Simulation-based Evaluation of Dynamic Attack and Defense in Computer Networks

Dissertation
zur Erlangung des Grades eines
Doktors der Naturwissenschaften (Dr. rer. nat.)
am Fachbereich Mathematik und Informatik
der Freien Universität Berlin

vorgelegt von

Alexander Bajic

Berlin 2021

Erstgutachter:	Prof. Dr. Marian Margraf
Zweitgutachter:	Prof. Dr. Florian Tschorsch
Tag der Disputation:	28.01.2022

Selbstständigkeitserklärung

Name: Bajic
Vorname: Alexander

Ich erkläre gegenüber der Freien Universität Berlin, dass ich die vorliegende Dissertation selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe. Die vorliegende Arbeit ist frei von Plagiaten. Alle Ausführungen, die wörtlich oder inhaltlich aus anderen Schriften entnommen sind, habe ich als solche kenntlich gemacht. Diese Dissertation wurde in gleicher oder ähnlicher Form noch in keinem früheren Promotionsverfahren eingereicht.

Mit einer Prüfung meiner Arbeit durch ein Plagiatsprüfungsprogramm erkläre ich mich einverstanden.

Datum: _____ Unterschrift: _____

Abstract

Research and development on IT security yields many different techniques and concepts to thwart off attackers in computer systems and networks, all of which come with different advantages and disadvantages. Choosing a capable solution that fits one's needs is a crucial, but also very challenging task. This is especially true for new emerging security paradigms where empirical data from the real world are still missing.

One of these IT security paradigms is *Moving Target Defense* (*MTD*). Through repeatedly altering a system's configuration and/or appearance, *MTD* intends to divert attackers, inhibit reconnaissance and increase effort of attacks. Many publications have emerged from research in this field, suggesting new techniques or providing practical implementations. However, it is not ultimately clear how effective these *Moving Target Defense* techniques actually are, and how they compare to each other or conventional defense mechanisms. Previous research mainly focused on the evaluation of individual techniques or on the comparison of at most two or three of them in very limited theoretical scenarios. What has been lacking so far is a flexible framework for assessing and fairly comparing the effectiveness of different defense techniques in diverse, and most importantly, realistic scenarios. This would assist researchers in benchmarking proposed techniques, and practitioners in selecting appropriate defenses that fit their scenarios.

To solve this problem, defense techniques and the attacks they are supposed to protect against need to be assessed under realistic conditions to be able to deliver meaningful results in the first place. Furthermore, such evaluation must not be limited to defenses arising from *MTD* or any other emerging paradigm for that matter, but be applicable to traditional and established defenses alike, to allow for a fair comparison. To this end, an attack simulation-based evaluation framework is proposed that — based on detailed modeling — is able to compare different types of defenses under realistic conditions to produce meaningful results on defense effects.

Case studies conducted with this framework, where different *Moving Target Defenses* are evaluated in a realistically modeled corporate network, reveal interesting and novel findings. Despite high-ranked publications saying otherwise, one of the most frequently suggested *MTD* techniques, virtual machine (VM) migration, may in fact have a negative effect on security. However, observed defense effects vary depending on details of the environment they are assessed in, implying that generalizing on the basis of incidental evidence is not advisable. Consequently, for evaluation to be fair and findings to be meaningful, detailed and realistic modeling is not sufficient but evaluation in diverse settings is equally important. To account for this newly determined requirement, the framework is extended with functionality allowing for the automated generation and diversification of realistic benchmark networks. Using this feature, simulation can automatically be scaled with ease, painting a more fine-grained picture of defense effects and

their distribution. Analysis of simulation results obtained from 500 of such benchmark networks not only confirms findings of the first case study but also reveals additional effects of the employed defenses, thus further emphasizing the need to base evaluation on a broad range of diverse networks.

Apart from defense evaluation, anonymous and dynamic routing is inspected more closely as one form of *Moving Target Defense*. Despite being regarded as a promising approach, respective techniques are yet underrepresented in *MTD* research. To this end, suggested network-layer anonymity protocols for application in the internet are investigated and shortcomings identified to subsequently propose an improved anonymous and dynamic routing protocol that can generally be applied in the context of closed corporate networks and the internet alike.

Acknowledgement

I can only imagine, how this endeavor would have turned out, if it was not for the support of the many people I met along the way, who helped me ask the right questions, challenged my ideas, and provided guidance in phases of doubt and uncertainty. First and foremost, I have to thank Dr. Georg T. Becker, who could have had an easier life if he had not taken on the role of a post-doc teaching me the do's and don'ts of good research and writing papers. Without him, I would still be sitting there, tweaking my first paper, thinking it was not good enough to be published. Fortunately, I was wrong, and even though the first published article was only an initial step, it gave me confidence in what I was doing — Thank you, Georg, for all the invaluable ideas, discussions, and for clearing all my reoccurring doubts and occasional frustration.

Furthermore, I must express my sincere gratitude to Prof. Dr. Marian Margraf who supervised this work and contributed to its quality through feedback from many fruitful and supportive discussions that helped me getting new perspectives and encouraged me to proceed. The constructive and generally positive nature of this relationship had a considerable impact on my motivation. Similarly, I want to thank Prof. Dr. Florian Tschorsch for the valuable discussions and feedback, turning my attention to aspects I would have missed otherwise, thus increasing the quality of this thesis. Each of our conversations sparked insights and ideas.

Who should not go unnoticed in this context are the people from the *Digital Society Institute* at *ESMT Berlin*, with whom I spent four amazing years. In particular, I want to thank Dr. Sandro Gaycken and Martin Schallbruch for creating an environment that enabled me to pursue professional and academic goals, but also Dr. Marco W. Soijer who was considerably involved in initiating the overarching project from the industry side and shaping it to be conducted on a scientific level[1].

Also, I must thank Dr. Pawel Swierczynski and Nicholas de Laczkovich for proofreading parts of this work and helping me to find the right words to improve its linguistic correctness as well as readability, and – together with Tobias Sudbrock – for providing moral support on so many occasions and just listening.

Last but not least, I want to thank my wife Katja for her continuous support and encouragement throughout the last years. Pursuing a PhD, at least for me, was more than a nine-to-five job, requiring long working hours and busy weekends, from time to time, that ultimately affected her everyday life, as well.

[1] The work presented in this thesis has been conducted as part of a collaborative research project funded by Cyber Works AG and, later on, Rheinmetall AG.

Contents

List of Figures

List of Tables

List of Listings

List of Abbreviations

AD	Active Directory
AES	Advanced Encryption Standard
AI	Artificial Intelligence
APT	Advanced Persistent Threat
ARP	Address Resolution Protocol
AS	Autonomous System
ASLR	Address Space Layout Randomization
AST	Abstract Syntax Tree
BGP	Border Gateway Protocol
CAIDA	Center for Applied Internet Data Analysis
CAN	Controller Area Network
CBC	Cipher Block Chaining
CDF	Cumulative Distribution Function
CEO	Chief Executive Officer
CFG	Context-free Grammar
CGI	Common Gateway Interface
CPRNG	Cryptographic Pseudo Random Number Generator
CPS	Cyber-Physical Systems
CPU	Central Processing Unit
CRM	Customer Relationship Management
CSV	Comma-separated Values
CVE	Common Vulnerabilities and Exposures
CVSS	Common Vulnerability Scoring System
DDoS	Distributed Denial of Service
DMZ	Demilitarized Zone
DNS	Domain Name System
DoS	Denial of Service
DSR	Data Space Randomization
EBNF	Extended Backus-Naur Form
ECDH	Elliptic-curve Diffie-Hellman
GCM	Galois/Counter Mode

HARM	Hierarchical Attack Relationship Model
HDD	Hard Disk Drive
HR	Human Resources
IDS	Intrusion Detection System
IoT	Internet of Things
IP	Internet Protocol
IR	Intermediate Representation
ISO	International Organization for Standardization
IT	Information Technology
IV	Initialization Vector
JSON	JavaScript Object Notation
KPI	Key Performance Indicator
LAN	Local Area Network
LAP	Lightweight Anonymity and Privacy
MAC	Message Authentication Code
MMU	Memory Management Unit
MS	Microsoft
MTD	Moving Target Defense
NASR	Network Address Space Randomization
NLP	Natural Language Processing
NOP	No Operation
OS	Operating System
OSI	Open Systems Interconnection
PC	Personal Computer
PEG	Parsing Expression Grammar
PHP	PHP: Hypertext Preprocessor
RAM	Random Access Memory
RCE	Remote Code Execution
RDP	Remote Desktop Protocol
SCNR	System Call Number Randomization
SDN	Software Defined Networking
SMB	Server Message Block
SME	Small and medium-sized Enterprises
SQL	Structured Query Language

SSD	Solid-state Drive
SSH	Secure Shell
UI	User Interface
UML	Unified Modeling Language
USB	Universal Serial Bus
VM	Virtual Machine
VSS	Variable-size Segments
WAN	Wide Area Network
XML	Extensible Markup Language
XOR	Exclusive Or

1. Introduction

Information technology (IT) plays an important role in modern society, be it for private purposes, the public sector, or businesses. Communication with friends, relatives, but also colleagues, nowadays mostly takes place via internet-based platforms. Banking is done online, and also transactions between banks are processed digitally in near real time. Even our most critical infrastructure, such as water and electricity supply, is highly dependent on IT and is becoming increasingly networked to align and automate processes. Simply put, modern society is built upon the advantages and conveniences we derive from the use of IT and its networking — but there are also risks involved. For as long as IT has existed, adversaries with diverse motives have tried to compromise it. Depending on the goal of such attacks, the consequences may range from disruptive and unpleasant to dangerous and life-threatening. *WannaCry* and *NotPetya* have shown that entire industries can be affected, even if these attacks were not necessarily targeted against specific companies. The infection of nuclear power plants and uranium enrichment plants with *Stuxnet* has made clear that even air-gapped systems are threatened when dealing with a sufficiently advanced and motivated attacker. Similarly, the case reveals that attacks of this kind can assume life-threatening dimensions.

Hence, there have always been efforts to protect IT systems. Various solutions have been proposed and implemented to counteract adversaries in different stages of an attack that, according to the frequently cited [37, 45, 46, 112, 118, 134, 140] *Cyber Kill Chain*® from Lockheed Martin [82], can be divided into seven phases ranging from *reconnaissance*, via *weaponization*, *delivery*, *exploitation*, *installation*, and *command & control*, to *actions on objectives*. While this subsumption is not authoritative, it illustrates that successful attacks are not singular events but the results of numerous intermediate steps, or phases for that matter. Consequently, effective defense is not only about keeping adversaries from completing their attacks, but disrupting them during any of these steps, starting with impeding reconnaissance needed to prepare attacks in the first place. Among these existing solutions are firewalls to prevent unauthorized access to networks or network segments, intrusion detection systems to detect such unauthorized access, and anti-virus software to protect endpoints, representing commonly known and well-established countermeasures. However, in the face of ever-growing threats to networks and systems, new concepts and paradigms emerge that intend to improve security and go beyond established approaches. One of these paradigms is *Moving Target Defense* (*MTD*), which intends to deter the attacker by repeatedly changing parameters of systems and networks to invalidate knowledge acquired during reconnaissance, thus impeding attacks. The term *Moving Target Defense* was coined around the year 2009 and reached considerable popularity since then, as is indicated by the continuously in-

creasing number of related publications per year[1]. Yet, the idea of active defense and deception is not entirely new and also not exclusive to *MTD*. These concepts have been used before, with some of the work dating back to a time long before the advent of *Moving Target Defense*. Some recent publications that qualify as *MTD* refrain from using this label but classify themselves as *Cyber Deception*. Mechanisms suggested in this context do not aim at invalidating gained knowledge but increase the difficulty of determining useful information by inserting dummy traffic, for example.

Irrespective of how emerging paradigms and proposed defenses are labeled, research and development to improve the security of computer systems and networks have been recognized as important topics. However, to employ such newly proposed techniques in a useful way, their impact on security and how they compare to existing defense schemes must be understood.

1.1. Motivation and Problem Statement

Research on new concepts and techniques to secure networks and systems, as well as their subsequent development are indispensable to keep up with sophisticated attackers. Yet, while suggested approaches regularly come with promising concepts and tailor-made scenarios to support their utility, the question on how to quantify and compare their impact on security often remains unanswered. This, however, is relevant to practitioners and researchers alike. The former need to make informed decisions about which defenses to employ in a given scenario to maximize security at reasonable costs. The latter, in turn, who develop such techniques, need to confirm that intended and achieved effects match, or compare the performance of their approach to that of existing solutions.

While many systems and defense techniques can be analyzed using static methods such as well-established attack graphs and trees, this proves to be difficult for *MTD* techniques for the dynamically occurring state changes caused by the defender actions. In the course of performing a single defense action, the corresponding attack graph may change drastically and would need to be renewed. Furthermore, the time instances at which these defenses are triggered may neither be random nor predefined, but can also be dependent on the output of an intrusion detection system (IDS). Generating a graph that considers all these possibilities, even for small systems, would quickly result in excessive numbers of possible states and transitions between them [123], thus becoming unmanageable and computationally expensive, if feasible at all. What is more, an attack (and defense) graph like this would also not serve one of its primary purposes, that is providing a comprehensive overview.

To this end, academic research not only produced a considerable amount of publications that suggest new defensive techniques, but also such that address the problem of defense evaluation, with approaches ranging from mathematical formalization to real testbeds. Existing work in this field strongly focuses on answering the question to what extent a specific defense effect improves security. Maleki *et al.* [89], for example, in-

[1] Specific numbers on the development of annual publications related to *Moving Target Defense* can be retrieved through Google Scholar, for example.

vestigate how an attacker's success probability changes if she cannot rely on previous reconnaissance but has to probe IP addresses over and over again in order to compromise targets, as required addresses are repeatedly changed by a respective defensive technique. Based on Markov modeling, the authors show that at sufficiently high frequencies, and especially in the case of multi-target attacks that require knowledge of multiple targets' addresses at the same time, attacker success is limited since maintaining correct knowledge on numerous addresses is very unlikely in the presence of repeated IP address randomization. However, the requirement that addresses of all involved targets must be known at the same time to perform a successful multi-target attack stems from the assumed effect that the shuffling of a target's IP address does in fact revoke the attacker's winning state on said target, which may be questioned. Another interesting question not being asked is that of other potential effects, including those that are detrimental to the defender or legitimate user of the system, such as service disruption or even security degradation. While the existence of such negative effects remains to be shown, for some *MTD* techniques, arguments supporting this idea are just as convincing as those used to convey positive effects. As Chapter 3 will further elaborate, defense effects are frequently assumed rather than tested in existing research on *Moving Target Defense* evaluation, with potentially negative effects receiving only little attention. This is partly motivated by the claim to keep models simple, which is plausible for analytical approaches to evaluation that cannot handle complex state representations. Yet, simulation-based approaches regularly strive for the same goal as to improve efficiency. However, this poses a problem, because for evaluation to deliver meaningful insights, it must be based on a defense's actual effects, including those that have negative implications. Otherwise there is a risk that obtained results simply cannot provide a realistic impression of a defense's real-world performance. Yet, this requires higher levels of detail and complexity. Testbeds, in turn, that provide the ultimate level of detail come with relatively high costs, and require considerable effort in maintenance and setup for relying on operational implementations of techniques.

What is more, to allow for a fair comparison of defenses, uniform metrics are needed that are applicable across different techniques and, most importantly, do not prefer one over the other. In that sense, deriving an attacker's success chance from guessing entropy of valid IP addresses, as in the previous example, will only insufficiently account for the security impact of defenses whose effects comprise other factors such as connectivity to a migrated VM within the network, for example, that are not subject to successful guessing or probing. Consequently, a fair and realistic approach to comparative evaluation must employ generic metrics that are able to capture various effects. Given that a large proportion of existing approaches to evaluating *MTD* regularly focuses on investigating individual techniques rather than comparing utility of different ones, this requirement has been addresses only rarely.

Considering the above, the questions arise whether there is an efficient way to reveal effects rather than assume them, how to quantify their impact on security, and how to fairly compare findings. To answer these questions, a coherent evaluation framework is presented that addresses the challenges of determining the actual effects of defenses, account for potential security improvement as well as degradation, and provide met-

rics capable of fairly comparing defense performance across numerous techniques and hundreds of independent investigations. Based on detailed modeling and simulation, this framework can incorporate well-known and established static defenses, as well as dynamic defenses arising from paradigms such as *MTD* and *Cyber Deception*. At the same time, it allows to represent differently skilled adversaries equipped with capabilities to perform both real and hypothetical attacks to simulate realistic threats and what-if scenarios alike, and track their progress in the presence of different defenses. Using a sophisticated modeling language, attacks, defenses, and legitimate actions, as well as the scenarios they operate in can easily be modeled with a high level of detail, so that ultimate effects of defenses need not be defined, but can be observed as a result of the involved actions' underlying mechanisms.

1.2. Contribution

The presented work makes several contributions to the state of evaluating *Moving Target Defenses* and potential *MTD* techniques themselves which are summarized as follows:

- **Modeling and simulation framework for realistic evaluation:** A modeling and simulation framework is presented that surpasses existing approaches to defense evaluation with regard to both accuracy and significance. By employing a flexible modeling language, systems and actors can be modeled at an arbitrarily high level of detail, allowing for a more realistic simulation and meaningful results. Furthermore, the framework can incorporate any number of attackers and defenders, so that both static and active defenses can be evaluated, making the framework more versatile. Defense-independent metrics ensure that attacker progress is measured irrespective of employed defense techniques, thus allowing for quantitative comparison across simulations. At the same time, detailed transaction logs allow for qualitative analysis, providing a better understanding of cause and effect of different actions. Parts of this framework have been published in a paper [19] presented at the *Nordic Conference on Secure IT Systems 2018* that was recognized with the Best Paper Award.

- **Revealing defense-induced security degradation:** With help of the aforementioned framework and a high-detail model, this work is the first to reveal that VM migration, one of the most frequently proposed *MTD* techniques, may in fact have a negative impact on security. Observed security degradation is related to environmental factors that have been ignored by other approaches so far. This is not only an interesting new finding, but also contradicts the numerous high-ranked publications [3, 46, 61, 64] that based their conclusions on strong assumptions, low-detail modeling, and, most importantly, single examples that have been specifically constructed and do not consider the environment a defense operates in. This experiment and its findings have also been part of the aforementioned publication [19] presented at the *Nordic Conference on Secure IT Systems 2018*.

- **Benchmark network fuzzing for automated scenario diversification:** Another major shortcoming of existing approaches to evaluation is the tendency to generalize on the basis of single experiments providing incidental evidence of a defense's effect. To not fall victim to the same mistake, the framework is extended with an automated diversification algorithm that is referred to as benchmark fuzzing. Based on this, arbitrary numbers of diverse high-detail benchmark networks can be generated automatically, instead of modeled manually. Using this mechanism, the proposed simulation framework may not only employ a considerably higher level of detail than other approaches, but also significantly scale the sample size of investigated scenarios which, in existing work, is typically one. This benchmark fuzzing is done to generate 500 networks, providing insights into a wide spectrum of defense effects, their frequency of occurrence, and environmental conditions they are related to. Being the first to show that the effects of specific defenses are not solely positive or negative but span a scale with results depending on individual factors, this work supports practitioners and researchers alike in selecting and developing adequate defenses on the basis of data. Furthermore, metrics are introduced that aggregate such amounts of data resulting from simulation at a larger scale, thus allowing for a comprehensive overview of defense performance. A related article [21] has been published in *Springer International Journal of Information Security (2021)*.

- **Qualitative assessment of *Moving Target Defenses*:** Apart from the defenses that are investigated experimentally, this work presents a critical assessment of numerous *Moving Target Defenses* and the analogies used to promote them. Using the *MTD*-supporting work's arguments and narratives against itself, negative effects as well as ineffectiveness of supposedly useful *MTD* techniques are well-reasoned on a solely logical level. Considering that such effects have not been identified so far, the presented work continues to point out why individual evaluation approaches fail to reveal potential security degradation or sheer irrelevance of movement. A related publication [20] has been presented at the *Mal-IoT Workshop, co-located with the ACM International Conference on Computing Frontiers 2020*.

- **Anonymous and dynamic routing-enabled *Moving Target Defense*:** Considering anonymous and dynamic routing which is slowly gaining traction in the *MTD* community, this work presents novel attacks on PHI, an anonymity protocol for the network layer representing the current state of research in this domain. Furthermore, an improved protocol is presented. This protocol, dPHI, is designed to operate in networks at the scale of the internet, yet remains generally applicable to smaller corporate networks, thus potentially useful for the purpose of *MTD*. It overcomes limitations of the protocols it is based on, yielding higher anonymity with comparable, partly even better, performance. Introduced attacks and the improved dPHI protocol have been published [18] in the *Proceedings on Privacy Enhancing Technologies Symposium 2020, Issue 3*.

1.3. Thesis Structure

The following Chapter 2 provides background information on topics that are relevant to this work. This includes an introduction to the concept of *Moving Target Defense*, as well as an overview of techniques that have been proposed in this context. Furthermore, fundamental concepts of logic programming in Prolog are outlined for being integral to the proposed evaluation framework. Afterwards, Chapter 3 presents a critical view on *Moving Target Defense*, the analogies used to promote it, and existing approaches to *MTD* evaluation. In this course, argumentative disparities related to proposed techniques are revealed, none of which are accounted for in existing approaches to evaluation. Chapter 4 focuses on the proposed framework, starting with collecting requirements that ought to be fulfilled in order to allow for realistic evaluation. Subsequently, relevant terms in the context of this framework will be presented, followed by an overview of the framework itself. Afterwards, its components and the way they interact are explained in detail, divided into its two major building blocks, the modeling language and the simulation engine.

Having understood how the framework operates, an experiment is presented in Chapter 5, illustrating that the framework is indeed capable of generating novel insights. Based on two variations of the same network and employing differently skilled attackers, the performance of popular *MTD* techniques is investigated, revealing that these may also have security degrading effects. While these findings corroborate the framework's capability to yield meaningful results, they also emphasize the need for testing in numerous diverse scenarios to fully understand a defense technique's impact. Consequently, Chapter 6 introduces extensions to the framework that concern capabilities of the modeling language and provide usable means to automate the generation of large numbers of unique, yet realistic scenarios. Using this feature in combination with an extended scenario definition, large-scale simulation is conducted. Generated results not only deliver incidental evidence on defense effects but a broad spectrum of possible outcomes and how these are distributed.

Chapter 7 makes a shift away from evaluation and instead focuses on anonymous and dynamic routing as a potential form of *Moving Target Defense*. First, existing protocols are presented, one of which is investigated in more detail. Critical errors that enable de-anonymizing attacks are revealed and elaborated to subsequently propose solutions. Based on this, an improved protocol is presented that is tested against the previously identified attacks, as well as additional ones that result from an extended threat scenario. The chapter ends with a quantitative analysis of the proposed protocol with regard to anonymity and performance.

Finally, a discussion follows in Chapter 8. First, findings will be summarized with regard to insights obtained from experiments, as well as the general research question. Based on this, related work is discussed that is closer to the scope of the framework and its findings to allow for comparison. This extends the broader existing work discussed in the course of Chapter 3, where limitations throughout the spectrum of evaluation schemes have been addressed. Afterwards, the work is concluded before presenting an outlook on potential future work.

2. Background

This chapter provides an introduction to the *Moving Target Defense* paradigm and presents related *MTD* techniques that have emerged from research and development in this field. Furthermore, fundamentals of the Prolog programming language will be introduced. These are relevant to the operations of the simulator that will be covered later.

2.1. The Moving Target Defense Paradigm

Many of the active defenses proposed throughout the last years, qualify as *Moving Target Defense*s or have specifically been presented as such. Consequently, *MTD* is of certain relevance in the scope of this work that intends to evaluate such active defenses. The following sections provide an explanation and definition of the concept itself and introduce specific defenses that have been suggested, grouped along the general domains they operate in.

2.1.1. Explanation and Definition

Moving Target Defense is an IT security paradigm that advocates proactive defense. Through mutating a system's appearance, knowledge that an attacker may have previously acquired is repeatedly invalidated. In consequence, attacks that rely on such outdated knowledge may fail, forcing the attacker to redo reconnaissance over and over again. This, in turn, is not only time-consuming and thus costly, but also risky as it increases chances of being detected for leaving traces and potentially triggering alerts. Such proactive defense behavior is opposed to classical concepts of IT security that are mostly static in nature. These comprise network segmentation, firewalling, intrusion detection or end-point security, for example, all of which are intended to impede or defend attacks, yet do not repeatedly change the system's appearance, thus allowing an attacker to take her time to collect information and prepare the next step.

In the context of IT Security, the term *Moving Target Defense* is being used at least since the year 2009 and has been widely adopted by academia and industry by now. So far, there is no official definition of *MTD*. However, the one provided by the US Department of Homeland Security in 2011 appears to be generally accepted (originally from [66]):

> *Moving Target Defense (MTD) is the concept of controlling change across multiple system dimensions in order to increase uncertainty and apparent complexity for attackers, reduce their window of opportunity and increase the costs of their probing and attack efforts.*

The notion of this definition can be found throughout literature, where similar phrasings are used whenever research on *Moving Target Defense* is being motivated, implying that there is a common understanding. A term frequently used in this context is *attack surface*, representing the part of a system that is subject to reconnaissance and potential attacks by an adversary. The effectiveness of *MTD* techniques is often argued with a shifting and potential reduction of attack surface. Manadhata and Wing [90] even propose using it as a metric to determine the (likely) security of a software product by measuring the ways an attacker can interact with the software that had formerly lead to exploits. Simply put, the idea is to limit the number of attack vectors to reduce the adversary's opportunities, thus lowering the chances of successful attacks. Yet, *MTD* does not necessarily aim at decreasing the attack surface, but obscure and hide it from the adversary by constantly changing it. To describe this goal, Zhuang*et al.* [149] introduced the term *exploration surface*. Exploration surface is the space in which the attacker believes the attack surface to be. Consequently, the goal of the attacker is to find (parts) of the attack surface within the exploration surface to launch the attack. In that sense, *MTD* intends to maximize exploration surface while keeping the attacker from decreasing it through reconnaissance. Further work that is dedicated to establishing a theoretical foundation of *MTD* has been presented by Zhuang *et al.* [148, 146] and Green *et al.* [59] among others.

2.1.2. Defense Techniques

The presented definition is very general in that it merely requires controlled change intended to impede attackers for a technique to be classified as *Moving Target Defense*. How and where this change happens is intentionally left open, resulting in a plethora of techniques that operate on different levels of IT systems. And despite the lack of any formal scheme to categorize *MTD* techniques, differentiation along the domains they operate in, is common practice, thus bringing some order to the otherwise confusing quantity of techniques. In their technical report on *Moving Target Defense* from 2013, as well as a compact survey from 2014, Okhravi *et al.* [100, 101] not only presented the state of research on *Moving Target Defense* at that time, but also compiled a list of categories, representing these domains. They differentiate between techniques that add dynamics (i.e., movement or repeated change) to networks, platforms, runtime environments, software, and data. The same categories are still in use in the aforementioned technical report's second edition presented by Ward *et al.* [131] that was published in 2018. A short description of the scope of these categories is given in the following:

- **Network-based** *MTD* techniques consider the shape and appearance of networks, intended to increase the difficulty of network-borne attacks through impeding reconnaissance. Respective adaptation may require configuration of the hosts in a network, but may also be deployed within the network infrastructure and operate transparently to the connected nodes. Utilizing features from software defined networking, some of the *MTD* techniques proposed in this category exhibit relative maturity and are frequently supported by prototypical implementations.

- **Platform-based** defenses consider specifics of the computing platform and other low-level resources that services run and rely on, as to impede attacks specifically going for related vulnerabilities. This usually comprises the layers up until the operating system and includes characteristics of employed hardware, but also hypervisors as in the case of VMs, for example.

- **Runtime environment-based** defenses comprise techniques that affect the behavior or appearance of the direct environment that services and applications operate in. In most cases, this environment is provided by respective operating systems, so that corresponding defenses affect the way resources are made available and accessed or interaction is handled.

- **Application-based** techniques directly affect a given application's shape and/or how it processes data irrespective of the system that the application runs on. Attacks that are supposed to be fended off by defenses in this domain are those relying on knowledge of the application's inner structure, for example.

- **Data-based** defenses address the randomization or obfuscation of data that are processed by applications. This serves to mitigate attacks that rely on malformed messages or inputs which aim at provoking unintended behavior or even exploiting vulnerabilities.

These categories are neither unified nor universally accepted, yet, irrespective of deviations in wording, can repeatedly be found throughout *MTD* literature.

To give an impression of the topic's breadth, various *MTD* techniques, grouped along the aforementioned categories are presented in the following. In case of ambiguity with regard to appropriate category selection or whenever a proposed technique simply affects multiple domains, remarks are made accordingly. Considering the amount of published work in this field, the list of techniques presented here can not be exhaustive. However, the selection was composed as to include those techniques that may be considered particularly relevant for their frequent occurrence in related research. For further information on *MTD* techniques, the attacks they are supposed to defend against, the interested reader is referred to the recent survey from Cho *et al.* [39] as well as the aforementioned work from Ward *et al.* [131].

Network-based Defenses

A frequently suggested network-based defense is the randomization of addresses and ports of communicating entities in networks. The intention behind this is to invalidate learned addresses in short intervals so that subsequent attacks that rely on such information ultimately fail. In addition, the shuffling of addresses obfuscates the real size of a network by confronting the attacker with numerous addresses when eavesdropping on traffic or actively probing. The general mechanism is frequently referred to as Network Address Space Randomization (NASR) and has been proposed in many different forms, some of which even go beyond the IP protocol. The earliest proposal of this kind dates

back to 2001, that is before the advent of *MTD*, suggesting the schedule-based reassignment of IPv4 addresses and was presented by Kewley *et al.* [80]. Their approach was not only a theoretical proposal but comprised a prototypical implementation. To further increase the attacker's uncertainty, approaches such as MT6D from Dunlop *et al.* [53] resort to employing IPv6, sourcing potential addresses from an even larger pool thus making guessing but also scanning significantly harder. More recent work took up on this topic and suggested similar schemes, yet benefiting from technological progress. In this regard, Chang *et al.* [34], and Narantuya *et al.* [97] implemented their IP randomization schemes (short: IP-Shuffling) with help of SDN. The approach and prototypical implementation of MacFarland *et al.* [88] also relies on SDN, yet goes so far as to also shuffle MAC addresses. Interestingly, they classify their approach as a host-based defense since adaptation does not happen "within" the network, but on each and every host. Generally speaking, proposals and implementations of NASR are comparatively mature as the work from Li *et al.* [86] and Yackoski *et al.* [138, 139] illustrates, which turned into a commercial product [47]. In recent years, this concept has also been suggested to improve IoT security. The MT6D-inspired approach from Zeitz *et al.* [141], and the one from Nizzi *et al.* [99] specifically address NASR in the context of IoT, considering limitations such as constrained power consumption, storage, as well as computational resources, which are common in the IoT context. Yet, as mentioned before, the scope of randomizing addresses of communicating entities has been extended since, going beyond application in IP networks. The proposal from Woo *et al.* [134] transfers the concept to CAN bus IDs in order to protect against spoofing. However, their approach to switching addresses after each message is not random but based on splitting the 27-bit ID into a 5-bit fixed ID for all 15 devices in the network and then use the remaining 22 bits to create an unpredictable ID so that attackers cannot simply spoof messages.

Other proposals advocate the migration of virtual machines (VMs) across different hypervisors in order to move critical services out of the attacker's firing line. This migration can either be done in a state-preserving or state-resetting fashion. The state-preserving type is what is referred to as live migration in the context of this work and suggests that VMs are moved during runtime as to keep respective services operational and preserve information of currently processed tasks. In contrast, the state-resetting type is referred to as cold migration and advocates VMs to be started from scratch, once deployed at their new location. Depending on whether this is done through effectively copying virtual HDDs to the target as advocated by Wang *et al.* [129], or by booting up duplicate VMs that are maintained at different locations, this type of migration may, in the latter case, fend off adversaries that reached persistence at the former location. The Mayflies framework from Ahmed and Bhargava [5, 6], as well as the scheme proposed by Zhuang *et al.* [150] pursue such an approach. However, the framework from Ahmed and Bhargava also considers other adaptations that will be covered in the context of platform-based defenses. Debroy *et al.* [50] suggest migration specifically for the purpose of mitigating distributed denial of service (DDoS) attacks. Their intention is to proactively, and in the worst case reactively, migrate applications to other presumably secure locations, which is why they refer to their approach as application migration. However, what they actually do is migrating VM snapshots, thus, from a technical per-

spective, implementing a form of VM live migration even though this serves to relocate the contained target application. Furthermore, the approach also stipulates to inject dummy traffic, once a VM under attack has been migrated to deceive attackers into thinking that their attack is still in progress. The authors explain how this mechanism is supposed to work from a rather high-level perspective, but do not provide technical details on how this scheme may be implemented. Interestingly, publications that are concerned with VM migration more often discuss how to efficiently and effectively employ it in defense strategies rather than suggest practical implementations.

Approaches that also involve migration, yet not for the sake of defending migrated nodes, have been proposed by Venkatesan *et al.* [127] and Almohri *et al.* [11]. The former propose to proactively relocate network probes that monitor passing traffic in order to increase chances of detecting intruders. The suggested placement and rotation scheme takes the respective network's hierarchy into consideration as to determine locations where attackers will most likely pass during lateral movement. The latter suggest repeated relocation of honeypots and decoy VMs to deceive attackers during reconnaissance, increase their risk of being detected, and detract them from targeting critical machines and services.

Another form of network-based defense or deception emerges from dynamic and/or anonymous routing. The approach from Hong *et al.* [65] suggests an SDN-enabled scheme that implements dynamic routing through continuously changing the network topology, thus affecting which paths are chosen for routing communication flows. Depending on the type of attacker and her location within the network, this may impede reconnaissance and also avert attacks that rely on access to communication flows. Another SDN-based approach has been suggested by Skowyra *et al.* [115] and intends to not only add dynamics to routing but also anonymize communication by exchanging identifiers in packet headers on the different layers of the network stack. To accommodate this, proxies are located in front of communicating entities to transparently apply changes to outgoing and incoming traffic. These proxies, in turn, cooperate with a trusted SDN controller that installs routes between hosts based on exchanged identifiers. In 2003, long before the rise of SDN, Touch *et al.* [121] presented DynaBone (i.e. dynamic backbone) that advocates the instantiation of several virtual overlay networks across which traffic should be multiplexed. While primarily intended to layer different DDoS attack mitigations to increase difficulty for attackers, these virtual networks also allow to implement dynamic routing which is handled by a dedicated *proactive/reactive multiplexer* (short: PRM). Revere [76] is a similar approach that also relies on overlay networks for establishing dynamic routes, yet for the sake of delivering security updates. Zhu *et al.* [145] assume a wireless context where communication is subject to jamming attacks and suggest that for routing to a given destination, source and intermediate nodes always pick two next hops, one for real and one for false communication flows. These false flows are inserted to confuse attackers, yet cannot be identified as such for using encryption. Not knowing which path the "real" communication is going, an adversary has to invest more resources to disrupt communication. While the scope of this approach is on wireless networks and jamming attacks, a similar scheme may be employed in wired networks that rely on source routing to defend against attackers who may reside on individual

paths. Compared to previously introduced approaches, network-based *MTD* techniques that are based on anonymous or dynamic routing are relatively scarce, despite their ability to impede reconnaissance or averting attacks. However, outside the *Moving Target Defense* paradigm, anonymous routing is a vivid and established research branch from which numerous suggested protocols have emerged. These vary with regard to performance, realized anonymity but also the scope in which they can be applied. To account for this topic's relevance, not only in the context of *MTD* but communication in general, Chapter 7 presents an excursus into this field. There, different approaches to anonymous routing are presented, one of which is discussed in detail for being applicable in both public networks such as the internet, as well as corporate networks without relying on one central trusted instance as was the case with the SDN-based approaches. Furthermore, the discussed protocol is analyzed with regard to information leakage that jeopardizes claimed anonymity to subsequently propose an improved protocol resolving identified issues.

