén integráljuk. A biztonság sérülése jelentős pénzügyi veszteséget, hírnév-károsodást és működési zavarokat okozhatnak, ezért elengedhetetlen, hogy a potenciális sebezhetőségeket már a tervezés korai szakaszában proaktívan azonosítsák és javítsák. Ennek támogatására projektünk megoldást kínál a biztonsági elemzéshez, amely magas szintű rendszermodellekből támadási fák létrehozását automatizálja Answer Set Programming (ASP) és Prolog segítségével. Ez a megközelítés megkönnyíti egy rendszer gyenge pontjainak korai felismerését, és gyors, automatizált folyamat révén hatékonyabb és proaktívabb biztonsági tervezést tesz lehetővé.

A rendszermodelleket ArchiMate modellek formájában vizsgáljuk. Az ArchiMate egy széles körben használt modellező nyelv a vállalati architektúrához, amely átfogó képet ad egy szervezet folyamatairól, alkalmazásairól és technológiai rétegeiről. Projektünk e modellek felhasználásával áthidalja a magas szintű architektúra modell és a biztonsági elemzés közötti szakadékot, és egy módszert kínál a potenciális biztonsági sebezhetőségek azonosítására és kezelésére.

Módszerünk az ArchiMate fájlok ASP- és Prolog-kóddá történő átalakításával kezdődik. Ez az átalakítás magában foglalja az ArchiMate-modell elemzését és a releváns architektúraelemek, azok tulajdonságainak és a köztük lévő kapcsolatoknak a kinyerését, amelyeket aztán ASP-be és Prologba kódolunk. Az ASP, amely a komplex keresési problémákra szabott deklaratív programozási megközelítéséről ismert, fontos szerepet játszik a modellen belül a hosztok, folyamatok és szolgáltatások közötti további, közvetett kapcsolatok feltárásában. Eközben a logikai érvelési képességeiről ismert Prolog a modellek feletti következtetés és az előre meghatározott támadási szabályok alkalmazásával támadási fákat generál. Ezeket a szabályokat úgy alakítottuk ki, hogy reális, magas szintű támadási vektorokat és támadói taktikákat, technikákat és eljárásokat (TTP) tükrözzenek, és ezáltal alapul szolgáljanak a lehetséges rendszerhiányosságok felderítésének.

Projektünk a támadási fák automatizált generálásán alapul, amely a biztonsági elemzés egyik alapvető eszköze: lehetővé teszi a rendszerben érdekelték számára a potenciális fenyegetések feltérképezését, a kockázatelemzést és az egyes kockázatcsökkentési stratégiák rangsorolását. A generált támadási fák az architektúramodellből a támadási szabályok segítségével jönnek létre, biztosítva, hogy ezek a fák a szervezet aktuális biztonsági állapotához igazodva a lehető legrelevánsabbak legyenek. A logikai érvelés és a deklaratív problémamegoldás kihasználásával rendszerünk hatékonyan vizsgálja a lehetséges támadási utak széles skáláját, alapos képet nyújtva az adott rendszert fenyegető veszélyekről. Ez a megközelítés jelentősen csökkenti a támadási fák generálásához hagyományosan szükséges manuális munkát és speciális szakértői háttérrel, így a felhasználók szélesebb köre számára válik elérhetővé. Még ennél is fontosabb, hogy lehetővé teszi a biztonsági tesztesetek automatikus generálását és végrehajtását, valamint a feltárt sebezhetőségekkel kapcsolatos magyarázatok automatikus előállítását, amelyek alapvető fontosságúak lehetnek a rendszertervezők számára.

Abstract

Ensuring IT systems' resilience against cyber attacks is an important concern and a non-trivial challenge. The shift towards a security-by-design approach emphasizes the importance of integrating security measures from the outset, rather than as an afterthought. Security breaches can lead to significant financial loss, reputational damage, and disruption of operations, making it imperative to proactively identify and address potential vulnerabilities early in the design phase of any initiative. To support this, our project introduces a novel method for security analysis, automating the creation of attack trees from high-level system models using Answer Set Programming (ASP) and Prolog. This approach facilitates early detection of weaknesses, enabling more efficient and proactive security planning through a fast, automated process.

We assume that system models are provided in the form of ArchiMate models. ArchiMate is a widely used modeling language for enterprise architecture, which provides a comprehensive view of an organization's processes, applications, and technology layers. Our project bridges the gap between high-level architecture and detailed security analysis by leveraging these models, offering a method for identifying and addressing potential security vulnerabilities.

Our methodology begins with a meticulous process of transforming ArchiMate files into ASP and Prolog code. This transformation involves parsing the ArchiMate model and extracting relevant architectural elements, their properties, and the relationships between them, which are then encoded into ASP and Prolog. ASP, known for its declarative programming approach tailored for complex search problems, is instrumental in exploring additional, indirect relationships and connections between hosts, processes, and services within the model. Meanwhile, Prolog, renowned for its logical reasoning capabilities, generates attack trees by reasoning over the models and applying predefined attack rules. These rules are carefully crafted to reflect realistic, high-level attack vectors and adversarial tactics, techniques, and procedures (TTPs), forming the foundation for detecting potential system weaknesses.

The core contribution of our project lies in the automated generation of attack trees, one of the essential tools in security analysis, enabling stakeholders to visualize potential threats, assess risks, and prioritize mitigation strategies. The generated attack trees result from the dynamic interaction between the encoded architectural model and the attack rules, ensuring that they are contextually relevant to the organization's security posture. By leveraging logical reasoning and declarative problem-solving, our system efficiently explores a wide range of potential attack paths, offering a thorough understanding of the threats to the system at hand. This approach significantly reduces the manual effort and expertise traditionally required in attack tree generation, making it accessible to a broader range of users. More importantly, it also paves the way for automated generation and execution of security test cases and for automated provision of information on discovered vulnerabilities, which can be crucial input to system designers.

Chapter 1

Introduction

In today’s increasingly interconnected digital landscape, ensuring robust cybersecurity is more critical than ever. The rapid expansion of IT systems, particularly in sectors such as IoT (Internet of Things), has introduced a wide array of vulnerabilities, making them prime targets for cyberattacks. Organizations are increasingly shifting towards a security-by-design approach, emphasizing the integration of security measures from the earliest stages of system development. This shift seeks to address the growing complexity and interconnected nature of IT infrastructures, which can significantly magnify the impact of security breaches. Such breaches not only result in financial losses and operational disruptions but also damage reputations, making early identification and mitigation of potential vulnerabilities essential.

The DOSS (Secure-By-Design IoT Operation With Supply Chain Control) project¹, funded by the European Union, is one initiative at the forefront of addressing these challenges. By focusing on the security of IoT systems from inception to deployment, DOSS aims to establish a secure “Supply Trust Chain” that spans the entire lifecycle of IoT components. A crucial aspect of this initiative is the use of Digital Cybersecurity Twins (DCTs), which simulate real-world IoT architectures to conduct security testing and uncover potential vulnerabilities before they can be exploited in live environments.

One of the key components of effective cybersecurity analysis is the creation of attack trees, which provide a structured way to model potential threats, evaluate risks, and prioritize security measures. Attack trees break down how an adversary might compromise a system, allowing security analysts to identify and address the most critical vulnerabilities. However, generating attack trees manually can be time-consuming and requires significant expertise, creating a barrier for many organizations.

Attack trees operate based on precondition logic, where a main goal is defined and broken down into smaller, actionable subgoals recursively. The leaves of the tree represent specific actions or steps required to achieve the overarching goal at the root. For example, if the goal is to gain terminal or shell access to a computer system, the subgoals might be: reaching the system (e.g., via a network connection) and obtaining the necessary credentials to log in. To acquire the credentials, one could employ methods like spear-phishing or brute-force attacks. The subgoals of the other step in the attack tree are determining how to gain network access, such as bypassing a firewall or delivering malware through an email. Each action is represented as a subgoal, with its own set of prerequisites. For instance, to bypass a firewall, techniques like exploiting open ports or misconfigurations might be used, while sending malware through an email would involve crafting a phishing

¹<https://dossproject.eu/> Last accessed: 29th of October 2024

attack to trick a user into executing the malicious file. This structured approach ultimately forms an attack tree, mapping out the potential pathways to achieve the desired objective.

Historically, the manual generation of attack trees, a method used to systematically analyze potential threats, has been a labor-intensive and expertise-driven task. Despite their effectiveness, the manual creation of these trees poses challenges in scalability and consistency, particularly in dynamic or large-scale environments. To address this issue, there has been growing interest in automating the generation of attack trees using advanced computational techniques.

This document presents a solution for automating the generation of attack trees from high-level system models. Specifically, our method leverages Answer Set Programming (ASP)² and Prolog³ to translate ArchiMate models⁴—widely used for enterprise architecture—into actionable attack trees. This automation significantly reduces the manual effort required, allowing for more efficient and proactive security analysis. By breaking down the process into two phases, our framework generates the attack tree, and then in another module, which is not part of this document, generates executable attack steps via templates, providing a comprehensive view of potential attack paths within the system. Our approach generates attack trees in a model-driven way, which is the key to speed and effectiveness, and keeps close contact with the model ensuring the tree truly represents valid attack scenarios. Our attack trees are crucial points for generating test cases for automated penetration tests and give feedback to system designers to make IT infrastructures, and IoT systems safer.

In the following chapters, we delve into the methodology behind this approach. The organization of the content of this paper is the following: The Background chapter provides an overview of the DOSS project, which serves as the broader framework within which our project operates. It outlines the goals and significance of the DOSS initiative. Following that, the Related Work chapter examines previous research on attack trees and graph-based methods, highlighting key advancements in their automated generation, which forms the foundation for our approach. The Design chapter presents the architecture of the project, detailing the logical components and the methodology used to structure the system. This is followed by the Implementation chapter, where the practical aspects of the project are explored, providing a detailed explanation of how the system’s components were developed, along with relevant examples. The Evaluation chapter demonstrates the system in action, showcasing an example of generating an attack tree for a specific system and identifying potential vulnerabilities. Finally, the Conclusion chapter summarizes the key findings of the project, reflecting on the outcomes and contributions of the research.

²<https://potassco.org/> Last accessed: 29th October 2024

³<https://www.swi-prolog.org/> Last accessed: 29th October 2024

⁴<https://www.opengroup.org/archimate-forum/archimate-overview> Last accessed: 29th October 2024

Chapter 2

Background

The increasing reliance on information technology systems in various sectors underscores the critical need for robust cybersecurity measures. In recent years, the complexity of cyberattacks and the interconnected nature of modern IT systems have highlighted the importance of a security-by-design approach.

2.1 The DOSS project

The DOSS (Secure-By-Design IoT Operation With Supply Chain Control) project funded by the European Union plays a pivotal role in this field. The project aims to address the challenges posed by increasingly complex and interconnected systems, particularly in the IoT (Internet of Things) domain. By introducing a comprehensive security framework that integrates multiple stakeholders in the supply chain, DOSS focuses on creating a “Supply Trust Chain” (see Fig. 2.1) that ensures the security of IoT components from their inception through their operational lifespan. This is achieved through formalized data exchange, component testing, and architecture modeling, which are vital to establishing trust among stakeholders in the IoT ecosystem.

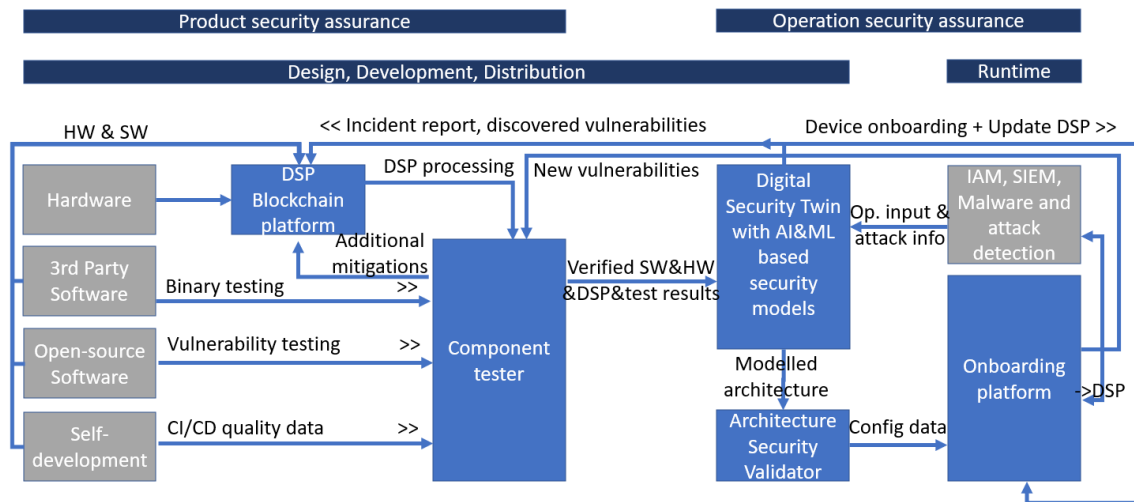


Figure 2.1: The "Supply Thrust Chain" of the DOSS project¹

¹<https://dossproject.eu/technology/#concept> Last accessed: 6th of October 2024

2.1.1 Using digital twins for security testing

A key component of the DOSS project is the Digital Cybersecurity Twin (DCT), which is a digital twin used for testing security properties and identifying vulnerabilities in IoT systems in their development phase.

Digital twins, which NASA first introduced, offer numerous benefits in the design, testing, and operation of cyber-physical systems. They enable cost-effective development by allowing various design options to be explored without the need for physical prototypes. Digital twins can execute test cases faster and more safely during testing than on real devices. In the operational phase, digital twins can replicate state transitions of the physical system, enabling close monitoring, behavior prediction, fault forecasting, and predictive maintenance. They can also be used as simulators for cost-effective and safe training of operating personnel.

In the DOSS project, the vision is that digital twins simulate real-world IoT architectures and offer the possibility of security testing and identifying potential security vulnerabilities. By employing AI-driven models and automated attack scenario simulations, the DCT provides valuable insights into system weaknesses, allowing stakeholders to test countermeasures before deploying them in live environments. This not only enhances the security posture of IoT architectures but also significantly reduces the time and costs associated with traditional security testing and certification processes.

The protection of IoT architectures within the DOSS project is implemented through a multi-layered approach. Initially, all hardware and software components undergo rigorous testing, and only those that pass are integrated into the system modeling phase. The Digital Cybersecurity Twin then simulates these IoT architectures at various abstraction levels, enabling the identification of vulnerabilities and the development of protective measures. Once the architecture design is complete, it is evaluated by the Architecture Security Validator, which ensures that it complies with relevant legal regulations and industry standards. After successfully passing these checks, the architecture is considered secure and ready for deployment, aided by automated onboarding technology. Post-deployment, the architecture is continuously monitored by various security modules—such as access management, attack detection, malware detection, and control flow integrity validators—to maintain and update its security status through the Device Security Passport (DSP), ensuring ongoing protection against emerging threats².

2.2 BME and the DOSS project

The Budapest University of Technology and Economics is a partner in the DOSS project, and it leads the work package on the design and implementation of a framework that allows for security testing of digital twins of IoT systems. We develop a framework that consists of multiple modules, each responsible for a specific subtask of handling security testing jobs. We assume that a low-level representation of the IoT system under development is provided by the system developer in the form of an infrastructure-as-code (IaC) description, together with the actual software components of the system and a high-level architecture model. The first step is then to extract relevant system information from the high-level model, from which we can identify impactful attack goals specific to the system. These goals mainly focus on breaking a component of the CIA triad (confidentiality, integrity, and availability) in an infrastructure, which is the key to modern IT security policies. After

²<https://dossproject.eu/technology/#concept> Last accessed: 6th of October 2024

generating these goals, we generate attack trees for them automatically. Attack trees encode the possible ways the attack goals can be reached. We use attack trees then to derive executable test cases. Such test cases are used to check if certain pre-conditions of the attack goals can be satisfied, or not, in the system at hand. Test cases are organized into a test plan and executed in an automated way. Finally, the test results and the attack trees are used to identify vulnerabilities and generate recommendations for the system developer on how to fix them.

This paper deals with the subtask of generating attack trees in an automated way. Hence, in the rest of this paper, we focus solely on this problem.

2.2.1 The role of attack trees

The generation of attack trees serves a special purpose in the project as it is the basis of the automatic test case generation for automated penetration testing.

The attack tree generation (see Fig. 2.2) uses declarative programming and pre-condition logic to break the goal into smaller subgoals from a high-level system model. The goal of this phase is to explore the potential attack paths and necessary preconditions for them to satisfy.

The next phase uses parts of the generated tree as templates: it generates test cases based on it and the system input - Infrastructure-as-Code files. Besides our main goal (i.e., attack tree generation), we provide information about this second phase in the Design chapter with an example.

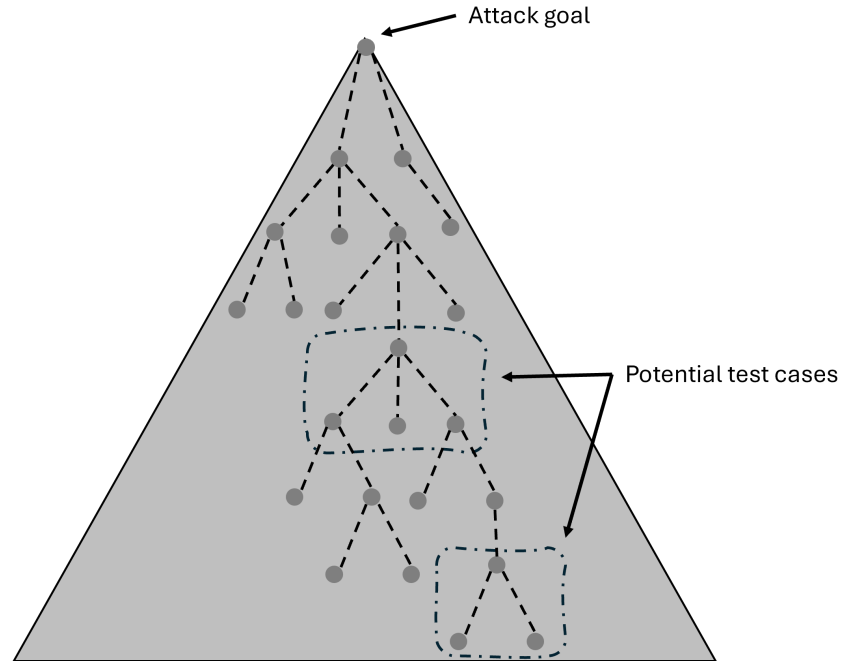


Figure 2.2: Our approach to build attack trees and identify executable test cases

Chapter 3

Related Work

The automatic generation of attack trees and attack graphs has gained significant attention in the field of cybersecurity. As systems grow more complex and organizations face an expanding array of threats, the demand for effective tools to detect and assess vulnerabilities has never been greater. This chapter explores key methodologies in this area, positioning our research within the broader context of advancements in automated attack tree generation.

3.1 Threat modeling - attack trees and graphs

Attack trees and attack graphs are both logical, graphical models used to map out potential strategies attackers might employ to compromise a system. First introduced by Bruce Schneier in 1999 [14], attack trees have become a widely adopted tool for security professionals. These trees break down potential threats into smaller, more manageable components, offering a structured, hierarchical view of the attacker's potential actions. At the top of the hierarchy lies the attacker's ultimate goal, while child nodes represent the various sub-goals or steps required to reach that objective. This method has proven valuable for systematically identifying and analyzing security threats, particularly in isolating potential points of failure or vulnerability within a system. Attack trees can easily discover and help eliminate threats, some extended attack tree specifications can even give additional options to extend the capabilities of these trees making them the ultimate tool for security analysis.

Attack graphs, which were initially explored through the work of M. Dacier in the 1990s [2, 3, 4], offer more flexible modeling techniques that can represent a wider range of attack scenarios, often involving multiple paths and objectives. These graphs allow for the illustration of interconnected vulnerabilities across different parts of a system. Attack graphs provide a detailed visualization of how attackers can exploit system weaknesses, identifying multiple paths through which an attacker might achieve their goals. This versatility makes attack graphs especially useful in large-scale environments with complex architectures, where vulnerabilities and attack vectors are highly interdependent.

Both attack trees and graphs are indispensable tools for threat modeling and security assessments. They help analysts systematically understand how attackers may exploit vulnerabilities, anticipate potential attack strategies, and evaluate the risk posed by different types of attacks. However, while manually constructing these models has been common practice, it is time-consuming, prone to error, and often lacks scalability. Especially in

large, dynamic systems where attack surfaces continuously evolve, the complexity of manually creating accurate models grows exponentially. This need for more efficient solutions has driven a strong interest in automating the generation of attack trees and graphs. Automated approaches can improve accuracy, scalability, and efficiency, allowing security professionals to focus on mitigation strategies rather than the tedious task of diagram construction.

3.2 Syntax of an attack tree and graph

Despite their widespread use, one of the persistent challenges in leveraging attack trees and attack graphs effectively is the lack of a standardized syntax. As noted by Lallie et al. [11], more than seventy distinct configurations for attack graphs and over twenty variations for attack trees have been described in the academic literature. These different configurations use various representations to depict key elements such as preconditions, postconditions, and exploits. Some configurations represent exploits with shapes like rectangles or ellipses, while others use different forms for preconditions, or even leave preconditions implicit. The diversity of visual syntax complicates efforts to compare or integrate different models, leading to inconsistencies that hinder both cognitive clarity and practical application. These inconsistencies hinder the cognitive clarity of the diagrams, making them harder to interpret, especially for non-experts who may not be familiar with the underlying visual conventions.