Platform-based Defenses

From a technical perspective, aforementioned proposals to implement VM migration can also be considered as platform-based defenses since re-location to another hypervisor may come along with changes in hardware that affect the underlying computing platform. Considering that attacks such as Meltdown and Spectre are (partly) specific to the utilized CPU architecture and may be exploited in virtualized environments, it becomes apparent that VM migration may have practical implications in this regard. As a result, a defender may choose to migrate VMs across hypervisors with heterogeneous hardware configurations to avert such hardware-specific attacks. Another defense that addresses the computing platform has been proposed by Dai and Adegbija [48] and intends to deflect side-channel attacks on caches. To accommodate this, a so-called "tuner" repeatedly changes the configuration of runtime configurable cache at intervals that are shorter than the time needed by an attacker to reverse-engineer the cache's index mapping and probing for cache hits and misses.

However, according to the previously presented classification, platform-based defenses are not only concerned with mitigating attacks that are related to the underlying hardware but also operating systems. Aforementioned work from Ahmed and Bhargava [5, 6, 7] that suggested a form of VM migration also considers the adaptation of these with regard to utilized operating systems and related parameters. As a result, VMs that are re-instantiated elsewhere will provide the same service but may have changed from a Windows to a Linux platform, for example, forcing the adversary to perform reconnaissance all over again to prepare her attack. Considering that the scheme from Ahmed and Bhargava relies on spawning VMs from ready-made templates, quickly "exchanging" an OS is practicable. Thompson *et al.* [120] propose a similar scheme to overcome OS-inherent weaknesses. Their approach also relies on ready-made VMs for quick interchangeability of operating systems, yet does not advocate the relocation of VMs. With rotation intervals as short as 60 seconds, reconnaissance was hardly feasible anymore, according to the authors' findings.

Wahab *et al.* [3] specifically propose to migrate services to protect these against attackers. With help of their risk assessment algorithm, they identify risky VMs whose services should be migrated, as well as safe VMs that may serve as a migration target. However, the authors do not provide a practical implementation on how services are supposed to be migrated independently from the VMs they are located in. Instead, they rely on such features to exist in a cloud environment and conduct their experiments with help of CloudSim [32].

Runtime Environment-based Defenses

One of the most commonly known defensive mechanisms that concern the runtime environment is Address Space Layout Randomization [119] (short: ASLR) that was first released in the year 2000. Since then it has been widely accepted, further improved, and is now integral to most modern operating systems. What ASLR does is, simply put, randomizing the location of data in memory and providing virtual memory addresses to applications, thus obfuscating the real location of information or executable code. The intention of this obfuscation is to increase difficulty of exploiting buffer overflows and deflect attacks that rely on knowledge of the location of data in RAM such as *return-to-libc* attacks. Its wide acceptance and success[1], as well as the fact that it fits the general notion of *Moving Target Defense* lead to ASLR being frequently used to motivate the idea and utility of *MTD*.

Yet other defenses operating on this level are Instruction Set Randomization (ISR) [77, 104], and System Call Number Randomization (SCNR) [72] that intend to keep attackers from executing successfully injected code by inserting an additional abstraction layer for communication with the underlying system. The basic idea of ISR is to make the runtime environment incompatible with injected code so that instructions are simply not understood. To accomplish this, the approach from Kc *et al.* , for example, creates unique execution environments for each and every process, that rely on modified instructions to operate properly. Assuming that the attacker does not know how the execution environment translates randomized instruction sets to those of the underlying system, it is unlikely for attackers to successfully inject instructions that cause intended behavior. Yet, while such measures seem capable of impeding attacks by not causing the intended effect, the injection of illegal instructions may still crash the respective process. This may be acceptable in many cases. However, in the context of security critical systems, a Denial of Service (DoS) may be just as fatal as successful compromisation. Potteiger *et al.* [105, 106] consider the case of cyber-physical systems (CPS) in contexts where reliability and availability are at least as important as defending against attacks. So, as to account for this, they propose a scheme that does not only employ ISR but also detects successful code injection to reconfigure the respective controller accordingly to avert any sort of unintended behavior. System Call Number Randomization, on the other hand, only concerns the mapping of numbers to specific system calls that

[1] Despite recurring issues related to faulty implementations, as well as successful attacks like the one from Gras *et al.* [57] that is based on side-channeling the memory management unit (MMU), ASLR is considered to be an effective defense.

is usually static. Through randomizing these upon process instantiation and inserting an abstraction layer that takes care of de-randomizing call numbers during operations, subsequently injected code relying on such system calls will not match the randomization scheme and therefore not cause intended effects.

However, while ASLR operates transparently towards applications, this is not the case with SCNR and ISR. These mechanisms cannot be exclusively implemented in the underlying system (e.g. the kernel) that provides the runtime environment, but also require modifications to the application in order keep legitimate code operable, while impeding execution of injected code. The approach presented by Jiang *et al.* [72] solves this through intercepting process invocation to inspect code and replace respective system calls. Nevertheless, this requires further modifications to the kernel and imposes additional load.

Application-based Defenses

Making applications dynamic to implement some form of *Moving Target Defense* can be achieved in different ways. One branch of research in this field is concerned with introducing variety to application binaries and is frequently referred to as software diversity. In their work from 2011, Jackson *et al.* [71] propose compiler-based randomization. Given that functionality programmed in a high-level language can, on a low level, be realized in different ways, the authors suggest to have the compiler randomize the produced machine code. As a result, every binary will be different so that specifically crafted exploits relying on characteristics of a certain binary do not work on a large scale. In subsequent work, Jackson *et al.* [70] proposed software diversification solely based on the random insertion of NOPs. Wu *et al.* [135] specifically suggest software diversification based on existing binaries. To accomplish this, they propose to first derive LLVM[2] intermediate representations (IR) from said binaries. Afterwards, different diversifying operations are performed on these representations before, finally, building new binaries that implement the same functionality, yet exhibit a different internal structure. In turn, Taguinod *et al.* [117] take up on software diversification at another level. Instead of introducing randomness with help of the compiler or subsequent modification of binaries, they suggest the translation of source code to other programming languages to yield binaries that are inherently different. Obviously, the challenge of this approach lies in the sophistication of said translator to correctly process and translate instructions, especially those that may be unique to one particular language and do not have a counter part in the target language. A general overview of approaches to software diversity and attacks that are of particular relevance in this regard have been presented by Larsen *et al.* [83].

Other approaches to application-based *MTD* simply suggest to rely on different applications that serve the same purpose, yet do not exhibit the same weaknesses so that attackers cannot easily scale their attacks. Taguinod *et al.* [117] and Zhang *et al.* [142] propose to do so for databases and their corresponding servers. Being integral to most

[2]LLVM used to be an acronym, standing for Low-level Virtual Machine. However, in the course of evolving into an umbrella project for different compiler and toolchain technologies, the abbreviation was removed and LLVM became a term of its own.

web applications, databases are frequently targeted by so-called SQL injection attacks that, if successful, may result in compromisation of data or even the underlying systems. This is achieved by injecting queries in unintended ways that bypass input sanitation. By switching between different types of databases and database server applications that also come with different dialects, this injection is supposed to become harder as specific servers may not be vulnerable to certain exploits or simply not "understand" the injected query's dialect. Boyd and Keromytis [27] propose a somewhat different approach that also intends to prevent SQL injection attacks, yet resorts to exchanging instructions in the SQL-server's parser and the scripts that perform queries on the application's behalf. As a result, an attacker that manages to inject queries will not succeed unless learning the instruction mapping. This is similar to the previously introduced concept of ISR, yet transferred to the application level. However, applications may just as well be diversified through their configuration. Even though this does not affect binaries so that exploits targeting characteristics of the application's inner structure will still scale, it changes behavior and, on a higher level, appearance at runtime. This, in turn, may impact the application's attack surface and, consequently, the set of applicable attacks that an adversary may resort to. Corresponding approaches stem from research on evolutionary algorithms and have been suggested by John *et al.* [73], Lucas *et al.* [87], as well as Collado *et al.* [42, 113]. They propose to diversify web server configurations so that these vary in potentially security relevant details, yet, yield the same observable experience from the user's perspective. Furthermore, they investigate which of these configurations perform best with regard to number and severity of attacks to subsequently recombine individual parameters in order to derive more secure configurations.

Data-based Defenses

Defenses that are based on dynamic data are generally concerned with how data are represented when stored or in memory. In this regard, Bhatkar and Sekar [24] suggest a scheme they refer to as *Data Space Randomization* (DSR) that is intended to add another layer of security for data in memory, once an attacker managed to bypass defensive techniques such as ASLR and ISR. The main idea of this approach is to simply XOR data with an unknown mask, yielding high entropy (e.g. 2^{32} for integers and pointers on a 32-bit architecture) that cannot be efficiently bruteforced. A similar approach has been suggested by Cadar *et al.* [30]. Both schemes require availability of the corresponding application's source code to insert respective operations. However, Cadar *et al.* also compute equivalence classes for instruction operands to assign random bit masks for XOR-operations on a per class basis which incurs less overhead and may increase efficiency of their approach. In the presence of viable attacks on existing DSR schemes, Rajasekaran *et al.* [108] propose another approach to DSR. While, in principle, employing the same scheme of masking data, their framework repeatedly re-randomizes masks so that the shape of data continuously changes. From a technical perspective, these data randomizing techniques rely on symmetric encryption to protect data in RAM. And while neither of the approaches presented here has been proposed in the context of *Moving Target Defense*, they fit the general definition.

At this point, the ambiguity in classifying the individual techniques should be recalled. This specifically concerns approaches to defend against SQL injection attacks. These have been categorized as application-based defenses in this work for effectively being implemented in server applications and related scripts (e.g. PHP or CGI) to diversify their behavior with regard to processing queries. Interestingly, Ward *et al.* [131] categorized these techniques as defenses that rely on dynamic data rather than dynamic applications for protecting against potentially malicious user-generated input data. However, this appears inconsistent considering that most other techniques have been categorized with regard to what is being randomized.

2.2. Logic Programming with Prolog

The concept of logic programming and implementations thereof date back to the late 1960s and early 1970s, arising in the context of research on artificial intelligence (AI) and natural language processing [43] (NLP). One of the earliest publications in that regard, also introducing the clausal form as is common in Prolog, was submitted by Cordell Green [58] and relied on Lisp. Prolog itself first appeared in 1971 and had been developed by Alain Colmerauer, Philippe Roussel and Robert Kowalski [43, 81]. While it is definitely not the only representative of logic programming languages, it is one of the most popular ones. The Prolog distribution which is of particular interest in the context of this work for being integral to the framework presented in Chapter 4, is SWI-Prolog. It dates back to 1987, is actively maintained, and subject to ongoing development.

An important strength of Prolog is, simply put, that it allows to query information from given knowledge bases that is not the result of mere look-ups, but incorporates the evaluation of rules that are part of said knowledge bases. In consequence, results of such queries may reveal information that is not explicitly given but deduced from existing data and defined relations. To be fair, this can equally be achieved with means of imperative programming. However, fine-grained instructions would be required as to specify how inputs ought to be processed in order to produce the required output, instead of just querying for the desired information. While this is definitely not a suitable way to solve every kind of problem, it sure is to evaluate *what-if*-scenarios and determine the different actor's options, as is relevant for the simulation framework. Since Prolog's declarative nature makes it inherently different from commonly used imperative programming languages, providing a short introduction will help to understand the presented work.

In the following sections, some of Prolog's basic concepts are presented. These are not exhaustive and solely serve to better understand how the framework's simulation engine operates and lay the foundation for the low-level representation of scenario descriptions which are presented in Chapter 4.4.2 and serve as input for the simulation engine. For an introduction to Prolog that goes beyond what is needed to understand the simulation engine and its inputs, the interested reader is referred to the work of Ivan Bratko [28] and Blackburn *et al.* [25].

2.2.1. Facts, Rules and Deductive Reasoning

Knowledge bases in Prolog generally consist of fact collections and rules. Fact collections contain structured information that can simply be retrieved or processed in the course of evaluating rules. Both information and rules are accessed through so called predicates that may also serve to characterize the type of, or relation between, enclosed information. What may sound confusing is best illustrated with a small example. Listing 2.1 depicts a simple knowledge base that only consist of the predicates `produces` and `needsCooling`. The former serves to store information on the kind of components a given manufacturer produces, the latter is to tell if a given component type needs some sort of cooling. Note that predicates start with minuscule letters. The terms enclosed in the parentheses following the predicate's name can be of different shape to represent different types of information:

- Terms that only consist of alphanumerical symbols starting with a minuscule letter are generally treated as strings (e.g. `somePredicate(word)`).
- Terms that consist of alphanumerical symbols starting with a capital letter are also treated as strings, yet only if the whole term is enclosed in single quotes as in `someOtherPredicate('Word')`.
- Otherwise, if omitting the quotes, terms starting with a capital letter will be treated as variables that may be instantiated to any value.
- Numerals simply represent integers and do not need any special character to mark them as such as illustrated in `numberOfLegs(horse,4)`.
- A list may also represent a single (compound) term that comprises a potentially extensive amount of yet other terms (e.g. `listPredicate([one,two,three])`).

Furthermore, these terms may also comprise yet other predicates. However, where this is needed and how it works goes beyond a small introduction and is also not needed for the simulation engine. The number of terms that a predicate refers to is called its arity. For the first predicate in Listing 2.1, `produces`, this arity is *2*. For the second predicate in the given example, it is *1*. It is common practice to provide information on arity when referring to predicates, especially since the same predicate can be used multiple times with different arity within the same knowledge base. The default notation is to simply append the arity to the predicate, separating them with a dash as in `produces/2` and `needsCooling/1`.

```
1 // knowledge base consisting of the two predicates 'produces'
      and 'needsCooling'
2 produces(intel,cpu).
3 produces(amd,cpu).
4 produces(micron,ssd).
5 needsCooling(cpu).
```

Listing 2.1: Fact collections in Prolog.

```
1 // the known predicates
2 produces(intel,cpu).
3 produces(amd,cpu).
4 produces(micron,ssd).
5 needsCooling(cpu).
6 // predicate 'produces' can also refer to a rule and recursively query
      the same predicate.
7 produces(Manufacturer,memory):-produces(Manufacturer,cpu).
```

Listing 2.2: Deductive reasoning at the example of a simple rule.

This information can subsequently be queried. Assuming that the knowledge base from Listing 2.1 has been loaded, one might simply "ask" if Intel produces CPUs by typing `produces(intel,cpu).` to which Prolog will answer `yes` or `true`, depending on which implementation of Prolog is being used. Consequently, a query such as `produces(intel,ram).` will return `no` or `false` as the given fact collection does not say otherwise.

However, what is `true` or `false` may not only depend on facts but also rules. For example, assuming that any manufacturer that produces CPUs also produces memory chips, there is no need to explicitly add respective facts to the knowledge base. Instead, a rule that allows to deduce production of memory chips in case of producing CPUs is sufficient. Listing 2.2 extends the previous fact collection with such a rule. There are several things to observe in this example. First, the same predicate that has been used for mere fact statements can also be used to represent rules that are of the same arity. Second, the first term of the rule's head, that is the part left of `:-`, starts with a capital letter marking it as a variable, and third, Prolog allows for recursion. Usage of a variable to represent the manufacturer allows this variable to take any form and have Prolog deduce whether this can evaluate to `true`. When loading this knowledge base, a query such as `produces(intel,memory).` would cause Prolog to replace all occurrences of the term `Manufacturer` with `intel` and see if its one and only goal, that is the expression right of `:-`, can be fulfilled. Ultimately, Prolog will return `true`. Deductive reasoning based on the fact `produces(intel,cpu).` in combination with a rule that states that every CPU manufacturer also produces memory is sufficient to come to this conclusion.

Obviously, the same rule might apply to SSD manufacturers. To include these, the rule can simply be duplicated while changing the goal's second term from `cpu` to `ssd`. In such cases, Prolog will check both rules for goal satisfaction and return `true` or `false` accordingly. However, another option would be to append the alternative goal to the existing rule, combining the two with a logical `OR` that is represented as a semicolon (`;`) in Prolog.

2.2.2. Unification

Taking this concept of deductive reasoning a little further and incorporating the other predicate (`needsCooling/1`), one might introduce a rule to determine if a given manu-

```
1 produces(intel,cpu).
2 produces(amd,cpu).
3 produces(micron,ssd).
4 needsCooling(cpu).
5 // extending the previous rule to include SSDs
6 produces(Manufacturer,memory):-produces(Manufacturer,cpu);
7                                 produces(Manufacturer,ssd).
8 // new rule for the unification example; will return true or false if
      'Manufacturer' is specified, otherwise valid isntances of '
      Manufactuer' are returned
9 mustProcureCoolers(Manufacturer):-produces(Manufacturer,Component),
10                                   needsCooling(Component).
```

Listing 2.3: Deductive reasoning and unification in Prolog.

facturer must procure coolers which are needed for CPUs, yet not for solid-state drives. A corresponding rule is depicted in Listing 2.3, line nine. As can be seen, the rule `mustProcureCoolers/1` has two goals combined through a comma (`,`) representing the logical `AND` in Prolog.

When querying `mustProcureCoolers(intel).` this rule first checks the predicate `produces/2` for the components produced by the specified manufacturer. The variable `Component` serves as a placeholder that might be instantiated to any value derived from the lookup on `produces/2`. In the given knowledge base, there is only one possible value that `Component` can be instantiated to which is `cpu`. Subsequently, this value is used when checking for fulfillment of the second goal as it utilizes the same variable `Component` which has just been instantiated. This process of instantiating variables to specific values in the course of verifying goals is referred to as unification. This concept of unification can be illustrated with even simpler examples. While the very first query from the previous section — `produces(intel,cpu).` — would trigger Prolog to check if the given goal can be satisfied and report `true` or `false` accordingly, it may be adapted to query which known manufacturers produce CPUs in the first place. This is simply done by not specifying the first term but inserting an uninstantiated variable as in `produces(X,cpu).` when querying. Based on this, Prolog will instantiate `X` to any value for which this expression is fulfilled and return it. In the given example, both `intel` and `amd` will be returned as possible instances of `X`, one after another. Unification does not only work for values obtained from fact collections but may equally be used with rules. In that sense, `mustProcureCoolers/1` need not be queried with a specific value but can also be evaluated using a variable that will be unified with any value for which this rule evaluates to `true`.

The existence of extensive literature implies that there is more to be said about Prolog than what is contained in this short introduction. However, the fundamental concepts needed to understand the simulation engine in Chapter 4 are summarized quickly. Obviously, knowledge bases that served as examples have been chosen to be very simple as to not obstruct in illustrating the notion of deductive reasoning and unification.

3. A Critical View on Moving Target Defense

The previous introduction to the general concept of *Moving Target Defense* and respective techniques presented the topic in a neutral way. Yet, in how far *MTD* really is the game changer that many related papers claim it is [38, 45, 46, 143], remains to be seen until respective mechanisms have sufficiently been tested in real-world scenarios. The different techniques come with convincing narratives, explaining how and why movement, under certain conditions, may impede the attacker and improve security. However, one might construct equally convincing scenarios in which movement may have unintended, though plausible side effects that actually cause security degradation. On other occasions, movement may simply have no effect at all. So far, defensive techniques proposed since research on *MTD* kick-started mostly lack the necessary maturity and have only been implemented in prototypical ways, at best. Only few proposals (e.g. Li *et al.* [86]) allegedly made it into commercial products which are still hardly available though. In consequence, empirical data to compare intended effects with real effects observed in operational networks and systems are missing.

To this end, a lot of research on frameworks to model and formally analyze *MTD* techniques has been conducted. Unfortunately, said work is only rarely considers potential negative effects *MTD* techniques might cause and if so, only to a limited extent that regularly does not go beyond cost incurred through resource overhead and service degradation. Indeed, it seems that there is a bit of an *MTD* bubble in which movement is promoted as the answer without really questioning the merits. On a plainly argumentative level, this chapter elaborates why one needs to be careful when considering *MTD* techniques[1]. For this purpose, some of the most figurative analogies used to motivate *Moving Target Defense* by providing examples of how movement is integral to averting threats in other domains, are shortly presented and contrasted with plausible real-world scenarios in which specific techniques may in fact reduce security. This serves to illustrate that, depending on the context, movement may not be such a good idea after all. There is merit in the main idea of *MTD* and there are applications where it can probably improve security. But as a few examples show, there are also cases where movement can potentially make things worse.

(a) Dogfight Analogy (b) Flock of Birds Analogy (c) Shell Game Analogy

Figure 3.1.: Illustration of different possible analogies for *Moving Target Defense* which result in different perspective on movements.

3.1. Euphemistic Analogies

Moving Target Defense is frequently motivated with analogies from the real world. Some of the most figurative ones are presented in the following, together with short explanations on how they presumably relate to defending IT networks and systems.

The Dogfight

Zhuang *et al.* [148] compare *MTD* to a military strategy that is colloquially referred to as "dogfight" (Figure 3.1(a)). They outline that in an air-to-air combat situation it would be suicidal if a pilot who is being chased by an enemy aircraft continued flying forward in a straight line as this gave the enemy enough time to properly aim and fire. Instead, the pilot will spin the aircraft and make unpredictable turns to get rid of the persecutor and not become an easy target. It is quite conceivable that in this situation, movement is absolutely critical to avert successful attacks. Furthermore, Zhuang *et al.* compare the state of IT security to sitting in a bunker, waiting to be overwhelmed by the enemy. *Moving Target Defense* supposedly changes this situation. The lesson to be learned from these analogies is that standing still gives an adversary the necessary time to prepare his or her targeted attack that could have been avoided through movement. This relates to previously introduced *VM migration*, for example, where movement is intended to get virtual machines out of the attacker's reach, thus aborting ongoing attacks and forcing the adversary to pick another target or engage in reconnaissance again.

The Flock of Birds

Another analogy compares *MTD* to a flock of birds and has been used by companies[2] to commercially advertise the concept of *MTD*. In a flock of birds, each individual is constantly moving, thus changing the shape of the flock (see Figure 3.1(b)). A single bird would not stand a chance against a predator. Yet, the constant unpredictable movement

[1]The work in this chapter has been published [20] and presented at the *Mal-IoT Workshop, co-located with the ACM International Conference on Computing Frontiers 2020.*

[2]https://blog.cryptomove.com/moving-target-defense-recent-trends-253ce784a680

within the flock makes it impossible to single out one specific bird and consequently increases difficulty of a targeted attack. In that sense, this analogy fits the ideas of *IP shuffling* or *VM migration*, for example, where randomly relocating VMs or assigning addresses from a preferably large address space are supposed to prevent adversaries from keeping track of single entities. Furthermore, for an attacker who attempts to identify potential targets through sniffing traffic or other passive means, *IP shuffling* will bloat the number of observed addresses over time making it harder to pick a valid target, similar to picking a target from a large flock.

The Shell Game

The US Department of Homeland Security project homepage [1], Zheng and Namin [143], as well as Cho *et al.* [39] employ yet another analogy to describe the concept of *Moving Target Defense*. They compare it to the shell game in which a ball is put under one of three identical shells as illustrated in Figure 3.1(c). The shells are moved rapidly to confuse the players who can bet under which shell the ball lies. With increasing speed, it becomes harder to keep track of the ball's actual location. Furthermore, once the player (representing the attacker in this analogy) loses track of the ball's position, every choice she can make is no better than wild guessing. This analogy generally translates to the frequency at which movement of network and system characteristics occurs, as to invalidate the adversary's previous observations. While there are limits to how frequently different defenses can be triggered, be it because of physical conditions or the disruption of legitimate services, the shorter the intervals, the sooner the attacker has to return to the reconnaissance phase. In fact, if movement intervals are shorter than the time needed for the attacker to scout out the system again, reconnaissance may even become obsolete, forcing the attacker to resort to other means or simply guessing.

3.2. Reconsidering Analogies and Potential Effects

While the previously presented analogies may be convincing to a certain degree, they are still only analogies that do not and can not capture the reality of attacks on IT systems. However, even if putting inevitable inaccuracy aside, the question arises why only favorable analogies should be applicable and not the unfavorable ones. The following sections will illustrate that, on the basis of analogies, movement may equally be considered disadvantageous or generally useless. Furthermore, some of the techniques that have been presented in Chapter 2.1.2 will be revisited to elaborate on how and why the form of movement they suggest may actually be suitable examples for such negative effects or sheer irrelevance of movement.

3.2.1. Analogies With a Twist

To illustrate that movement is not necessarily always a good idea, *Moving Target Defense* shall be compared to deer hunting. While this, again, is just a figurative analogy, it makes clear that characterizing the effects of *MTD* is not that simple. Furthermore, the shell

(a) Deer Hunting Analogy

(b) Truth about the Shell Game

Figure 3.2.: (a) An efficient hunter will wait in a deer stand, exploiting the prey's movement. (b) The shell game's difficulty does not result from movement but sleight of hand.

game analogy is revisited. By having a closer look at what the shell game actually is, the message that this analogy conveys takes an ironic turn. Surprisingly, this changed meaning may still apply to introduced *MTD* techniques.

Deer Hunting

The meaning of the deer hunting analogy highly depends on the question whether or not a deer becomes a harder target if it is moving around. And this, in turn, depends on the hunter's strategy. Should the hunter employ a stalking approach in which he or she systematically sweeps through the hunting ground, this might be the case. By moving around, the deer might elegantly evade the also moving hunter and remain undetected for a longer time. However, stalking a deer can be tiresome and difficult, as the deer will flee when it notices movement or smells the hunter. Therefore, most hunters rely on a different approach: They simply find a suitable location or set up a deer stand where they can hide and wait for a deer to show up as depicted in Figure 3.2(a). And in this case moving around does not help the deer, but is actually the only reason why it might get shot in the first place. If the deer would stay put, the hunter would never catch it (unless it does so right in front of the hunter). Transferring this to the domain of IT security and *Moving Target Defense*, an adversary who is aware of the presence of *MTD* techniques may observe the system and wait for a configuration more vulnerable to her attacks that would not work if any of the other configurations had been statically employed.

The lesson to be learned from this example is that just as there are positive analogies supporting the idea of movement for the sake of security, there are equally plausible analogies suggesting the opposite. Researchers who propose and evaluate *Moving Target Defenses* should not only consider how movement benefits the defender, but also how movement may benefit the attacker.

The Shell Game Revisited

While the shell game's primary intention is to convey the notion that rapid movement is a suitable measure to avert attackers, those who employ this analogy obviously disregard the fact that this game is a fraud. That is, the difficulty of the shell game is not the result of movement, but sleight of hand where the player who moves the cups will hide the ball in the back of his hand as depicted in Figure 3.2(b). Quick movement merely serves as a distraction and is intended to make believe that success and failure solely depend on attention and the difficulty of keeping track of the "target". To what extent analogies are capable of adequately representing *Moving Target Defenses* and the effects they may have on systems and networks has been questioned on several occasions in this chapter. It is remarkable, though, that this analogy is not even suitable to promote the benefits of movement.

Sticking to the roles of the defender being the player who moves the cups and the attacker being the one who is trying to find the ball, the attacker still loses. However, considering the changed notion of the shell game, this does not result from movement.

3.2.2. When Movement May Hurt

This chapter discusses a few *MTD* techniques where movement may plausibly be expected to lower security, raising doubts about its unconfined usefulness. Despite existing research claiming security benefits, a critical consideration of potential effects appears necessary, since similar arguments as those that are used to convey movement-related security improvements, can be employed to imply security degradation. In particular Virtual Machine shuffling, OS rotation, application rotation, as well as configuration parameter randomization are being discussed. Note that these *MTD* techniques are not necessarily the only ones that may have negative effects on security but have been chosen for their frequent occurrence in *MTD* research.

Virtual Machine Shuffling

Shuffling or generally the migration of virtual machines has been addressed by several approaches to *MTD* evaluation [6, 7, 9, 10, 11, 45, 64, 150] and can be done in different ways as the various proposals from the Background Chapter indicate. Hong and Kim [64], for example, suggest state preserving live migration of VMs within a pool of physical hosts in the network. Using their modeling approach called HARM, they show that this VM shuffling increases security as it moves a VM with its critical application out of the attacker's firing line by changing its location and connectivity.

If the physical position of the VM does not matter for an attack, moving the VM has no effect since the logical communication must remain intact for the critical service to remain operational. However, there are several attacks in which the physical location matters: For one, there are cross-VM attacks in which an attacker may gain control over or at least interact with VMs running on the same hypervisor. For another, co-located VMs can frequently, that is in most default configurations (e.g. Xen), communicate unhindered through the hypervisor's virtual switch, thus mitigating any firewalling that

could have been in place at the VM's previous location. This may enable further attack paths, especially in small enterprises, where VMs are not isolated to the maximum as is or should be the case in Amazon's AWS or Microsoft's Azure, for example.

With this in mind, the "stalking hunter" should be recalled. If the attacker wants to perform a cross-VM attack, she needs to gain control over a VM on the same hypervisor. During the time it takes the attacker to get access to a VM on a particular hypervisor, however, the VM shuffling defense might have moved the target, thwarting any effort. In consequence, similar to a hunter stalking the prey, movement in form of VM shuffling may increase attacker cost. Yet, "stalking" a particular VM is only possible if it takes more or less the same effort to get control over any VM on any hypervisor. In practice, however, this is not necessarily the case. Unless the adversary is able to instantiate her own VM, she has to compromise an existing one to gain access. Additionally, she needs to be able to reach this VM which might be protected through a firewall. Therefore, the attacker will likely be able to attack a small subset of VMs, only. Similarly, even if the attacker is allowed to start her own VMs, she will most probably not be able to influence on which physical host this is done, thus impeding efficient "stalking" of the target.

In fact, the waiting hunter analogy is much more fitting: Having gained control over a VM, be it through compromisation or legitimate instantiation, the attacker, in the presence of VM shuffling, simply has to wait until the target moves into the desired position. By only attacking a single VM, the attacker can essentially reach any other VM by simply waiting for VMs to be shuffled. Hence, the attack surface has increased significantly while the overall security of the system decreased — due to VM shuffling. This is a hypothetical, yet plausible scenario that does not interfere with the concept of VM migration. In that sense, VM migration may open up new attack paths, thus drastically reducing the overall security of the network.

Operating System Rotation

OS shuffling or rotation is a *MTD* technique that is based on the idea that constantly switching operating systems makes it harder for an attacker to penetrate a network. The reasoning behind this is that if an attacker knows an exploit against one OS, switching the OS will fend him off. Since the attacker does not know which VM will run which OS at what point in time, attacking will be difficult. OS rotation, among other things, has been addressed and deemed useful in [5, 6, 7, 120]. However, within these approaches, the effectiveness of OS rotation to improve security was never questioned but only postulated. Instead, the focus of evaluation was on feasibility and optimization of OS rotation to reduce downtime. Once again, the hunting analogy can be applied. If an attacker has an exploit for a specific OS, then using this exploit (after it has been developed) is typically a matter of seconds. So the difficulty lies in obtaining or developing an exploit in the first place. In the OS rotation scenario, the attacker only needs to find an exploit for one of the OSes that are being rotated and then wait for the desired OS to run on the target. For inevitable overhead, OS rotation can not be done every few seconds so that there will be ample time to attack the OS once the desired OS is rotated in. In effect, OS rotation enables new attack chains that would not exist without it.

Note that OS rotation is not the same as OS diversification. Using diverse operating systems across systems within the network, it may be harder for an attacker to spread throughout the network and reach his ultimate goal. But while OS diversification is able to cut off attack chains, OS rotation enables new attack chains that may result in a net security loss.

Application Rotation and Parameter Mutation

John *et al.* [73, 87] have proposed application configuration parameter mutation as an *MTD* technique. The idea is to use genetic algorithms that constantly re-evaluate and change configurations to find (near) optimal settings for an application. However, they also measure configuration distance to ensure that old and new settings are not identical. They use an Apache Web-Server running on Redhat Enterprise Linux as a prototypical example.

Constantly verifying if the used configuration is still the most secure makes sense — if an attack becomes known, systems should quickly be adjusted accordingly. However, what benefit is gained by not keeping the most secure configuration until a more secure configuration is found? Switching between configurations may actually increase the attack surface: Should there be no vulnerability that can be exploited based on the currently employed configuration, the attacker only has to wait until the configuration is changed to try again, thus increasing his chances. Since there is only a limited number of reasonable configurations, it is also not problematic for an attacker to wait for a configuration that he is crafting an exploit for to reappear. Hence, while continuously optimizing the configuration of an application is a reasonable strategy, the forced constant "movement" of the configuration to hopefully equally, but potentially less secure settings, decreases security. It seems movement is only done so that it qualifies as a *MTD* technique.

Other approaches that do not only consider an application's configuration but the application itself, have been proposed by Taguinod *et al.* [117] and Zhang *et al.* [142]. At the example of SQL servers they suggest that the entire application is exchanged with another one that is capable of fulfilling the same task. However, this is subject to the same limitations as mentioned before.

Casola *et al.* [33] even propose to constantly switch between a set of different encryption systems. The authors implemented their prototype on a small resource constrained OS to prove feasibility. They argued that a brute force attack becomes more difficult, as the attacker does not know which cryptosystem is being used at a given time. While noting that there might be more efficient attacks than brute force on an algorithm, they claim that this does not influence the security benefit from switching algorithms. They also note that when the attacker does not know the used key length, brute force attempts are more difficult. Yet, why would it be more secure to switch between two key lengths? To attack an 80 bit key using brute-force requires, on average, 2^{48} times less effort than attacking a 128 bit key. Hence, from a security perspective using the longest key is the most secure, not switching between them. Furthermore, the likelihood that an attacker finds a flaw in one of, say, three encryption algorithms or implementations is higher

than in only one. Consequently, by constantly switching between encryption algorithms, the attack surface rather increases, thus potentially reducing overall security. Updating keys or using longer keys would be a much more sound and efficient solution if one is concerned about brute-force attacks.

3.2.3. When Movement May Work

Of course, there are also cases in which movement makes a system more secure and the positive *MTD* analogies of the dogfight or the flock of birds can be applied. ASLR, NASR, as well as anonymous and dynamic routing, for example, are techniques suggesting movement of properties that may plausibly impede an attacker during reconnaissance and execution when repeatedly being changed, yet are not subject to vulnerabilities themselves. To better illustrate what this means: Changing a host's IP address as advocated by NASR does not directly affect vulnerability for the simple reason that, in general, one address is not more or less secure than another one. Different operating systems, on the other hand, exhibit different vulnerabilities so that exchanging these for one another may directly affect a system's security level. Note that this does not mean that there are no downsides to these techniques. At least, they incur additional load on respective systems. However, an adversary cannot simply exploit movement by waiting. Obviously, the techniques discussed in the following are not necessarily the only ones that may have a positive effect on security.

Address Space Layout Randomization (ASLR)

The most prominent example is ASLR in which real memory addresses are obfuscated by the OS. The "movement" of addresses increases the difficulty of reliably exploiting buffer overflows, for example, due to unknown memory offsets. Despite the existence of viable attacks on ASLR, the scheme has shown to raise the bar for attackers. The comprehensive overview of suggested techniques presented in Chapter 2.1.2 lists many more randomization techniques based on dynamic software, dynamic data, or dynamic runtime environments that intend to increase the difficulty of exploit development. The survey from Ward *et al.* [131] further extends this list with examples including commercial approaches in which an application is recompiled for every new user so that an exploit developed for one target does not work for another. It is interesting to note that 54 out of 89 *MTD* techniques listed by Ward *et al.* are categorized as exploit development preventing, many of which have not been proposed as *Moving Target Defenses*. Still, the numerous frameworks and theoretic approaches to *MTD* evaluation in existing literature do not consider assessing the effectiveness of exploit prevention, but instead focus on the other phases of the attack chain.