The cognitive effectiveness of attack trees and graphs is another critical issue that has not been sufficiently addressed. Nevertheless, a balance between consistency and flexibility is often necessary. While standardized models provide a common framework, they may not always capture the nuances of specific systems. A more adaptive approach to visual syntax, allowing for customization based on the context, can offer greater clarity without sacrificing interpretability.

Beyond visual syntax, one of the central challenges in attack tree and graph modeling lies in the logical representation of attack paths and scenarios. Logical modeling refers to the process of capturing the underlying relationships between preconditions, exploits, and system states in a structured, rule-based format. These representations use formal logic, such as conjunctions, disjunctions, and (some of them) sequential dependencies, to model how an attacker can progress through a system based on certain conditions being met.

In an attack tree, multiple sub-goals might need to be achieved simultaneously (represented by an “AND” condition) for an attacker to reach a higher-level objective. Similarly, attack trees can model the possibility of multiple attack paths by using OR logic, where the fulfillment of any one of several conditions can lead to a successful compromise. The clear and accurate representation of these logical relationships is critical for understanding how vulnerabilities can be exploited and how to mitigate potential threats effectively.

3.3 Opportunities in automation

Given the limitations of manual modeling and the challenges posed by inconsistent syntax, automation presents a promising solution for improving the reliability and scalability of attack tree and graph generation. Automated systems can leverage advanced algorithms to analyze system configurations, identify vulnerabilities, and generate attack trees or graphs that accurately reflect the attack surface.

Moreover, automation offers the opportunity to enforce more standardized representations of attack models, helping to overcome the inconsistencies in visual syntax that have plagued manually created models.

3.3.1 Attack tree and graph generation

A recent survey by four researchers at the Technical University of Denmark classifies the methods for automating the generation of attack trees and graphs into three primary approaches: model-driven, analysis-driven, and vulnerability-driven[9]. They also identified that these approaches often overlap, with some works utilizing combinations of model-driven, vulnerability-driven, and analysis-driven methods. This hybridization reflects a need to tackle limitations such as scalability, meaning that multi-faceted solutions are increasingly common.

3.3.1.1 Model-driven approach

The model-driven approach begins with a system model, using specified rules to generate the tree or graph. A well-done example of this is the research of Ivanova et al. [8]. They present a method for transforming graphical system models into attack trees, considering the physical, digital, and social aspects of an organization. Their approach applies a set of transformations to model elements like locations, policies, and assets, recursively generating attack steps that include both technical and social engineering actions. A key feature is the handling of data mobility, allowing for more dynamic attack scenarios. While demonstrated using one modeling language, the authors argue their method applies to various graphical systems and attack models.

In another model-driven approach, Gadyatskaya [5] and colleagues proposed a method for generating attack-defense trees from socio-technical system models. This work introduces the concept of attack-defense bundles. These bundles are generated alongside the socio-technical model and can be used to synthesize attack-defense trees. This approach extends basic attack models by incorporating potential countermeasures directly into the model, which allows for basic attack-defense analysis and facilitates the selection and maintenance of security controls that are not initially represented in the model. These methods may face complexity problems in the attack tree or graph generation depending on the system for attack tree generation defined.

3.3.1.2 Analysis-driven approach

The analysis-driven method expands on this by integrating system analysis results, often utilizing formal methods such as model checking. Sheyner et al. [15, 16] introduced a notable early example to analyze system properties and generate attack graphs based on network vulnerabilities. Similarly, Ibrahim et al. [6, 7]. used the AADL (Architecture Analysis and Design Language) to generate attack graphs for SCADA systems.

Bezawanda et al. [1] introduced AGBuilder, an innovative AI tool designed for generating and refining attack graphs using event logs. Their approach employs natural language processing to convert vulnerability descriptions from databases like CVE into structured planning problems. This enables the automatic generation of attack graphs that reflect cause-consequence relationships between system states. A significant advantage of AGBuilder is its support for incremental updates, allowing for dynamic adaptation to changes in system vulnerabilities without the need for complete regeneration of attack graphs. This

capability is crucial for maintaining resilience against evolving cyber threats by facilitating the integration and reuse of smaller attack graphs into larger network models.

These methods often incorporate formal security properties, such as reachability or the compromise of key assets, allowing automated tools to explore possible attack vectors and generate models that represent realistic attack paths. However, as highlighted in the survey by Konsta et al. [9], one challenge these methods face is scalability, as they often struggle with the state-space explosion inherent in large systems.

3.3.1.3 Vulnerability-driven approach

Vulnerability-driven approaches leverage known vulnerability data, such as the Common Vulnerabilities and Exposures (CVE) database, to generate attack trees or graphs by matching system components to existing vulnerability templates. Phillips' and Swiler's work [12] on network vulnerability analysis exemplifies this approach, where they iteratively match attack templates to nodes in a network.

Later, Swiler et al. [17] expanded this approach by incorporating detailed attacker profiles, allowing their tool to refine attack graph generation based on the attacker's assumed knowledge and resources. This profile-based refinement introduces a more dynamic aspect to the template-matching process, as different attacker assumptions generate varying attack paths.

Recent developments in cybersecurity have harnessed machine learning to anticipate attack trajectories by constructing attack graphs, a method exemplified by Koo et al. [10]. Their 2022 study presents a pioneering framework that utilizes deep learning for generating attack graphs. By implementing a two-phase approach—comprising learning and generation stages—the model first learns from existing attack graphs based on a vulnerability database and then applies this knowledge to create attack graphs using only network topology and system information. This innovative approach showcases its potential for scalability, however, the model's dependence on existing vulnerability databases introduces challenges in adapting to novel threats, particularly in dynamic environments such as the Internet of Things (IoT). Despite this, the method's encoding-decoding model offers a cost-effective and efficient means.

Additionally, the tool developed by Pinchinat et al. [13] goes beyond static vulnerability data by using system logs to synthesize attack trees. Their method parses logs using context-free grammar to generate attack traces that are then formalized into attack trees. This semi-automated process can dynamically respond to the system's operational environment, updating the attack model in response to observed events.

3.3.1.4 Our solution

From the above methods for generating an attack tree, we used the Model-Driven approach. Our method leverages an initial system description - a high-level model - using Archimate, which is then translated into Prolog and Answer Set Programming (ASP) to explore and generate attack models. This approach aligns with methodologies that directly translate system models into security models using predefined transformation rules. We chose this approach because it offers a structured and systematic way to represent potential security threats based on the architectural design of the system, ensuring that security analysis is closely aligned with the actual system architecture. Furthermore, this method effectively generates attack trees while minimizing time complexity, which is fur-

ther reduced through the use of ASP, making our program a better time-space complexity software compared to other solutions.

Chapter 4

Design

In this chapter, we describe the design and organization of our program, highlighting the critical modules that enable the efficient generation of attack trees. Each module plays a specific role in transforming a high-level system model into a declarative form, generating the attack tree, handling the complexities that arise during tree generation, and finally exporting and analyzing the results. The following section introduces the main model of the program and the subsequent ones provide a detailed breakdown of these modules and their contributions to the overall design.

4.1 The basis of the program - the declarative model

At the core of the program lies the declarative model (see Appendix), which offers a high-level abstraction of the system. It defines key elements such as the actors, services, data flow, and the behavior of the system and potential attackers. This influences all the parts of the system, it is the base of the attack tree generation: it defines the precondition logic, which creates the trees, and the system to be tested.

4.1.1 Model Components

The system is represented through a structured hierarchy of conceptual elements, which serve as the building blocks of the declarative model. A host refers to any device or machine within the network, while gateways are a specialized subset of hosts that connect networks. A network represents a set of computers capable of local communication representing real-world scenarios where devices are connected to the same subnet and can reach each other.

An agent is an abstract entity representing actors with specific behaviors. This can include running processes, human agents: system administrators, or more specialized entities such as attackers. Processes provide services within the system.

A datum encompasses any piece of information within the system, such as readings from a sensor. A credential is a specific type of data such as passwords, encryption keys, or biometric identifiers like fingerprints. They are used to access, use resources, and get privileges.

A channel represents a data communication pathway, for example, a link between remote processes using it for data exchange. Storage refers to any physical or logical space on a host where data is maintained, such as a database disk or buffer.

Finally, privileges define access controls within the system. They are for checking whether an agent can use services, send or receive data over channels, interact with storage, or escalate access to administrative levels (e.g., root or user privileges) on a host or device.

4.1.2 The static system model

With the help of the model components, we can define the entities of the system, but to make the model represent a real system, it is necessary to capture the relationships and connections between them. This is the responsibility of the static system model.

It establishes several key relationships, such as a host being connected to a network, representing the communication link between devices. It also defines where a process runs, illustrating which computational, logical tasks are carried out on specific hosts. Furthermore, the model captures how processes provide services, representing the functionality offered to other parts of the system, and how processes rely on other services or data to function correctly.

In addition to these basic relationships, the static system model also addresses how services are exposed to the rest of the system: when a service is exposed, it becomes available for use by other entities, this represents eg. open ports. Each service can offer specific capabilities, such as the ability to access another service, perform certain operations, or control processes. This includes actions like stopping or updating a process, or interacting with data: reading or writing it.

The “can attach to” relation represents a human entity eg. a sysadmin, who can be able to connect to a host of the system or a network.

Data management and communication are also key aspects of the static system model. The model tracks how data flows through the system, representing how a particular piece of data may be a copy or function of another. Data moves across the system through communication channels, which carry information between processes, hosts, and networks. These channels are vital for maintaining the flow of information and ensuring that processes can interact with the data they need.

Storage within the system is also addressed by the static model, which represents where data is stored and how it is managed. Data is typically stored on hosts in various forms, such as databases or buffers, which are crucial for maintaining the system’s persistent state and supporting ongoing operations.

Finally, the static system model accounts for the role of agents within the system, whether they are human users or automated processes. Agents can possess credentials, such as passwords or encryption keys, which grant them access to specific parts of the system. These agents may also have privileges that allow them to perform actions like receiving or sending data on a channel, storing or retrieving data from storage, get user or root rights, and using a certain service.

4.1.3 Dynamic system model

The dynamic system model, as the name suggests, is system-specific and varies from one system to another. This model defines, for example, the privileges granted when a particular credential is obtained, or the prerequisites for using a service—whether a specific privilege is required or if access is granted simply by reaching the service through an open port.

The dynamic model also outlines the rules about how data could be sent or received over channels, as well as how data can be stored or retrieved from a storage medium. Additionally, it defines the exact privileges necessary to escalate access, such as gaining root or user-level permissions on a host. These system-specific rules shape how interactions can occur within the network and what are the preconditions of these.

4.1.4 General system rules

The general rules are responsible for defining the basic preconditions that must be satisfied for entities within any system to perform specific actions, such as accessing services, performing operations, stopping or updating processes, and interacting with data, channels, and storage.

The capability of accessing services is based on the relationships between agents, processes, and services. An agent or process can access a service if it can use another service that provides access or, in the case of processes, if they share the same host or network with another process providing the service and the service is exposed. Similarly, a human can access services by attaching to the relevant host or network where the service is provided in the last case the service should be exposed.

The capability of performing operations depends on whether the agent or process can use a service that offers the required operational capability.

The capability of stopping or updating processes follows a similar logic. An agent or process can stop or update another process if it can use a service that offers such capabilities. This ensures that only authorized entities can control or modify active processes.

Regarding the capability of accessing channels, processes can interact with a communication channel if the channel traverses the host on which the process runs or if both the host and channel are connected to the same network. For humans, channel access is similarly tied to their ability to attach to the relevant host or network.

For storage access, both processes and humans can interact with storage if the storage resides on the host they are connected to. Processes must run on the host where the storage exists, while humans must attach to the host that manages the storage.

The capability of reading and writing data is managed through several preconditions. Agents and processes can read or write data based on their access to services that provide these capabilities, the ability to send or receive data over channels, or retrieve and store data in storage systems. Additionally, agents can read data by accessing a copy of it or write data if they can modify a related piece of data that the original data is derived from.

Finally, the capability of obtaining credentials is based on whether an agent already possesses the credential or can read a copy of it from stored or transmitted data.

4.1.5 Attacker-specific rules

The attacker's capabilities are determined by the control they achieve over agents, processes, or other system resources. Once the attacker has gained control over certain agents, they can exploit and manipulate various system components. These rules model the main behavior of an attack, making these essential during the automated attack tree generation.

The attacker can access services if they control an agent with the necessary permissions. If an agent (eg. a process) has access to a particular service, the attacker inherits

this capability through their control of the agent. Similarly, the attacker can use any service the controlled agent can use.

In addition to accessing and using services, the attacker can perform operations if the controlled agent has the authority to do so. For instance, if an agent can execute specific commands or initiate processes, the attacker gains this capability by proxy. The ability to perform operations extends the attacker's reach into system control mechanisms, allowing them to execute actions that may affect the system's state.

One of the most critical aspects of the attacker's potential is the ability to obtain privileges. When the controlled agent can escalate their privileges—such as gaining administrative rights—the attacker can leverage this ability to gain higher levels of control over system components. Privilege escalation is a common tactic in attacks aimed at gaining broader system control which may be an ideal subgoal to achieve our main goal in the root of the attack tree.

Similarly, the attacker can obtain credentials by directly compromising them or through the controlled agent's access. If an agent can obtain credentials such as passwords or encryption keys, the attacker can also acquire these credentials. This allows them to bypass authentication mechanisms and gain unauthorized access to secure areas of the system. In some cases, the attacker may not need to control an agent but can instead directly compromise credentials, further enhancing their control over the system.

In terms of connectivity, the attacker can attach to networks or hosts by controlling a human entity that has the necessary permissions. For example, if a system administrator or user can attach to a specific network or host, the attacker can exploit this capability. This enables the attacker to move laterally within the network, potentially compromising other systems connected to the same environment. This function would help us determine which roles, users in the system may have too much permission.

Furthermore, the attacker can access communication channels if they control processes or humans with access to those channels. This allows the attacker to intercept, manipulate, or disrupt communications between system components. Similarly, access to storage systems becomes possible if the attacker controls processes or human users with access to those storage units. This enables the attacker to read from or write to data storage, data exfiltration, or corruption. This can also happen if the attacker can gain full control over a machine to which the storage is connected.

The attacker can also achieve typical low-level attack goals by targeting processes and hosts. They can control processes by updating them, injecting malicious code, or fully controlling the host on which the process runs. Through these methods, the attacker can corrupt processes, disable them, or exhaust their resources, resulting in service disruption or denial.

When attacking hosts, the attacker can gain full or partial control depending on the level of privileges they acquire. By obtaining root privileges, the attacker can fully control the host, granting them the highest level of system access. Partial control can be achieved by exploiting user-level privileges or by controlling specific processes that run on the host. In either case, the attacker's control over the host allows them to compromise the system's confidentiality, integrity, or availability.

Finally, high-level attack goals can be achieved by targeting data and services. The attacker can make data unavailable by corrupting or encrypting it, rendering it inaccessible to legitimate users. They can also alter services by controlling the processes that provide them, or make services unavailable through overloading, disabling processes, or

disrupting dependencies. These actions result in service outages, data loss, or severe system instability, all of which can be leveraged by the attacker to achieve their broader objectives.

4.1.6 Potential test cases and testable subgoals

The following outlines testable low-level attack goals that represent practical and achievable objectives for an attacker within a system. The word “testable” means that these goals are the ones that will be sent for the next phase of the project. To help this process we designed the system model to give the next layer additional help (see Fig.4.1). Besides the testable precondition, it gives additional helper nodes that contain contextual information.

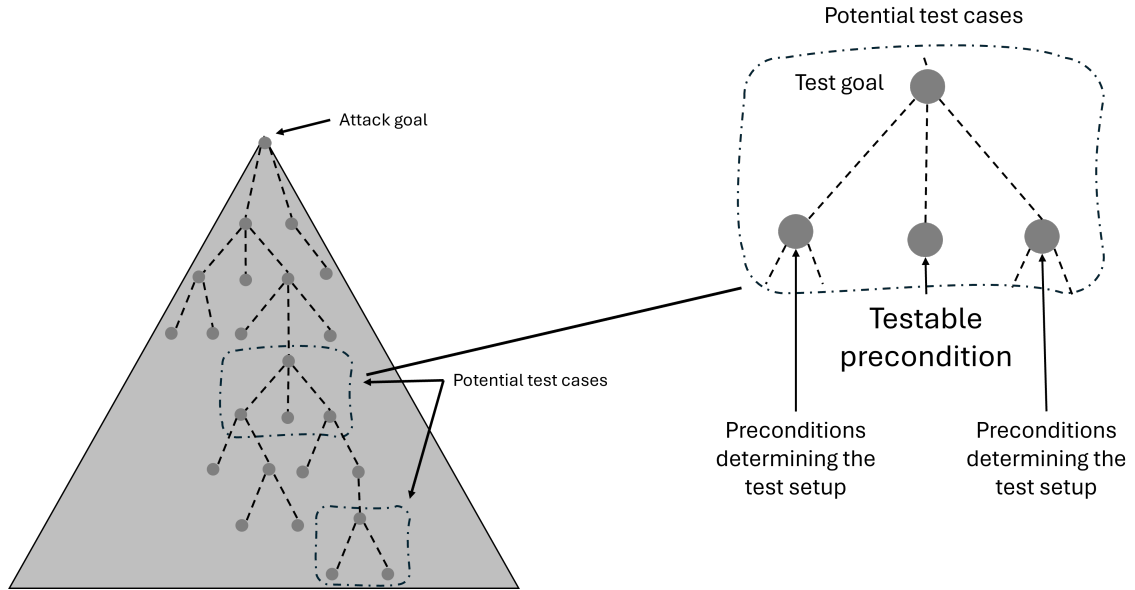


Figure 4.1: Template-based testable subgoals

As this group of rules is the key to a successful attack, let’s take an example to better understand how these rules work and why are they so important in the whole system (See Fig.4.2).

At a certain point in the tree, we got a potential test case with the test goal of: “attacker can exhaust web server process”. The preconditions for this are that the webserver provides two services on port 80 and 8080 and the attacker can access the first one. With the help of the precondition logic, the preconditions of this access can be further explored, now say that it is through an internet connection and port 80 is an open port. The testable precondition is that the attacker could overload the webserver through port 80. Using the next phases of the project this test case matches a certain template/test case configuration and from that, it creates an executable test case. After the execution, an observer process can determine the result of the attack, and if it was successful, the test goal becomes “True”. After running these tests, with the help of the precondition logic, the truth value of each node can be determined (even the main attack goal’s value) and so the penetration test is done.

Test case template example: DoSHTTP

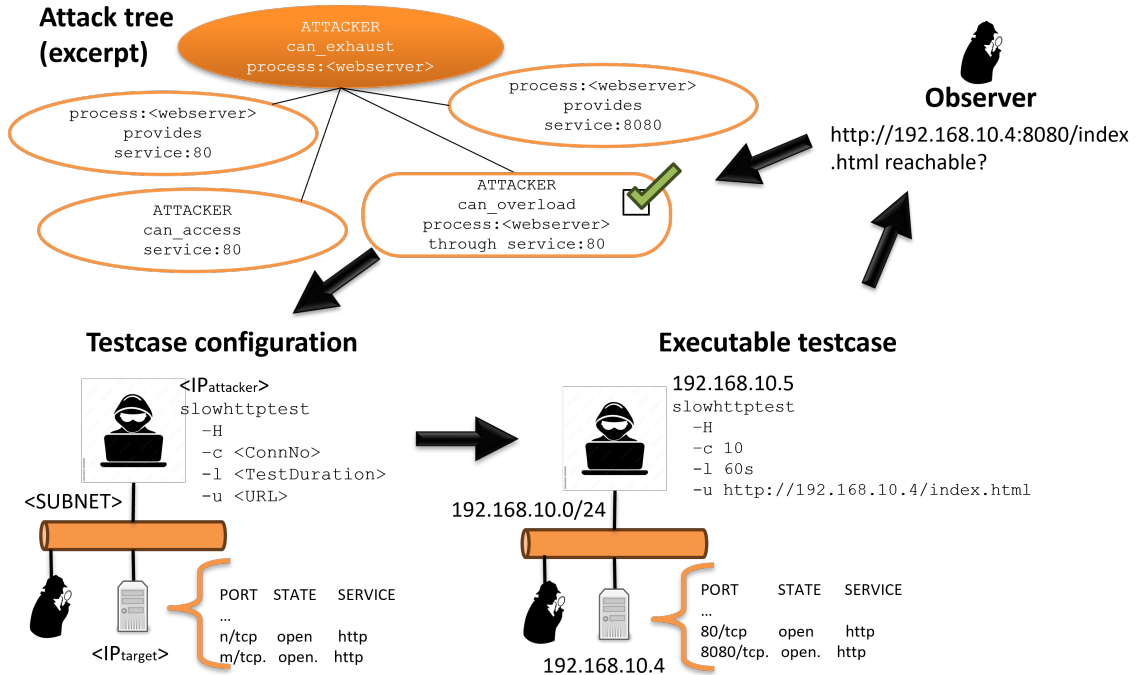


Figure 4.2: Example of a test case

These goals include controlling a human agent through various social engineering methods such as bribing, extortion, or exploiting insider threats. While this remains a theoretical possibility in the model, it won't be actively tested. Another fundamental attack vector involves compromising credentials through methods like password guessing or exploiting default passwords.

The model also addresses process-level attacks through services. When an attacker can access a service provided by a process, they might attempt several approaches: crashing the process through techniques like fuzzing, injecting malicious code into the process, or implementing denial-of-service attacks through process overloading. Additionally, the attacker might focus on overloading specific services to prevent legitimate users from accessing them.