Network Address Space Randomization (NASR)

Another technique that potentially benefits security is IP shuffling or network address space randomization in general. This assumption is based on the fact that changing addresses has no impact on a target's vulnerability per se for addresses themselves not

being security critical. The deer hunter analogy is therefore not applicable. On the other hand, constantly changing network addresses may indeed limit the time frame an attacker can use obtained information, thus making some attacks more difficult. The attacker also has a harder time learning the network layout and do reconnaissance to identify high priority targets. Yet, this only holds true as long as the attacker has no efficient way of obtaining addresses or layout information. Considering that legitimate clients must be able to communicate with servers, implemented NASR schemes must provide means allowing for unobstructed communication. These may simply be based on using DNS names that are presumably unknown to the attacker and cannot be sniffed or other more sophisticated authentication schemes. However, such means that allow for unimpeded communication may also be used by an attacker, once necessary resources have been compromised. In such cases NASR is of no benefit, but at least does no harm either.

Anonymous and Dynamic Routing

Another technique that has the potential to improve security is anonymous and dynamic routing. Its effectiveness, however, may depend on communication behavior and the threat scenario. First of all, anonymous routing itself does not necessarily fit the definition of *Moving Target Defense*. Assuming that anonymous routing manages to keep senders and receivers secret, there is no movement at all but a reduction of available information. While this may be even better, it still does not involve movement. Nevertheless, dynamic routing can be considered a *MTD* technique, which in combination with sender and receiver anonymity may offer a higher degree of security. In a threat scenario where an adversary needs to intercept all traffic, dynamic routing impedes the attacker unless he manages to control each of the selected paths to re-assemble all data. Depending on whether dynamic routing is employed on a per packet basis, instead of per flow, the attacker might not even be able to re-assemble a single payload if not eavesdropping on all paths. However, should the attacker not be interested in what is being communicated but only in the fact that two entities communicate, dynamic routing increases chances that the adversary may catch at least some packages that reveal who is communicating with whom. In this threat scenario, anonymizing or at least obscuring communicating entities is advisable. While obscuring with help of aforementioned NASR may raise the bar for an attacker to identify communicating entities despite changing addresses, full anonymity is even better. It should be noted, though, that the ability to route dynamically, as well as the degree to which obscuring or anonymizing addresses can avert identification depend on the network topology. The excursus in Chapter 7 further elaborates on this topic.

3.2.4. When Movement Does Not Matter

Chances are that movement of certain system or network properties may not have practical effects. However, it is advisable to differentiate between the reasons why that is. In a somewhat ironic yet illustrative way, Evans *et al.* [56] outline how different *Moving*

Target Defenses are incapable of defending against certain types of attacks. Neither improving nor lowering security, the examples that the authors choose convey the meaning that selected defenses have no effect at all. However, in the case of Evans *et al.* this is simply due to the fact that defenses and attacks are deliberately chosen to operate on different levels so that there is basically no interference. ASLR, for example¸ will not be able to prevent SQL-injection attacks since it has got nothing to do with how a related application handles inputs.

Yet, despite the fact that the form of movement introduced by a specific technique may not affect all kinds of attacks, there are also examples where movement makes no sense from any point of view. One such technique is the CAN ID shuffling scheme suggested by Woo *et al.* [134] which has already been mentioned in Chapter 2.1.2 when introducing selected *MTD* techniques. The technique supposedly makes it harder for an attacker to send a malicious message to a device by switching the address (CAN ID) in unpredictable ways. And in fact, this increases difficulty for an attacker. However, the effect is not due to randomly changing the ID, but using the last 22 bits of the ID header as an unpredictable authentication tag. Typically, an authentication tag would also ensure integrity of the message, and not only the address. Nevertheless, this increases security, albeit not due to any address randomization but the inclusion of a cryptographic authentication tag. In that sense, this technique fits the revised shell game analogy, where movement only seemingly contributes to the effect that in reality depends on something entirely different.

Without further data from the real world or results from adequate evaluation schemes, it is hard to tell which other *MTD* techniques may actually have a positive impact on security, though be it not for the sake of movement. The case of Woo *et al.* allowed for such evaluation plainly on the basis of dissecting the suggested scheme, yet others may require testing and experimentation.

3.3. Status Quo

Supported by figurative analogies, this chapter outlined that frequently proposed and cited *Moving Target Defenses* bear the potential to not only improve, but also degrade security. While these negative effects of specific techniques have only been derived on a plainly logical level and not shown in the real world, neither have their benefits in most cases. So far, existing approaches to evaluating *MTD* techniques mostly considered the benefits of movement, thus causing analysis to be biased in favor of movement. Downsides, when taken into account at all, are regularly limited to cost incurred through excess resource consumption or service degradation, but never include security reduction. Considering the potential security gains as well as losses, research on *MTD* should incorporate both positive and negative effects into theoretical *MTD* frameworks and analysis techniques. In particular two simple rules should be kept in mind when analyzing or developing *MTD* techniques:

1. **Movement can hurt:** Possibly negative security implications of movement must always be considered.

2. **Do not move for movement's sake:** It should be verified that the security benefit from an *MTD* technique really stems from the movement aspect.

With *MTD* being promoted by funding agencies and specific workshops, there has been an influx of papers proposing movement for a wide range of applications. Hopefully, cautionary notes help researchers to keep a balanced view on *MTD* and start "separating the wheat from the chaff". Research should focus on the security goals to be reached, that may be supported by concepts such as *MTD*. Yet, in the end, the technologically most sound solution should be chosen. If movement is able to deceive an attacker, it may be reasonable to employ it. But if a static system can achieve the same, one should not hesitate to promote this technology despite not being able to attribute it to a popular trend in academic research.

3.3.1. Limitations of Evaluation Approaches

The reason that existing approaches to evaluation fail to reveal defense-induced security degradation may be due to some repeatedly observable limitations related to different aspects of respective evaluation schemes. One such aspect is the question on what is actually being evaluated. What may frequently go unnoticed is that some of the suggested evaluation approaches investigate defense effects, while others evaluate defense strategies. The difference is that the former intend to find out about the impact defenses have on system and attacker. The latter, in turn, address the question of how to efficiently use these defenses in the presence of certain attacker types or environmental constraints, while assuming that intended effects are also achieved. Another aspect is the underlying method that is used to perform analysis in the first place. This is hardly ever the same for any two evaluation approaches. Yet, differentiation between those that rely on either mathematical formalization, simulation, or (virtual) testbeds seems appropriate. These categories are very inclusive so that approaches they subsume still differ. The chosen method also has implications with regard to size and complexity of models that analysis is based on, thus ultimately influencing evaluation results and their validity.

A considerable number of suggested approaches to evaluating *MTD* techniques are game theoretic in nature and rely on mathematical formalization. These regularly yield probabilities, quantifying how chances of a successful attack (or defense) are affected by a specific technique, thus overcoming the comparability issue. Yet, their ability to model complex situations is limited. What is more, most game theoretic approaches focus on optimizing attack and defense strategies or finding equilibria in the presence of actors of certain capability. This, however, requires in-depth knowledge on defense effects which are mostly just assumed. Respective evaluation schemes range from Stackelberg Games [3, 122], Zero-Determinant Theory [130], and empirical game-theoretic analysis [107], through different forms of Markov modeling [7, 45, 46, 84, 89, 136, 144], to stochastic petri nets [14, 40]. While such approaches regularly consider that movement may be constrained, some of the proposed work [38, 94, 129] is more elaborate in this regard, taking potential service degradation and excess resource consumption into account. Negative effects on security, however, are out of scope.

Furthermore, there are simulation-based approaches to *MTD* evaluation. Being generally able to imitate operation of real-world processes, simulation can handle higher degrees of complexity that go beyond the capabilities of mathematical formalization and investigate problems where there is no analytical solution [23]. However, to what degree simulation generates meaningful insights into the effects of *MTD* techniques, obviously depends on the respective approach's scope, as well as quality and detail of modeling. Hong and Kim [64], for example, propose to evaluate *MTD* techniques with help of simulation on a multi-layered model that is supposedly able to incorporate a high level of detail. Yet, for their case study, they only consider a small simplified network segment with a very limited threat scenario and, most notably, assume perfect knowledge about the attacker's state. Succeeding work based on their approach [8, 10, 55] extends the scenarios that simulation is performed on, but still suffers from ignoring realistic threats that may result from co-location of VMs, for example. Yet other simulation-based approaches [50, 51] do not intend to determine defense effects but rather to optimize defense utilization to mitigate DDoS attacks. In this regard, they are comparable to the previously referenced mathematical approaches that focused on developing strategies. Similarly, early work from Zhuang *et al.* [147, 150] suggests simulation to test performance of different attacks and defenses in dependency of characteristic parameters such as timing, frequency and expected effects that are assumed to be known.

In contrast, there are approaches that rely on testbeds [41, 98, 118] and analyzing raw data from real-world measurements [140], offering an unmatched level of detail that ultimately supports the validity of generated evaluation results. However, these are scarce for causing considerable effort in maintaining and extending them, even if virtualized. Firstly, a defense must be fully implemented to be evaluated, there is no quick testing of an idea or concept. Secondly, monitoring the system state as to determine whether or not a defense was able to prevent attacks requires further effort, all of which is multiplied by the number of scenarios a defense is supposed to be tested in.

A viable approach to defense evaluation that accounts for the needs of proactive defense should not presume effects but determine them. Furthermore, it should be able to incorporate the level of detail that is necessary to yield realistic results, and be maintainable and scalable enough to easily extend testing.

4. A Modeling and Simulation Framework

In this chapter, a modeling and simulation framework is presented that overcomes previously identified limitations of existing approaches to defense evaluation, making a step towards fair and realistic quantification and comparison of the security impact of different defense techniques[1]. The framework consists of independent components to handle the various steps from initial modeling, through simulation, to the processing of results. To motivate why the framework has been designed the way it is and to allow for a better understanding of subsequent descriptions and explanations, general requirements towards evaluation will be presented first, followed by an introduction to terms that have a specific meaning in the context of the framework. Afterwards, an overview of the framework's general structure will be presented, before providing a detailed description of the framework which is divided into two major parts, the modeling language and the simulation engine. Finally, a meta model is defined that prescribes how modeled entities should look like. This is to establish a minimum level of detail and ensure compatibility of said entities.

4.1. Towards Fair and Realistic Evaluation

Chapter 3 outlined that existing approaches to evaluating *MTD* have a tendency to only consider beneficial effects and ignore potential downsides. While there was no proof but only claims that such negative effects exist, the arguments that have been presented to justify such claims are not less plausible than those used to convey the beneficial effects. Consequently, for evaluation results to yield meaningful insights, the chosen approach must be able to reveal both positive and negative effects.

To accomplish this, the suggested framework must fulfill certain requirements. These are presented in the following, together with explanations on why fulfilling them supposedly resolves limitations and shortcomings of existing approaches. However, the sole definition of requirements does not yet solve any problems which is why Chapter 4.1.2 will introduce some fundamental design choices that have been made to address said requirements and tackle the challenges related to realistic and fair evaluation.

4.1.1. Requirements

To be able to realistically analyze and compare different defense techniques, an evaluation framework must fulfill, or – on occasions where modeling and simulation incorporate

[1] Parts of the work presented in this chapter have been published [19] in the proceedings of the *Nordic Conference on Secure IT Systems 2018*.

user input – allow for the fulfillment of the following requirements. Note, however, that these requirements are still relatively general, only specifying what aspects should be considered, but not how to consider them. How the suggested framework addresses these, is covered in the remainder of this chapter.

- *Handling dynamic changes:* The key property of paradigms such as *Moving Target Defense* is the proactive and flexible modification of a system's appearance. This means that the attacker is not the only one who may incur changes to the system. As a result, a suitable analysis technique needs to be able to incorporate numerous actors, account for the effects of their actions, and consider interdependencies that affect their subsequent options.
- *Modeling granularity:* Analysis can only reveal effects on characteristics that have been modeled in the first place. An over-simplified model that does not consider that co-located VMs may interact with each other, for example, will not be able to determine effects that result from such interaction. In consequence, obtaining meaningful results is only possible if the modeled networks and systems, with their OSes and applications are represented in a realistic way.
- *Realistic attacker capabilities:* Attackers must be able to realistically interact with the system. While exploits are often used to penetrate a network, once attackers are inside, a common strategy is to exfiltrate credentials and then use legitimate means to laterally move within the network. Consequently, not only exploits but also legitimate actions must be incorporated. Furthermore, attacker actions must be equipped with realistic requirements that are checked against the modeled system's state to decide whether or not they can be performed.
- *Realistic defense modeling:* Similarly, defender actions need to be modeled accurately. That is, representing a defense action's impact on the system through the state changes it causes and not through the intended high-level effect. Otherwise, security-relevant side effects will go unnoticed.
- *Stateful attacker:* To account for the fact that reconnaissance is integral to successful attacks and that one of the primary purposes of proactive defense is to invalidate previously acquired information, the framework must incorporate an actor's knowledge. Otherwise, the effects of invalidating previously learned IP addresses through means like NASR could not be adequately represented.
- *Common and quantifiable metrics:* To be universally applicable, metrics that are used for evaluation must be independent from any given defense technique. Furthermore, for subsequent comparison to be fair, they must be able to equally capture the different techniques' effects as to not prefer one over the other.

4.1.2. Fundamental Design Choices

Based on the aforementioned requirements and insights from existing approaches to evaluation, fundamental design choices have been made that are expected to support the

realistic evaluation of defenses. How a framework implements these, is not important at this point but will be addressed later in this chapter. However, introducing and motivating them here helps to understand how the toolchain sketched in Chapter 4.3 came into existence and which assumptions it is based on.

To perform analysis, event-discrete simulation has been chosen for its ability to incorporate interaction of multiple actors, be it attackers or defenders, and account for intermediate state changes caused by proactive dynamic defense or any other action which may ultimately affect subsequent options of all other actors. Knowledge that actors may have acquired is part of this state representation, thus equally subject to manipulation by successful actions. Limitations presented in Chapter 3.3.1 illustrated that evaluation on the basis of mathematical models, though elegant and computationally less expensive, is not able to incorporate a state model with the desired level of detail but requires considerable simplification and strong assumptions. Testbeds, on the other hand, constitute the opposite extreme. Being real systems, their level of detail is unsurpassed, yet their setup and management requires a lot of effort, even when virtualized.

In the course of simulation, to determine any actor's options to interact with the system based on the current system state, deductive reasoning is used. As previous research by Ou *et al.* [102] has shown, Prolog is suitable for this purpose, especially since some implementations such as XSB and SWI-Prolog support tabling [110]. This feature, simply put, allows to selectively store intermediate results for specific predicates with given arity, so that values, instead of re-computing them over and over again, can be looked up. As a result, the performance of Prolog is further improved by significantly reducing computational effort for answering subsequent queries, which is especially useful when the number of potentially executable functions is high and their dependencies are "sufficiently" complex. Existing work uses deductive reasoning to derive complete trees instead of next steps only (e.g. MulVal). Yet, in the presence of multiple state-manipulating actors, this is barely feasible as has been mentioned before. Furthermore, the complexity of the resulting graph would render it incomprehensible, partly taking its purpose to the absurd.

Additionally, a modeling language is provided, forming an abstraction layer between the user and the framework. This way modeling systems and actions is unified and independent from the specific implementation of the simulator used to determine an actor's options to interact with the system and apply resulting state changes. The reasons to do so are simple. First, the toolchain may be subject to further development in the future. Should there be a better way to do simulation or determine an actor's next steps, and the framework be adapted accordingly, previously crafted models should not be invalidated but benefit from the frameworks improvement. Second, the format required by the simulation engine to perform deductive reasoning is not user-friendly and comparatively extensive, thus emphasizing the utility of a dedicated modeling language. After all, if the framework is supposed to enable IT security practitioners and researchers alike to check on various defenses' impact on security in numerous different settings, it must not suffer from obscure formatting or syntax required for subsequent processing. Instead, it should be human-readable and as usable as possible. Third, introducing such an abstraction

layer allows for the integration of high-level features that may aggregate otherwise extensive code into single instructions. This way, the required detail for modeling systems and actions can be reached with less effort.

A simplified overview and summary of a framework that adheres to these design choices is presented in Chapter 4.3, followed by detailed descriptions of the main components in the subsequent sections.

4.2. Definition of Terms

Before going into details of the language's features and syntax, and illustrating these on the basis of practical examples, it is necessary to establish a common understanding of a few specific terms that are used in the following sections. While being particularly relevant to understanding the modeling language, these terms denote concepts that are referred to throughout the whole framework. These terms and the meaning they have within the scope of this framework are presented in the following.

Elements are the building-blocks of every scenario and represent more or less complex data structures that can be related to one another through references. Elements can have an arbitrary amount of attributes that can be elements themselves, thus allowing for the definition of large trees. However, elements may also be completely unrelated. Their primary purpose is to allow for the encapsulation of information that somewhat belongs together. Employing the example of a computer, such an element may be a PC, that, in turn, consist of numerous other elements such as a CPU, RAM, or an operating system.

Attributes refer to the properties that elements may have to characterize their shape and behavior during simulation. Such attributes may be elements themselves (i.e., sub-elements) that have attributes of their own. However, attributes may as well be primitive data types such as strings to simply define an element's name, for example. Regarding an element as the root of a tree, primitive attributes represent leaves, whereas complex attributes (i.e. sub-elements) pose as nodes that increase the tree's depth. Depending on the required level of modeling detail, the user of the language may opt for either one or the other. Attributes may not only refer to single data points but also lists.

Types are used to classify elements and their attributes. In the framework, there exist three predefined primitive types, `string`, `boolean`, and `int` that are used to store either text, `true` or `false` values, or numbers. These primitive types can only be used for element attributes but never exist without a corresponding element. On the other hand, there are no predefined types for elements. Instead, the user of the language can freely define an element's type upon its declaration.

Identifiers are unique names that are used to unambiguously refer to elements. This is especially important for elements that pose as roots, as they have no super-ordinate element that refers to them through one of its attributes. Sub-elements have identifiers, as well, yet accessing them through a parent element's attribute is easier in most cases. Depending on how a sub-element is instantiated, its unique identifier may even be unknown to the user so that access through attributes is the only way.

Templates are ready-made "blueprints" that can be used to instantiate elements with a single instruction, similar to classes known from object-oriented programming. The contents of templates can range from a hand full of primitive attributes to complex structures comprising numerous sub-elements and functions. Templates can be freely defined by the user to serve any purpose and must have unique names to allow for unambiguous instantiation calls.

Functions are, just like elements and their attributes, part of the modeled network. Their purpose is to enable interaction with the system, be it to simply check if certain conditions are met or for the sake of manipulating the system state. Functions can be defined at will, either in the scenario definition or one of the templates it utilizes. Functions are generally global so that their names have to be unique.

Parameters serve the purpose of making both functions and templates flexible. They are passed upon function call or template-based element instantiation and provide values that shape the appearance of an element to be created or specify aspects of a function's operation, respectively.

4.3. Framework Overview

The modeling and simulation framework consist of different components. Figure 4.1 provides a simplified overview, depicting information items as squares and processing steps as dashed rounded boxes.

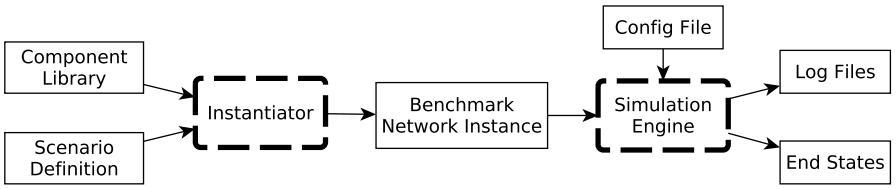


Figure 4.1.: Overview of the attack simulation framework

The first processing is done by the instantiator that transforms any given scenario definition into a corresponding benchmark network instance. Such a scenario definition dictates the composition of the system or network that is supposed to be investigated. To keep this definition as compact and comprehensible as possible, it utilizes the component library to reuse previously defined elements that can be of any size and complexity. This reduces effort when creating large and high-detailed scenarios, as reoccurring elements do not need to be defined over and over again, similar to the concept of classes from object-oriented programming. The scenario definition, as well as the templates in the component library are written in a human-readable modeling language that employs an intuitive syntax and provides specific features to allow for a compact description of systems as well as actions.

The resulting benchmark network instance is a single Prolog file that follows specific conventions with regard to representing facts and rules to allow for automated processing

by the simulation engine. As opposed to the high-level scenario definition, the benchmark network instance is a self-contained monolith, providing a coherent description of a network's full state including the actors, as well as possible actions and their requirements. Consequently, it is not as compact and easy to comprehend anymore. Based on this input, the simulation engine can determine the different actors' possibilities to interact with the system, trigger actions, and apply state changes accordingly. How the simulator operates exactly is determined by the simulation configuration file. This file is used to configure the simulator with regard to available defenses, attacks and legitimate actions, as well as the number of rounds to simulate. The intention behind using such a simulation configuration instead of "hard-coding" respective properties into the scenario definition is, simply, to increase flexibility and avoid repeatedly passing through the whole toolchain. This way, the same benchmark network can easily be tested for different attackers and defenders without the need to apply any changes to the scenario.

Throughout the simulation, the simulation engine keeps track of all events and writes them into log files for subsequent analysis. These logged events comprise anything from first initialization of attacks, defenses and legitimate actions, through occurring state changes, to assets acquired by the attacker(s). Additionally, the simulation's final state is exported for further investigation or later continuation of the simulation.

4.4. Modeling Language

Considering the fact that both JSON and XML serve the primary purpose of describing structured information, the question may arise, if it was not easier to simply resort to employing one of these instead of developing a new language. While this would have saved time also with regard to the required translator, it would not have fulfilled all requirements, though. First, while both JSON and XML are human-readable and allow for a coherent state description, they are far from comprehensible. Once the system to be modeled exceeds even the smallest degree of complexity that might not yet even be sufficient for the envisaged level of detail, JSON and XML documents become impossible to read or maintain without additional tools. Even though JSON omits a lot of the overhead that is mandatory in XML, the degree to which this improves on comprehensibility is negligible in the presence of large and detailed system models. Furthermore, neither of them provide any operators or keywords that are intended to reduce modeling effort or provide any other sort of flexibility for that matter. Instantiating complete elements from reusable templates with a simple keyword such as `new`, for example, is not possible without any non-standard preprocessing. Finally, the scenarios needed for simulation are not only mere state descriptions but also incorporate functions as means for attackers and defenders to interact with the system. To a certain extent, functions with their conditions and effects could probably be represented as nodes with specific attributes. However, considering the descriptive nature of both JSON and XML, resorting to this practice feels more like a work-around and is anything but intuitive.

In consequence, the high level modeling language used in the scenario definition file and the component library, has been specifically designed and developed for this frame-

work. As outlined in Chapter 4.1.2, the modeling language serves as an abstraction layer between the user and the framework that may provide at least three very obvious advantages:

- Simplification of the modeling process by providing an intuitive and common syntax so that scenarios are easy to create and maintain.
- Introduction of advanced language features to reduce modeling effort, increase efficiency and keep definitions compact.
- Provisioning of a unified way to modeling systems and functions that is independent from subsequent processing so that changes to the framework do not invalidate previously defined scenarios.

To what extent these advantages can be realized, depends on different aspects. The two first advantages solely depend on the language's design that will be covered in the following Chapter 4.4.1. There, a coherent description of the modeling language's syntax and features will be provided, together with practical examples to give an introduction on how to use it when modeling systems and functions. The third advantage from the above list requires the implementation of a translator to transform the high-level description, written in said modeling language, into a low-level representation that is compatible with the framework's other components taking care of subsequent processing. This low-level representation consists of valid Prolog facts and rules that follow a certain structure as expected by the simulation engine. Considerable effort was put into developing a structure for this low-level representation so that it is straightforward to derive from the high-level modeling language and, at the same time, suitable for automated processing by the simulator. Details on this low-level representation will be provided in Chapter 4.4.2.

4.4.1. High-level Language Properties and Syntax

The modeling language employs a syntax that is very similar to that of well-known object-oriented programming languages such as Java or C++. Its primary purpose is to enable the efficient description of a system's state and the definition of functions in order to interact with it. The language comes with a manageable amount of keywords, and only has to comply with few conventions in order to be processable. As a result, it is easy to grasp and does not leave much space for erroneous usage. The following outline is divided into three parts, with the first one presenting the basics of describing system states which is fairly simple. The second one introduces functions, elaborating on some noteworthy particularities, whereas the third part serves to explain special functors that can be used within functions.

System State Description

Most instructions to describe the scenario's state are, simply put, declarations and assignments, both of which always require a type to be given first. Explicit declaration

is only needed for elements, whereas element attributes, or rather the attribute keys through which stored values and other elements are referenced, are always declared implicitly. That means, an element attribute of type `string` that is referred to as `name` (i.e., the attribute key), for example, is automatically declared upon assigning a string to it but cannot be declared in advance without such assignment. However, should an attribute be assigned with another element instead of a simple string, said element must have been declared at some previous point. There is one exception to this rule, namely when the keyword `new` is used to instantiate a new element upon assignment, but this will be covered later on. Note, that the ability to assign elements as attributes of other elements allows for the description of arbitrarily complex structures that can be regarded as directed graphs. When assigning lists instead of singletons, square brackets are appended to the type. Likewise, the list of values to be assigned is enclosed in square brackets with its values separated by commas.

Element attributes are generally accessed through the element's identifier in conjunction with the respective attribute key, both of which are connected through a dot (`.`), as is common in many languages. In large structures (i.e., the aforementioned directed graphs), this scheme of simply appending a dot and the respective attribute key can be carried forward to an arbitrary extent to reach attributes of subordinate elements at any depth. It should be noted that identifiers, as well as element attributes always start with a minuscule letter. Also, every instruction is terminated with a semicolon (`;`).

```
1  // declare elements of different types
2  application app1;
3  application app2;
4  os os1;
5  node pc1;
6  interface if1;
7
8  // give them primitive attributes of type string
9  string app1.name="cron";
10 string app2.name="nginx";
11 string pc1.type="server";
12
13 // assign element as attribute
14 os pc1.os=os1;
15 interface pc1.interfaces=if1;
16 interface app2.bindingInterface=if1;
17
18 // assign lists
19 application[] os1.apps=[app1,app2];
20
21 // subordinate elements for detailing
22 processor cpu1;
23 int cpu1.cores=8;
24 processor pc1.cpu=cpu1;
25
26 // indirect access to attributes
27 int pc1.cpu.cacheSize=512;
```

Listing 4.1: Element declaration and value assignment

A few exemplary instructions to demonstrate how a simple description of a computer, referred to as `pc1`, may look like are given in Listing 4.1, with lines two to six depicting element declarations that solely require type and identifier (e.g. `application app1`) to be specified. Lines nine to eleven, in turn, equip these with attributes of type `string` to specify `name` and generic `type` respectively. Additionally, lines 14 to 16, as well as 24 illustrate how to set elements to be attributes of other elements to enable more detailed modeling, while line 19 shows how to assign multiple values, in this case elements, at once. Besides, what has not been mentioned before, this code snippet demonstrates the usage of double dashes `//` to initiate comments. These always affect the whole line and cannot be placed behind a statement that is supposed to be processed. For better comprehension and to illustrate the concept of these descriptions forming a directed graph, Figure 4.2 provides a visualization of the structure that results from the provided code snippet with squares representing both elements and simple attributes, and arrows respective attribute keys. While this is only a small example, it is indicative of the language's ability to define complex scenarios.

What this example also shows is that the assignment of subordinate elements merely forms relations. Whether these are integral to the parent element, thus increasing its level of detail as in lines 14, 15 and 24, where `pc1` is extended through specifying and assigning subordinate elements and their attributes, or simply express a relation like that of an application's binding interface as in line 16, does not matter to the language. What these relations mean is ultimately dependent on how the corresponding functions use such information. To prevent any ambiguity with regard to semantics and ensure compatibility of subsequently defined elements and functions, it is advisable to define a meta model specifying required attributes and how they should be treated. For the scenarios used in the case studies in Chapters 5 and 6.4, a corresponding meta model is described in Chapter 4.6.

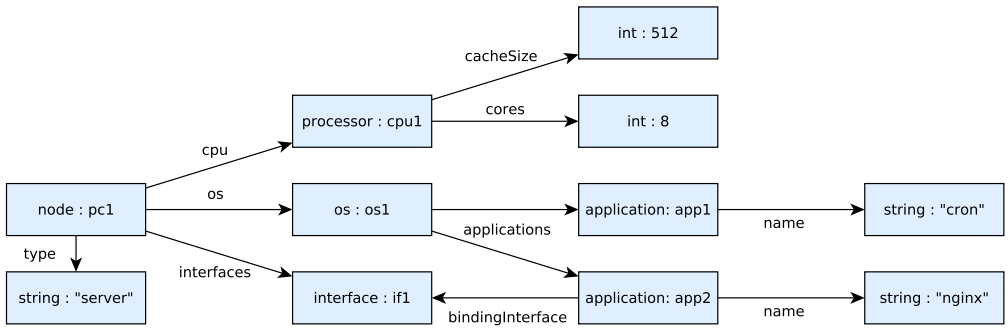


Figure 4.2.: Graphical representation of structure from code snippet in Listing 4.1.

But what if the modeled node had, say, 100 attributes, half of which are other elements with attributes of their own and several nodes would be needed for a given scenario? Being able to model elements at a high level of detail as to create realistic scenarios, raises the question of how to reuse such detailed models. While the process of describing elements is straightforward, repeating descriptions over and over again to create a mul-

```
// init-instruction naming four expected parameters
init(TYPE,APPS,CORES,CACHE);

// upon usage, names are enclosed in $-signs
string type=$TYPE$;

// create os element
os os;

// this assignment expects APPS to be a list
application[] os.apps=$APPS$;

// templates can directly define subordinate elements and specify
    their attributes
processor cpu;
int cpu.cores=$CORES$;
int cpu.cacheSize=$CACHE$;
```

Listing 4.2: Example template file to instantiate a simple node

titude of similar elements is highly inefficient, prone to errors and, additionally, bloats the scenario beyond comprehension. This jeopardizes at least one of the aforementioned advantages that a dedicated modeling language should deliver. This is where templates come into play. A simple `new`-statement allows to instantiate previously defined elements of arbitrary complexity that can be customized upon creation by providing respective parameters to the constructor, just like classes in object-oriented programming. Getting back to the example of a node, the necessary specification could be saved in a template file (e.g. `node.template`) that follows the exact same syntax as introduced before. However, to enable the processing of variable parameters that are passed on when calling `new`, the template starts with a special `init`-statement, specifying how many parameters it expects and under which reference they can be accessed throughout this template. A sample is depicted in Listing 4.2.

Note that the parameters' names used for referencing corresponding values are in all capital letters. From a technical perspective it is sufficient for these terms to only start with a capital letter, yet, for improved distinguishability, it is recommended to use all capital letters. These variables can then be used throughout the template to specify the shape of the newly created element by simply enclosing them in dollar-signs (`$`) and placing them where needed. Listing 4.3 illustrates how to use the keyword `new` in order to instantiate elements from the previously defined template. Note that the identifiers of elements that are defined within a template directly serve as attribute keys to reference them through their parent element in the same way as is done for attributes of primitive type. Therefore, they do not need to be globally unique as is required for identifiers used at the top level of the scenario definition, but only within the template. From a technical perspective, the translator takes care that newly created sub-elements still get a globally unique identifier as this is required by the simulation engine, yet not by the modeling language.

```
// the initial instructions are basically the same as before
application app1;
application app2;
application app3;

string app1.name="cron";
string app2.name="nginx";
string app3.name="word";

// element 'pc1' is instantiated from template and customizing
    parameters are passed on
node pc1=new node("server",[app1,app2],8,512);

// reuse of the template to instantiate another, different pc
node pc2=new node("client",[app3],4,512);
```

Listing 4.3: Instantiation call

To further extend the degree to which code is reused, instantiation of subordinate elements such as the CPU from the example in Listing 4.2 may, again, be externalized into additional templates. As a result, calling the `node` template could trigger calls to other templates, thus subsequently instantiating respective elements and using them as attributes. In this case, the code that is needed in the node template to instantiate a new CPU would be reduced to a single line. While the CPU's complexity is manageable in the given example, this will come in handy when reaching higher levels of detail.

Function Definition

Having introduced the basics of system modeling, the next step is to have a look at functions. By enabling potentially state-changing interaction with the system, functions are what makes simulation possible in the first place. However, before going into details on syntax, keywords, and operators, some important specifics of how functions work in the context of the framework and its modeling language should be addressed. In this framework, functions are not intended to perform arithmetic operations or data processing in general. Functions specify conditions that must be met to enable their execution, together with effects that may subsequently be applied as to change the system's state. In that sense, functions are not imperative but rather declarative in nature. Similar to a fine-grained SQL query, a function's set of conditions characterizes the shape of elements it may be applied to, yet does not specify how retrieval of such elements is implemented. The question on how this is done in practice is of no concern here and left to subsequent processing.

As is common in most languages, parameters can be passed on to functions. Assuming there is a function that expects a parameter of type `node`, for example, then providing a specific `node` element (i.e., through its identifier) upon function call will result in a check to determine whether or not related conditions are fulfilled for the respective `node` element. If so, the function call will return `true` along with a list of potential effects

to be applied to the system state. However, leaving the expected parameter "blank" by providing an uninitialized variable instead of an existing `node` element's identifier, the function operates as a generator that will output all nodes that meet its conditions, each one at a time. This is, in fact, one very important feature of functions within this framework and ultimately required by the simulation engine. Yet, for this to work, function parameters and defined conditions must not be independent. This will be covered in more detail later on in Chapter 4.5, where the simulation engine's functioning is explained.

While functions serve the purpose of allowing interaction with the system by modifying its state, they may as well have no state-changing effect. In that case, the evaluation of a function's conditions merely determines whether or not execution is possible. Causing no state changes, such functions do not directly contribute to the simulation's progress. Yet, they can be employed as "helpers" for other functions that may simply use them as additional conditions that are either fulfilled or not, thus further narrowing down the result set of the calling function. How functions work, is best illustrated with help of working examples. Starting with a simple function, the syntax will become clear very quickly and some fundamental operators will be explained.

Listing 4.4 presents the necessary code for a simple function that checks for operational nodes. Prior to definition, functions need to be declared in the same way as has been shown for elements in the previous section. In addition, functions must be equipped with certain attributes that determine how they are processed by the simulation engine, which is done analogous to the specification of attributes for elements. These comprise duration and success rate of both setup and execution of a specific function (i.e., `setupTime`, `setupProb`, `execTime` and `execProb`) as shown in lines four to seven in Listing 4.4. In the context of this framework, it is assumed that actions must be setup prior to initial execution. How long this setup takes and how likely it is to succeed is determined by respective numeric values. The idea behind this is to account for the different types of actions that may be modeled. Something like performing an HTTP-GET request does not pose a challenge and needs less preparation than crafting a zero-day exploit, for example, which is usually time-consuming and may even fail. However, once an action has been setup successfully, its performance is shaped by duration and success rate of execution, determined by two other values. Lastly, a fifth attribute determines whether or not a function is `public`. If so, it can be called by actors depending on the fulfillment of its conditions. On the other hand, should a function not be `public`, it can only be called from within other functions, thus explicitly turning it into a helper function. The first half of Listing 4.4 covers the function's declaration and the specification of said attributes that have been chosen to cost minimal time (i.e. one round) in setup and execution while exhibiting a success rate of 100 % in both regards. The function definition starts in line ten. The conditions being checked here are made up to serve the purpose of this example and may look different in a more sophisticated scenario. Having no effect, this function is what has previously been called a "helper" and was chosen for the simplicity that results from the absence of effects.