More sophisticated attack patterns involve privilege escalation scenarios. In cases where an attacker already controls a process running on a host, they might attempt to break out of their current privileges to gain full control over the host system. This essentially represents a privilege escalation attack, where the goal is to obtain root-level access from user-level privileges. A more challenging scenario occurs when the attacker only has access to a service provided by a process, rather than controlling the process itself. From this position, they might attempt to gain either full or partial control over the host system, with partial control typically meaning user-level access. This could be achieved through techniques such as code injection via a process's service. Each of these attack goals is designed to be practical and testable, allowing for systematic evaluation of system security through automated testing approaches.

4.2 Components, interfaces of the program

After creating the model of the system, we defined key requirements to ensure the program's input and output were both reliable and efficient. These requirements shaped the design of the system, allowing us to divide the program into well-defined modules, each responsible for handling specific tasks in the attack tree generation process. This modular approach not only promotes clear organization but also makes the program work smoothly and efficiently. To illustrate the system's operations, we created a flowchart of it, which can be seen on Fig. 4.3.

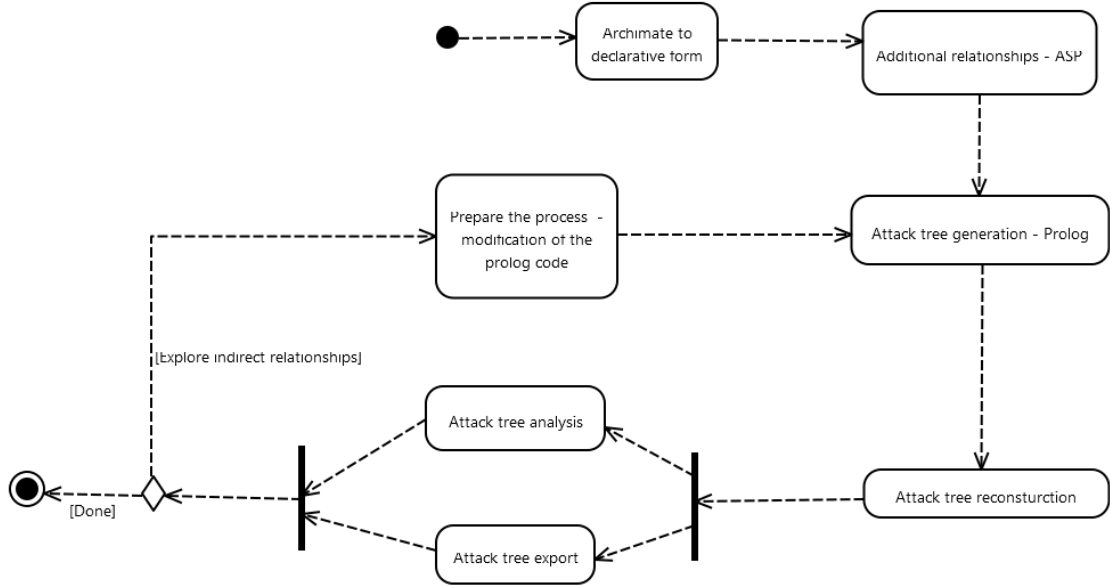


Figure 4.3: Flowchart of the system

4.2.1 Inputs, interfaces

4.2.1.1 The instructions - goal of the tree

One of the key inputs to the system is the goal of the attack tree, represented by the root node. As part of our automated penetration testing framework, the project requires the development of an interface to capture this goal. The goal is a high-level objective based on system parameters provided by another component, and it is designed to reflect violations of the CIA triad (Confidentiality, Integrity, Availability). This interface is implemented in a declarative format using Prolog, which will be discussed in detail later.

4.2.1.2 The high-level model - Archimate

There are several approaches to modeling an IT system, as the same system can be represented in various ways. To facilitate efficient information extraction from a high-level model, it was necessary to develop a specific method for structuring the system model. This method ensures that the model is fully compatible with our program and can be seamlessly integrated into our processes. By refining this approach, we aim to standardize the way the system is depicted, making it easier to interpret and utilize within the framework of our software.

4.2.2 The core

4.2.2.1 Turning system model into a declarative model

After exporting the system model from ArchiMate into files representing relations, elements, and properties, we convert the data into ASP and Prolog formats (discussed in the following subsections). This part of the program reads the model files, maps the elements to declarative system nodes such as hosts, processes, and network devices, and analyzes the relationships between them while incorporating additional properties, such as whether a port of a host is open. By following these steps, we successfully generate the declarative system model, laying the foundation for constructing the attack tree.

4.2.2.2 The tree generation - Prolog

Prolog, a logic programming language rooted in formal logic, is particularly well-suited for tasks that involve reasoning and deduction, such as generating attack trees. The declarative nature of Prolog, where logic and relationships are expressed rather than sequences of commands, greatly aids in this process. By leveraging Prolog, we can easily model complex systems in a way that mirrors real-world logical interactions. After creating the declarative system model, we can use additional rules to explore potential threats by applying the pre-condition logic embedded within these rules. These rules handle various critical actions, such as determining access to a node, identifying how a service can be overloaded, or other forms of system vulnerability.

Together, these sets of rules (explained in Section 4.1) allow for the construction of an attack tree. With Prolog's logical inference capabilities, we can generate this attack tree dynamically: based on a special query from the other module of the main penetration test automation project. Once the attack tree is defined, the next step is to traverse it systematically, ensuring each potential attack path is identified and saved in a format that can be used for further analysis or future reference, ensuring compatibility for ongoing threat assessments or defense strategies. This approach ensures both flexibility and expandability in the exploration of security vulnerabilities.

4.2.2.3 Reducing complexity - ASP

There might be some problems during the generation of the trees: as discussed in the related work chapter (see Chapter 3), generating an attack tree can lead to exponential growth in runtime complexity. This occurs because there may be multiple ways to achieve a given goal, and some of these paths can be quite long or complex. For example, consider a scenario with three hosts (hostA, hostB, hostC) behind a firewall on the same intranet. If a subgoal in the tree is to gain administrator privileges on hostA, the first step is to establish access to it. (A subgoal is a precondition of the main goal - a certain node in the tree at a certain depth.) This could be done directly through the firewall, or, if access has already been gained to hostB (from a prior action), we might use hostB as a stepping stone. There may be many more such alternative paths. As this is only a subgoal, it might appear in multiple attack scenarios, therefore the node could be evaluated multiple times. As a result, the number of possible routes to the main goal (root of the tree) can grow exponentially.

To minimize this complexity, we have opted to use Answer Set Programming (ASP) to explore indirect relationships between services, processes, and hosts in the model. These

relationships are then added as additional system facts. In the example above, the multiple paths to access hostA are abstracted into a single “`ecan_access hostA`” relation. If needed, the system can later reveal the specific paths (See Fig. 4.3) - and explore the indirect relationships, but during the generation of the main attack tree, this relation is treated as a fact rather than a query. This simplification applies only to basic relations, such as access or usage. Importantly, attacker actions, like compromising a host, are excluded from this process to ensure they are explicitly represented in the tree and do not disappear during abstraction. For instance, when an access expression has two children—one representing an attacker action (e.g., compromising a host) and the other representing access through a service—using ASP in this case would simplify the main tree by only showing the access expression. This abstraction hides its children to prevent exponential growth in the attack tree generation. As a result, attacker actions will only appear if specifically queried, rather than in the main tree itself. This could make the attack tree harder to interpret, as the attacker actions could remain hidden.

4.2.2.4 Reconstructing the tree

Once the attack tree has been generated, we need to reconstruct its structure from the output provided by the Prolog module. This process is handled using a script, which systematically organizes the Prolog results back into a cohesive tree format.

The Prolog output provides logical statements that describe how each node relates to others using parent-child relationships. The script interprets these statements and maps them into a hierarchical structure, where each node represents a specific event, condition, or action.

During reconstruction, the script ensures that each node is accurately positioned within the tree, preserving the logical relationships between parent and child nodes. This step is crucial, as it ensures that the attack tree not only reflects all possible attack paths but does so in a manner that is clear, organized, and easy to analyze. The reconstructed tree is then ready for further processing, including visualization, analysis, and export to a database for more advanced querying and reporting.

4.2.3 Results, output interfaces

4.2.3.1 Analyzing results - recommendations from the high-level attack tree

The resulting attack tree serves as an effective representation of the system’s vulnerabilities, providing a clear and structured overview of potential threats. Each leaf node in the tree represents a critical step that could lead to the system’s compromise, highlighting key areas where security breaches may occur. By mapping out these potential attack paths, the attack tree becomes a valuable tool for system designers, especially during the early stages of project development, offering a comprehensive form of threat analysis. This proactive approach aligns well with the security-by-design principle, ensuring that security considerations are embedded into the system from the outset. Furthermore, the leaf nodes can be easily expressed in disjunctive normal form (DNF), a logical structure that simplifies the breakdown of the system’s security goals into individual, actionable components. This logical representation aids in identifying and prioritizing the most critical threats, allowing for targeted efforts to mitigate them and strengthen the system’s overall security posture. In this way, the attack tree not only identifies vulnerabilities but also guides designers in reinforcing the system against its most significant risks.

4.2.3.2 Exporting

The final process of the program is exporting, this is for preparing the output attack tree for further usage for the next modules of the main project.

Chapter 5

Implementation

This chapter provides a detailed breakdown of the program’s components and outlines the system’s implementation.

5.1 The declarative form of the system model

First of all, we had to convert the system model described in Section 4.1 to declarative format. This process was simple, as illustrated in the example below (See Fig. 5.1). However, we made some adjustments by introducing additional rules. These are related to the compatibility of the ASP output: eg. when we check if no external facts (marked with “e”) are coming from ASP before we test a clause (See Fig. 5.1, right column, second line).

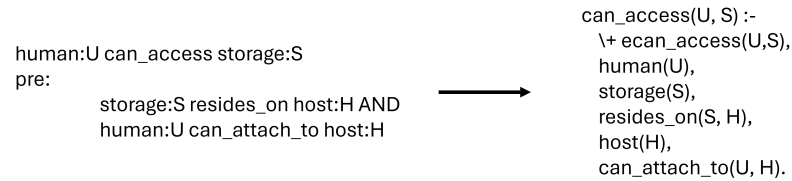


Figure 5.1: Declarative model into prolog conversion

5.2 Translating graphical into logical model

This translation was quite straightforward, we exported the archimate file into Comma-separated values (.csv) files and directly translated it into using a simple Python script. We translated the archimate elements into hosts, processes, services..., and relationships into connections such as whether a process runs on a host. To be easily trackable we used the archimate id during the translation to not lose the origin of a certain element. (Eg. there can be two separate services named ssh, running on two computers, which can lead to swapping these while interpreting the attack tree). The properties in the archimate model became additional facts in the system, eg. if a service is exposed. A small example shows in Fig. 5.2 how an element from the archimate model becomes a host and how a relationship becomes a declarative “is_connected_to” relation with the previously declared host.

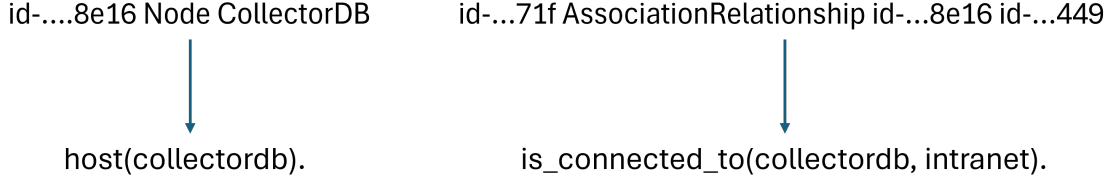


Figure 5.2: Archimate to declarative model translation

5.3 Generating system facts from indirect relationships - ASP

Using the generated declarative model from the previous component, we can generate the indirect facts from it. Before we do that we removed all the attacker rules from the system rules to ensure that only the non-attack related rules produce the indirect relation facts. We used Clingo¹, which is an answer set solver for ASP. The output of the ASP is modified so, that each fact derived this way has an “e” character at the start of the fact, which symbolizes that this is an indirect relationship, and if the user wants to know how it has been generated, it can be explored using Prolog.

5.4 Generating the tree - Prolog

We utilized the system model, system rules, and the output from the ASP-based component to generate the attack tree. During this process, we traced the precondition logic defined by the rules, starting from the input attack goal and continuing until a system fact or low-level attack rule was reached. To mitigate the potential exponential growth of the precondition logic tree—an issue only partially alleviated by ASP—we imposed a limit on the tree traversal depth.

We gather subtrees that are in an “OR” relationship during the process: when we identify a subtree where the root represents the main goal and the leaves consist only of system facts or low-level attack rules, we save it. These subtrees are minimal trees that satisfy the goal: they only contain “AND” relations. Each node in the tree is assigned a unique ID, which aids in reconstructing the tree structure from these subtrees. An example can be seen in Fig.5.3: the program first tried to check if the right precondition could satisfy the goal, but it could not. Then it tried the left, the left has 3 preconditions connected with the “AND” relation, which is represented by an arc on the figure. It first checked the two on the side, then the middle one. This one has two preconditions, connected with the “OR” relation, it checked the left one, which was a fact or a low-level rule (it became green). Then this tree was saved as a tree that satisfies all the requirements.

¹<https://potassco.org/clingo/>

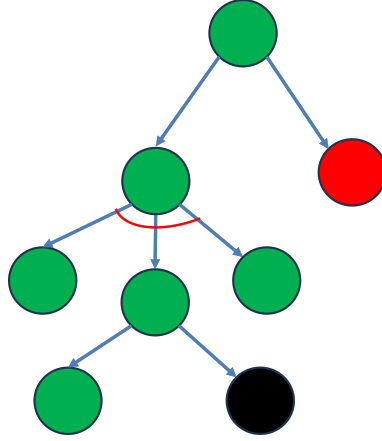


Figure 5.3: Tree traversal

While traversing, we can run into an issue with recursive rules. Some rules naturally form multiple looping sequences of different lengths. For example, a rule could allow us to access Service A through Service B, and then return to Service A, forming a loop like: can access A $\rightarrow$ can access B $\rightarrow$ can access A $\rightarrow$ can access B, and so on. In more complex cases, multiple rules might interact this way, creating many indirect recursive loops.

To solve this, we assume that if a node in the attack tree has a precondition that references another node already present higher up in the tree, it is evaluated as false (see Fig. 5.4). The reasoning is straightforward: if a node has a precondition that points to an ancestor node on the path to the root, checking this condition would lead to infinite recursion. While assuming the condition is true may seem plausible, it introduces uncertainty, as it cannot be fully verified. Therefore, the safest assumption is that such a condition is false. If it were true, it would not pose a problem: the precondition logic and the backtracking ensure that the tree traversal will still discover any valid paths where the condition holds elsewhere. We did this by tracking the called predicates and its parameters and checking if something already appeared previously and if so, we returned “False”. But why did not we have any problem with this while using ASP? The secret hides behind the nature of ASP: it automatically eliminates recursion.

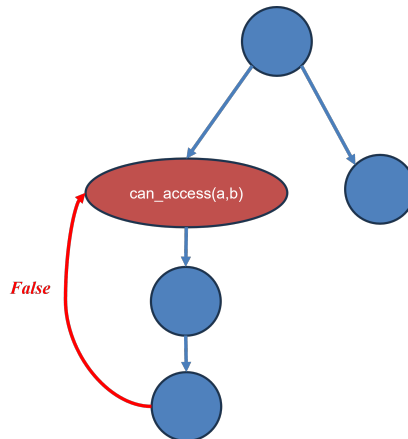


Figure 5.4: Recursion handling

5.5 Reconstructing the attack tree

The reconstruction was necessary because our subtrees are minimal and together these create the attack tree (See Fig.5.5). The image shows an example of what may be a result of an attack tree generation and how it should be reconstructed.

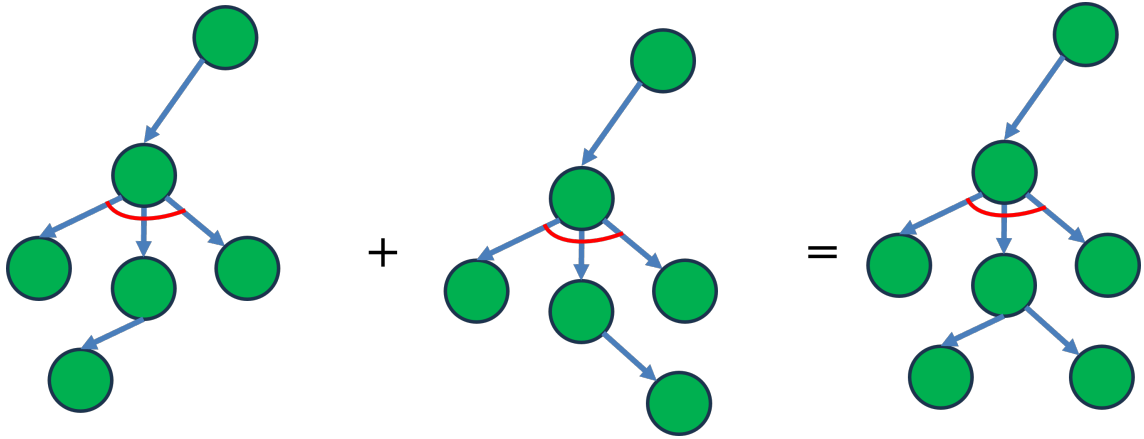


Figure 5.5: Reconstruction of the attack tree

To solve this first, we added explicit “AND” nodes into the subtrees that help us differentiate if we are looking at an “OR” or an “AND” in a situation where a node has multiple children in the original tree (it was originally the arc symbol on Fig.5.5). After that operation the reconstruction was straightforward, we needed just traverse through the subtrees and merge the nodes if their parents were the same recursively. The only exception was the newly added AND nodes, which could be merged only if their children were the same too.

5.6 Exporting the results - neo4j

To effectively manage and analyze the output of attack tree generation, we export the resulting data into a Neo4j graph database². Neo4j is a highly efficient native graph database that excels at storing and querying complex data relationships, making it an ideal solution for managing attack trees within our broader automated penetration testing framework, where this project serves as a vital submodule.

Neo4j facilitates the storage of attack tree nodes as graph nodes and the relationships between these nodes as edges. This lightweight representation allows for easy management of the attack tree, creating an interface for subsequent modules of the automated penetration testing framework.

²<https://neo4j.com/> Last accessed: 3rd of November 2024

Chapter 6

Evaluation

To check whether our program works well, we created an example system and generated attack trees for various attack goals for this system. Our example system is quite small compared to a real-world system, but we believe that it is sufficient to test the capabilities of our program, and hence, to validate our approach. In particular, we can verify the program’s output manually and check if it gives the correct output.

6.1 The example system

Our example IoT system contains some sensors, whose sensor readings are collected and stored by a host, and the stored data is used by another host, where some process makes majority decisions based on the stored sensor readings. More specifically, the system (see Fig. 6.1) contains 3 temperature sensors that are connected to the internet and two hosts that are connected to an intranet, where the two networks are separated by a firewall. One of the hosts runs a collector process that fetches sensor readings from the sensors, and a database process that stores them. The other host runs a voter process that reads data from the database and makes decisions. For example, the voter may raise alarms if there are any anomalies (too high or too low sensed values) or missing data.

We have prepared the model with properties that represent this imagined system: each node has running processes, and processes provide services. For example, the sensor can be queried remotely and sensor readings are sent from them to the collector process through an unencrypted channel. This channel traverses the firewall, which provides access to the internet-connected sensors from hosts on the intranet through an outbound filtering service. The collector process collects the data from the sensors and then uses the “database write” service of the database process to store the data. Then the voter process can read the stored data through the “database read” service of the database. The use of these services requires privileges that can be obtained by possessing the appropriate credentials. Each intranet-connected device also runs a management process that provides a remote maintenance service, which can be used with specific credentials. A distinguished human agent, the “sysadmin”, has the management credentials and can maintain the imagined system.

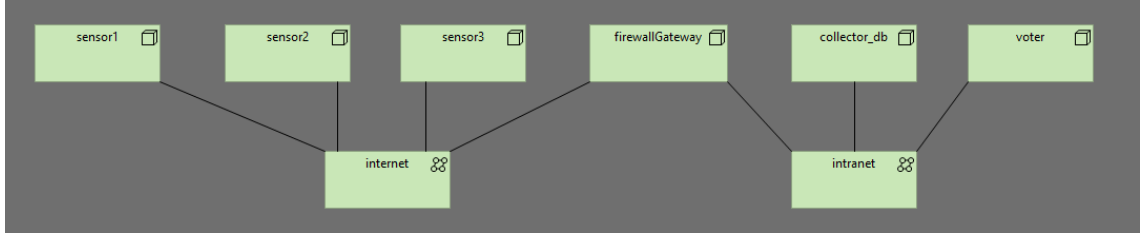


Figure 6.1: Network-view of the test system in Archimate

6.2 The test runs

We used our program to generate attack trees for the example system. For this purpose, we specified three different attack goals. In the following subsection, we will show the results that we got.

6.2.1 Making unavailable a sensing service

An attacker goal’s might be to create chaos in a system. In our case, this can be achieved by making at least two sensors unavailable. To see if and how this is possible, we asked the program to build an attack tree for the main goal “can_make_unavailable(a_attacker, s_sensing1)”. The predicate is quite easy to understand. The “a” character means that the attacker is an attacker-human agent, and the “s” means that sensing1 is a service. In this situation, the attacker is connected to the internet and that is all the attacker has at the beginning.

We generated an attack tree for this at a maximum depth level of 9 - this means that the farthest node from the root is 9 nodes away. Based on the circumstances of the situation the tree gives several attack options. An attacker easily gets access to the sensor, as it is connected to the internet as well as the attacker. This way he can overload it, or try to get control over the sensor process by injecting code into it. He might also try to crash the sensing process through the sensing service that it provides or stop the sensing process by gaining full control over the device that runs this process.

As the generated attack tree is quite large, we cannot visualize the entire tree. Rather, we show a part of the tree for illustration in Fig. 6.2.