The first thing to note is that a function definition always starts with the function name that must begin with a lowercase letter. This name has to be unique since functions are

```
1 // necessary function declaration
2 function operationalNode;
3 boolean operationalNode.public="true";
4 int operationalNode.setupTime=1;
5 int operationalNode.setupProb=100;
6 int operationalNode.execTime=1;
7 int operationalNode.execProb=100;
8
9 //function definition
10 operationalNode(node PC)
11 {
12   PC.powered=="true" &&
13   PC.interfaces==INTERFACE &&
14   INTERFACE.state=="connected"
15 };
```

Listing 4.4: Definition of a simple helper function that does not cause any state changes.

globally accessible irrespective of where (i.e., in which template or even the main scenario file) they have been defined. Following the function name, enclosed in parentheses, aforementioned parameters are given. These are represented as pairs with the first term specifying the parameter's type, and the second term specifying the parameter's name for reference throughout the function's body. Should there be more than one parameter, these pairs are simply separated by commas (`,`). It should be noted that the parameters' names are written in all capital letters that, analogous to the parameters processed upon template call, mark them as variables. However, from a technical perspective, it would be sufficient for parameters to only begin with a capital letter to be identified as variables. Together, function name and parameter list form the function head which is followed by the function body that, enclosed in curly brackets, contains all conditions and potential effects. Processing-wise, a function is handled as a large instruction, which is why it must be terminated with a semicolon (`;`), just like instructions that are used to describe the system state.

Inside the body of the function depicted in Listing 4.4, there are different conditions that a node must fulfill in order to be considered operational within the scope of this example, with line 12 simply stating that a node must be `powered`. Whether or not this is given, shall be determined by comparing the value of the primitive attribute `powered` of the corresponding `node` element referred to as `PC` (i.e. the parameter name from the function head) with a provided reference value. To do so, the comparison operator `==` is used. The example gets interesting with lines 13 and 14, where another variable named `INTERFACE` is introduced that can be identified as such for being spelled in capital letters only. The intention behind these two lines is to check if the node in question has an interface assigned to it and whether or not this interface's state is `connected`. Just like in the previous code samples, interfaces are subordinate elements of nodes that have attributes themselves, and one of these is being checked here to make for a simple example. Remember that elements, including the subordinate interface, have identifiers and that if an element is assigned to be the attribute of another element, it is technically

```
operationalNode(node PC)
{
  PC.powered=="true" &&
  PC.interfaces.state=="connected"
};
```

Listing 4.5: Definition of a simple helper function that does not cause any state changes in an alternative notation.

the subordinate element's identifier that is set to be the attribute value, thus basically serving as a pointer. This has been shown in Listing 4.1, line 14, for example. However, in the context of checking whether or not a node is operational, the exact `interface` element assigned to a specific `node` element is of no interest. What matters is that there is one interface, any interface, assigned to this node and that its state is indeed `connected`. Hence, a variable is used to represent whatever interface is "found" at `PC.interfaces`.

However, the ability to chain attributes through connecting dots (`.`) known from the introduction to state descriptions also exists for functions. This means, instead of referencing the potentially existing interface with help of the variable `INTERFACE` to access its attribute `state`, one may as well point directly to said sub-attribute. An adapted version of function `operationalNode()` is depicted in Listing 4.5. Depending on how many attributes of said `interface` need to be accessed, the one or the other notation may result in cleaner code and support structuring.

Irrespective of the chosen notation, in the given example, all specified conditions must be met for the node to be considered operational. This is why they are combined through two ampersand symbols `&&`, representing the `AND`-operator. A list of all basic operators that can be used within functions is given in Table 4.1, together with their descriptions, as well as examples that illustrate usage. There may be cases, in which it is necessary to group combinations of conditions as to determine the order in which they are resolved. By default, `AND` has a higher precedence than `OR`, meaning that expressions like `A && B || C && D` are implicitly treated as `(A && B) || (C && D)`. Using parentheses, this behavior can be altered, which is especially useful when offering alternative sets of conditions to decide whether a function can be executed or not.

Having understood this simple example, more complex functions shall be considered that are not limited to using comparison operators and may also have effects. For this purpose the model is extended to also include the ability to remotely start a computer. The corresponding function will illustrate two important things: utilization of helper functions and specification of effects. Listing 4.6 depicts the respective code snippet.

```
wakeOnLan(node PC)
{
  operationalNode(PC) &&
  ! ( PC.running == "true" )
  set boolean PC.running="true"
};
```

Listing 4.6: Definition of a simple effective function that employs a helper function.

Table 4.1.: Operators to be used within functions.

Operator	Symbol	Usage	Description
`AND`	&&	`c1 && c2`	Logical `AND` requiring both conditions `c1` and `c2` to be fulfilled as to return `true` for the whole expression.
`OR`	||	`c1 || c2`	Logical `OR` requiring either condition `c1` or `c2` or both to be fulfilled as to return `true` for the whole expression.
`EQUALS`	==	`v1 == v2`	Compares values `v1` and `v2` for equality and, if so, returns `true`. Instead of specific values, references may be used, as well.
`NOT`	!	`! (c1)`	Negates the result of condition `c1`, so that expression evaluates to `true` if `c1` is `false`.

While still being a relatively simple example, the function `wakeOnLan()` shows that existing functionality can easily be reused by calling `operationalNode()` in line three. This provides an easy way to encapsulate frequently required conditions and include them with one simple line of code. Apart from the checks that are performed in the course of calling `operationalNode()`, to "start the node" it is also necessary that the node is not already running. Line four inserts the respective condition that also illustrates on how the `NOT`-operator is used. Of course, the condition may equally be changed to `PC.running=="false"` to achieve the same result without using the `NOT`-operator. However, this would require that the state of `PC.running` is maintained in any case, that is, also when the node is not running. Yet, depending on the meta model, the attribute `running` might only come into existence upon calling `wakeOnLan()` or any other function to power on the node. In that case, there would be no previous state in which `PC.running=="false"` would ever be fulfilled so that using the `NOT`-operator is obviously mandatory. Line five finally specifies the function's effect before the function body is closed. Note that this statement starts with the keyword `set`, followed by the word `boolean` to indicate the type of the information that is being set. The third part of the statement specifies the information that is supposed to be persisted. In this case, the value `true` shall be assigned to the attribute `running` of the respective node referenced through `PC`. Variables that are used in the course of specifying conditions can evenhandedly be used when describing the function's effects. Functions can have an arbitrary number of effects that are simply separated through line breaks or white spaces.

Apart from setting, there is also the possibility to add or delete information, using `add` or `del` instead of `set`. The difference between setting and adding is that `set` will overwrite previously stored information that could have been referenced through `PC.running`, whereas `add` does not interfere with any existing value and simply appends the specified information. However, if using `set` in a context where there is no previous value to be overwritten, it will automatically behave like `add`. Consequently, repeated

adding turns singletons into lists. Deleting works the other way around and removes the data point specified in the third part of the statement if the type fits. With help of these modifiers, functions may alter the system state thus potentially affecting the different actors' subsequent options to interact with the system. No matter which type of modification is used, the syntax is exactly the same as introduced with the example of `set`. A compact overview that summarizes the three modifiers is given in Table 4.2, together with examples and descriptions.

Note that setting, adding or deleting can only be done for singletons. That means, the information right of the equal sign of any such instruction must not be a list instance. As a result, an instruction like `add application os1.apps=app1` cannot be adapted to `add application os1.apps=[app1,app2,app3]` in order to assign several applications at once. Instead, either one `add` instruction per application instance is needed, or a variable must be used in such assignments, whose specific value results from another instruction that serves as a generator and outputs singletons. This causes said variable to take different single values during unification, thus resulting in an individual `add`-instruction for each instance of `APP`. An example of how this works is given in Listing 4.7 that simply removes all applications from `PC` for which the attribute `updated` is not `true`.

This section gave an introduction to the definition of functions, focusing on their general structure and syntactical aspects that must be adhered to. Also, and probably most importantly, the application of effects has been presented that is integral to interaction simulation. However, despite being sufficient for some illustrative examples, the conditions to determine when and how functions may be executed that have been presented so far, are yet limiting.

Table 4.2.: Modifiers to specify function effects.

Modifier	Example	Description
`set`	`set boolean PC.running="true"`	The `set`-modifier writes the given value to the specified attribute in a way such that any previously defined value is overwritten.
`add`	`add application OS.apps=app3`	The `add`-modifier writes the given value to the specified attribute in a way such that it is appended to potentially existing values. In case of one existing value, this modifier turns singletons into lists.
`del`	`del application OS.apps=app3`	The `del`-modifier removes the given value from the attribute. When the last value assigned to an attribute is being removed, the whole attribute is removed from the model since values are conditional to their existence. Using an uninitialized variable (e.g. `VAR`) instead of a specific value (i.e. `app3` in the given example) when deleting, causes all values to be removed at once, thus removing the whole attribute.

```
removeOutdatedApps(node PC)
{
  PC.os.apps == APP &&
  ! ( APP.updated == "true" )

  del application PC.os.apps=APP
};
```

Listing 4.7: Assigning multiple values.

Built-in Functors

While using `AND` and `OR` to combine simple checks for equality (or inequality) of attributes may get one further than expected, detailed interaction modeling may require a little more. For this purpose, the language has been equipped with functors that can be used within function definitions, primarily to refine conditions. They implement mechanisms beyond the abilities of the operators presented in Table 4.1, thus allowing for more fine-grained functions. In the following, purpose and usage of these functors is explained:

- `isInList(x,y)` checks whether or not `x` is in the list that is referenced through `y`. The shape of `y` is that of an attribute reference such as `os1.apps` from line 19 of Listing 4.1 which points to a list of values. Whether the element whose attributes are being referenced here is explicitly given through an identifier (e.g. `os1`) or instead by using a variable, does not matter. Also, as was the case with previous attribute references, it is possible to reference attributes that are deeper within a hierarchy of elements by chaining attribute keys using dots (`.`) as delimiters. However, `x` is either a specific value or a variable that may hold a specific value. It cannot be instantiated as a reference to some element's attribute (e.g. `app1.name`). Should this functor be used with `x` being an uninitialized variable, it will serve as a generator, repeatedly yielding one specific instance of `x` for every value referred to through `y`.

- `isOfType(x,y)` checks whether or not `y` is of type `x`. This functor is only applicable to elements and not primitive types such as `string` or `int`. Whereas `x` must be specified, `y` may either be an element's identifier or a variable referring to an element. Should this functor be used with an uninitialized variable serving as `y`, then it will work as a generator and return one instance at a time of said variable for each element within the model that is of the specified type.

- `findAllWhere(x,(conditions))` creates a list of all elements `x` that meet the specified `conditions`. This functor must be used in combination with an assignment like `LIST=findAllWhere(X,(isOfType(application,X)))`, specifying a variable that is supposed to hold the generated list. Any conditions that can be used throughout functions and exhibit the aforementioned generator property can be used as `conditions` in the sense of this functor. These comprise, for example, the checks for equality (or inequality), as well as the two previously mentioned

functors. Just like in functions, these conditions can be combined through the `AND` and `OR` operators. Note that this functor does not generate single results one after another but a list instance holding several values at once. Given that the application of effects always concerns singletons, such a list can only be an intermediate result of the function it is generated in and will require further processing.

- `pickFromList(x)` will randomly pick one item from list `x`. This functor must also be used in an assignment that specifies the variable which is supposed to hold the value picked from `x`. An example building upon the result from the previous functor `findAllWhere()` could look like `VALUE=pickFromList(LIST)`. Note that this functor expects `x` to be a list instance and not a reference to an attribute which may hold several values. Furthermore, from a technical perspective, this could be a generator, yielding one of the items enclosed in `x` at a time. However, the functor is implemented such that it only yields one of the enclosed items that has been picked randomly.

- `isMember(x,y)` is similar to the previously introduced `isInList()`-functor in that it checks whether or not `x` is in list `y`. However, in this case, `y` is not a reference to an attribute that may hold a list of values but instead is a list instance as the ones resulting from using the `findAllWhere()`-functor. Assuming that `x` has been specified, this functor will return `true` or `false`, depending on `x`'s presence in `y`. Yet, should `x` be an uninitialized variable, it will serve as a generator, yielding one result per element in `y`.

- `append(x,y)` appends element `x` to list `y` and returns a new list instance, which is why this functor must be used in an assignment (e.g. `NEWLIST=append(X,LIST)`). Again, note that `y` is not a reference to an attribute holding several values but a list instance as resulting from `findAllWhere()`, for example.

- `concat(x,y)` concatenates two strings `x` and `y` and returns the result which must be assigned to a new variable as in `WORD=concat("Hello","World")`. Both `x` and `y` may either be explicit strings or variables that hold strings. Attribute references cannot be used as arguments. Note that this functor may cause unintended behavior if used with uninitialized variables for yielding an infinite number of results.

- `len(x)` returns the length of `x` that must be a list instance. This functor must be used in an assignment like `LEN=len(LIST)` to store the returned value.

- `getHash(x)` is used in an assignment such as `HASH=getHash("password")` to generate an md5 hash from `x` and store it in `HASH`. The practical use of this functor is limited except for delivering checksums to easily detect manipulation of attributes that may have been caused by previous state-changing actions.

Obviously, the first several functors are more sophisticated than `len()` or `concat()`, for example. Therefore, Listing 4.8 presents a minimal working example that utilizes the initial four functors `isInList()`, `isOfType()`, `findAllWhere()`, and `pickFromList()`.

```
1 // create node
2 node pc1;
3 // create os
4 os os1;
5 string os1.type="client";
6 // assign os to node
7 os pc1.os=os1;
8 // create pool of applications, assuming a template exists
9 application pc1.os.apps=new application("MailClient");
10 application pc1.os.apps=new application("WordProcessor");
11 application pc1.os.apps=new application("Browser");
12 //declare function - omitting timings here
13 function deleteRandomApp;
14 ...
15 deleteRandomApp(node PC){
16   isOfType(node,PC) &&
17   APPS=findAllWhere(X,(isInList(X,PC.os.apps))) &&
18   APP=pickFromList(APPS)
19
20   del application PC.os.apps=APP
21 };
```

Listing 4.8: Minimal working example for usage of first functors `isInList()`, `isOfType()`, `findAllWhere()`, and `pickFromList()`.

Presenting how these may be used in practice should clarify how to employ the remaining ones. To better illustrate what function `deleteRandomApp()` does, the related state description it operates on is also provided. Obligatory function attributes (line 14) have been omitted, though. Note that if the depicted function is used as a generator, and not called with a specific node instance as a parameter, the first condition of the function in line 16 serves to narrow down the space of elements from which to instantiate the variable `PC`. Considering that the given scenario only consists of five elements, this is not really necessary as the search space is small and there are no other elements exhibiting an attribute named `os` that, in turn, holds an attribute called apps. Yet, in larger scenarios this may increase performance.

4.4.2. Low-level Representation in Prolog

Considering that the previously introduced high-level language serves as an abstraction layer to simplify modeling and decouple it from whatever happens in subsequent processing, there must be a low-level representation that suits the simulation engine's requirements. Furthermore, there must be a way to transform the former into the latter. As has been mentioned before, this low-level representation is based on SWI-Prolog and a corresponding translator has been implemented, both of which will be introduced in the following. Processing-wise, it would now be the translator's turn to generate the required Prolog facts and rules. However, for the sake of didactics and to better reflect the actual development process, the low-level representation is introduced first.

Describing States With Fact Collections

Knowing what can be done with the high-level language, there are two requirements that appear to be of particular importance when it comes to the low-level representation:

- The low-level model must allow for the adequate description of all elements, their attributes and relations, as well as functions provided through high-level descriptions so that transformation does not cause any information loss.
- Low-level representations that adhere to this model must be automatically processable by the simulation engine irrespective of what has been modeled or any semantics.

In order to develop a suitable model, these requirements must be broken down, resulting in subordinate tasks. Deciding on a structure on how to represent the state description introduced in Chapter 4.4.1 appears to be a reasonable first step. The previous introduction to Prolog in Chapter 2.2 presented collections of facts as means to store information. Yet, while predicates and arity of such collections may usually be chosen freely, the need to ensure interoperability with the simulation engine and the intent to keep its complexity as low as possible, suggests to agree on a unified structure with only few predicates and fixed arity. From a structural point of view, a single predicate with arity three appears sufficient to represent elements with all their attributes. The predicate's first atom may serve as the element's identifier, the second one as attribute key and the third one as the corresponding value. Likewise, by using identifiers of existing elements as values, this allows to define relations to form trees or graphs.

The structure that has ultimately been employed is only slightly more complex. To also incorporate types, the previously suggested triple was extended to a quadruple, with `elementAttribute` serving as the respective predicate that was chosen freely. In addition, a second predicate with arity two has been introduced to solely store element identifiers and their respective types. This predicate is named `element`. Based on these, all state information from the high-level description can easily be represented. Listing 4.9 provides an example of how this looks in practice, depicting the Prolog representation of the high-level description from Listing 4.1 where different elements with only few attributes had been defined and related to each other. As illustrated, the first term of `element/2` specifies the respective element's type, whereas the second one holds the identifier through which it is referred. The four terms employed in `elementAttribute/4` represent element identifier, attribute type, attribute key and attribute value in this exact order. Please note that there is probably no limit to the number of alternative low-level representations that would make for an equally suitable solution so that this is not the only way to do it, but one of many.

While being considerably more extensive, the low-level representation's structure is relatively simple and straightforward. Predicate `elementAttribute/4` is fully capable of representing structures of related elements irrespective of their depth, it just needs more space to do so. Consequently, translating the one into the other does not pose any challenges. However, there is more to an adequate low-level representation, namely

```
element(application,app1).
element(application,app2).
element(os,os1).
element(node,pc1).
element(interface,if1).
element(processor,cpu1).
elementAttribute(app1,string,name,'cron').
elementAttribute(app2,string,name,'nginx').
elementAttribute(pc1,string,type,'server').
elementAttribute(pc1,os,os,os1).
elementAttribute(pc1,interface,interfaces,if1).
elementAttribute(app2,interface,bindingInterface,if1).
elementAttribute(os1,application,apps,app1).
elementAttribute(os1,application,apps,app2).
elementAttribute(cpu1,int,cores,8).
elementAttribute(pc1,processor,cpu,cpu1).
elementAttribute(cpu1,int,cacheSize,512).
```

Listing 4.9: Low-level representation of element declaration and value assignment

functions. In Prolog, there is no such thing as functions. Yet, as introduced in Chapter 2.2, there are rules and these are suitable to implement required functionality as will be shown in the following. However, finding a suitable representation that serves the intended purpose of functions is more challenging than it was for mere state descriptions.

Representing Functions Through Rules

Just like functions, rules can generally be divided into head and body. However, to adequately represent functions as introduced in Chapter 4.4.1, a rule's head is considerably more complicated than the corresponding function head, whereas a rule's body merely contains respective conditions. Hence, for better comprehension, these two parts of rules will be introduced separately, starting off with the rule head.

To ensure that the simulation engine is able to determine all actions, decide upon their feasibility and ultimately perform them, a defined structure is needed. Given the fact, that all functions must be declared as such, the simulation engine can easily determine all functions and even distinguish between those that are publicly available and those that are not by checking their attribute `public`. To provide a unified way of calling all functions, a dedicated predicate with arity four has been introduced that is `eval/4` and characterizes the default rule head. The number of parameters a function may have is not reflected by the predicate's arity since these are enclosed in a list that is treated as only one of the four terms. Would the arity be dependent on a function's number of parameters, the simulation engine would have to count these first to determine the correct predicate. While this is definitely possible, putting all parameters into a list treated as one compound term appeared to be more convenient.

The first term of predicate `eval/4` specifies the function name as its sole identifier, which is why these names must be unique. Furthermore, the simulation engine only

Table 4.3.: Overview of terms used in the default rule head.

Term	Description	State
# 1	Holds the unique function name.	Must be specified, which is done automatically by the simulation engine, based on available functions.
# 2	Used for the current actor.	Must also be specified, which is done by the simulation engine, based on a strict order in which actors take turns.
# 3	List of all function parameters.	Uninitialized when determining all available options. Specific values provided upon execution attempt which is timed by simulation engine.
# 4	Holds result that is either 0 or effect quadruple.	Always kept uninitialized to retrieve result from rule evaluation based on specific combination of function and parameters.

expects declared functions, and nothing else, to be valid first terms, implying that declared names and names from function definitions must match. The second term is used to provide the identifier of the actor on whose behalf a function is called. Even though the actor could be provided through one of the function's parameters, passing it through a dedicated term is convenient for two reasons. First, the simulation engine has no semantic understanding of a function's parameters, so presuming that any actor used in the parameters should be the currently active actor may cause unintended behavior. Second, as has been outlined before, functions may be used as generators to determine valid parameters in the first place. In fact, this is of utmost importance for the simulation engine when determining viable next steps. Therefore, during simulation, parameters will frequently be kept uninitialized, while the actor always has to be specified. Otherwise, an action may be deemed feasible just because any actor can call the corresponding function. The aforementioned list of function parameters serves as the third term of `eval/4`. Depending on the simulation engine's state, this term is either uninitialized to determine possible options for a given function, or initialized to check if conditions are still fulfilled before respective effects are applied. The final term in the rule's head is reserved for its "results" that relate to a function's effects as described in Chapter 4.4.1. This term may either hold the integer `0` in case of helper functions, or a quadruple specifying one effect of said function. Should a function have more than one effect, there will be several instances of the corresponding rule, one for each effect. During simulation, all of these will be evaluated to determine all effects that need to be applied. Yet, this does not cause any additional computational effort, as will be explained in Chapter 4.5, but avoids further complication of the rule head and simplifies processing by the simulation engine. A summary of all four terms is given in Table 4.3. To better illustrate the previous outline, Listing 4.10 shows the head of the rule that represents function `operationalNode(node PC)` from prior examples. For the sake of simplicity, placeholders are employed where usually conditions would be in the rule's body. These will be covered later on.

```
eval(operationalNode,ACTOR,[PC],0):-
  <condition 1>,
  ...
  <condition n>.
```

Listing 4.10: Rule head for function `operationalNode()`

It should be noted that while Prolog comes with built-in predicates that allow to set/add (assert) and delete (retract) information upon rule processing, instantaneous effect application is not desirable so that these predicates are not part of any rules. Not only would such behavior render separately defined chances and timings obsolete that have been set as additional attributes, it would also make it impossible to check if a function's conditions are fulfilled without directly applying its effects. In consequence, effects are better returned as results of unification during rule evaluation to have the simulation engine take care of their processing.

As opposed to the rule head, the body is comparatively simple for only comprising the conditions that decide whether or not a corresponding action can be performed. However, the low-level representation of these conditions, which are technically speaking goals in Prolog, look a lot more complicated than their high-level counterparts. As was the case in the high-level language, these are combined through logical AND and OR operators whose precedence can easily be altered by using parentheses in equal measure. The conditions and functors that can be used in the modeling language have counterparts in the low-level representation. Some of these are simple queries of the predicates used to describe the system state that have previously been introduced, others employing built-in predicates and predefined rules. To understand the low-level representations of conditions presented in the following, some basic Prolog operators from Chapter 2.2 should be remembered. In Prolog, a simple comma (,) represents a logical AND, whereas a semicolon (;) expresses a logical OR. The NOT operator that is referred to as an exclamation mark (!) in the modeling language, is represented through `\+`. A dot (.) indicates any statement's end, that includes both facts and rules as a whole.

In the low-level representation, checks for equality and inequality, as well as the functors `isInList()` and `isOfType()` from Chapter 4.4.1 are realized as mere queries of the predicates `element/2` and `elementAttribute/4` and do not require any sophisticated functionality. In that sense, checking whether a node referred to as `PC` is really `powered` as expressed by condition `PC.powered=="true"` from the `operationalNode()`-example is represented as `elementAttribute(PC,boolean,powered,'true')`. When processed by Prolog, this query will determine whether or not a given node instance meets this condition or, when operating as a generator, reveal all instances of `PC` that do. Depending on the degree to which attribute keys are concatenated to refer to attributes of subordinate elements as in `PC.interface.state=="connected"`, for example, such queries must be extended to check on chains of element attributes. In the same way, `isOfType()` is represented by queries on the predicate `element/2`. The rule corresponding to function `operationalNode()`, which already served to illustrate the structure of

```
1 eval(operationalNode,ACTOR,[PC],0):-
2   (element(node,PC)),
3   (elementAttribute(PC,boolean,powered,'true')),
4   (elementAttribute(PC,interface,interfaces,RAND0829),
5   elementAttribute(RAND0829,string,state,'connected')).
```

Listing 4.11: Example of a rule solely relying on querying the fact collection

rule heads, gets by with these simple queries. To make for a better example, assume that the representation of `operationalNode()` as given in Listing 4.5 is augmented with a check to determine whether the element referred to as `PC` really is of type `node` so that the low-level representation also contains a query to `element/2`. This is simply done by inserting the additional condition `isOfType(node,PC)` at the beginning of the high-level definition. Listing 4.11 depicts the corresponding rule in Prolog.

As can be seen, all conditions are queries to the fact collection represented through predicates `element/2` and `elementAttribute/4`, not requiring any deductive reasoning but only unification to be evaluated. These queries are combined through commas (`,`) representing the logical `AND` operator. The fact that they are all enclosed in parentheses which are used to determine precedence, has no impact in the given example. However, it is the translator's standard procedure to enclose goals that result from conditions and functors in the high-level language in parentheses to mitigate any ambiguity considering evaluation order, especially when low-level representations consist of numerous statements. Lines five and six are noteworthy since these, together, represent condition `PC.interfaces.state=="connected"` that makes use of concatenation to refer to attributes at a deeper level. Considering the previous explanation on how information in the predicate `elementAttribute/4` is structured, there is no way that querying a single entry can represent such a condition. Therefore, queries must be composed to check on related entries in `elementAttribute/4`. To accomplish this, unique variables are used to establish links between elements and the attributes of their subordinate elements. Through unification, Prolog will take care to find a match, if possible. Otherwise this goal cannot be satisfied for the current instance of `PC` so that the whole function either fails or another `PC` instance has to be checked.

Other conditions that may be used in the high-level function definition (i.e., self-defined functions and functors other than `isOfType()` and `isInList()`), do not solely rely on fact queries. While calls to self-defined functions are simply represented by incorporating the related rule, the other high-level functors rely on built-in predicates from Prolog or pre-defined rules that are inherent to the simulation engine. Table 4.4 gives a comprehensive overview of these and aligns them with their corresponding high-level representation. For completeness sake, previously explained fact queries are included here, as well.

Table 4.4.: Overview of high-level functors and conditions together with their low-level representations. Note that from a technical perspective, the `append`-functor causes items to be prepended. However, the order of lists is not relevant to any of the functors or operators.

High-level condition/functor	**Low-level representation**
PC.powered==”true”	elementAttribute(PC,boolean,powered,'true')
! (PC.powered==”true”)	\+(elementAttribute(PC,boolean,powered,'true'))
PC.interfaces.state==”connected”	(elementAttribute(PC,interface,interfaces,RANDXYZ), elementAttribute(RANDXYZ,string,state,'connected'))
isInList(APP,pc1.apps)	elementAttribute(pc1,application,apps,APP)
isOfType(node,PC)	element(node,PC)
LIST=findAllWhere(X,(goals))	findall(X,(goals),LIST)
ITEM=pickFromList(LIST)	choose(LIST,ITEM)
isMember(ITEM,LIST)	member(ITEM,LIST)
NEW=append(ITEM,LIST)	NEW==[ITEM|LIST]
WORD=concat(”one”,”two”)	atom_concat(”one”,”two”,WORD)
LEN=len(LIST)	length(LIST,LEN)
HASH=getHash(VALUE)	md5_hash(VALUE,HASH,[])

4.4.3. Translator

The translator is responsible to transform the high-level definitions written in the modeling language into low-level representations in Prolog. Previous examples of state and interaction descriptions in both representations illustrate that this can generally be done on a per instruction basis, only requiring contextual information on few occasions. This is the case when processing templates, for example.

The translator is implemented in Python and relies on the Parsimonious library [54] from Erik Rose, a parser which is based on parsing expression grammars (PEG). Processing of all files that need to be translated is done in a multi-stepped approach to resolve different statements and instructions in a convenient order. Furthermore, translation is done recursively to process templates at any depth of the described structure, creating an instance of the main `worker` for every element instantiation triggered by the `new` keyword followed by the respective template name. Instances of the `worker` class process complete files in consideration of provided contextual information which may comprise parameters for the `init`-statement of a template, as well as the identifier of the parent element from which instantiation of the subordinate element has been initiated. The first `worker` instance that processes the main scenario definition file is the only one operating without contextual information since this file represents the scenario's root.

To translate a scenario definition, the only mandatory input is the definition file itself. This, however, assumes that the corresponding library where potentially referenced tem-

```
elementdef = ws type ws identifier ws eos
ws = " "*
type = lowercase alphanum*
identifier = lowercase alphanum*
eos = ws ";" ws
lowercase = ~"[a-z]"
uppercase = ~"[A-Z]"
alphanum = digit / letter
digit = ~"[0-9]"
letter = uppercase / lowercase
```

Listing 4.12: EBNF extract for parsing element declarations

plates are stored, is in the same location as the definition. Should deviating component libraries be used, these must be specified through providing path information.

General Working Mode

The parsing of any files is preceded by completely loading the file into memory. On this occasion, comments may already be filtered out and possibly included snippets be integrated accordingly. These snippets represent externalized code that is stored outside the main scenario definition. This may be useful to maintain function properties such as timing and success probabilities separately. Once the file has been fully loaded, the parser evaluates it with help of a grammar that specifies symbols and defines the patterns that are needed to recognize these. Defined symbols can be used to describe more complex symbols thus avoiding redundant definition. Said grammar is formally described in extended Backus-Naur form (EBNF) [16, 133] that must cover all symbols the parser might encounter. To give a simple example, a grammar may state that the symbol `word` is defined as a continuous sequence of characters ranging from `a` to `z`. The symbol `sentence`, in turn, may be defined as a sequence of symbols of type `word` that are delimited by white spaces, ending with a full stop. Based on this grammar, the parser would be able to recognize and dissect sentences and words, as long as the analyzed text does not contain capital letters, numbers or any other symbol that has not been defined. For better comprehension, an extract of the EBNF that is used to process element declarations is depicted in Listing 4.12. While this is only a simple example, it conveys the notion of how grammar definition works. Note that this extract contains unnecessary redundancy with regard to symbols `type` and `identifier` which, however, improves readability in this example.

If the provided document has been parsed successfully, an abstract syntax tree (AST) is generated. This is a hierarchical representation of all identified symbols and how they are related to allow for subsequent processing. This tree is traversed bottom up by a node visitor, starting with the leaves. In this course, identified symbols are processed, transformed if needed, and returned to their parent node. Since specific values that are used within conditions are the same for both high-level and low-level representations on many occasions, the need for transforming is limited to nodes in the AST that, together

with their child nodes, represent complete expressions. Considering this, the single values of a condition that serves to perform comparison for equality do not need to be transformed, for example. Instead, the node that handles processing of the condition as a whole will simply take the values from the child nodes and assemble the corresponding expression for the low-level representation. This, in turn, will be returned to the next higher node in the AST for further processing. To better illustrate this, Figure 4.3 depicts a simplified extract from an AST where the focus is on the functor `isOfType()`, indicating which operation is performed at what point during traversal.

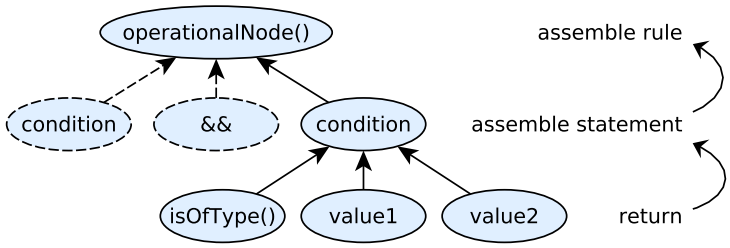


Figure 4.3.: Traversal of the AST to translate high-level to low-level representation.

Multi-stepped Processing of Files

The outlined process of parsing and transforming of high-level scenario definitions is divided into distinct steps to incorporate contextual information where needed and allow for pre-processing of certain instructions. This pre-processing concerns included snippet files, for example. To avoid parsing and interpreting yet another instruction, pre-processing takes care of dereferencing included snippets and incorporates their contents as if reading one continuous file. Whenever elements are instantiated from templates, contextual information is required to correctly process enclosed instructions and assign the resulting element and its attributes accordingly. However, the parsimonious library can only process context-free grammars (CFG). While this comes with benefits as the utilized EBNF is comparatively user-friendly, it impedes the building and processing of one overarching AST. In consequence, transforming high-level to low-level representation in one go is hardly feasible. To circumvent this, the aforementioned recursive instantiation of `workers` has been implemented, allowing to process templates in the context that they have been called in.

Once all referenced snippets and templates have been processed in their respective contexts, the translator generates an intermediate representation of the scenario. This is an extensive monolith which is still based on the modeling language, yet independent from all previously referenced content. It contains all required information as if snippets had not been externalized and all elements were defined manually without resorting to templates. Based on this intermediate state, translation can be done truly context-free since all calls to templates from the different levels in the scenario's hierarchy, and the variable parameters they employ, have already been resolved.

4.5. Simulation Engine

On the basis of translated scenarios, the simulation engine takes over, fulfilling the primary purpose of this framework: controlling interaction simulation of all involved actors and applying resulting state changes accordingly. Yet, furthermore, the simulation engine is also responsible to log the progress of the different actors for subsequent evaluation. The simulator's general mode of operation, as well as the representation of progress are covered in the following. Finally, some details with regard to its implementation are addressed.

4.5.1. Operational Mode

The simulation of attack and defense is performed in an event-discrete fashion with time represented through rounds for which the simulation engine maintains a counter representing the time-wise progression of the simulation. Consequently, all public attacker and defender actions take a certain amount of rounds to execute, serving to put the different kinds of actions into perspective and make simple tasks consume less "time" (i.e., rounds) than complicated ones. Furthermore, actions are characterized by individual success probabilities to account for the possibility of complex tasks to fail. Before any of these actions can be used effectively, though, they must initially be set up which takes additional rounds and is subject to yet another probability. However, this setup is only needed once per action and actor. The intention behind this is to reflect time and effort involved in preparing specific actions such as exploits, as well as the chances that this ultimately fails and related effort was in vain. Legitimate actions, on the other hand, that represent intended interaction with systems and software may have setup times as low as one round, succeeding with a chance as high as 100%. The attributes of a function that account for these probabilities and durations have already been introduced in the context of defining functions in the modeling language in Chapter 4.4.1.

All actors have the opportunity to start actions once per round, one after another in a fixed order. This order is determined through the order in which actors are defined in the scenario. Once it is a given actor's turn, she may initiate as many actions as the current system state allows for. Which actions are feasible and can therefore be initiated is determined with help of unification. For a given actor and the known list of generally available actions, the simulator checks if, and for which input parameters, a Prolog rule evaluates to `true`. A list of ongoing actions is maintained for each actor, keeping track of which actions have been started when and in which round they are supposed to be finished. In the beginning of every new round, these lists are checked by the simulation engine. This is not only done to check for due actions, but verify that all ongoing actions are still generally feasible and in line with the current system state. Should actions have become impossible due to one or more previous state changes, the simulation engine will detect this and abort respective actions. This is the case when a defense may have moved a target out of the attacker's reach, for example, but also when the system detects that actions would cause a state change that is already present. While this may sound abstract, a simple example would be if the attacker engages in

different actions that all yield the same effect and one of them succeeds earlier. In this case, redundant actions are terminated. When checking all ongoing actions as described, the simulator will ultimately come across due actions, once in a while. If these are still feasible, the simulator will attempt to execute them. The order of attempting execution is the same as the order in which these have been enlisted. Whether or not this attempt is successful depends on the respective action's probability. A dice roll is used to decide whether the action was successful or not and, if so, the state is modified accordingly. This processing of queued setups and executions is depicted in Algorithm 1.