Let’s see how this partial attack tree could be interpreted, and what a small, simplified subtree looks like (this one has a depth of 3 on the image). The attacker wants to make the sensor1 service unavailable. To achieve this, he could disable “p_sensor1”, which is the process that provides the service. This could be achieved by crashing the process through the sensing service. The attacker needs access to the sensor, but this is already given: he is connected to the internet as well as the sensor, and the port of the sensing service is exposed. In this case, the “can_disable” node of the tree represents a testable sub-goal, where the “provides” and “can_access” nodes are the pre-conditions for the test setup, and the “can_crash” node can be associated with a test tool (e.g., a fuzzer) that can try to crash the process by feeding it with various inputs. If the test is executed and it results in successfully crashing the sensing process on the sensor device, then the sensing service becomes unavailable and the root goal can be fulfilled. Following the definition of DNF, we can represent it with the leaves of subtrees connected by “AND” and these clauses connected by “OR”. Since this attack tree is relatively small, its DNF (disjunctive normal form) would include only a few leaves. To create a simplified version—showing what

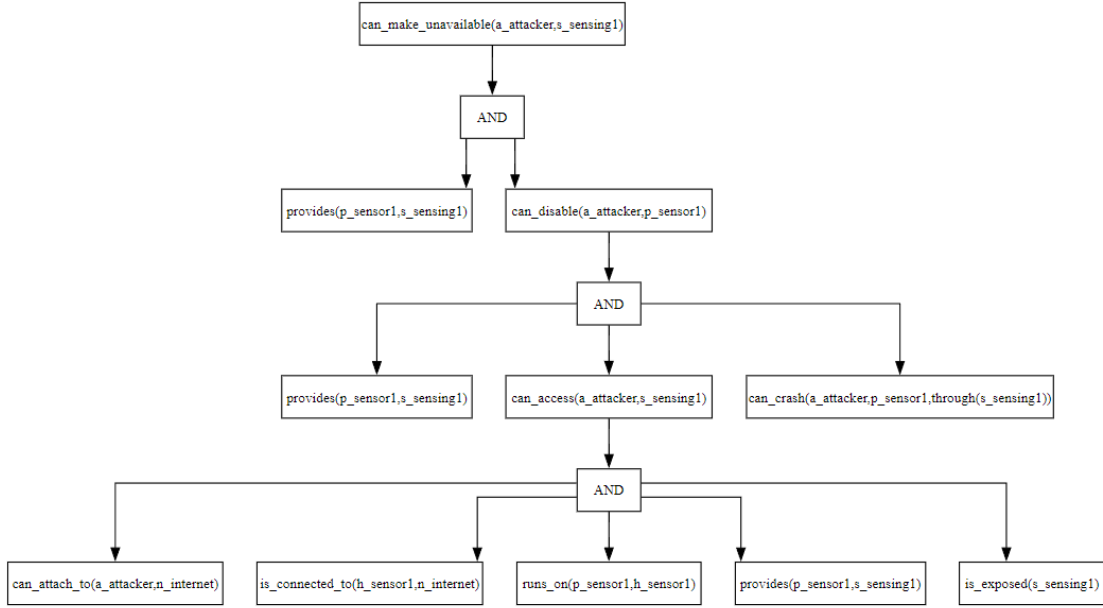


Figure 6.2: A small, simplified subtree - Example I.

conditions the attacker needs to satisfy for a successful attack—we can remove the system facts. The remaining node is “can_crash”, which represents a testable condition. This means that the necessary system conditions are present for the attack, the attacker might use these conditions to crash the service and make it unavailable. However, to reduce the system’s vulnerability, we may adjust the system facts. For example, by changing “is_exposed” (indicating the service is directly exposed, such as having an open port in this example), we can make the system more secure. This could be achieved, for instance, by using a firewall.

6.2.2 Tampering data flow

Creating chaos can be achieved in another way: focusing on the sensor data instead of the sensing service. We can try to sabotage the data at any time when it is traveling from the sensor to the voter.

We used “can_make_unavailable(a_attacker, d_stored_sensor3_data)” for this test. We would like to build an attack tree with the main goal that the attacker wants to make the sensor3 data unavailable, which is stored in the database. This can be done in many ways; one example is by modifying the sensor data in the storage where it is stored or another one is by corrupting the sensor data when it is travelling through channels.

Let’s see an example subtree from this case too (see. Fig.6.3).

This example tree shows that the data is stored on the storage drive of the database, so the attacker wants to get full control over the host that runs the database to write on this drive. This can be achieved through the database service, for example, by getting control over the database process. As this subtree (even after simplification) is very large, we could only put a part of it into the document. We only show how the firewall can access the database, while the other branch that would show how the firewall can be controlled cannot be seen. The first precondition is easy to understand: the firewall and the host that runs the database process are on the same network (i.e., the intranet). The second

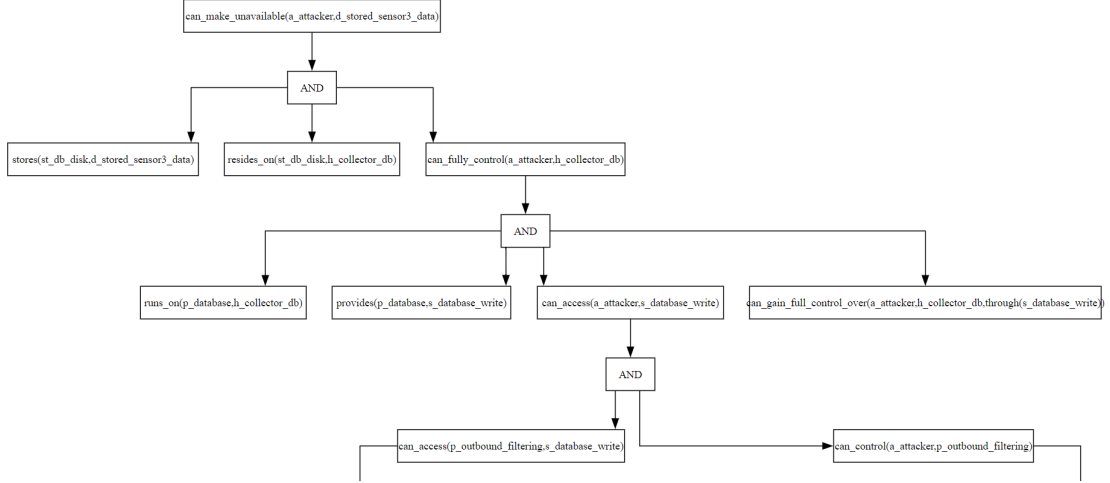


Figure 6.3: A small, simplified subtree - Example II.

precondition can be achieved because the attacker can access the firewall management service and can try to inject code into its process.

The part of the DNF in this tree contains the injection attack, the exposed firewall management console, and the exposed database write service. Besides these, it contains a potential privilege escalation too (`can_gain_full_control_over`).

6.2.3 Modifying the data

The alarm can be misled if the attacker can modify the data to show too low or too high values. To try this out we built an attack tree with the main goal of writing the data in the database. To achieve this - based on the generated attack tree - the attacker might send malicious packets through the firewall (after compromising it), try to get access to the exposed database service, and try to guess the password (eg. brute force), which gives write privilege to the data in the database. The goal can also be done by gaining full control over the host machine, by attacking the management service or the database read service of the computer using exploits. This way we can get write access to the disk containing the data and directly write it. He can try to inject malicious code into the management process or the database process and then escalate privileges to get full access to the device and then write the data.

An interesting example attack tree is about writing the data through the data channels (see Fig. 6.4).

The stored sensor2 data is a function of the data from the sensor because it is a copy of it (this can be interpreted as this is an identity function), and this data is carried on a data channel. If the attacker can send on this channel, he can modify the data. To do this, he needs access to it and obtain the necessary privileges. As the subtree is too wide and deep, we had to cut it at this point. The attacker can get access to it through the firewall as it travels through it, this can be done by attaching to it by controlling a human agent. He can obtain the privilege by injecting code into the management process. The data travels through the firewall, and it does not require any credentials (this data flow is not encrypted), so he can directly modify this data.

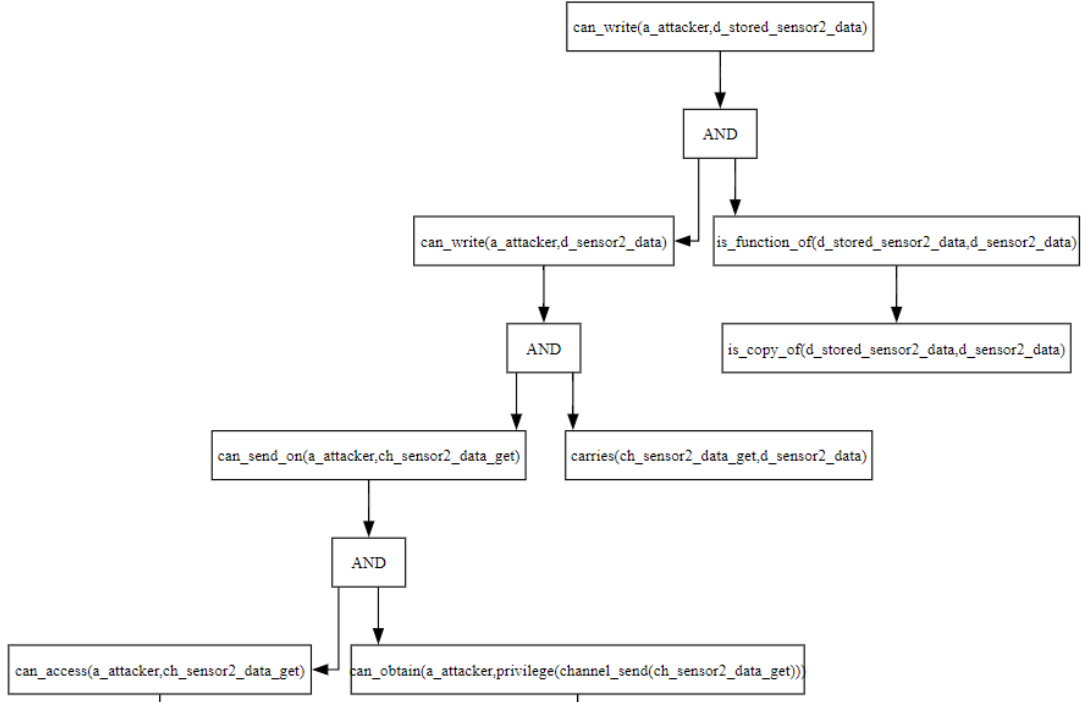


Figure 6.4: A small, simplified subtree - Example III.

The DNF part of this subtree contains the controlling of the human agent, the fact that through the firewall data can be sent (there is no encryption in the connection), the code injection into the management process, and the open port of its service.

6.3 Analyzing the results

Based on the DNF of the trees, we can analyze them and check the most common methods to attack the system. Let's check the second example ("can_make_unavailable(a_attacker,d_stored_sensor3_data)"). Since this system is quite small, we can only make minor changes to make it more secure. However, in a larger, real-world system, this is the part where the security-by-design approach becomes important.

We have gathered some specific information from its DNF, as shown in Fig 6.5. The chart illustrates the percentage of subtrees (attack scenarios) in which a particular leaf appears. Based on this data, we can recommend several system improvements. For instance, enforcing a strong database password could reduce the probability of a successful attack by 4.3%. Additionally, attackers might target the firewall and collector management software. By hardening these systems—such as by sanitizing inputs to prevent code injection and limiting management panel functions—we can significantly reduce risks.

Furthermore, insider threats (e.g., coercion or extortion) could allow unauthorized access to the collector management software or even the firewall. To mitigate this, in large organizations, we could recommend that administrators should be granted only the minimum privileges necessary, with no single "super-admin" role.

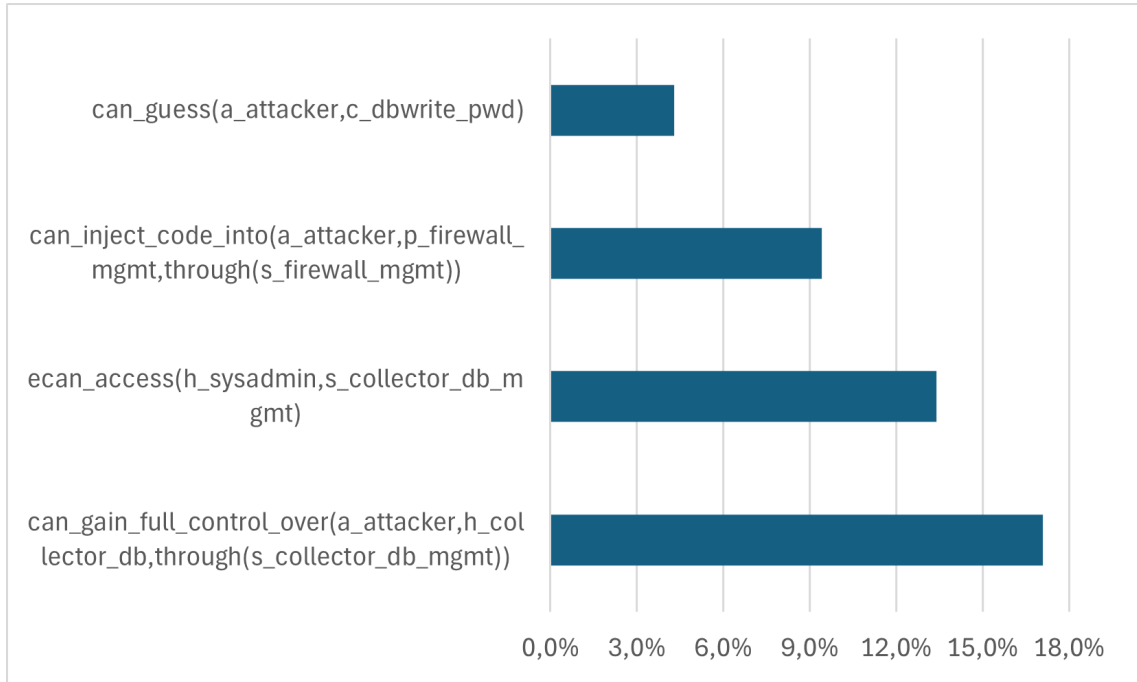


Figure 6.5: Frequency of the leaves in subtrees (DNF)

6.4 Optimization results

Besides testing the correctness of the program, we wanted to check how much the usage of ASP optimized the process of attack tree generation, we used attack trees at depth level 8 on each example from the previous sections. The vertical axis on Fig 6.6 shows how many nodes are in a tree with and without using ASP. In each test, the difference was about 30%. This means that the program needs noticeably less computation to generate and process a tree when using ASP: during the tree generation, where it does not need to evaluate these expressions (multiple times), as well as the merging process, where it does not need to handle these nodes.

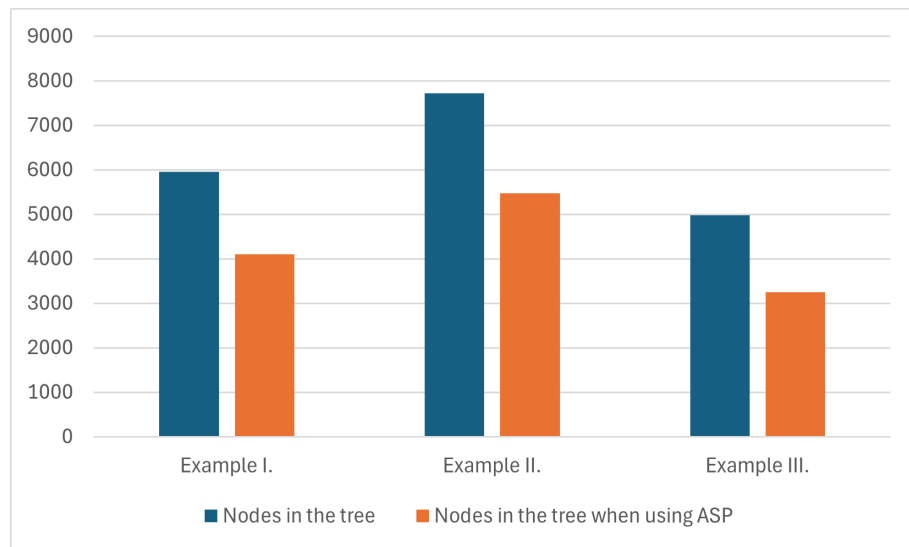


Figure 6.6: Number of nodes with and without ASP usage

Chapter 7

Conclusion

In our research, we solved the task of building a system for automating attack tree generation to assess cybersecurity risks effectively. This project emphasizes the importance of integrating security at the earliest stages of system design, utilizing high-level models and automated methods to ensure proactive defense strategies. By leveraging Answer Set Programming (ASP) and Prolog, we translated architectural models into attack trees that map out potential vulnerabilities within the system. This approach not only speeds up the identification of risks but also minimizes manual work, making advanced security analysis accessible to a broader audience.

The attack trees created in this process provide a comprehensive view of possible attack paths, each mapped to specific weaknesses within the system. Testing and evaluation on a sample IoT system showed how the model finds potential vulnerabilities, from sensor data manipulation to data flow tampering. Each generated attack tree points directly to areas that would benefit from heightened security, such as adding authentication protocols, reinforcing data channels, or limiting permissions on critical nodes.

Looking forward, we will extend the declarative model as well as the test system. We will focus on further optimizing the attack tree generation, and continue working on the automatization of penetration testing using the program demonstrated.

Acknowledgements

The research presented in this paper was supported by the European Union’s Horizon Europe Research and Innovation Program through the DOSS Project (Grant Number 101120270).

I would like to thank Dr. Levente Buttyán for his endless support throughout the entire project and for developing the program’s declarative model. I would like to thank Dr. Dorottya Papp for the example test case in section 4.1.6 and for Figure 4.2, which represents this example. I’m also very grateful to Dr. András Pataricza and András Földvári for their valuable advice on the attack trees and the ArchiMate-to-declarative conversion, as well as for creating the test system in ArchiMate.

Bibliography

- [1] Bruhadeshwar Bezawada, Indrajit Ray, and Kushagra Tiwary. Agbuilder: An ai tool for automated attack graph building, analysis, and refinement. In *IFIP Annual Conference on Data and Applications Security and Privacy*. Springer, 2019.
- [2] M. Dacier. *Towards quantitative evaluation of computer security*. PhD thesis, Institut National Polytechnique de Toulouse, 1994.
- [3] M. Dacier and Y. Deswarte. Privilege graph: an extension to the typed access matrix model. In *3rd European Symposium on Research in Computer Security, ESORICS94*, pages 319–334. Springer, 1994.
- [4] M. Dacier, Y. Deswarte, and M. Kaâniche. Models and tools for quantitative assessment of operational security. In *Information Systems Security*, pages 177–186, Yorktown Heights, New York, USA, 1996. IBM TJ Watson Research Center.
- [5] Olga Gadyatskaya. How to generate security cameras: Towards defence generation for socio-technical systems. In *Graphical Models for Security*. Springer International Publishing, 2016.
- [6] Alaa T. Al Ghazo, Mariam Ibrahim, Hao Ren, and Ratnesh Kumar. A2g2v: Automatic attack graph generation and visualization and its applications to computer and scada networks. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 2020.
- [7] Mariam Ibrahim and Ahmad Alsheikh. Automatic hybrid attack graph (ahag) generation for complex engineering systems. *Processes*, 2019.
- [8] Marieta Georgieva Ivanova, Christian W. Probst, Rene Rydhof Hansen, and Florian Kammüller. Transforming graphical system models to graphical attack models. In *Graphical Models for Security*. Springer International Publishing, 2016.
- [9] Alyzia-Maria Konsta, Alberto Lluch Lafuente, Beatrice Spiga, and Nicola Dragoni. Survey: automatic generation of attack trees and attack graphs. *Computers & Security*, 2023. <https://doi.org/10.48550/arXiv.2302.14479>.
- [10] Kijong Koo, Daesung Moon, Jun-Ho Huh, Se-Hoon Jung, and Hansung Lee. Attack graph generation with machine learning for network security. *Electronics*, 2022.
- [11] Harjinder Singh Lallie, Kurt Debattista, and Jay Bal. A review of attack graph and attack tree visual syntax in cyber security. *Computer Science Review*, 2020.
- [12] Cynthia Phillips and Laura Painton Swiler. A graph-based system for network-vulnerability analysis. In *Proceedings of the 1998 Workshop on New Security Paradigms, NSPW '98*, New York, NY, USA, 1998. Association for Computing Machinery.

- [13] Sophie Pinchinat, François Schwarzenruber, and Sébastien Le Cong. Library-based attack tree synthesis. In *Graphical Models for Security*. Springer International Publishing, 2020.
- [14] Bruce Schneier. Attack trees. *Dr. Dobbs's Journal*, 1999.
- [15] O. Sheyner, J. Haines, S. Jha, R. Lippmann, and J.M. Wing. Automated generation and analysis of attack graphs. In *Proceedings 2002 IEEE Symposium on Security and Privacy*. IEEE, 2002.
- [16] Oleg Sheyner and Jeannette Wing. Tools for generating and analyzing attack graphs. In *International Symposium on Formal Methods for Components and Objects*. Springer, 2003.
- [17] L.P. Swiler, C. Phillips, D. Ellis, and S. Chakerian. Computer-attack graph generation tool. In *Proceedings DARPA Information Survivability Conference and Exposition II, DISCEX'01*, 2001.

Appendix

Declaration

For this paper, I used DeepL, Grammarly, ChatGPT, and Claude Sonnet. DeepL was used for translating and paraphrasing. Grammarly was used to make grammar and spelling corrections. I used ChatGPT and Claude Sonnet to avoid repeating words and to rework sentences for better flow and clarity.