As has been mentioned before, to represent functions with multiple effects, one instance of the corresponding low-level rule exists for any single effect. This is intended to keep rule heads simple and not maintain lists of lists in the fourth term of `eval/4`. Except for the respective effect, these rules are absolutely identical. While one might think that this causes additional load for repeated evaluation of the same rule, a convenient feature named tabling takes care of this. With tabling enabled, goals in Prolog are not re-evaluated but looked up if they have been evaluated before. This way, no further computational effort is needed so that evaluating multiple versions of the same rule does not slow down simulation. Furthermore, values that can be looked up also accelerate evaluation of other rules if required goals are the same. To make sure that this tabling does not interfere with the simulation engine's task to determine feasible actions on the basis of the current state, generated look-up tables are deleted after every successful state change. Once checking and attempted execution of due actions is done, the actors may start over with enlisting actions.

With regard to attacker behavior, the simulation engine employs a greedy attacker that tries to perform all available attack actions in parallel. Hence, once an attack action becomes available (i.e., the action's requirements are met), the attacker will start it. But once an action has been started, the same action cannot be initiated with the same parameters again, as long as it is in the list of ongoing attacker actions. For example, an attacker can start a phishing attack against five different targets in a single round. However, once she has started a phishing attack against a target, the attacker has to wait until this phishing attack was either successful, failed or was defended, before launching another phishing attack against the same target. But if learning of a new target, the attacker is free to start a phishing attack against this new target any time.

4.5.2. Measuring and Logging Progress

The attackers' progress is measured in form of revenue which is generated through compromising certain resources that have been assigned a specific value. This value can be freely chosen on a per resource basis. The simulation engine will extract this value from any resource that is equipped with an attribute named `externalValue` and assign it to the corresponding attacker, once she gets hold of it. For each round the simulation engine stores the generated revenue and relates the accumulated amount to the number of rounds that it took the attacker to reach it. Hence, the simulation engine does not report costs on a per action basis but counts the overall time till compromise and the maximum revenue generated in a given scenario. These two measures are not specific

Algorithm 1 Processing queue of actions in setup and execution

Require: $currentRound <= maxRounds$

```
1: procedure processActionQueue(currentRound)
2:     for actor ∈ ListOfActors do                          ▷ this is an ordered list
3:         for action ∈ actor.ongoingSetups do              ▷ check setups first
4:             if action.dueRound == currentRound then
5:                 randomNumber ← getRand(0, 1)             ▷ get rand between 0 and 1
6:                 if randomNumber <= action.setupProb then
7:                     actor.availableActions.append(action)
8:                 end if
9:                 actor.ongoingSetups.remove(action)
10:            end if
11:        end for
12:        for action ∈ actor.ongoingExecs do               ▷ check executions second
13:            fulfilled ← checkRequirements(action)        ▷ action requirements met?
14:            if fulfilled == True then
15:                if action.dueRound == currentRound then
16:                    randomNumber ← getRand(0, 1)
17:                    if randomNumber <= action.executionProb then
18:                        applyEffectsOf(action)           ▷ state changes incurred
19:                        actor.ongoingExecs.remove(action)
20:                    end if
21:                end if
22:            else                                         ▷ abort if requirements not met
23:                actor.ongoingExecs.remove(action)
24:            end if
25:        end for
26:    end for
27: end procedure
```

to any chosen defense and serve as comparators to measure the impact a defense has on attacker success. This attack strategy and metric is similar to the method used by P^{2}CySeMoL [63] and pwnPr3d [75]. The alternative would be to limit the number of actions an attacker can execute in parallel and assign costs to each action. However, this would require an intelligent attacker with a strategy that strives to make optimal decisions. Furthermore, these decisions would need to incorporate observable system behavior (i.e. noticeable effects of the currently employed defense) while still being subject to imperfect knowledge. Considering this, the attacker itself would become an influencing factor in the experiment so that evaluation results would not only reflect effects of the defense under test but also changes in attacker behavior, thus impeding fair comparison. In this regard, a greedy attacker is suitable for the specific purpose since all employed defenses "face" the same potent attacker, making comparison ultimately fair.

Apart from revenue, the simulator keeps track of all initiated, aborted, and successfully executed actions on a per round basis for each actor. For successful actions, the corresponding state changes are logged, as well. In addition to the quantitative analysis, these data allow for a qualitative analysis and enable the user of the framework to investigate how and why different states came about. While measuring and logging revenue is important in order to quantify the attacker's success under different conditions, the simulation engine itself does not require this and will operate irrespective of any such values being assigned to resources. Once the defined number of rounds to simulate has been reached, the simulation engine will export said data in form of CSV files for each actor. Furthermore, the current state of the Prolog instance will be saved for subsequent investigation, if needed, or simply to continue simulation at a later point in time.

4.5.3. Implementation

The simulation engine is based on Python (3.7) and SWI-Prolog [69], one of the most actively maintained open-source Prolog implementations that was initially released in 1987. To bridge these two, the PySwip library [137] is used that allows to perform queries to Prolog databases from within Python programs. To accomplish this, the Python-based part of the simulator maintains a dedicated Prolog instance that has been loaded with the scenario's low-level representation. For the sake of simplicity, Python takes care of all organizational aspects that are related to the simulation process. This includes keeping track of rounds, durations, and probabilities, but also logging all interaction, no matter if successful or not. Furthermore, Python also handles setup and execution queuing, and only resorts to Prolog to determine available actions, verify fulfillment of their requirements, and apply state changes, if necessary. This determining of available actions makes extensive use of previously introduced unification, allowing to simply query for all combinations of rules and respective parameters that evaluate to true at any given time for the current actor. The resulting list is what the calling Python script treats as executable actions. Compatibility is ensured through the unified structure of the low-level representation thus enabling Python to retrieve required information and perform all needed operations on the basis of only few predefined queries. As a result, the simulation engine is fully independent from what has been modeled and can equally be used to simulate interaction of any kind in any other domain.

4.6. A Unified Model for Networked Systems and Interaction

So far, this chapter gave a comprehensive introduction to the framework, focusing on the modeling language to describe scenarios and actions, as well as the simulator that processes these scenarios to determine the different actors' possible steps, apply respective state changes, and ultimately reveal how different conditions affected their progress. However, the preceding outline of the simulator's working mode revealed, that the simulator is fully agnostic to what has been modeled. This means that, apart from the necessity of defining actors and actions, there are no further requirements or support mechanisms ensuring that modeled actions are actually compatible with the shape of

the described system to perform simulation on. As a result, different attacks or defenses might not be executable and their effects never applied, simply for relying on conditions that consider types of elements and specific attributes that are not part of the model. To illustrate this, neither the modeling language nor the simulator dictate how a compromised host shall be represented. This could simply be done by equipping a host with a boolean variable such as `compromised` and setting it to either `true` or `false`, for example. However, this is just one way to represent compromisation of hosts with many others being equally plausible. How this is ultimately done, is of no importance at this point and highly dependent on what is being modeled and at which level of detail, emphasizing the framework's flexibility. What is important though, is to use element types and their attributes consistently across actions and state descriptions.

This is where the meta model comes into play, prescribing how the different elements that form scenarios should "look like" to be compatible with each other and meet the required level of detail to yield meaningful results in the course of simulation. While the definition of such a meta model is not necessary from a technical point of view, it simplifies modeling and ultimately helps to understand how scenarios implementing it are generally structured. Note that the meta model only covers the basic forms of elements and their attributes. In consequence, it includes the general structure of an application, for example, yet not specific instances thereof such as a potential web server or corresponding browser. These will be introduced in the course of the respective cases studies, where needed.

In general, the meta model can be designed freely and must comply with only few conventions that serve to avoid otherwise required configuration and allow for unobstructed processing. Remarks will be made on occasions where this is relevant. For better comprehension, elements of the meta model with all their attributes and relations are also visualized in a UML class diagram that is depicted in Figure 4.4. Considering that the modeling language allows to aggregate these into templates, which resembles the concept of classes from object-oriented programming, this representation appears convenient. Even though the model is comparatively simple, the textual description to elaborate on meaning and purpose of different elements and attributes may appear confusing at times, which is why repeated consultation of Figure 4.4 is advised to get a better understanding of the model. It should be noted that relations of parent to child elements are always depicted as aggregations and never as compositions. While it may seem more realistic to mark the relationship of node to OS as a composition, there is no such dependency on a technical level, as a respective OS will still exist even though its parent node may have been deleted.

4.6.1. Nodes, Operating Systems and Applications

These three types of elements are the building blocks of every scenario. In the modeled network, nodes represent all physical and virtual hosts which, in turn, are the basis for operating systems and applications that enable most of the interaction in scenarios. Nodes are comparatively simple. By default they are characterized by seven attributes, four of which are of type `string` referred to as `name`, `type`, `cpu`, and `hasVuln`. While

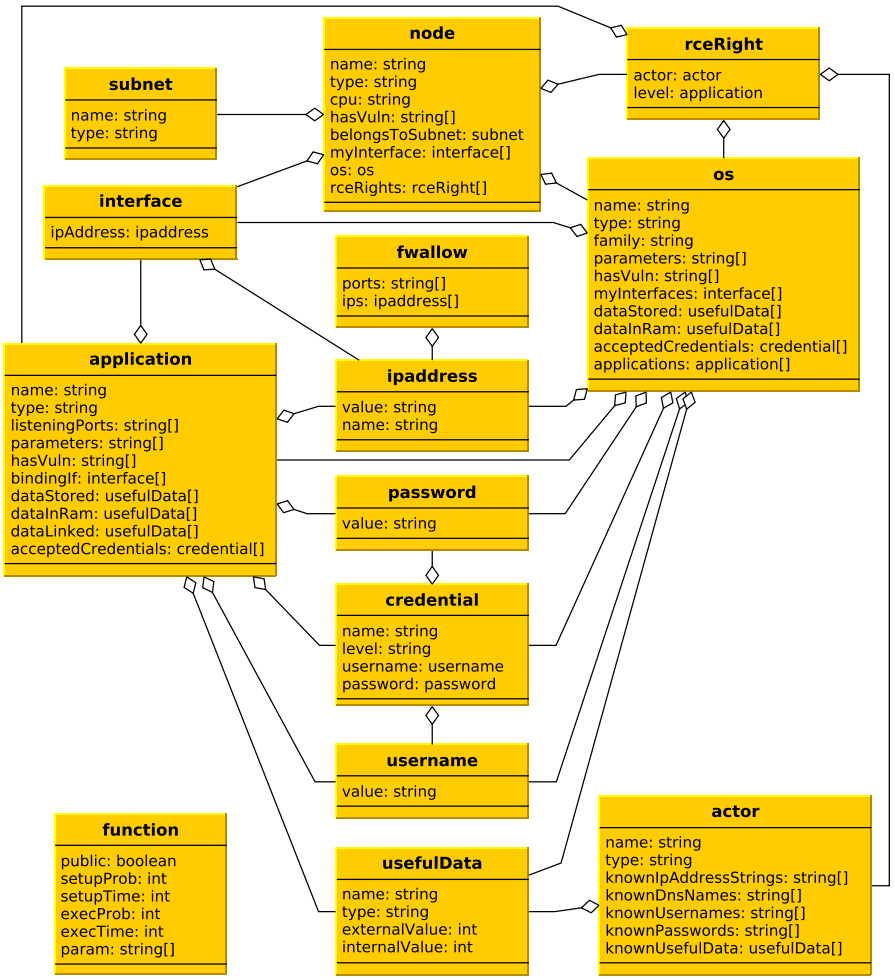


Figure 4.4.: Meta model for a detailed representation of network scenarios.

`name` only serves as a human readable identifier (as opposed to the identifier used to unambiguously refer to the `node` element itself), `type` can be used to differentiate between servers or clients, for example. Specifying `cpu` may serve to distinguish between certain capabilities or limitations of a given node and, in combination with vulnerabilities entered in the string list `hasVuln`, decide upon the applicability of exploits such as *Meltdown* and *Spectre*, for example. The other three characteristics of a node refer to its OS, installed network interfaces and the subnet(s) that the respective node is physically attached to. The meta model assumes that a `node` only has one OS referenced through `os`, while there may be numerous interfaces and subnets. In most cases, a node will have only one network interface so that the list that the attribute key `myInterfaces` refers to only contains one `interface` element's identifier. Firewalls, however, are nodes, too. And to make any sense, they require at least two interfaces. Similarly, a node may be attached to several subnets which are referred to through attribute key `belongsToSubnet`.

In this regard, information on subnets might have equally been part of the node's corresponding `interface` elements. However, in the scope of this model, interfaces and their configuration only serve to decide on the feasibility of communication on a logical level, that is on or above layer three of the ISO OSI model. Subnets, on the other hand, represent affiliation to the same broadcast domain where ability to communicate is subject to fewer restrictions. How the meta model handles communication will be covered in more detail in Chapter 4.6.5. Being elements themselves, `interface` and `subnet` instances have attributes of their own. Subnets bear two attributes of type `string` that are `name` and `type`. While the former serves to equip a subnet with a descriptive name, the latter may characterize what kind of nodes it hosts (e.g. servers, clients, security cameras etc.). Interfaces, on the other hand, hold yet another element which is of type `ipaddress` that will be covered later on in the context of how the meta model implements communication.

Elements of type `os` represent the respective nodes' operating systems and exhibit a higher degree of complexity. Once again, a `string` attribute serves to equip the `os` element with a descriptive name. Furthermore, `string` attributes `family` and `type` shall be specified. The former to differentiate between general OS families such as Windows, Linux or Unix, for example, the latter to specify whether this is the respective operating system's server, client or maybe even embedded version. In what granularity this is used may depend on the scenario. More importantly, an `os` instance also holds a list of interfaces that is, in most cases, the same as that of the super-ordinate `node` element which is why the respective list may simply be passed on during OS instantiation. Deviations are possible, though. Maintaining such a list within the `os` element serves the purpose of easily passing these to subordinate elements of type `application` that the `os` element is equipped with through its `applications` list. On the other hand, if needed, network-enabled functions that may be considered OS-inherent can simply be linked to elements in the `myInterfaces` list without requiring instantiation of a dedicated `application` element. Examples of functions that extend the basic shape of an OS to address certain use cases will be covered in the context of the case studies. To exert control over such potentially OS-inherent functionality or behavior, a `string` list referred to as `parameters` is maintained individually for each `os` instance. There, arbitrary values can be enlisted that may be incorporated by functions as to entail their outcome. However, the meta model does not prescribe to what extent this list is used, as this depends on specific implementation of functionality for individual scenarios. Apart from this, each `os` instance holds a `string` list referred to as `hasVuln` where respective entries are made to assign certain vulnerabilities. This works analogous to the `parameters` list, yet is intended to specifically effect availability or outcome of exploits. However, operating systems may also hold data, which can be obtained by an attacker to either extract information for subsequent attacks or simply to generate revenue. These data are organized in lists that are referred to as `dataInRam` and `dataStored`. While the former represents information in RAM which may only be accessible under certain conditions, the latter represents information that is accessible through the file system and can be read with help of functions that are inherent to the OS or dedicated applications, for instance. Both lists are of type `usefulData` which is usually used for data that hold revenue. However, the

framework does not perform strict type checking so that any kind of data that is used in the meta model can simply be stored in these lists. In consequence, though, respective element types must be checked upon access through functions in order to be handled correctly. Should operating systems provide functionality that requires identification and/or authentication, a list named `acceptedCredentials` shall be maintained for each OS instance that refers to elements of type `credential`. Credentials are combinations of other elements that represent usernames and passwords, and will be covered in more detail in Chapter 4.6.2.

Lastly, there are elements of type `application` that, by quantity and complexity, make for a large part of the scenario descriptions that will be presented in the case studies and also provide most of the available functionality for interaction. While the specific functions they provide and the individual attributes they may require highly depend on the kind of application that is being modeled, they share a common basic structure, resembling that of operating systems. As was the case with other element types, string `name` is used to provide a descriptive name, whereas `type` serves to characterize the nature of the application, which may be a web or mail server, for example. Also, network-enabled applications maintain their own list of interfaces that they listen on for incoming communication. This list is referred to as `bindingIf` and is a subset of the `myInterfaces` list of the super-ordinate `os` element. The reason this a is a subset is simply to reflect cases in which services running on nodes with multiple connected interfaces must not be accessible through all of these. Practical examples from everyday life are web-based UIs of routers, for example, which, at all cost must not be accessible through the WAN interface but only the internal LAN. But also SQL servers frequently listen on their link local address only, at least in default configuration. Since the meta model's representation of network communication also considers ports, `listeningPorts` must be specified. And considering that some applications may listen on numerous ports, this is a list. Despite the fact that ports are numerals, they do not serve to perform any arithmetic operations which is why the respective list is of type `string` and not `int`. In equal manner to operating systems, `application` elements maintain lists for `dataInRam`, as well as `dataStored` that may contain any type of information. The reasons that applications hold such lists in addition to the lists maintained by their respective `os` element are three-fold. First, it makes applications more self-contained. Second, in the presence of multiple applications per OS, it would become impossible to tell which data belong to which application without additional information. Third, depending on the degree to which the attacker may have gained access to the system in question, she might only be able to access data of said application but not the rest of the system. In such cases, compromised data are simply those that are subordinate to the compromised application. On occasions where the attacker gained full access to a system (i.e., the OS), the function that enables the attacker to extract data will simply iterate through all subordinate applications. Besides the two lists that refer to data that are physically located on the node where the application resides, there is a third list containing data that is referred to as `dataLinked`. This list may hold elements from different sources, including `dataStored`, that can be accessed through said application. This additional list is simply intended to account for cases in which an application like a Wordpress installation, for

example, provides access to data that are not stored locally but in a remote database. Depending on the application in question and potential vulnerabilities, an attacker may acquire such information without compromising the node or application where data actually reside. Should applications need to identify legitimate users to authorize certain types of interaction or access to data, a list of `acceptedCredentials` can be maintained in the same way as for operating systems. Furthermore, string lists `parameters` and `hasVuln` are maintained for each application instance, serving the same purpose as in the case of operating systems.

4.6.2. Assets and Information

The meta model comprises a set of defined element types to represent different kinds of information, as well as assets that hold numeric values serving to generate revenue upon acquisition by an actor. These element types are `ipaddress`, `username`, `password`, `credential`, as well as `usefulData`, and will be explained in the following.

Elements of type `ipaddress` hold two `string` attributes, the actual IP address referred to as `value`, as well as a corresponding DNS name that is referred to as `name`. Both are sufficient to unambiguously identify targets for network communication and are therefore unique within a given scenario. Instances of type `username` only have one `string` attribute named `value` that specifies a given user's name. This kind of information serves to reflect ownership of assets and may also be used for identification to decide upon availability of certain functions. To make this identification unambiguous, such names are unique within the scope of one modeled application. However, across applications, usernames may be reused which does not only concern respective strings, but also complete elements in case some sort of federated access is being replicated. Similarly, `password` elements have only one string attribute, `value`, that holds a password. This information, in combination with a corresponding `username`, is used for authentication. Obviously, passwords need not be unique, neither within the scope of the scenario, nor within that of applications. Together, pairs of `username` and `password` elements form `credentials`. These are elements of their own, intended to explicitly link username and password to form valid combinations for authentication. Furthermore, `credential` elements contain information on the related privilege level. Upon describing that data elements may be assigned to operating systems and applications alike, and elaborating on the purpose of this distinction, the idea of differentiating between levels of access already became apparent. While obviously different sets of credentials may serve to access different OSes and applications, the resulting privileges may be basic or elevated, the former representing regular users, the latter administrators or super users. This privilege level is determined by the `string` attribute `level` and can be used for arbitrary differentiation. Apart from this, `credential` elements have a string attribute to equip them with a descriptive `name`. Lastly, there are elements of type `usefulData`. These do not contain any information that is critical for further interaction with the system so that their acquisition has no impact on the actor's further progress. Instead, they hold two attributes of type `int`, `externalValue` and `internalValue`. The former one is where numeric values for generating attacker revenue are assigned which is counted to

measure the attacker's progress. The latter may be used to assign values representing another perspective on the assets worth, for example from a defender's point of view, allowing for a more distinctive analysis if needed. Also, there are two string attributes to specify `name` and `type` to add meaning to the respective elements and simplify retrieval when analyzing results manually.

Elements of all these types are regularly located (or rather referenced) in the lists `dataStored`, `dataInRam`, and `dataLinked` that are maintained for operating systems and applications, respectively. This is to reflect, that not only valuable assets may be located in systems and applications, but also addresses and credentials, which may be extracted from saved configuration files and the like. Furthermore, usernames may be inferred from the local file structure or simply polling directory information from an Active Directory (AD) server. Passwords, on the other hand, may be present in RAM. It is the scenario designer's obligation to ensure plausible distribution of such data. The fact that these lists for referencing data are of type `usefulData` does not matter since strict type checking is not enforced but must be applied upon extracting said information.

Apart from such "storage" locations, respective elements are employed in the context where they fulfill their primary purpose. For IP addresses, this is obviously the interfaces where they serve to identify targets for network communication. Credentials, as outlined before, may be referenced in a list named `acceptedCredentials` should the respective OS or application need such information for implementing authentication or the like. However, it should be noted that information from such attributes cannot simply be read and extracted by an actor for not representing accessible information in the sense of the meta model.

4.6.3. Attacker Model

One of the requirements listed in the beginning of Chapter 4.1.1 stated that attackers must be stateful to realistically represent lateral movement and its requirements. That means, apart from affecting the system under attack, successful actions of the attacker may also serve to acquire knowledge that she can utilize in subsequent attack steps. However, as mentioned before, the simulation engine is generally agnostic to what is being simulated and only requires elements of type `actor` to exist in order to attribute performed actions correctly. Additional attributes and their semantics play no role in the simulator's operation. Consequently, maintaining attributes that describe the attacker's state and their potential relevance in fulfilling the requirements of different actions must be considered in the meta model. For this purpose, the following attributes have been defined: `knownIpAddressStrings`, `knownDnsNames`, `knownUsernames`, `knownPasswords`, and `knownUsefulData`.

The chosen attribute keys already indicate what kind of information they refer to, representing all types of data comprised in the meta model. However, it should be noted that these lists only contain strings that have been extracted from elements. This means, that the actor's list of `knownPasswords` does not contain references to elements of type `password`, but only the values that are enclosed in said elements. The reason to do so is simple and founded in the working mode of the simulator: Whenever handling elements,

it is in fact only their identifiers that are being passed or returned by the different functions, similar to pointers in C. As a result, assigning an element of type `password` to the attacker's knowledge will only duplicate the reference but not produce a deep copy. While this is intended behavior, it makes assigning elements to attacker knowledge as a consequence of successful reconnaissance inadequate to represent acquisition of information that can "expire" as any subsequent changes to said element would inevitably be known to the attacker. Therefore, to effectively implement and test schemes that rely on invalidating acquired information such as Network Address Space Randomization or simple password renewal policies, attacker knowledge is based on extracted values of simple types which are always duplicated when assigned. Therefore, all functions that augment attacker knowledge must be designed so that the corresponding effect defining which information is added, extracts values from respective elements but must not assign the element itself. In effect, subsequent changes to the system state will cause attacker knowledge to deviate more and more, so that actions requiring such information become unavailable unless reconnaissance is repeated. The outlined scheme is applied across all types of information. Elements of type `usefulData` are the only exception to this rule. Neither being required in any further attack steps for simply measuring progress nor being subject to any modification by a defense scheme, these elements may simply be assigned to the attacker knowledge.

4.6.4. Representing Compromisation

The meta model generally differentiates between two types of access an adversary may have to a system, one allowing to read provided resources, the other enabling execution of code on the respective system. The former may simply result from having gained legitimate access to a system's SMB service or the company's CRM, for example, which allow for reading potentially relevant data, thus generating revenue. Similarly, exploitable vulnerabilities in applications and OSes may allow to do so, yet without requiring preceding authentication and authorization. Either way, the system in question is the target of an action that is launched from somewhere else within "reachable distance". The meta model assumes that any sort of reading, no matter if legitimate or enabled by vulnerabilities, does not incur any changes on the target system. The other type of access goes beyond simply reading but allows to exert control over the target, thus becoming a new source from which further attacks can be launched. Obviously, this ability to (remotely) execute code may subsequently allow to read stored data, which is why this type of access comes with more advantages for the attacker. To acquire such capabilities, the adversary must either get access to a legitimate service (e.g. SSH or RDP) that is intended to exert remote control, or find and exploit a vulnerability that grants such privileges. To what degree this is possible, obviously depends on the functionality implemented in the specific scenario. Performing an action that results in the ability to remotely control a host is encompassed by a state change of the target system, indicating, which actor is able to operate on which privilege level. To accomplish this, the meta model includes a dedicated element type that is referred to as `rceRight`, derived from RCE which stands for remote code execution. Whenever an actor performs an action with said effect, an

`rceRight` element will dynamically be created and augmented with respective attributes to specify details and be attached to the `node` element that hosts the compromised `os` or `application` instance. For this purpose, the attribute `rceRights` is reserved. Attaching such elements to the corresponding node has been chosen for the simplicity that results from only checking nodes and not requiring to traverse all OSes and application which occur in higher quantities. Considering that each OS and application are subordinate to a particular node this appears convenient and allows to easily check for such privileges.

Elements of this type are characterized by two attributes. One is named `actor`, referencing the `actor` instance that obtained the privilege, and the second one named `level` of type `application`, referencing the application or OS over which the actor my exert control now. Here again, the specified type poses no impediment to assigning an element of type `os`. Functions that check on this attribute must determine the type and allow or disallow certain actions accordingly. Being able to control the OS, that is if the level attribute refers to an `os` instance and not an `application`, is the meta model's representation of elevated privileges, allowing the respective actor to also read information from subordinate applications. Should the level attribute refer to an application, on the other hand, reading is limited to data enclosed in said application, resembling user-level privileges. However, both types enable the actor to use the respective node as a source for further actions and try to engage in communication with nodes that may have previously been unreachable.

4.6.5. Communication

While different types of communication such as exchanging e-mails or transporting data via exchangeable media (e.g. USB sticks) may be implemented through extending the meta model, the built-in type of communication is network-based. For this purpose, aforementioned elements of type `ipaddress` that hold unique addresses and DNS names serve to identify connected nodes and allow to engage in communication depending on the fulfillment of certain requirements. This is checked with help of a function written in the modeling language itself that does not have any effect but simply evaluates to `true` or `false`, which according to the previous outlines in Chapter 4.4.1 makes it a helper function. What this function basically does is to check if any of the nodes that the current actor may use as sources is able to reach the target address and port. Nodes that may serve as source are simply those that have an element of type `rceRight` attached to their `rceRights` list, where the current actor is set as `actor`. For any of these nodes to be able to reach the target, it either needs to be part of the same subnet as the target, or requires the existence of a firewall rule that specifically allows communication for involved IP addresses on the respective port. The logic behind this is that nodes belonging to the same subnet may communicate unimpeded for not being separated through a firewall. Communication across subnets, in turn, is assumed to be completely blocked, unless there is at least one rule that explicitly allows communication for given source and destination. Such firewall rules are represented through yet another element type which is referred to as `fwallow`. Elements of this type are characterized by two attributes, the first being a list of `ipaddress` elements referred to as `ips`, and another

```
tryConnect(node SNODE,actor ACTOR,interface SIF,ipAddress SIP,node
    DNODE,interface DIF,ipAddress DIP,port DPORT)
{
  isOfType(node,DNODE) &&
  isOfType(node,SNODE) &&
  isOfType(actor,ACTOR) &&
  isInList(RCENODE,SNODE.rceRights)&&
  RCENODE.actor==ACTOR &&
  // find the node belonging to the destination ip
  DIPADDRESS.value==DIP &&
  DIF.ipAddress==DIPADDRESS &&
  isInList(DIF,DNODE.myInterfaces) &&
  // check if they are in the same subnet else check via checkfw
  ( ( isInList(SUBNET,SNODE.belongsToSubnet) &&
  isInList(SUBNET,DNODE.belongsToSubnet) ) ||
  ( isInList(SIF,SNODE.myInterfaces) &&
  SIF.ipAddress==SIPADDRESS &&
  checkfw(SIPADDRESS,DIPADDRESS,DPORT) ) )
};

checkfw(ipAddress SIP,ipAddress DIP,port DPORT)
{
  isOfType(fwallow,FWRULE) &&
  isInList(SIP,FWRULE.ips) &&
  isInList(DIP,FWRULE.ips) &&
  isInList(DPORT,FWRULE.ports)
};
```

Listing 4.13: Function tryConnect and its helper to determine ability to communicate.

one being a list of strings referred to as `ports` containing ports. The effect of such a firewall rule is that communication between all referenced IP addresses is generally allowed for all listed ports. The function `tryConnect` which performs these checks to reach a specific target is depicted in Listing 4.13, together with another helper function that solely serves to determine if at least one fitting `fwallow` element exists.

4.6.6. Timings and Probabilities of Functions

In the course of explaining how functions are declared and defined (Chapter 4.4.1), as well as the introduction to the simulation engine's working mode (Chapter 4.5.1), it was made clear that the framework expects an element of type `function` to be declared for each defined function within the scenario. Furthermore, these may have attributes, one of which is mandatory, others conditional, and yet another optional. Considering this, such elements and related attributes must obviously be part of the meta model and are — in contrast to any other aspect of the meta model — prescribed by the simulation engine. The attribute that is required for all functions is the `boolean` named `public` which determines whether or not a respective function can be called directly by an actor during simulation or solely serves as a helper that can only be called in the context

of other functions. Depending on this boolean's shape, further attributes of type `int` may be required. These are aforementioned chances and duration of setup and execution, referred to as `setupProb` and `setupTime`, as well as `execProb` and `execTime` that must be specified for public functions only. While these may also be defined for functions that are not public (i.e., a helper function), the simulation engine will ignore such properties and only consider said attributes of the function that utilizes the helper. The one attribute that is fully optional is the `string` list referred to as `param` that allows to assign tags to functions. Respective values are only considered if the simulation engine has been initialized with a list of valid tags that restricts which functions the simulator can choose from. This is part of the simulation configuration file introduced in Chapter 4.3 and useful when intending to repeat a simulation in the presence of differently skilled actors, which otherwise would require to change the model and translate the scenario again. Being part of the system state, the attributes of a function element can, from a technical perspective, be modified in the course of interaction simulation. However, serving the purpose of controlling function behavior, the meta model forbids such modification.

4.6.7. Interplay

Based on the previous specifications, nodes can be modeled that are equipped with certain types of operating systems, which in turn provide the environment for applications. Both OSes and applications have been defined to hold information in dedicated lists (e.g. `dataStored`) that may be accessed in the course of data exfiltration. Yet, others such as `acceptedCredentials` solely serve to link certain information to an application or OS, in this case for the sake of discriminating and authenticating valid users, but cannot be read by an attacker who managed to compromise a node and obtain reading or execution privileges. Consequently, functions defined to interact with a scenario implementing this meta model must not only adhere to the chosen structure of different element types, attributes, and their interrelations, but also need to consider their semantic meaning. In this regard, a function that allows the attacker to read all useful data that are subordinate to a certain node, might be executable for correctly relating to elements and attributes that exist in the model, yet not fulfill its intended purpose as it would also yield elements from the aforementioned `acceptedCredentials` list. Similarly, functions must consider the model's representation of an actor's control over a certain node by checking respective elements of type `rceRight` and related privilege levels when determining whether or not different actions can be performed on certain targets. Furthermore, the meta model dictates how network connectivity shall be represented to incorporate aspects such as firewalling but also consider the existence of different, separated subnets and account for their implications on the nodes' ability to communicate. Adherence to the meta model's representation of network connectivity is enforced through the aforementioned helper function `tryConnect` that is to be used within functions that rely on connectivity.

The logical next step would be to define functions. However, even though the presented meta model may already suggest some plausible functions, it does not prescribe any form of interaction. Doing so would dictate capabilities of the actors and impose an interaction

scheme that may be out of the modeled scenario's scope. Consequently, functions that are in line with the meta model are subject to the case studies, where they serve to enable interaction of actors that suit the given scenario, as well as its threat and defense model.

5. Evaluating the Security Impact of Defense Techniques

So far, the modeling and simulation framework has only been described, covering the modeling language and its primary features, as well as an introduction on how to use it. Furthermore, the working mode of the simulation engine has been outlined to provide a basic understanding of how modeled states and actions are processed to produce results. What is still missing, though, is a demonstration of its capabilities on the basis of a realistic scenario. To this end, this chapter presents a case study that has been conducted with help of the presented framework[1]. Using the framework's modeling language, a sample network representing the IT infrastructure of an SME has been modeled to investigate the performance of four defenses in the presence of differently skilled attackers. While intended to provide insights on the effects of these defenses and their impact on security, this investigation also serves to check on the approach's viability.

The investigated defenses are outlined in the following section, followed by a coherent description of the sample network's shape and the assumed software landscape. Afterwards, modeled vulnerabilities, corresponding exploits, as well as legitimate actions are presented that characterize the attacker and her ability to interact with the system state, before presenting experimental results and the insights they yield. These, however, also introduce a twist, showing that minute changes to the scenario under test may have a considerable effect on simulation results, thus raising concerns about the validity of findings that have been obtained from single investigations. Based on these insights, the chapter concludes with an assessment of whether evaluation based on detailed modeling and simulation is conducive to gaining a better understanding of *MTD* techniques and their security implications, while raising further questions.

5.1. Investigated Defenses

The defense techniques that are employed in the experiment to investigate their impact on security through affecting attacker progress are *IP shuffling*, *VM live migration*, *VM cold migration*, and *VM resetting*. The first three have been chosen for their frequent occurrence in *MTD* literature and the existence of prototypical implementations. *VM resetting*, was included for being integral to *cold migration* and having a potential impact on security itself. However, the modeling language does not restrict which defenese may be modeled so that others could have been implemented just the same. To put

[1] The presented case study and its findings have been published in the proceedings of the *Nordic Conference on Secure IT Systems 2018* [19]. The publication was recognized with the Best Paper Award.

performance of the different defenses into perspective, simulations will also be conducted with *no defense* enabled, serving as a base line.

Table 5.1.: List of defenses that are employed in the case study for comparison of performance, as opposed to using *no defense*.

Name	Description	Impact
Live migration	The VM is migrated from one physical host to another without loosing its state.	Moving a VM changes the physical connectivity and hence communications paths.
VM resetting	The VM is restarted from read-only memory, loosing all state information.	Any remote code execution privileges on the VM previously gained by the attacker are removed.
IP shuffling	A new IP address is assigned to the VM.	Knowledge of the IP address previously gained by the attacker is removed.
Cold migration	The VM is migrated to another physical host, restarted from read-only memory, and assigned a new IP.	This is the combination of live migration, VM resetting, and IP shuffling.

IP shuffling is one of the most frequently suggested *Moving Target Defenses* [34, 53, 80, 86, 97], advocating the repeated change of IP addresses of communicating entities, may also comprise ports, and in rare cases even MAC addresses [88]. The intention is to impede the attacker by invalidating previously acquired knowledge of addresses and making guessing significantly harder by leveraging large address spaces. The implementation employed in the case study only considers IP addresses and is therefore denoted as *IP shuffling*.

VM live and cold migration propose to repeatedly relocate VMs across various physical hosts. Doing so for the purpose of defense has been addressed in previous work dealing with *MTD* and network defense in general [3, 11, 44, 61, 64, 98, 129], with suggested schemes generally differing in whether a VM's state is preserved or not. State preserving migration that is denoted as *live migration* in the following intends to seamlessly move a VM out of the attacker's reach while keeping it as is. Attacks that *live migration* is supposed to fend off are such that rely on certain connectivity [64] or co-location of VMs [61, 129]. Another form of VM migration that is denoted as *cold migration* suggests to not only migrate VMs but reset them to a default state, reverting any potential modifications caused by the attacker [5, 45]. Additionally, such VMs (may) receive new IP addresses (e.g. through DHCP), thus implementing a form of IP shuffling whenever migrated. In the scope of this work, *cold migration* combines moving and resetting of VMs while providing them with new IP addresses.

VM resetting suggests the sole recurring reset of virtual machines. Prior to the advent of *MTD*, approaches such as SCIT [22] already promoted the idea of resetting VMs to previous, presumably secure states to prevent attackers from reaching persistence. Though not being a *Moving Target Defense* in the classical sense, it has been included

to determine its individual impact on security. This allows to dissect performance measurements of *VM cold migration* and relate its results to those of the defenses it is composed of.

A brief summary of employed techniques can be found in Table 5.1. Note that the real-world implementation and application of such defenses may come with challenges. For example, the blunt shuffling of IP addresses might not only impede attackers but also break legitimate communication if no precautions are taken. To account for this, the previously presented meta model also includes DNS names that are assumed to be known to legitimate users, allowing to resolve correct IP addresses at any time. Likewise, the resetting of VMs renders any service useless that stores data locally, which is why VMs that host databases or serve as storage are exempt from resetting and cold migration.

5.2. Network Layout and Software Landscape

Figure 5.1 shows the network setup for the envisioned small enterprise network. The network is separated into a DMZ with servers accessible from the internet, an intranet with clients, and a server subnet. The communication between subnets as well as between machines within the server subnet is subject to firewalling. Furthermore, no machine beyond the DMZ is directly reachable from the internet. In the DMZ two Xen servers are assumed that form a pool of hypervisors for three VMs. These comprise a Microsoft Exchange server running on Windows Server, and two VMs running on Ubuntu Server. One for the company's Drupal-based website, and one for a Tomcat server that hosts applications such as time tracking that are accessible to employees from the intranet as well as from the internet after log-in. In the server subnet there are four hosts, three of which form another pool of Xen servers to host VMs, and one Ubuntu Server machine serving as a storage system for backups. The VMs in this second pool comprise:

- A Windows-based Active Directory Server acting as the domain controller, providing authentication services and network file sharing.
- A server running Base CRM, a proprietary customer relationship management system, based on Ubuntu Server.
- A server for accounting applications such as Datev, based on Windows Server.
- Another Tomcat server that exclusively runs applications for the HR department, based on Ubuntu Server.
- A Veritas Netbackup server to centrally command and control the backup agents on the various backup clients, based on CentOS server.
- Two Ubuntu-based servers for DevOps purposes (e.g. Jenkins and Jira).
- Four SQL servers, two of which are based on Ubuntu Server (for the Tomcat HR and Base CRM) and the other two being based on Windows Server (for Active Directory and Exchange).

Finally, client computers are assumed to be located in the subnet called "Intranet", which is connected to the server subnet and the DMZ through the second firewall. All clients are based on Windows 10 and differ in the user groups that operate them. They are equipped with the MS Office suite and backup agents. Employed applications and operating systems are based on respective element types that have been defined in the context of the unified meta model in Chapter 4.6 but also extend these to implement required attributes and functionality where needed. One somewhat extraordinary extension in this regard is the Xen operating system's capability to host virtual nodes. Every Xen instance holds a list of type `node` that is named `vms` where references to attached VMs are maintained. Apart from being subordinate elements of an OS, virtual nodes do not differ from physical nodes and are based on the same template.

In the given example network, eight different assets are modeled that can be obtained by an attacker, yielding different amounts of revenue which add up to 100 points in total. Four of these `usefulData` elements are supposed to represent customer data, two of which yield 15 points each, the other two 10 points each. Additionally, there are two `usefulData` elements representing financial data, yielding 15 points each, as well as credit card information and HR data for 10 points, each. All of this data can be accessed in different ways. One way to access customer data is through the Base CRM frontend, if respective credentials have been obtained from the various back-office clients. Another option is to directly query the SQL server where data are effectively stored, given the fact that the attacker was able to obtain username and password. Yet another possibility is to compromise the operating system that the SQL server is running on and exfiltrate the database. The financial data can be accessed through the CEO's computer or through her e-mail account, again opening up different ways to acquire this information. HR data can be obtained through compromising either the Tomcat server in the server subnet or the respective SQL server where data is stored. Finally, credit card information can be retrieved through access to the assistant's computer or its backup.

The fact that the sample network utilizes Xen hypervisors to host VMs for different purposes allows to incorporate VM migration in the scenario. From a practical point of view it does not make sense to shuffle VMs from the DMZ with those from the server subnet. Hence, VM shuffling is only used to move VMs across hosts that belong to the same pool. To simulate the changed physical connectivity mentioned by Hong and Kim [64], hosts from the server subnet are directly attached to the free ports of the routing firewall `fw2`. By default, most hypervisors use a virtual switch that the hosted VMs are attached to. Therefore, VMs located on the same host are assumed to be connected by the virtual switch of the hypervisor and can communicate with each other regardless of the firewall setting in `fw2`. To summarize, the firewall `fw2` limits communication between VMs located on different hosts, whereas communication between VMs on the same host is not restricted.

It should be noted that the four VMs that serve as SQL servers for the different applications are never migrated but strictly allocated to host 6. This is due to the fact that the migration of VMs that contain large databases poses additional challenges in order to maintain availability and consistency. Additionally, VM resetting conflicts with the database's primary purpose to persist data.

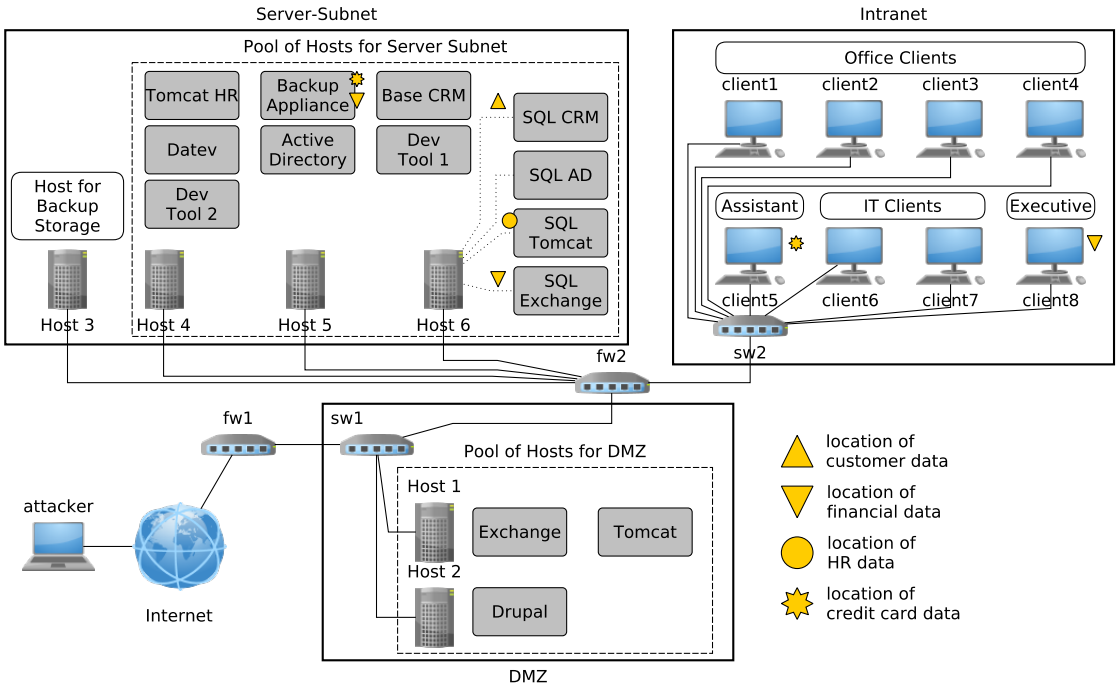


Figure 5.1.: The network used in the case study, representing a fairly typical small enterprise setup.

5.3. Vulnerabilities and Attack Steps

Choosing realistic vulnerabilities and exploits, as well as legitimate actions that contribute to the attacker's progress is crucial for a fair and realistic evaluation. Detailed protocols of successful attacks like that of *Phineas Fisher* on *Hacking Team* [103] have revealed that for attackers to reach their goals, legitimate actions are as relevant as vulnerabilities and their exploits. Consequently, legitimate actions are integral to the model. However, the question arises how to choose vulnerabilities, functions and exploits. This was done as follows: For the presented sample network, specific and commonly used software has been chosen and the *CVE* database and *Metasploit* database searched for related entries for the years 2016 to 2018. For each *CVE* entry with a high *CVSS* score, applicability of the vulnerability in the given scenario was manually checked. In particular, vulnerability exploits that result in either remote code execution, privilege escalation or the retrieval of information (e.g credentials, RAM content etc.) have been chosen. Based on this, exploits have been modeled for the respective vulnerability with a high level of detail that can entail a range of requirements that need to be met. Similarly, realistic legitimate functions of the assumed applications and systems have been implemented that are equally capable of providing the attacker with execution privileges or valuable information. Examples of such legitimate functions are remote shell or desk-

top access for operating systems to gain remote code execution privileges, ARP cache lookups to retrieve IP addresses, or SQL queries to obtain information from databases.

One important aspect is defining the duration of an exploit as well as the attack success probability. Although the *CVE* entries include parameters that are related to an exploit's duration and likelihood of success (e.g. attack complexity, exploit code maturity etc.), specific figures for these measures cannot be derived from a given score. Therefore, values for duration and success rate have been manually determined based on a vulnerability's description, the underlying mechanism, and the availability of exploit code (e.g. in *Metasploit*), noting that these values could potentially be optimized with data obtained from real world attacks.

This totaled in 16 exploits, as well as 10 legitimate actions an attacker can call, resulting in 26 executable actions from an attacker's perspective. Table 5.2 gives an overview of these, including the respective *CVE* entry number(s) if applicable, a simplified effect description, and the attacker type that is able to use them. Details on requirements and effects of these actions can be found in Appendix A.

Table 5.2.: List of actions available for differently skilled attackers (year), *CVE* entry for exploits if applicable, and corresponding effect.

Action	*CVE* entry	Effect	Skill
`tomPrivEscalation`	2016-9775, 2016-9774	elevate RCE on target OS	2016
`privEscalationWindows`	2016-0026	elevate RCE on target OS	2016
`backupServerRCE`	2016-7399	read data from multiple OS	2016
`phishingDocRCE`	2016-0099	RCE on target OS	2016
`tomHttpPutRCE`	2017-12615, 2017-12617	RCE on target app	2017+
`jmxTomcatVulnerability`	not available	RCE on target OS	2017+
`privEscalationUbuntu`	2017-0358	elevate RCE on target OS	2017+
`eternalBlueRCE`	2017-0143 to 2017-0148	RCE on target OS	2017+
`redirectBackupToCloud`	2017-6409	read data from multiple OS	2017+
`backupClientRCE`	2017-8895	RCE on target OS	2017+
`clientRCEoverServer`	2017-6407	RCE on target OS	2017+
`meltdown`	2017-5715, 2017-5753	read data from target node	2017+
`drupalRCE`	2017-5715, 2017-5753	RCE on target app	2017+
`sendMailExchangeRCE`	2018-8154	RCE on target app	2017+
`exchangeDefenderRCE`	2018-0986	RCE on target app	2017+
`readData`	legitimate	read data from target OS	any
`pingscan`	legitimate	read IP from target OS	any
`arpCache`	legitimate	read IP from target OS	any
`configureAdClients`	legitimate	RCE on target OS	any
`getCustomerData`	legitimate	read data from target app	any
`getMail`	legitimate	read data from target app	any
`remoteDbManagement`	legitimate	read data from target app	any
`sqlQuery`	legitimate	read data from target app	any
`remoteShellLinux`	legitimate	RCE on target OS	any
`remoteShellWindows`	legitimate	RCE on target OS	any

5.4. Experimental Results

Two independent experiments have been conducted, one in which the attacker could utilize exploits based on vulnerabilities published in 2016 (4 exploits plus 10 legitimate functions) and one with exploits based on vulnerabilities from 2017/2018 (12 exploits and 10 legitimate functions). In both cases, performance was tested in the presence of *no defense technique*, *live migration*, *cold migration*, *IP shuffling*, and *VM resetting*, resulting in a total of ten setups for simulation. For every combination of defense and attacker type, simulation was started 100 times, consisting of 8000 rounds each. The reason to repeat simulation 100 times is that, while the modeled network itself is static, the numerous attacker and defender actions are subject to different probabilities to account for operations that may plausibly fail occasionally due to their difficulty, complexity, or sheer chance (e.g. phishing attack)[2]. Based on the 100 simulations per setup, averages have been formed that are less susceptible to outliers. Furthermore, three revenue thresholds have been defined. These are at 40, 75 and 100 points respectively to measure how many simulations reached these thresholds for a given number of rounds. The results are depicted in Figure 5.2.

In the experiment where exploits from 2017 and 2018 were used, no significant difference between having *no defense technique*, *IP shuffling* or *live migration* can be observed. Attacks succeeded fairly quick and all simulations achieved the maximum revenue of 100 points. When *VM resetting* or *cold migration* were enabled, it took more rounds for the attacker to reach revenue levels of 75 or 100. Hence, one can say that they had a positive impact on security. *Cold migration* is the combination of *live migration*, *IP shuffling* and *VM resetting*. The fact that *cold migration* and *VM resetting* performed nearly identical indicates that the security gain primarily results from *VM resetting* and not from migrating (shuffling) VMs.

In the second experiment only four exploits (i.e., those selected from the year 2016) and ten legitimate functions were available, resulting in fewer viable attack paths. In this case, all defenses performed similar for a revenue threshold of 40. However, revenue levels of 75 or 100 were only achieved when either *live migration* or *cold migration* was enabled. If *no defense technique*, *IP shuffling* or *VM resetting* was used, these revenue levels were never reached in any simulation run. The log data of the simulations reveal that there was only one possible attack path to achieve at least 75 points. The performed attacker actions are listed in Table 5.2. The first step is that the attacker launches a successful phishing attack against one of the clients. The attacker can then use the remote code execution privileges as well as the stolen DNS names to launch an attack based on exploit `backupserverRCE` (*CVE-2016-7399*) on the backup server. These first attack steps are independent from any of the used defense techniques and generate more than 40 revenue points for the attacker. This is due to data directly found on the attacked client and backup server, as well as using the Base CRM server with stolen credentials of the client. Besides data that directly generates revenue, additional data is stored in the backup. In particular, it also contains configuration files of the "Base CRM" and "Tomcat HR"

[2]Respective probabilities for the different actions can be found in Appendix A.

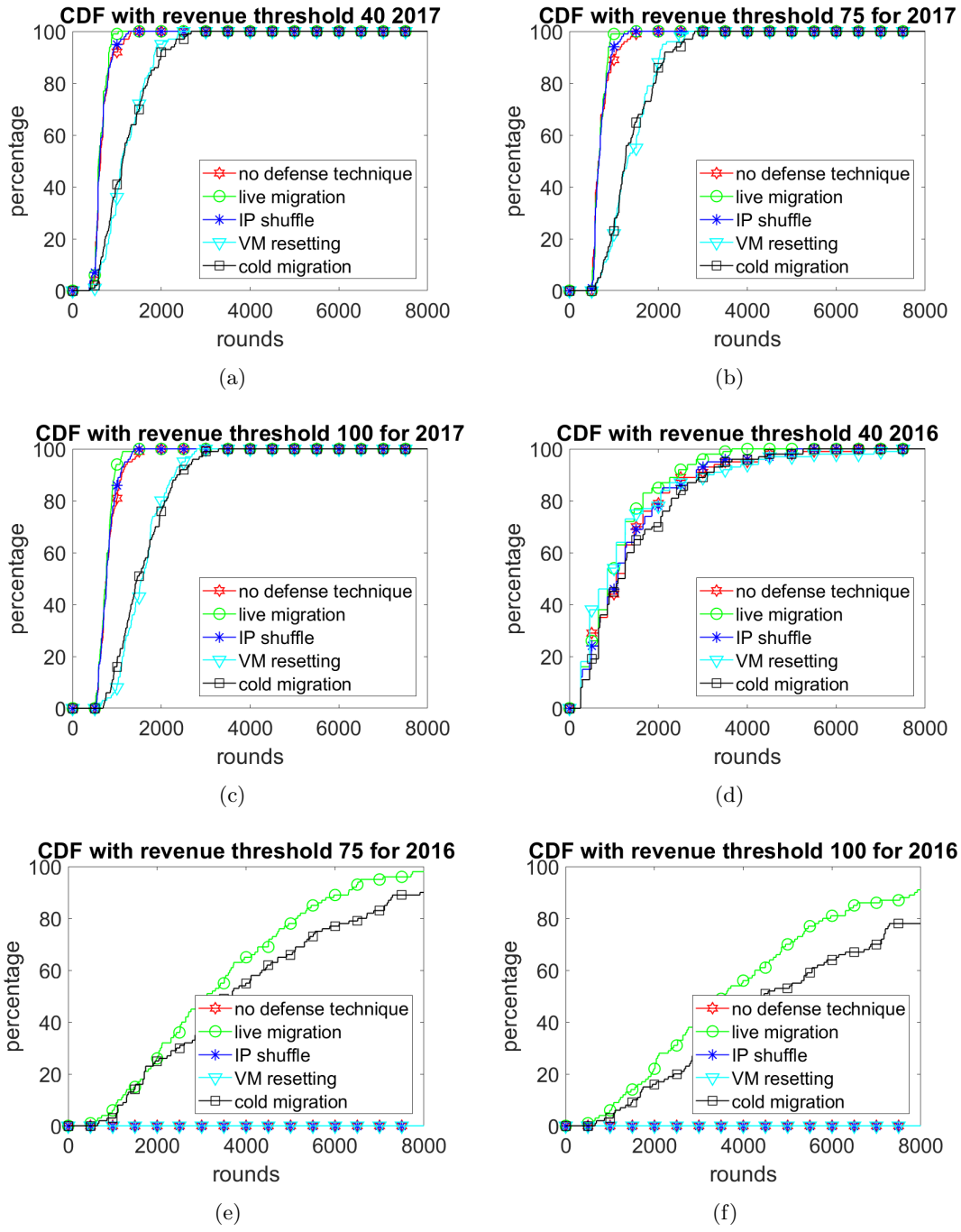


Figure 5.2.: Results of the attack simulation. Each defense was simulated 100 times for exploits based on 2017/2018 vulnerabilities (a-c) and 2016 vulnerabilities (d-f). Results are displayed with regard to reached threshold with the y-axis depicting the percentage of simulations that reached the respective success threshold for the given round (x-axis).

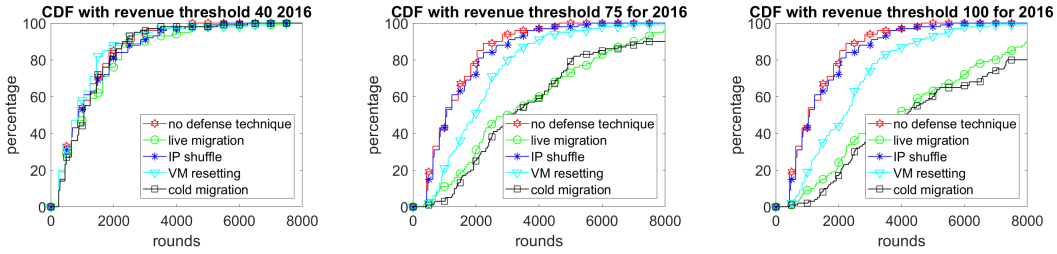
servers and the corresponding SQL credentials. These SQL credentials can then be used in the next attack step to retrieve customer data and HR data using regular SQL queries and database management commands. However, to perform these regular functions, the attacker needs to be able to communicate with the SQL servers on host 5 via the SQL port 3306. Both nodes that the attacker controls — the compromised client (`phishing`) as well as the backup server (`backupserverRCE`) — are not whitelisted to communicate on the SQL port. Since the backup server and the SQL server are located on different hosts in the beginning, the firewall blocks such communication attempts. Therefore, for *no defense technique*, *IP shuffling* or *VM resetting* the attacker cannot call these SQL functions and, in consequence, never reaches revenue levels of 75 or above. However, the firewall cannot block communication between VMs on the same host as they are, by default, attached to the same virtual switch. The log data reveals that when *live migration* or *cold migration* is enabled, the backup server is shuffled to the same host as the SQL databases roughly one-third of the times. Hence, whenever the backup server was on the same host as the SQL server, the attacker could retrieve data using SQL queries until another shuffle operation migrated the backup server to a different host.

Please note that this scenario is exactly as discussed by Hong and Kim [64] to assess the effectiveness of live migration. The migration of VMs changes the physical connectivity and with it the attack paths. However, as the experiment shows, this can have a significant negative impact on security as the migration of VMs enables attack paths that would otherwise not exist.

Evaluation With a Twist

In the experimental results depicted in Figure 5.2, migration had a negative impact on security. Only the removal of the attacker's RCE privileges (which is being done in *VM resetting* and *cold migration*) had a notably positive effect on security. However, resetting VMs only hindered the attacker and made attacks more difficult with regard to the required time (rounds) to a full compromise but could not completely fend off attacks. Of course, if resetting is done at much higher frequencies, it is possible to get results in which all attacks are defended. Such timings are not very realistic, though.

However, one can also produce scenarios in which migration has a measurable positive effect. When having a look at the experiment based on the 2016 exploits, the reason why the attack does not work if no migration is used is that the VMs of the backup server and the SQL servers are not on the same physical host. Therefore, to generate positive results for migration, the starting position of VMs was modified, moving the backup server to the same host as the SQL servers. Figure 5.3 shows the experimental results for this modified case of an attacker employing exploits from 2016. This time, migration contributed to security. The reason is that with migration turned on, the backup server and the SQL server were on different hosts two-thirds of the time, while for the other non-migrating defenses they were always on the same physical host. Hence, attacking was more difficult in that it took the attacker more rounds to exfiltrate data. Yet, it should be noted that the attacker was still able to exfiltrate all data within 8000 rounds in 90% (*live migration*) and 80% (*cold migration*) of the simulations.



(a) (b) (c)

Figure 5.3.: The same analysis as in Figure 5.2 (d-f) but this time the starting position of the backup server was on host 6 (as the SQL servers) instead of host 5.

The fact that attack simulation is not only able to measure revenue data but also log all steps performed by the attacker, allows to investigate why a defense performs a certain way. This is exactly what was done to understand and describe why migration performed so poorly for the 2016 scenario in Figure 5.2 (e,f), illustrating the advantages of being able to also perform qualitative analysis to put quantitative results into perspective. In that sense, simulation has shown to be viable, generating insights that other approaches were not able to deliver. However, another important lesson to be learned is that the results of attack and defense simulation can be heavily influenced through tuning the investigated scenario. Hence, evaluation on the basis of one single scenario does not allow for generalizing findings, despite being common practice among proposed schemes.

5.5. Summary

This chapter presented a case study in which experiments with *Moving Target Defenses* based on *VM live and cold migration*, *IP shuffling*, and *VM resetting* have been conducted, yielding important insights regarding the effects of different defenses, as well as evaluation in general. With regard to defense effects, results show that while random VM migration can have a positive effect on security, negative effects are equally possible. Through changing the physical connectivity in a network, VM migration influences attack steps. Should the initial network layout be beneficial for the attacker, moving VMs may improve security through increasing attack time as the attacker has to wait for VMs to be shuffled back to suitable positions. However, should the initial layout be comparably secure and not allow for attacks, random VM migration will eventually shuffle VMs such that an attack becomes feasible in the first place. That means, the potential positive impact is only an increase in attack time while the negative impact is that formerly impossible attacks may be enabled.

More importantly, though, apart from specific results on defense performance, the case study in this chapter has shown that attack simulation based on realistic exploits, functions and network setups is indeed capable of analyzing and comparing defense techniques, laying the corner stone for subsequent development of the framework and research on defense effects.

6. Benchmark Network Fuzzing

The case study and its results presented in the previous chapter revealed that simulation on the basis of detailed models is indeed capable of generating valuable insights. However, it also showed that seemingly minute changes in the scenario can have a considerable impact on evaluation results. In consequence, to evaluate fairly, defenses under investigation should be tested in a multitude of diverse scenarios as to include potentially relevant corner cases, revealing a larger spectrum of effects. This would allow for a coherent overview of a given technique's strengths and weaknesses, as well as the situations in which they come into effect. Considering this, the requirements for realistic analysis and comparison presented in Chapter 4.1.1 must be extended to also include the following:

- *A wide range of benchmark networks:* The performance of attacks and defenses may depend on environmental factors. Hence, to realistically compare defense techniques, the analysis should not be based on a single network configuration but on a wide range of configurations.
- *Aggregation of results:* Once investigated scenarios reach a certain quantity, individual treatment of results such as the previously employed CDF plots is neither efficient nor useful. In consequence, metrics are needed that aggregate data to yield comprehensible and meaningful insights.

Despite the simplifications and comfort features introduced by the modeling language, crafting detailed and realistic scenarios requires effort. For networks of the kind presented in the previous experiment and a modest number of variations thereof, this is definitely feasible. Yet, should the scenarios to be tested grow in size and complexity or the number of variations needed to be increased, this may become challenging. What is more, handcrafting variations to detect corner cases in which the effect of a defense takes a turn is prone to bias since the respective scenario designer might only consider such cases that she deems relevant for expecting a certain effect. Environmental conditions of unanticipated causes and effects may simply not be included and therefore not be detected.

In order to increase test coverage and avoid unintended bias, yet keeping modeling effort at a manageable level, this chapter introduces benchmark network fuzzing, enabling the automated diversification of benchmarks networks on the basis of a single scenario definition[1]. Depending on the required degree of freedom, such definitions may merely describe a scenario's basic structure while making specifics such as network size,

[1] Parts of the contents of this chapter have been published in an article [21] in the *Springer International Journal of Information Security*.

configuration of nodes, employed software but also distribution of vulnerabilities subject to probability. To accommodate this, the previously introduced modeling language is augmented with new functionality so that, from a technical perspective, it is no longer a simple modeling language to describe mere states and actions but is now capable of processing. This includes utilization of runtime variables, loops, if-clauses and probabilities, for example, to make scenario definitions flexible where needed. In consequence, one single definition may result in a vast amount of unique instances whose (dis)similarity depends on how extensively these features are used and in which combinations they occur. Each of these instances is a full state description defining every element in the network including attackers and defenders. However, there is no obligation to use any of these diversifying features. Fully deterministic scenario definitions as used in the first experiment work just as before.

The next section gives an introduction to these new language features and provides information on how to use them correctly, followed by a detailed outline as to how these new instructions are processed by the framework's translator, that is more of an interpreter at this point. Afterwards, to test these new capabilities of the framework, another experiment is conducted where size, complexity and number of scenarios are considerably increased with help of benchmark fuzzing to perform simulation at a larger scale. Before presenting and discussing obtained results, new metrics are presented that help make sense of the increased amount of data and allow for interesting insights. Finally, this chapter is concluded with a summary of how benchmark fuzzing contributes to realistic and meaningful evaluation.

6.1. New Language Features

Extending the modeling language's capabilities to use runtime variables, augment it with features such as loops and if-clauses, as well as additional operators to employ probability, provides several advantages that do not only simplify diversification but scenario definition in general. Some of the most obvious benefits resulting from these enhancements are the following:

- omit excessive repetitions with loops
- use conditions to switch between various alternatives within a scenario
- use runtime variables to store and retrieve information at any point
- maintain and extend lists of elements that are of special interest (e.g. group items that must be accessed and altered on particular occasions)
- employ probability to diversify single minute attributes as well as large parts of a scenario to create differently shaped instances of the same scenario

In consequence, the creation of numerous different, yet similar instances becomes much easier and less time consuming, and can even be fully automated when employing

probability to "decide" upon certain aspects that impact the instance's shape while creating it. To better illustrate how newly added features are used in practice, it makes sense to start off with runtime variables that the language is now able to process, since these play a major role when utilizing loops and if-clauses, for example.

Runtime variables extend the concept of parameters that are passed to the `init` instruction of templates. Since these may be different for any new instance of a given template, using them within the template had to be realized with help of placeholders that serve to represent specific values. To use them, their names simply had to be enclosed in `$`-signs upon usage and the translator would insert the corresponding value upon processing the given instruction. In a similar fashion, runtime variables are used. Coming into existence upon first usage, they do not need to be declared. An instruction as simple as `currentRound=1;` will cause a variable named `currentRound` to be created with the given value. Respective values may equally be strings or lists. Upon assigning a new value to a runtime variable, the old value will simply be overwritten. However, values may not only be set and retrieved but also processed in the sense of incrementing counters, concatenating strings or simply appending values to lists. This way, the translator can maintain a state of existing elements and related information which can be used to fine-tune subsequent scenario generation. Previously, augmenting elements with additional attributes only worked if all their identifiers were known and hard-coded into the definition. Now, through maintaining lists of element identifiers, these can be recalled at any point of subsequent translation to augment them with attributes or use them for other purposes, simply by iterating through said lists. This is especially useful when certain aspects have been made subject to probability so that there is no way of hard-coding related information in the definition. It should be noted that runtime variables are only valid in the scope of the file that is being processed. That means, a variable that is maintained in the main scenario definition is not known in the context of processing a template file which has been invoked by calling `new`, for example. Should this be needed, the respective variable must be passed to the template upon instantiation.

The syntax of using **for-loops** is similar to that of function definitions. The functor `for` initiates the loop and is followed by parentheses that specify the counting variable together with a specified range or list for iteration. Afterwards, the loop's body is specified which must be enclosed in curly brackets. Inside this body, the counter variable can be used in the same way as runtime variables. Being enclosed in `$`-signs, they will be easily detected as such and processed accordingly. Loops may be nested and their counting variables are only valid within the scope of the loop. Similarly, the **if-clause** starts with the keyword `if`, followed by parentheses containing the condition to be checked, as well as the curly brackets that enclose the instructions to be processed if the condition is met. Nesting if-clauses is possible to an arbitrary extent.

The full benefit of these augmentations to the modeling language arises from combining them, allowing to easily scale up numbers while preserving the ability for individual treatment in case of certain conditions. Listing 6.1 provides a code snippet showing how these features may be used. In this example, 100 nodes are instantiated according to the template presented in Listing 4.2 and directly equipped with an attribute to denote

```
subnet clients;
// creating 100 nodes for the client subnet
for($I$ in range(1,100)){
  node pc$I$=new node("client",[],4,512);
  subnet pc$I$.belongsToSubnet=clients;
  $allClients$+=[pc$I$];
}
// now, every fourth client is equipped with a Linux-based os
$counter$=1;
for($PC$ in $allClients$){
  if($counter$==5){
    $counter$=1;
  }
  if($counter$==4){
    os $PC$.os=new os("linux");
    $windowsClients$+=[$PC$];
  }
  $counter$+=1;
}
```

Listing 6.1: Employing runtime variables, for-loops, and if-clauses in practice.

them as members of the `subnet` named `clients`. Furthermore, each `node` element is appended to a list of runtime variables (i.e., `allClients`) for later reference. Afterwards, every fourth node is equipped with an operating system of the Linux-family, which is simply done by iterating through the list of all nodes and checking if the counter is four to make respective modifications.

These features allow to easily scale a given scenario and even quickly switch between alternatives that may have been defined in the scenario definition though simple conditioning. However, this is still deterministic and does not allow for automated variation. To accomplish this, one additional feature referred to as `PROB()` has been introduced that allows to leave some aspects to chance while processing the scenario. Through randomizing certain values, conditions that are subsequently checked may take different turns thus considerably changing the shape of the resulting scenario instance, allowing to generate large varieties. This probabilistic feature simply implements randomly picking one value from a list of values which may either happen assuming a uniform distribution of probabilities or by providing a second list of values that determines probabilities for the values in the first list based on the order of appearance. It should be noted that utilization of this feature differs from that of functors for requiring the respective expression to be enclosed in `$`-signs as if referring to a runtime variable. Why this is the case will be covered in the next section where implementational aspects of the extended modeling language are covered. Listing 6.2 provides a code snippet showing how to use this feature (in lines two, five, and eight), using the example of a randomly chosen number of nodes to instantiate and their potential vulnerability in dependence of the utilized operating system.

```
// depending on chance, either 10,100 or 200 nodes will be generated
$numOfNodes$=$PROB([10,100,200])$;
for($I$ in range(1,$numOfNodes$)){
  // os is chosen randomly: 50% for windows, the other two 25% each.
    Note that the last probability may be extrapolated
  $ostype$=$PROB(["windows","linux","osx"],[50,25])$;
  node pc$I$=new node("client",[],4,512);
  os pc$I$.os = new os($ostype$);
  $vulnerable$=$PROB([true,false],[10])$;
  if($ostype$=="linux"){
    if($vulnerable$==true){
      string pc$I$.os.hasVuln="shellshock";
    }
  }
}
```

Listing 6.2: Probabilistic scenario shaping.

While this is only a trivial example, it illustrates how the newly introduced features may be used to automatically diversify scenarios. Probabilistic determination of specific values was only done on three occasions, already yielding a high number of different potential outcomes. Obviously, these will mostly differ in the number of pc and utilized operating systems, which may be of no particular importance. Yet, practical implications are evident. Assuming that those features are used on a large scale to choose between equally plausible shapes of different aspects of networked systems and making subsequent choices dependent on previous ones, numbers of realistic scenarios can easily be scaled.

6.2. An Additional Processing Layer

Simply put, the modeling language has been augmented with features that go beyond mere descriptions but allow for additional processing by the translator itself, before producing the output in form a monolithic Prolog file. In this regard, the translator does not need to be rewritten from scratch but may similarly be augmented with capabilities to adequately handle newly introduced instructions. Processing wise, the effects of runtime variables, loops, if-clauses, as well as the probabilistic value selection can be evaluated before any form of translation is conducted. What this means is that the original translator can simply be equipped with a preprocessor that resolves all new instructions, producing a representation that can be translated just as before.

To accomplish this, parsing is extended to recognize the new instructions, first. This is done with an additional grammar that is considerably simpler than the one used to translate scenarios for only needing to tell apart descriptive instructions and those that need preprocessing. In consequence, the resulting AST is simpler which also benefits its traversal to produce an ordered list containing all instructions from the input file. While anything that concerns descriptions is basically just left as is, instructions requiring preprocessing are transformed into a structured format to simplify subsequent handling.

At this point, the preprocessor comes into play, working off ordered instructions to build and maintain the list of plain descriptions and variables, replace them where needed, unroll loops, an evaluate if-clauses. Furthermore, in this step, all probabilistic decisions are resolved and directly replaced by the value that has been determined on the basis of given options. This direct in-place substitution is the same as done for runtime variables and not handled by a specific procedure as is needed for loops or if-clauses. Therefore, to be detected as a value that should directly be replaced, it is simply enclosed in `$`-signs, serving as a distinctive identifying feature. The fact that this list contains all instructions, even the plainly descriptive ones, ensures that processing of runtime variables is in sync with processing of the rest of the scenario. If treated separately, additional effort would be needed to align which variable had what value at a given time to substitute placeholders correctly. The output that is generated is a representation of the scenario definition that is fully deterministic and does not contain any of the newly introduced instructions anymore. From this point on, the previously introduced translation process will pick up. This whole procedure is performed for every individual file that is being loaded, from the main scenario to the template files and snippets. Remember that the main worker of the translator processes all files recursively, so as to implement preprocessing, the procedure outlined above is simply incorporated in the worker, just before translation is invoked.

6.3. Metrics for a Comprehensive Analysis

For analyzing simulation results on a small scale, the previous experiment relied on CDF plots for different thresholds. This has shown to be a convenient way to get an impression of how the different defenses perform under certain conditions. Yet, when increasing the number scenarios, manually investigating different plots to decide which defense performed best under what conditions is hardly feasible and, most importantly, not efficient. To this end, new metrics are presented that deliver meaningful KPIs from simulation logs which, in turn, can be further processed to condense results.

The new metrics still rely on the attacker's progress in the presence of different defenses, measured in form of revenue gained through compromising resources of certain value. During simulation, this attacker revenue is logged in each round, with $r_j[i]$ denoting the revenue accumulated by the attacker up to round i for simulation j. However, instead of simply plotting this accumulated revenue, results are now captured with help of the following metrics: $rmax$ is the maximum accumulated revenue among all l independent simulations conducted for a given combination of network and defense, $ravg_j$ is the average accumulated revenue per round for simulation j, and $ravg^m$ the median of

these averages among all l simulations. These are formally defined as:

$$rmax_j = r_j[n] \tag{6.1}$$

$$rmax = \max_{j \leq l} \{rmax_j\} \tag{6.2}$$

$$ravg_j = \sum_{i \leq n} (r_j[i])\,/n \tag{6.3}$$

$$ravg^m = \underset{(j \leq l)}{\text{median}} \{ravg_j\} \tag{6.4}$$

In this regard, $rmax$ shows how far an attacker can possibly infiltrate a network. Deviations in this metric across defenses indicate if a defense was able to cut off attack avenues — or maybe even opened new ones. However, only looking at $rmax$ does not capture cases where a defense cannot prevent an attack, but slow it down. Many proposed *MTD* techniques actually do not claim to completely prevent attacks but impede the attacker as to make attacking more costly and risky. This is what $ravg_j$ is for. Relying on the average of accumulated revenue per round instead of gained revenue per round, $ravg_j$ accounts for both earliness and height of obtained revenue. As a result, it will yield lower values for simulations in which revenue acquisition was delayed, even though $rmax$ may be the same. This way the two metrics complement each other. Deriving $ravg^m$ from all $ravg_j$ serves to make this metric resistant to outliers and condense results from l independent simulations per network and defense into one single value. These two metrics are well suited to compare defenses within the same network. However, across different networks, where conditions might differ significantly and achievable revenue deviate, comparison of these values makes no sense. What can be done though, is classifying results on a per scenario basis to determine whether or not the $rmax$ value of the defense was higher, equal or lower than that of employing "no defense". If so, the defense is effective. Yet, should $rmax$ be higher, the defense actually has a negative impact on security. Doing the same with $ravg$ is not helpful, though, as this value has a high variance due to the probabilistic nature of the simulation. For this metric, classification is done on the basis of overlapping confidence intervals for $ravg^m$ to decide upon significantly higher or lower values. In Chapter 6.4.4, analysis of experimental results of the following case study will illustrate this classification scheme and the purpose it serves.

6.4. Fuzzing-enabled Defense Evaluation at Scale

For the second case study, a mid-sized corporate network consisting of nine subnets is assumed. These allow for a considerably more complex network hierarchy than the three subnets used in the previous experiment, let alone scenarios that consist of merely three hypervisors and five unspecific VMs [64]. While this generally requires more lateral movement for the attacker to compromise all valuable resources, it also provides more occasions for defenses to unfold their potential and learn about their effects. However, the most important difference is that this case study relies on the previously presented

fuzzing approach. That is, specific network instances are not modeled manually as before, but derived from a single high-detail scenario definition that serves as a template for automated diversification. To test this functionality and investigate the spectrum and quality of results obtained from such auto-generated network instances, this experiment was chosen to be larger. Instead of evaluating defenses in only two benchmark networks, this number is scaled up by factor 250 through utilizing the newly added features of the modeling language to automatically generate 500 network instances for simulation. These 500 instances share the same basic network layout and general connectivity but differ in a number of aspects which have been declared subject to probability and conditions. In particular these include:

- No. of clients and servers in different subnets
- OSes of clients and servers
- Server types (e.g. CRM, SQL, Webserver etc.)
- No. of XEN hosts within subnets to migrate VMs
- Existence and distribution of different vulnerabilities in:
  - Hardware
  - OS
  - Software
- Reuse of credentials
- Storing and caching of credentials
- Firewall misconfiguration

This allows to evaluate attack and defense performance in different, yet equally realistic scenarios and get a better understanding of their impact on security. Using the same defenses as in the first experiment allows to investigate if the previously provoked incidental security degradation caused by individual defenses can also be observed when employing automated and unbiased diversification. Should this be the case, benchmark fuzzing would not only be convenient but also suitable to produce scenarios that yield equally meaningful insights. Furthermore, the increased scale, complexity, and diversity may reveal additional effects and shed light on how these are distributed. In the following, the general network layout, software landscape, as well as vulnerabilities and attacker actions will be explained.

6.4.1. Network Layout

The scenario definition employs a high degree of freedom to generate diverse benchmark networks in large quantities. Yet, the general network structure is fixed, consisting of nine specific subnets. That means, irrespective of randomized properties such as

the number of hosts in the demilitarized zone (DMZ), their configuration, or installed applications, there will be a DMZ in every benchmark network. The same goes for other subnets and structural properties so that all instances share a general layout as shown in Figure 6.1. The depicted circles represent subnets that form the network. These are connected through different types of arrows, representing different types of possible communication between them. This is either IP and port-based, out of band via removable media, or through e-mails. Remember from the meta model that IP and port-based communication depends on firewall rules that are usually inserted on the basis of deployed applications, however, this time also based on potential misconfiguration.

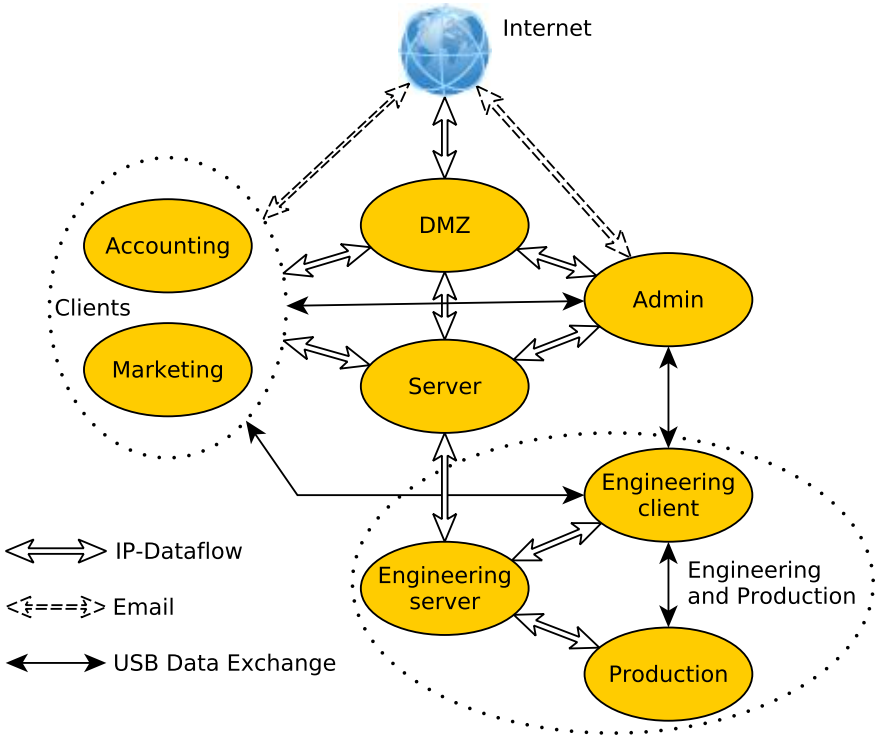


Figure 6.1.: Fixed subnets, represented as circles, that are randomly populated and interconnect through firewalls. Communication is possible as indicated by the different arrow types

The depicted structure has been chosen to account for company networks with production or development subnets that, to a certain degree, need to be isolated from regular network traffic for their criticality to operations or confidentiality reasons. As can be seen in Figure 6.1, network-based communication to "engineering and production" is restricted to the server subnet that may be required for keeping production numbers in sync with accounting, for example. However, there might also be unofficial communication from and to engineering clients via removable media to represent users that try to bypass inconvenient restrictions. This type of communication is implemented fairly simple through assigning a certain attribute to individual clients of the respective subnets,

marking those that may engage in such exchange. Communication is less restricted in the part of the network that can be considered as the office IT. There, clients access application servers in the DMZ and server subnet, while admins have to manage these. Exposed servers from the DMZ need to communicate with the server subnet for authentication services and the like. Additionally, clients from marketing and accounting, as well as admins are able to receive e-mails.

The ranges of the numbers of physical nodes, VMs, as well as applications that may be realized in the course of diversification for the different subnets in the scenario definition is given in Table 6.1. These are minimum and maximum numbers of instances of the given element type. As one can see, for applications in some client subnets, these numbers may rise quickly. This is due to conditions and probabilities employed in the scenario definition that ensure realistic scaling of client applications and other properties. For example, should the randomly picked number of virtual machines in the server subnet be high, conditions in the scenario will take care to equip these with applications that are randomly chosen from a list of potential server subnet applications. Otherwise the VMs would serve no purpose and just bloat the model. However, to be considered realistic, legitimate clients are needed to communicate with those servers. Therefore respective client applications are deployed on a set of randomly chosen clients that fit the application type to ensure that accounting software, for instance, is not deployed on the admin clients where they are most probably of no use. Depending on how these random numbers turn out, and how many clients have been instantiated in the respective subnet, this may lead to a high application count. At the same time, this pairing of client and server applications triggers the instantiation of firewall rules to allow for communication between respective addresses and ports. This logic has been used throughout the definition file to enable the generation of versatile and realistic benchmark networks.

Table 6.1.: Range of possible scenario size

Subnet	No. of Phys. Nodes	No. of VMs	No. of Apps
remote	1 - 3	0	6 - 45
internet	2	0	6 - 12
dmz	2 - 3	4 - 12	5 - 14
server	4 - 9	11 - 24	15 - 32
printer	1 - 3	0	0
engineering clients	6 - 9	0	3 - 51
engineering server	2 - 5	5 - 12	6 - 15
production	1 - 4	0	1 - 4
marketing	4 - 7	0	24 - 98
accounting	4 - 7	0	24 - 105
admin	2 - 4	0	2 - 4
Total	29 - 56	20 - 48	92 - 380

6.4.2. Software Landscape

The applications that are instantiated depending on probability and conditions have been selected with regard to functionality that is regularly required in a corporate context. These comprise Active Directory (AD) Servers, Microsoft Exchange, Apache Tomcat, SAP for production and accounting, but also different types of database servers, to name just a few. Applications come with typical actions one can reasonably expect to exist as to allow for interaction, extending the default templates presented in the meta model in Chapter 4.6. In that sense, the Apache2 server allows to retrieve websites, whereas the OSes allow to read from the file system, for example. On occasions where there is no obvious real world application as to represent a certain functionality, generic applications have been modeled and equipped with plausible actions. An example for such a generic application is the marketing server that is assumed to be accessed by client applications on nodes in the marketing subnet and allows, depending on the given privilege, to retrieve certain types of information.

Table 6.2.: Potential applications per subnet

Subnet	**Potential server applications**
dmz	Apache2, Tomcat, SQL, Exchange, Remote Shell/Desktop
server	Active Directory (AD), SQL, SAP, Max DB, Veritas Netbackup, Tomcat, Marketing (generic), Datev, SMB, Remote Shell/Desktop
engineering server	Git, Licensing (generic), SQL, SAP, Max DB, Remote Shell/Desktop
production	*generic application for deployment*
	Potential client applications
internet	DMZ: Apache2, Tomcat, Exchange
remote	DMZ: Apache2, Tomcat, Exchange; Server: AD
marketing	DMZ: Apache2, Tomcat, Exchange; Server: AD, Veritas Backup, Tomcat, Marketing
accounting	DMZ: Apache2, Tomcat, Exchange; Server: AD, Veritas Backup, Tomcat, Datev, SAP
admin	DMZ: Apache2, Tomcat, Exchange, SQL, Remote Shell/Desktop; Server: AD, Veritas Backup, Tomcat, Datev, SAP, SQL, Max DB, Remote Shell/Desktop

Table 6.2 gives an overview of the different applications that the benchmark network generator chooses from, in order to equip server and client nodes in the respective subnets. In this case study, more than one instance of an application can be deployed per subnet. This is not only true for SQL servers that are frequently used exclusively for specific services, but also for other applications. In these cases, the benchmark network generator re-evaluates related configuration parameters which are subject to automated diversification, including potentially existing vulnerabilities, so that the resulting application instance exhibits a different set of characteristics than previous ones. The client application listing contains additional information about the target subnet of a given

application. Just like applications, operating systems are subject to automated diversification, as well. The experiment differentiates between a Windows, Linux and XEN basis and determines parameters, as well as vulnerabilities accordingly.

6.4.3. Exploits and Legitimate Actions

As in the previous case study, the attacker's interaction is realized through actions that can generally be divided into exploits and legitimate actions. However, their number has been increased for this larger scenario, which is also due to the extended list of modeled software. These actions have been determined by scanning through the list of applications employed in the case study and identifying their core functionalities. Furthermore, basic OS-inherent actions that may be relevant in the course of an attack are included again. These comprise pinging hosts, opening remote shells, and reading from filesystems, to name just a few. In equal manner as before, vulnerabilities for employed software were obtained from the *CVE* database and *Metasploit*. These have been filtered with regard to age and score as to only consider those with a score higher than eight in a time range from the years 2016 to 2019. Vulnerabilities from this list have been selected and incorporated into the model if exploiting them yielded any of the following effects:

- allow restricted or privileged read access to data
- grant restricted or privileged remote code execution
- escalate privileges

In consequence, this list contains exploits known from the first experiment but also new ones that are related to the extended software catalog and the increased time span.

When comparing the different legitimate actions, but also exploits, similarities in underlying mechanisms, as well as effects become apparent. Actions to query SQL databases or fetch e-mails, for example, can be classified as authenticated reading of network-based resources. Many exploits, on the other hand, provide attackers with RCE privileges requiring no authentication at all. Based on these reoccurring patterns, generic actions and exploits have been crafted that are used to equip fictional applications such as the marketing server with plausible means of interaction.

All in all, this resulted in 48 actions an attacker can directly call, with Table 6.3 giving an overview of how these have been classified along different dimensions. While *type* differentiates between exploits and legitimate actions, *execution environment* distinguishes the setting in which these may be launched. In the context of this case study, this may either be through network connectivity to the target, RCE privileges on the target system, RCE privileges on any system that resides on the same physical host as the target, or through other channels such as e-mail or removable media. *Execution restriction* indicates whether the attacker needs to authenticate with help of credentials or not and, finally, *effect* categorizes actions with regard to the changes they cause to the system state. "Reading assets from applications" refers to obtaining aforementioned `usefulData` elements that have a defined value that serves to measure attacker revenue.

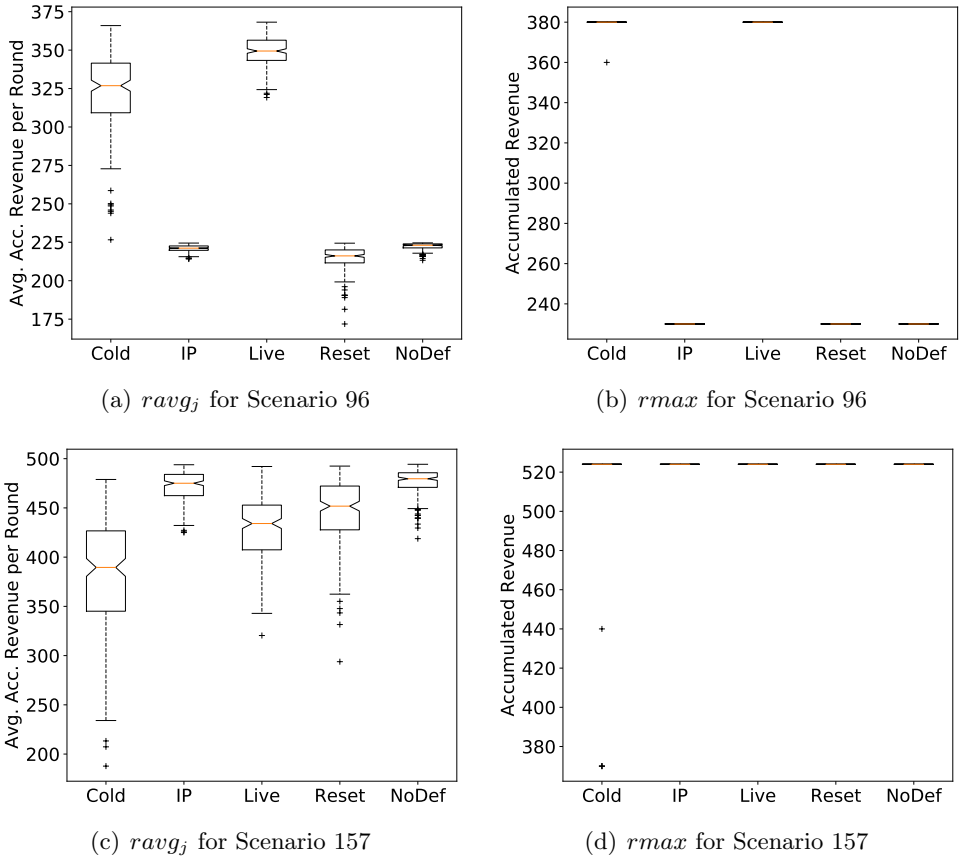
This may be credit card information stored in a database that the attacker can get hold of by different means. Differentiating between "full" or "restricted" simply refers to the privilege level. "Read only IP addresses" results in the attacker learning IP addresses. These have no inherent value but are needed for further attack steps. "Reading all data" from either applications or complete operating systems comprises extraction of any information, including `string` values for IP addresses, DNS names, usernames, passwords, etc. as well as `usefulData` elements. This may be the case when getting access to the filesystem to retrieve configuration files of different services as was also discussed in the context of the meta model. Lastly, "remote control" subsumes all actions that grant remote code execution, be it through legitimate ways such as a remote shell or through exploiting vulnerabilities. An exhaustive list of all implemented actions can be found in the supplemental material referred to in Appendix B, comprising definitions of their requirements and effects. Similar to the approach in the previous case study, duration and success rate of exploits have been manually determined on the basis of related vulnerabilities' descriptions, the underlying mechanisms, and the availability of exploit code (e.g. in *Metasploit*).

Table 6.3.: Attacker functions (48) grouped by specific attributes

Dimension	**Shape**	**Number**
Type	legitimate	24
	exploit	24
Execution Environment	Network	37
	Local System	4
	Local Host	4
	Others	3
Execution Restriction	Authenticated	21
	Unauthenticated	27
Effect	Read Assets from Application (full/restricted)	5/9
	Read all data from OS	7
	Read all data related to Application	3
	Read only IP addresses	2
	Remote Control (system/user)	20/2

6.4.4. Experimental Results

Due to the probabilistic nature of simulated actions, development of attacker revenue may vary across simulations, even when repeated for the same benchmark network and defense. Therefore, for every combination of the 500 benchmark networks, five defense



(a) $ravg_j$ for Scenario 96

(b) $rmax$ for Scenario 96

(c) $ravg_j$ for Scenario 157

(d) $rmax$ for Scenario 157

Figure 6.2.: Boxplots representing attacker performance for scenarios 96 and 157 based on *i*) the average accumulated revenue per round $ravg_j$ across different defenses and 200 independent simulations each and *ii*) the maximum accumulated revenue $rmax$ at the end of each simulation

configurations, and two additional control groups, $l = 200$ independent simulations are performed with $n = 12000$ rounds each. The number of rounds $n = 12000$ was chosen high enough so that for simulations with *no defense* in any given network, the attacker could, and did reach the maximum amount of achievable revenue at least once. However, on average, the maximum available revenue was obtained in 91.8% of the cases when *no defense* was employed. This results in a total of 700,000 simulations which is why simply plotting attacker revenue development for each of these will not yield any insights for the sheer amount of information. This is where the introduced metrics come into play.

Figure 6.2 depicts "notched" boxplots representing the average accumulated revenue per round $ravg_j$, as well as the maximum attacker revenue $rmax_j$ for networks 96 and 157 based on 200 simulations per defense. These were chosen exemplarily and the boxplots of all networks can be found in Appendix B, including data and Python scripts to generate them. In each plot, the upper and lower boxes depict the top and bottom quartiles,

representing the 25% of values above and below the median. The horizontal line that separates top and bottom quartile along the notch, is the median. For boxplots that display the average accumulated revenue per round, this is $ravg^m$. Whiskers, where present, indicate variability of values outside the quartiles, while outliers are represented as crosses. The notches in the boxplots represent the 95% confidence interval for the median, i.e., the area in which the real median is expected to lie based on the observed data with a probability larger or equal to 95%. This interval was calculated using the approach of McGill *et al.* [91] that relies on a Gaussian-based asymptotic approximation of the standard deviation and is defined as:

$$M \pm 1.7 \frac{1.25R}{1.35\sqrt{(N)}} \tag{6.5}$$

In this expression, M represents the observed median, R is the interquartile range defined through $R = Q3 - Q1$ and N is the number of observations which in the given case case is $l = 200$. To check on results, all simulations with *no defense* are performed three times in total, so that there are two control groups. Since these are based on the same inputs as the reference case, they are subject to the same probability distribution. Consequently, if sample size and simulation duration have been chosen adequately to employ said metrics, there should be no significant difference.

Figures 6.2(a) and 6.2(b) show boxplots of the average accumulated revenue per round and the maximum attacker revenue for network 96, that allow for interesting observations. Most notably, values of *rmax* are considerably higher for both types of *VM migration* as opposed to employing *no defense*. This means, that these defenses did not only fail to prevent attacks, but opened up new attack opportunities, just like in the first experiment's original network configuration for the 2016-attacker. The qualitative analysis of respective simulation logs reveals that this results from changed connectivity and co-location of VMs that occasionally increases the number of viable attack paths. The other defenses yield the same *rmax* as *no defense*, meaning that the number of successful attacks is the same. Looking at the average accumulated revenue one can see that *cold migration* and *live migration* also have a negative impact on this performance indicator, yielding higher values than *no defense*. *IP shuffling* has no considerable impact while *VM resetting* was able to slow down the attacker. Note that *cold migration* exhibits a lower average accumulated revenue per round than *live migration* which can be explained by the fact that *cold migration* also incorporates resetting of VMs.

In contrast, network 157 paints a different picture. As shown in Figure 6.2(d), in this network, *rmax* is the same across all defenses, meaning that none of them decreased or increased the number of successful attacks. However, as shown in Figure 6.2(c), both types of *VM migration*, as well as *VM resetting* have a throttling effect on the attacker. Furthermore, one can see that *cold migration*, being a combination of the other defenses, has the strongest impact. Judging from this case, *VM migration* appears to be a capable defense, similar to the findings from the previous experiment's second network configuration. The qualitative analysis of simulation logs reveals that the positive effect of *VM migration* in this network is due to disadvantageous initial placement of VMs. This

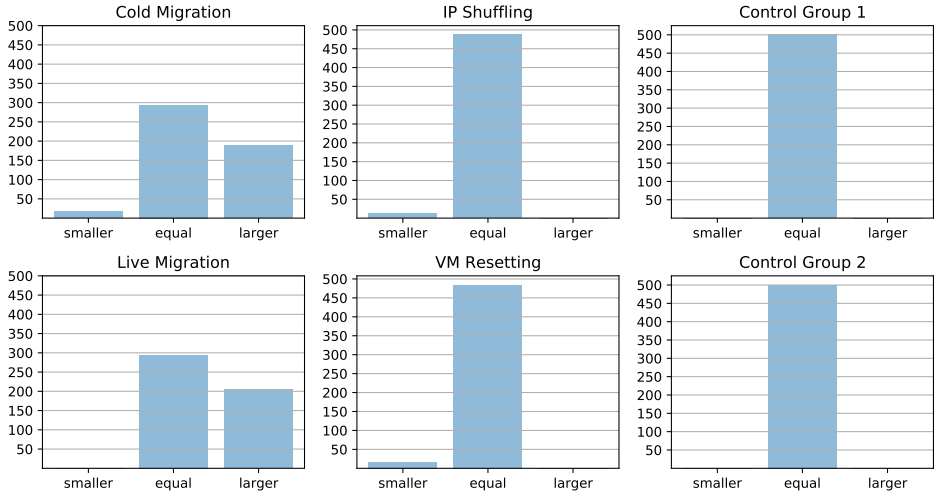
results in a relatively high number of viable attack paths so that subsequent relocation of VMs is more likely to cut existing paths off than establish new ones. Thus, VM migration slows down the attacker.

Comparing Defenses Among 500 Scenarios

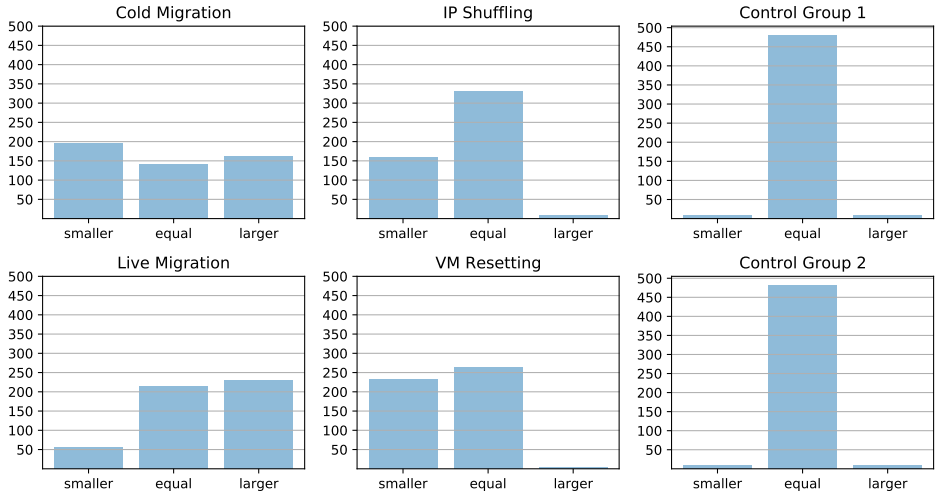
The important question that arises from these findings is which types of effects exist for the different defenses and how they are distributed. So far, the results from two specific networks have been investigated where different effects of defenses could be observed. However, this second experiment aspired to allow for investigation across numerous cases which is why simulation was conducted for 500 networks that have been generated with the presented benchmark fuzzing approach. Also, the claimed utility of the newly introduced metrics in aggregating data has not been shown yet since the boxplots still only concern one single network at a time.

For an overarching analysis, the maximum attacker revenue $rmax$, which basically represents the worst case attack scenario for a given network and defense shall be considered first. For each benchmark network, values of $rmax$ for all active defenses are compared with that of *no defense* to determine if these are smaller, equal, or larger, and categorize them accordingly. This ability to categorize allows for a compact representation in form of histograms, indicating how frequently different effects could be observed. Figure 6.3(a) shows respective histograms of all defenses together with the two control groups (where *no defense* was active) across all 500 scenarios. One can see that for *IP shuffling* and *VM resetting*, as well as the two control groups, $rmax$ is never higher than that of *no defense*, thus never causing any security degradation. On the other hand, in the presence of *cold migration* and *live migration* in 189, respectively 205, out of the 500 networks, $rmax$ increases. That means, in more than a third of all cases, migration actually has a negative impact on security. A reduction of maximum attacker revenue is only observed in very few cases (i.e., *cold migration* 17/500, *IP shuffling* 13/500, *live migration* 0/500 and *VM resetting* 16/500). In this regard, the considered defenses hardly ever keep the attacker from reaching her goal as compared to *no defense*. However, this is not too surprising, considering that the main goal of most *MTD* techniques is to slow an attacker down.

As opposed to categorizing and comparing $rmax$ across defenses, for which only maximum values matter, doing so for the average accumulated revenue per round requires a more sophisticated scheme. Looking at the boxplots in Figures 6.2(a) and 6.2(c), one can see that values for $ravg_j$ occasionally spread across wide ranges. Consequently, neither maximum nor minimum are able to adequately capture the observed effect. The median appears suitable, however, despite similarities, is hardly ever the same for different defenses, thus making it hard to decide when values are genuinely different or roughly the same. To this end, $ravg^m$ together with its 95% confidence interval shall be employed, which is indicated by the notches. For each defense it can then be checked whether the confidence interval lies entirely above or below the confidence interval of *no defense*. If so, its $ravg^m$ is classified as "larger", respectively "smaller". This is in line with McGill *et al.* [91] who state that non-overlapping confidence intervals indicate



(a) Max revenue



(b) Acc. Revenue per round

Figure 6.3.: Histogram of the attack simulations over 500 networks with **a** showing how often each defense resulted in a larger, the same, or smaller $rmax$ compared to *no defense*. In **b** the number of networks is shown in which $ravg^m$ for a chosen defense is significantly smaller or larger than that of *no defense*

significant difference. Should intervals overlap, values are generally classified as "equal".

Figure 6.3(b) depicts the resulting histogram of the average accumulated revenue per round. Again, including results from the two control groups for which simulations with *no defense* have been repeated. Note that a significance level of 95% means that there is a probability of up to 5% that a median is falsely classified as different. In the control groups, deviation in 4%, respectively 3.8% of the cases can be observed so that the assumption of a classification error smaller than or equal to 5% is not violated. Results show that *cold migration* and *live migration* cause a higher average accumulated attacker revenue in 32.4%, respectively 45.8% of the cases. *VM resetting* and *IP shuffling* exhibit negative effects in only 0.6%, respectively 1.6% of the cases, which, considering the confidence interval, might be the result of pure chance. In contrast, *live migration* reduces the average accumulated revenue in only 11.2% of networks which is the lowest followed by *IP shuffling* (32%), *cold migration* (39.2%), and *VM resetting* (46.6%).

In summary, *cold migration* may have both positive and negative effects. However, in the investigated benchmark networks, its positive impact on security is lower than that of using sole *VM resetting*. *Live migration* affects security mostly negatively. Consequently, both types of *VM migration* are not advisable since they open attack paths that would otherwise not exist. Instead, *VM resetting* and *IP shuffling* have the largest positive impact while hardly exhibiting any downsides. Among the two, *VM resetting* increases the security level considerably more (46.6%) than *IP shuffling* (32%). Note, that these findings concern the defenses' general effectiveness, given their duration times as delimiting factors. To what extent the most effective defense is attractive for a defender to employ, however, may depend on factors such as strategy, environmental conditions, and the cost that result from these. Nevertheless, in this specific scenario of a mid-sized enterprise network with given vulnerabilities and exploits, *VM resetting* performs best.

Coverage of Benchmark Fuzzing

The question, whether the introduced fuzzing approach is able to yield equally meaningful results as the manually tweaked benchmark networks, can be clearly answered with yes. Not only did fuzzing produce the same cases as seen in the initial case study, it also revealed new cases that had not been observed yet. Table 6.4 gives an overview of all encountered effects, ordered by frequency of occurrence. Two of the five most dominant effects that measurably deviate from employing *no defense* are the ones discussed at the examples of scenarios 96 and 157 where migration either degraded or improved security. However, extending simulation to include more networks of higher diversity reveals that *IP shuffling* was able to slow down the attacker in a considerable number of cases, as can be seen in both Table 6.4 and Figure 6.3(b). While not as effective as *VM resetting*, it still increases security. Furthermore, rare cases are revealed where *cold migration* and *live migration* cause $rmax$ to increase, for example, while at the same time, *cold migration* lowers $ravg^m$ that results from *VM resetting*. This highlights that relying on a few handcrafted benchmark networks is not enough to analyze defense techniques. Instead, it is advisable to base analysis on many benchmark networks that, with help of fuzzing, can be derived automatically from one single scenario definition.

Table 6.4.: Overview of observed defense performance combination for cold migration (C), live migration (L), VM resetting (R), and IP shuffling (I) with '+' indicating a larger, '-' a smaller attacker revenue

rmax CLRI	ravg CLRI	count	rmax CLRI	ravg CLRI	count	rmax CLRI	ravg CLRI	count
====	====	108	====	-+-+	2	-===	---=	1
++==	++==	74	-=--	-==-	2	=+==	-+-=	1
====	-==-	39	=+==	-==-	2	==-=	++--	1
++==	++-=	34	++==	----	2	++-=	++-=	1
++==	++--	28	-+-=	-+--	2	++==	==--	1
====	---=	26	-+--	-=--	2	++==	++-+	1
====	-=-=	24	====	++=-	2	++--	++--	1
====	----	23	====	==-=	2	-=-=	-=--	1
====	-=--	13	====	---+	2	++=-	++-=	1
====	-+-=	11	====	-+=-	2	++-=	-+--	1
++==	=+==	9	++==	=+--	2	++==	---=	1
====	=+-=	9	++==	-==-	2	====	++++	1
++==	-+-=	9	-===	-=--	2	====	=+==	1
====	++==	5	=+==	-+--	1	====	===+	1
====	++--	5	-+==	-==-	1	-===	----	1
++==	-+=-	5	=+==	-+=-	1	++=-	++--	1
++==	=+-=	4	=+==	-=-=	1	-=--	-=-=	1
====	-+--	3	====	=++=	1	====	+===	1
++==	-=-=	3	====	-===	1	==--	=+--	1
++==	++=-	3	-+=-	-=--	1	-+=-	-==-	1
++==	-=--	3	==-=	-+--	1	=+==	++=-	1
-+--	-+--	2	++==	++++	1	====	++-=	1
++==	====	2	====	===-	1			

6.5. Summary

This chapter introduced "benchmark fuzzing", an extension to the modeling and simulation framework that is based on substantial enhancements of the modeling language and the translator responsible for deriving valid Prolog code, allowing to automatically generate large numbers of realistic benchmark networks from a single scenario definition to address the previously identified need for evaluating defenses in a multitude of different, yet equally realistic settings. Scenario definitions that serve as input for this automated process might dictate the basic network structure, for example, while declaring other characteristics subject to probability. Networks generated from this definition will follow this prescribed structure, yet differ in aspects such as the number of nodes, deployed applications or any other parameter. This enables the framework to scale up evaluation with ease, reducing effort while increasing coverage. To test this new capability, as well as validate and extend findings on defense performance, further experiments have been conducted, using benchmark networks derived from "benchmark fuzzing". For this purpose, a scenario definition for a mid-sized corporate network was created and used

to automatically generate 500 benchmark network instances, serving as input for the simulator to evaluate the previously introduced defenses at a considerably larger scale. Furthermore, two new metrics have been introduced to quantify the security implications of the defenses under test. The first metric indicates how far an attacker can infiltrate the network for a given defense by measuring gained revenue, while the second metric exposes throttling effects on attacks.

Implementing these new features, the framework, previously focusing on modeling and simulation, advanced with regard to automation and scaling. Considering that previous experimentation had shown that for evaluation to be meaningful, it has to consider numerous different scenarios, automated benchmark network diversification and aggregated metrics to analyze results are not merely convenience features but actually hard requirements that the framework now fulfills.

7. Routing-based Moving Target Defense

While the previous chapters focused on evaluating *Moving Target Defenses*, the scope now shifts to anonymous and dynamic routing that is still underrepresented in *MTD* research. Though promising for considerably impeding reconnaissance if implemented and applied correctly, only few *MTD*-related publications address this topic. Furthermore, those publications that do cover anonymous and dynamic routing as a means of *Moving Target Defense*, regularly only consider dynamically changing communication paths through adapting the topology with help of SDN [65, 115]. The routing scheme presented here[1] is not tailored to the concept of *MTD*, yet may generally be employed for that purpose, as well. The following outline considers application in the internet for posing the biggest use case and providing the size, as well as layout for anonymous dynamic routing to unfold its potential.

The potential to improve security and privacy with help of specifically designed routing protocols stems from the fact that most communication networks, and in particular the internet, do not hide who is communicating with whom at what time. However, from a privacy perspective, such meta data are very critical as a lot information can be inferred from them. This applies to both global public networks such as the internet, as well as closed corporate networks. An attacker that gets hold of conveniently located nodes within a network will have an easy job during reconnaissance for obtaining a lot of relevant information through simply inspecting passing traffic. While reconnaissance is, in most cases, not an adversary's primary concern, it helps in preparing the next steps. Considering a global scale, an entity controlling an autonomous system (AS) can store meta data — who communicated with whom and at what time and which service — of all communication going through that AS. While not necessarily needing such information to prepare a subsequent attack, authoritarian regimes use such information to oppress and prosecute opponents. Furthermore, a malicious AS can manipulate the Border Gateway Protocol (BGP) to increase the traffic it can eavesdrop on [13, 126]. Technical changes to the internet infrastructure and the used protocols are therefore advocated by privacy-conscious actors [124]. The most famous initiative to counter surveillance is the Tor project [116] in which traffic is encrypted and relayed through hops to obfuscate the actual communication partners using onion routing. While Tor is still subject to traffic analysis and other attacks (e.g. [31, 74, 96]), it offers a good degree of privacy for the average user. However, the introduced overhead does not make it a viable solution to be used by every client and application. Tor's speed is often insufficient [52] and its scalability is limited [12, 93]. Transferring the concept of Tor to closed corporate networks is also challenging. By establishing an overlay

[1] The work presented in this chapter has been published [18] in the *Proceedings on Privacy Enhancing Technologies Symposium 2020, Issue 3*.

network which employs source-based routing, it cannot easily be adapted to operate effectively in a smaller scale. Furthermore, considerable adaptation on the client side would be required to make applications use it. Anonymous communication protocols with provable anonymity guarantees such as Mixnet-based systems exhibit an even worse performance than Tor [49]. For comprehensive surveys of such protocols see the work of Shirazi *et al.* [114] or Ren and Wu [109].

To tackle this problem, lightweight anonymity protocols have been proposed which focus on optimizing performance and deployment costs with the goal to allow large-scale utilization at the network layer. The Lightweight Anonymity and Privacy (LAP) protocol [67] is the first proposed protocol in this category. It is deployed at the network layer on top of the Internet Protocol and offers sender anonymity. Dovetail [111] builds upon LAP and adds receiver anonymity, to some degree, but requires user-controlled pathlet routing. HORNET [35] offers the highest degree of anonymity, yet requires source-based routing. The Path-hidden lightweight anonymity protocol (PHI) [36] builds upon the LAP and Dovetail protocols. Like LAP, PHI works with any inter-domain routing protocol such as BGP. From a privacy and security perspective anonymous and dynamic routing are favorable features. Performance requirements and costs are limiting factors, though, that impede the wide deployment of anonymity preserving protocols. Consequently, investigating how these can be realized in a cost-effective manner is an important task. Whether or not anonymity protocols will be deployed in global networks such as the internet is, in the end, a political question. However, as outlined before, anonymity protocols are not only interesting from a privacy perspective but also for network security. In large corporate networks they constitute a promising approach to decrease information leakage in case of compromised infrastructure. This, in turn, can hamper lateral movement, thus impeding attacks such as Advanced Persistent Threats (APT). With the rise of software-defined networking (SDN), implementing these protocols in IP networks has become very realistic and cost-efficient.

The contribution of the work in this chapter is twofold. Firstly, novel de-anonymization attacks against the PHI protocol are presented that significantly decrease receiver and sender anonymity. Secondly, an improved lightweight anonymity protocol named dependable PHI (dPHI) is proposed that withstands these attacks. The reason for focusing on PHI in the first place is that its application in closed corporate networks is generally feasible, while representing the state of the art for this kind of protocols. Suggesting an improved successor based on identified shortcomings is the obvious thing to do. However, before going into details of the PHI protocol and possible attacks on it, related work is presented to get an impression of the status quo in the field of anonymous and dynamic routing. Once PHI, its limitations, as well as the improved dPHI protocol have been presented, performance analysis is conducted to compare dPHI with other proposed protocols based on commonly used metrics. Finally, limitations are discussed.

7.1. Related Work

In the following, the most important lightweight anonymous routing protocols will be introduced that are relevant for PHI and dPHI. In the quantitative anonymity analysis in Chapter 7.5.2 not only PHI and dPHI are compared, but also LAP and HORNET for reference. Consequently, they are shortly introduced, as well.

LAP is the first network layer lightweight anonymity protocol and was presented in 2012 [67]. The goal of LAP is to provide sender anonymity while not impeding the routing of the network it is deployed in. The session establishment works as follows: A session request is sent from source to destination. In the session request message, a header field is used by each routing node to store information on how it routed the message. That is, each node stores the ingress port (from where the message has been received) and the egress port (to where the message is forwarded), encrypted with its individual secret key. This way, routing information can only be retrieved by the routing node that wrote the information, thus preventing other nodes from learning where the message originated. After the session request message reaches the destination, all further messages are exchanged based on the encrypted routing information in the header. Each routing node receiving a message decrypts the routing information and sends the message to the corresponding ingres or egres port. In consequence, an attacker that eavesdrop on a message does not learn the source address. However, an attacker can determine the distance to the source by counting the routing elements in the header. To obfuscate this information, LAP has the option to employ variable-size routing segments (VSS) with a parameter that will be denoted as VSS in the following[2]. In this case, for each routing element a random number (that does not exceed the VSS threshold) of dummy entries is added. This way the path length is obfuscated to a certain degree.

Dovetail was proposed two years after LAP by Sankey and Wright [111] and builds upon the basic idea of storing the routing information encrypted in the header. However, it adds some form of receiver anonymity by introducing a helper node to which session establishment requests are sent. The real destination of a message is encrypted with the helper nodes public key and, upon arrival at the helper node, decrypted and inserted into the message's header. The message is then forwarded to the destination via a tail node that lies on both the path from the source to the helper node, as well as from the helper node to the destination. One main difference to LAP (and PHI) is that Dovetail assumes pathlet routing, which is a client-controlled scheme. Hence, it interferes with any routing policy employed by the underlying infrastructure. Furthermore, the number of nodes on a path is not hidden in Dovetail so that an attacker can learn the distance to source or destination. Considering this, Dovetail proposes not to use shortest path routing, thus obfuscating the real distance by occasionally using longer paths than necessary.

HORNET [35] employs a source-based routing scheme in which the client chooses a path to the destination. For each routing node on the path, the client encrypts information on how to forward messages, using the respective routing node's public key, and stores it in the session establishment message. Each routing node receiving the session

[2] In LAP [67] this parameter was denoted as M.

establishment message decrypts its corresponding routing information and re-encrypts it with a secret symmetric key and stores it in the header. After session establishment, messages are exchanged as done in LAP and Dovetail using these encrypted routing information fields. Each routing node only learns its predecessor and successor on the path, offering the highest anonymity of the discussed protocols. However, in the internet, source-based routing undermines routing policies employed by intermediate nodes to balance load, for example. This makes adopting such routing policies unlikely. Besides anonymity, HORNET also introduces default payload encryption from hop to hop.

For completeness' sake, it should be noted that there have been several proposals to increase security in BGP inter-domain routing [29, 79, 85, 125, 132], including protocols that increase privacy [15, 60, 62]. However, privacy in these protocols does not refer to sender or receiver anonymity but rather focus on the privacy of the ASes and their business models. Mitseva *et al.* [95] compiled a comprehensive survey of such proposals.

7.2. System and Threat Model

The PHI and dPHI network models are similar to the internet architecture. It is assumed that two clients want to communicate over a network of routing nodes. In the internet analogy these routing nodes represent the interconnected Autonomous Systems (AS) that form the internet. Each node is connected with one or more other nodes over dedicated interfaces. Furthermore, each node can be connected to a multitude of clients. In the employed nomenclature, clients are denoted with lower case letters, while the routing nodes are denoted with capital letters, typically an A followed by an index.

Each communication is initiated by a client s, called source, that wants to communicate with a client d, called destination. The routing nodes that source and destination are attached to are referred to as A_s and A_d respectively. According to the used protocol, a path is established between s and d via routing nodes. The path between two nodes A_i and A_j is denoted with $P_{A_i-A_j}$ and comprises all nodes in the route path. An asterisk (*) indicates that the corresponding node is included in the path, i.e. $P_{A_i-A_j*}$ includes A_j but not A_i. In the following, $|P_{A_i-A_j}|$ is denoted as the number of nodes on the path $P_{A_i-A_j}$, i.e the distance between A_i and A_j, not counting A_i or A_j. Using the taxonomy of Kelly *et al.* [78], this translates to a *wired* network, with a *free* path topology and a *unicast* routing scheme.

7.2.1. Original Threat Model

Two different threat models are considered, with the first being the same as defined in PHI. It assumes that a single attacker controls exactly one routing node A_i and can ***i)*** read and store all packets passing through this node, ***ii)*** modify or stop all packets passing this node, and ***iii)*** send new packets originating from this node. Furthermore, it is assumed that the attacker knows the network topology and the routing policies of each individual node. Hence, for each packet the attacker can predict how a node would forward it. In principle, there are two attack goals:

- **Sender anonymity**: The attacker tries to de-anonymize the source of a communication request by minimizing the set of clients (the sender anonymity set size) that could have sent a message.
- **Receiver anonymity**: The attacker tries to de-anonymize the destination of a communication request. That is, he or she tries to minimize the set of clients (the receiver anonymity set size) that could be the recipient of a specific message.

According to the nomenclature of Kelly *et al.* [78] this translates to an adversary with *local* reachability, *dynamic* adaptability, and *active+passive/internal* attackability. The information the attacker can learn and the degree to which the attacker can de-anonymize the receiver or sender is defined in Chapter 7.5.1. The destination d does not need to be trusted by s when it comes to sender anonymity. That is, if the adversary is located at the exit node (and hence knows the destination), the security properties of dPHI and PHI hold even if d is malicious. Recently, Wails *et al.* [128] presented attacks against various anonymity protocols in case the source or destination moves within the network and the attacker located within the path between s and d is able to link sessions. While this is an interesting type of attack, it is not part of the threat model of dPHI.

7.2.2. Extended Threat Model

The original threat model from PHI can be seen as a realistic scenario in which a nation state actor has control over ASes which reside in her country. Due to their geographical proximity, these adjacent ASes could be modeled as a single large AS to be in-line with the threat model. Yet, getting control over an AS located geographically far away would be more difficult for such an actor. The assumption that an attacker controls a single AS is therefore realistic in many cases. However, one might argue that in the internet it is very easy to gain access to clients located at various different locations by setting up servers in different countries, using proxy servers, or renting botnets. Hence, it is reasonable to assume that an attacker does not only control a routing node but also a number of clients connected to different routing nodes. Therefore, the extended threat model assumes the same capabilities as before but add a fourth ability: ***iv)*** The attacker has full access to clients connected to different routing nodes. These clients can be used to receive and transmit messages.

7.3. The PHI Protocol and How to Attack It

PHI is a stateless routing protocol in which no information about sessions is stored within the routing nodes. Instead all necessary routing information for each AS is stored in the message header with the goal to anonymize the communication partners to malicious ASes or outsiders. In the following, a short introduction to the PHI protocol is given. For a detailed protocol description, the reader is referred to the original PHI publication [36].

Figure 7.1 gives an example of the routing process in PHI, where a source $s = c_1$ connected to AS A_1 wants to anonymously communicate with a destination $d = c_{14}$

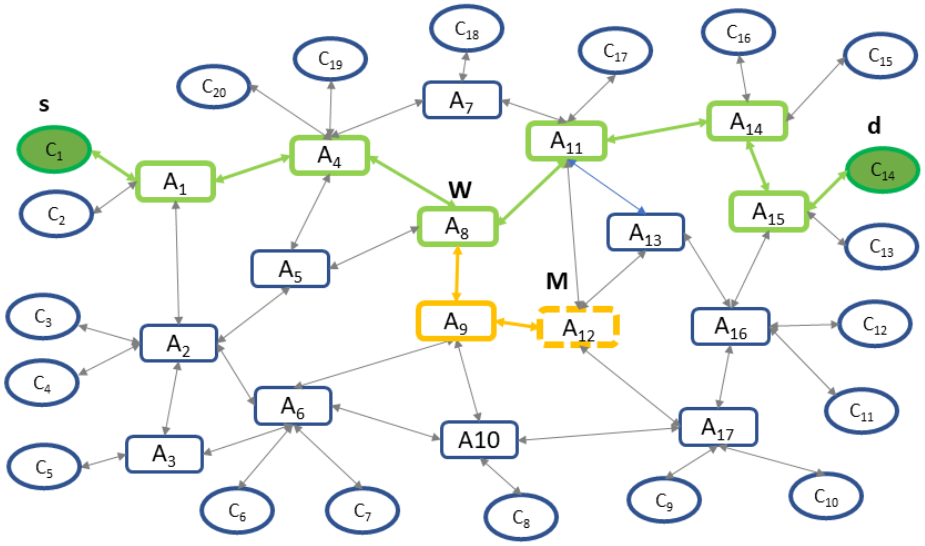


Figure 7.1.: An example routing procedure between source $s = c_1$, destination $d = c_{14}$ and helper node $M = A_{12}$ resulting in midway node $W = A_8$. The final path between s and d is depicted in green while yellow nodes are only used during setup.

connected to A_{15}. First, s chooses a helper node M and uses its public key to encrypt the destination address d. In this example $M = A_{12}$ was chosen. Afterwards, s sends a midway request to M that includes the encrypted destination d. Each node on path P_{s-M} stores its routing information, consisting of ingress and egress port to forward a given message, encrypted in the header's routing segment before forwarding it to the next node on the way to M. For this purpose, every node A_i has its own set of secret keys (k_i^{pos},k_i^{enc},k_i^{mac}), so that the encrypted routing information can only be decrypted and authenticated by the node that inserted it. The routing information is stored in the routing segment V at pseudo-random positions pos based on the node-specific secret key k_i^{pos}, session ID sid (which is derived from the session's fresh public key pub_s), and a Pseudo-Random-Function $PRF()$:

$$pos = \texttt{PRF}(k_i^{pos}, sid) \tag{7.1}$$

V is initialized by the sender s with random values that are indistinguishable from real routing information. This way, a node receiving a message does not learn anything about a message's previous path. Since a node does not know if a routing element already contains routing information, it may happen that two nodes write to the same routing element. In this case important information is lost so that the session establishment fails. Hence, session establishment in PHI is not dependable and may require several attempts to initiate a session. The authors of PHI proposed to send out four session requests in parallel to achieve a success rate of 90% with the default parameters in the

internet environment. The routing information R_i for node A_i consists of egress and ingress ports and is encrypted with:

$$c_i = \texttt{ENC}_{k_i^{enc}}(R_i||pos_{prev}||flags) \tag{7.2}$$

where pos_{prev} is the position of the routing segment which was inserted by the preceding routing node and transmitted in a specific header field. $flags$ contain additional protocol-specific information. A message authentication code (MAC) is used to verify the integrity of the current and previous routing element with:

$$m_i = \texttt{MAC}_{k_i^{mac}}(c_i||m_{i-1}) \tag{7.3}$$

where m_{i-1} is the MAC in the routing segment at position pos_{prev}. If M would directly forward incoming midway requests to d, the resulting path would be unnecessary long or violate routing policies such as valley-freeness [36]. For this reason, a backtracking phase was introduced in PHI. Instead of forwarding the message to d, it is sent back by helper node M to the previous node with d in the destination field. Each AS that receives such a midway request message decides based on the routing policy and the stored ingress information if it should become the midway node W. In the given example, a simple routing policy is used that checks if there exists a shortest path from the previous node to the destination which does not include the current node. If such a path exists, the message is sent back. Else, the current node becomes the midway node W. In Figure 7.1 this leads to selecting A_8 as the midway node W. Obviously, different routing policies to ensure goals such as valley-freeness, for example, can be employed just the same.

After midway node W has been found, the handshake message is forwarded to destination d. Each node on the path between W and d stores its routing information encrypted in the routing segment for later retrieval by the same node. Once d receives a handshake message it computes a session key k_{s-d} between s and d using ECDH key agreement with the session-dependent public key pub_s enclosed in the payload of the handshake message. The session ID is generated via a cryptographic hash with $sid = Hash(pub_s)$ to securely link it to the session-dependent public key pub_s and prevent man-in-the-middle attacks. Each AS on the path back to s uses its (encrypted) routing information stored in the routing segment to determine how to forward the message. When the message reaches s, the handshake is complete and s and d have established a shared session key and a PHI header to securely communicate with each other. Note that a session establishment will not succeed if an aforementioned collision occurs, which is why several session establishment requests are sent in parallel.

7.3.1. Attacks on PHI

In the following, several novel sender and receiver de-anonymization attacks against PHI are introduced. The quantitative impact on sender and receiver anonymity will be discussed at a later point in Chapter 7.5.2.

Passive Distance Leakage Attack

An attacker controlling a node $A_i \in P_{s-W}$ between the source and the midway node can store the routing segment V during the path request message. This allows the attacker to observe which elements have changed, once the handshake reply message is returned. Changed elements belong to nodes in $P_{A_i-M} \cup P_{W-d}$. Let the number of changed elements be a then $a = |P_{A_i-M}| + |P_{W-d}|$. Knowing M, the attacker can also predict $|P_{A_i-M}|$ and therefore learn $|P_{W-d}|$. However, the attacker does not know which node of the (known) path P_{A_i-M} is the midway node W. Consequently, she does not know $|P_{A_i-W}|$ and cannot directly compute the distance to d. Still, learning $|P_{W-d}|$ in conjunction with P_{A_i-M} may considerably reduce the anonymity set size.

Active Attack on PHI to Determine W and the Distance to d

The attacker on a node $A_i \in P_{s-W}$ between source and midway node can perform an active attack during the transmission phase to determine which elements belong to P_{W-M*} and which to path P_{A_i-d}. For this attack, it is assumed that the attacker is able to determine whether or not a data transmission reaches the destination d. Depending on the used application layer protocol, this could be inferred from re-transmissions from s or responses from d, for example. The attack is fairly simple: The attacker modifies single routing elements and observes if the message still reaches d. If not, the corresponding routing element belongs to P_{A_i-d} since only these elements impact successful routing from A_i to d. This way the attacker can learn $|P_{A_i-d}|$.

The attack's efficiency can be increased by combining it with the passive attack from before that reveals which elements in the routing segment belong to $P_{A_i-M} \cup P_{W-d}$. This way, the attacker only needs to test these. Furthermore, the routing elements belonging to P_{W-M} are the only ones that changed during the passive attack but whose manipulation did not result in a transmission failure. Hence, the attacker also learns the distance $|P_{W-M}|$ between midway and helper node. Since the path between the attacker node and the helper node is known, the attacker can use this information to precisely determine the identity of W. This, in turn, reveals information about the distance to destination d in a topology-based attack.

Distance Leakage to the Source

The same active attack as before can also be applied in a reverse order to determine the distance to s. Again, the attacker modifies routing elements of messages going from the attacker controlled node A_i to s and observes if the messages still arrive. The challenging part is that the source only expects a single valid path reply. Modifications of V during the transmission phase can easily be detected by s. In consequence, once a path reply reaches the source, it will not accept any further path replies. This problem can be circumvented with an attack strategy as follows: Starting with $i = 1$, the attacker modifies all but i routing elements in V. The attacker does this in all $\binom{l}{i}$ possible ways to send $\binom{l}{i}$ modified path reply messages. If the distance to the source is i, one of these path reply messages is valid and s sends back an answer. If not, the attacker knows that

the distance is greater than i and increases i by one and repeats the process. This way source s receives only a single valid setup message and will not get suspicious. Since the position $prev$ of the previous routing element is known, the attacker does not need to consider this position in her attack. Figure 7.2 illustrates this attack.

While the attack is efficient for small distances, the number of messages the attacker has to send increases exponentially with the distance to s and routing segment size m. However, in PHI the default segment size is $m = 12$, so that only 2048 messages need to be sent to test all distances from 1 to 12. Yet, for large sizes of $m = 48$, the attack quickly becomes impossible.

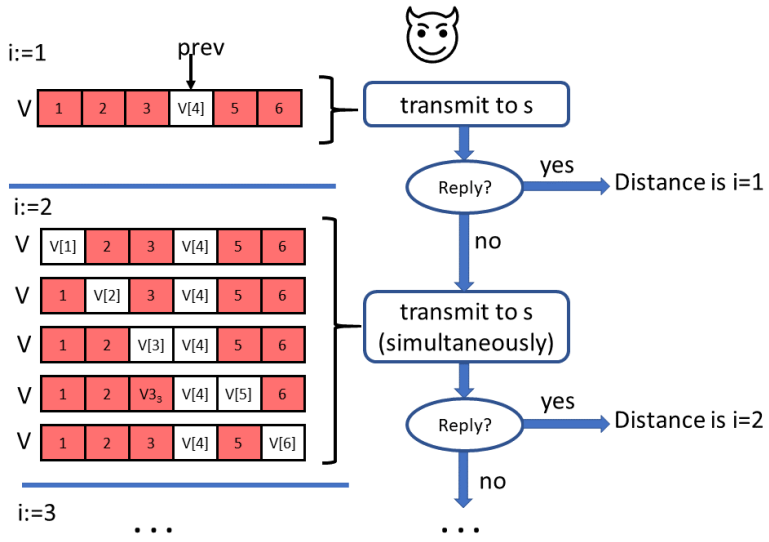


Figure 7.2.: Illustration of the attack to reveal the distance from A_i to s. In this example the routing segment V consists of six routing elements with $V[4]$ known to belong to the previous node. The routing segments the attacker modifies are indicated in red.

Attacks on Implementations Without Freshness

The PHI protocol does not specify the employed encryption scheme and in particular, it is not explicitly stated that freshness is needed for encryption. This omission could lead to implementations without freshness. Indeed, the implementation section in PHI [36] does not include any freshness and the performance was measured without additional seeds in the header. What follows, is a discussion on the implications should freshness be omitted in the implementation of PHI. Routing elements in PHI are encrypted without authentication using:

$$c_i = \mathtt{enc}_{k_i^{enc}}(R_i||pos_{prev}||flags) \tag{7.4}$$

with k_i^{enc} being the encryption key of the respective node, R_i the routing information, pos_{prev} the position of the routing element in V corresponding to the preceding node $i-1$,

and $flags$ containing protocol-specific information. Hence, the entropy of the plaintext is actually very low. If two messages of different sessions are routed in the same way, the corresponding ciphertexts will be identical if pos_{prev} is the same. Since $pos_{prev} \in \{1, .., 12\}$ with PHI's default parameters, chances for this to occur are considerably high. By comparing the ciphertexts c_i from different sessions, an attacker can detect routing elements that are equal with a high probability.

If an attacker starts many different sessions with publicly known destinations and different source addresses (e.g. by using proxies), she can craft a lookup table with the ciphertext and corresponding routing information for all ASes with moderate effort. Using such a lookup table, an attacker would in effect be able to decrypt the routing information without actually knowing k_i^{enc}. The only "freshness" stems from the fact that pos_{prev} is based on sid which is linked to a session's public key. Since the number of possible values for pos_{prev} is very small, so is the entropy. Furthermore, a malicious node can use the same sid and public key to send messages, mimicking the "forward to A_d phase" of the protocol. The attacker will not be able to finish the handshake due to the missing private key but can still observe the returned message header, thus detecting overlapping paths with only few messages. In particular, if the attacker wants to find out if a session belongs to a specific destination, a single message to d with the old session ID sid will do. If the suspicion is correct, the routing segment returned by the message from d will be identical.

Correlation of Failure Probability and Distance to Destination

Session establishment of the PHI protocol is unreliable with the failure probability depending on the distance between source s and helper node M, as well as the distance between midway node W and destination d. Details regarding the collision probability will be provided at a later point in Chapter 7.5.3. If the attacker is able to link different session establishments for the same destination d but different helper nodes M, these probabilities can be used to make predictions about d. The more data the attacker collects, the more accurate this prediction becomes.

How difficult it is to link session requests in practice, depends on the used application layer protocol and application. Obviously, payload encryption makes it considerably harder to link sessions when the attacker is not the entry node and therefore does not know s. However, for the entry node it is not unlikely that multiple session requests to the same (unknown) destination can be linked by observing the traffic flow and timing at a higher protocol level. Note that the attacker can drop some successful session establishment messages to force s to send out even more session establishment requests to get more data in shorter time.

Attacks Using Extended Threat Model to De-anonymize s

The previously described attacks are in line with PHI's original threat model presented in Chapter 7.2.1. Yet, in Chapter 7.2.2 an extended threat model has been introduced in which the adversary does not only control an AS A_i along the routing path but also

n clients $c_1, .., c_n$ connected to different ASes. This is a realistic assumption in an open network such as the internet, where an attacker could utilize proxies or resort to botnets. The attack discussed here assumes that the attacker has control over such clients.

The idea of the attack is to send modified backtracking messages from the attacker controlled AS A_i with different attacker controlled clients $c_1, .., c_n$ as the destination and the original routing segment V^1. Each client c_j receiving such a message records the routing segment V_j^1 and sends it to the attacker. The attacker then compares the routing segment V_j^1 with V^1 and counts the number of changed routing elements. The attacker knows that for a client $c_j \neq s$, one node in P_{s-A_i} is chosen as the midway node W_j. The number of changed routing elements observed in V_j^1 is equivalent to the distance $|P_{W_j-c_j}|$ between this midway node W_j and the attacker controlled client c_j. In a second step, the attacker can also determine $|P_{A_i-W_j}|$ by sending a modified backtracking message to client c_j, again, but this time with modifying some routing elements in V^1 to determine which modifications lead to message drops during backtracking. This is the same approach as done for the previously described active attacks to determine distances to d and s. The degree of de-anonymization in this attack is very high as the attacker can use many different clients, resulting in many different midway nodes W_j, to narrow down the possible positions of s. Note that the attack becomes easier if the attacker also exploits the missing freshness in PHI's encryption as identical ciphertexts show common paths between different nodes.

7.4. The Improved dPHI Protocol

The dPHI protocol is generally based on PHI, yet, implements several modifications to withstand the discovered attacks. The modifications are based on five main ideas: *i)* splitting the routing segment into two parts to prevent passive distance leakage, *ii)* circular insertion of routing elements instead of picking pseudo-random positions to avoid collision and enable integrity checks, *iii)* extending the backtracking phase back to source s and adding integrity checks of the routing segment, *iv)* addition of a midway nonce to prevent attackers from modifying the destination, and finally *v)* the consistent use of authenticated encryption with fresh IVs to ensure freshness and high entropy ciphertexts. In the following, these changes are discussed in more detail. Furthermore, a formal protocol description with pseudo code can be found in Appendix C.

7.4.1. Key Management

In the dPHI protocol, every node A_i has a secret symmetric key k_i that is only known to this one node and used to encrypt and decrypt its own information in the routing header. Furthermore, every node A_i that can serve as a helper node has an ECDH key pair $(pub_i, priv_i)$. The public key pub_i of the chosen helper node needs to be known to source s in advance. In addition to these static keys, the source s generates a session key k_{s-M} with helper node M and a session key k_{s-d} with destination d using ECDH key exchange during the dPHI session establishment. The established session key k_{s-d} can be used by both source and destination to exchange encrypted messages that are

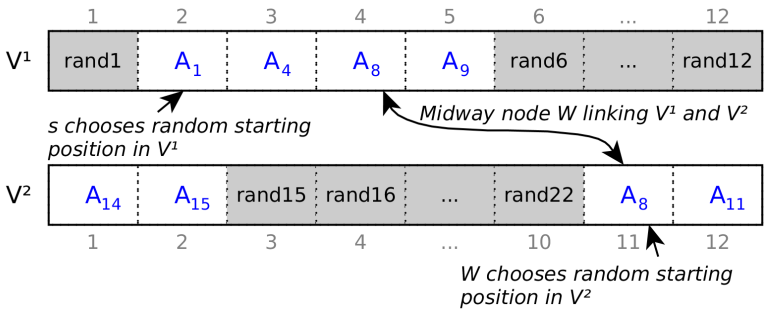


Figure 7.3.: Illustration of routing segments V^1 and V^2 for the example from Figure 7.1 with $A_s = A_1$, $M = A_9$, $W = A_8$ and $A_d = A_{15}$. Randomly initialized values are depicted in gray.

protected against man-in-the-middle attacks. Note that no payload encryption is done by nodes in dPHI and hence needs to be taken care of by the clients.

7.4.2. Authenticated Encryption

All symmetric encryption in dPHI is performed using an authenticated encryption algorithm such as AES-GCM [92]. An authenticated encryption algorithms gets three inputs, the key k, the plaintext p and additional authentication data a which is authenticated but not encrypted. The output is a triplet consisting of a freshly generated IV IV, authentication tag t and the corresponding cyphertext c with $(c, t, IV) = \texttt{enc}_k(p, a)$. In dPHI, the session ID *sid* is always part of the additional authentication data a to ensure that the ciphertext is securely linked to the current session.

7.4.3. Modification to the Routing Segment

Two major changes are applied to the routing segment V. The first is to split it in two, with V^1 storing routing information for nodes on path P_{s-M} and V^2 storing the routing information for nodes on P_{W-d}. Furthermore, routing elements are inserted into V^1 and V^2 in a circular manner, starting at random positions that are chosen during their initialization. Both V^1 and V^2 default to length $l = 12$ but can be adapted individually to fit other use cases. Figure 7.3 depicts the shape of V^1 and V^2 for the example from Figure 7.1. Figure 7.4 shows the complete dPHI header. V^1 is initialized by source s and V^2 by midway node W using keyed cryptographic pseudo random number generators (CPRNGs) based on fresh and secret seeds. While s initializes V^1 immediately, V^2 is initialized by W after the backtracking phase, before forwarding the handshake to d. The seed used to initialize V^2 is stored by W encrypted in the midway field in the dPHI header. As shown in Figure 7.3, midway node W stores its routing element in both V^1 and V^2 (encrypted with different IVs).

Another modification in dPHI is the routing information R that is encrypted and stored in a routing element. In PHI, it only consisted of ingress and egress ports. In

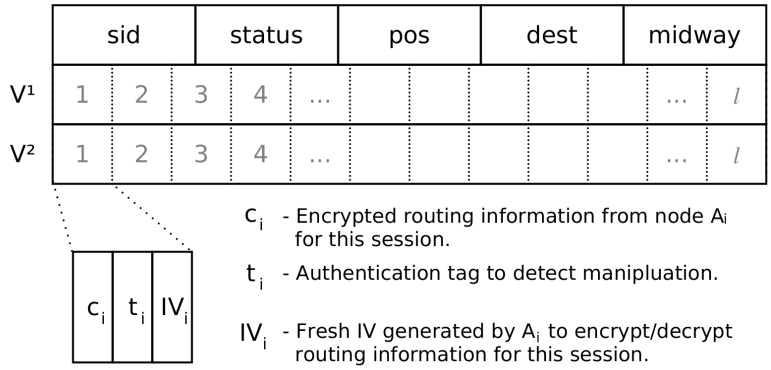


Figure 7.4.: Illustration of the dPHI header showing V^1 and V^2 that form the routing segment. Note that field sizes in this representation do not relate to real sizes of header elements.

dPHI, it also contains two pointers $posV1$ and $posV2$ indicating the positions where the current routing element is stored in V^1 and V^2. If it is only stored in one of the segments, the other pointer is null. These pointers are used to detect if the position of the routing elements have been altered and enable the midway node to find its field in the other routing segment. A *type* flag indicating that the node is the midway node is stored in R, as well. Equation 7.5 shows how a routing element consisting of (c_i, t_i, IV_i) is computed:

$$\begin{aligned} (c_i, t_i, IV_i) &= enc_{k_i}(R; sid||c_{pos-1}) \\ R &= (ingress||egress||type||posV1||posV2) \end{aligned} \tag{7.5}$$

See Algorithms 2 and 3 in Appendix C for details.

7.4.4. Backtracking to s and Verification of V^1 and V^2

In PHI, backtracking is done from helper node M to midway node W, where the message is forwarded to destination d. To enable the source to verify the integrity of V^1, backtracking in dPHI is changed as to reach all the way back to s. The node that becomes the new midway node W, encrypts the destination address d with its secret key and stores it in the destination address field in the header. Then, the message is sent back to s. Consequently, nodes between W and s do not learn about d, while midway node W can retrieve the destination using its secret key in the next phase of the protocol when s sends the handshake to d. This modified backtracking phase allows source s to check if V^1 has been tampered with by comparing it to the initial version of V^1 that s stored while preparing the request. In dPHI it is assumed that s knows the maximum distance to the chosen helper node M. The idea of this integrity check is that with knowledge about the distance, s can estimate which routing elements in V^1 should have changed and which not. If modifications outside the expected range occur, s drops the message

and may initiate a new attempt. Details of this step are provided in Algorithms 4-5 in Appendix C.

Midway node W performs the same integrity verification of V^2 as s does of V^1. W initializes V^2 when it forwards the message to d for the first time, based on a fresh seed and a keyed CPRNG (see Algorithm 7 in C.4). To be able to retrieve the seed when the message gets back, W encrypts the seed together with a close upper bound $dist_d$ of the distance to d with its own secret key and stores it in a dedicated midway field in the header. In addition to the session id sid, the ciphertext c_{V1} of W's routing element in V^1 is used as additional authentication data for this encryption.

$$midway_{(to\_d)} \leftarrow \texttt{enc}_{k_i}(seed||dist_d, sid||c_{V1}) \tag{7.6}$$

When the midway node receives the handshake reply from d, it decrypts the seed and distance value and verifies that only the expected routing elements in V^2 have changed. The additional authentication data securely links V^1 and V^2 so that an attacker cannot simply replace V^1 or V^2 as a whole without the midway node noticing. After verifying V^2, the midway node W replaces the midway field with a MAC of the entire routing segment comprising V^1 and V^2 to ensure this secure linkage for the remainder of the protocol.

7.4.5. Addition of the Midway Nonce

The last major change in dPHI is the addition of a midway nonce to prevent attacks in the extended threat model. During initialization, s generates a nonce n_{mid} (see Appendix C.2-C.5 for details). This nonce is encrypted with the shared session key k_{s-M} that s derived with help of M's publicly available key pub_M. To enable M to derive that exact same key, s includes pub_s of its freshly generated key pair in the payload when sending its request to M. The helper node M decrypts this nonce and puts it, unencrypted, in the midway field of the backtracking message that is sent back. Midway node W, in turn, computes a hash value of the midway nonce together with the destination $dest$ and routing segment V^1 and overwrites the midway field $midway_{(to\_s)}$:

$$midway_{(to\_s)} \leftarrow \texttt{Hash}(dest||n_{mid}||V^1) \tag{7.7}$$

When source s receives the message at the end of the backtracking phase, it verifies this hash value and that the correct destination $dest$ has been used. If this is not the case, the message is dropped.

7.5. Analysis of dPHI

To check if dPHI really is an improvement over PHI, it shall be analyzed with regard to three aspects. First, it needs to withstand the previously outlined attacks on PHI since these motivated the development of dPHI in the first place. Second, to determine the degree to which dPHI fulfills its primary purpose, that is providing anonymity of senders

and receivers against adversaries controlling routing nodes, a quantitative anonymity analysis is conducted. Third, to find out about the cost of the security improving modifications, dPHI is analyzed with regard to computational effort and goodput.

7.5.1. Security

In Chapter 7.3.1 several novel attacks on PHI have been presented. Whether or not the modifications introduced with dPHI are able to withstand these, also under consideration of the extended threat model, is discussed in the following.

Passive Distance Leakage Attack

The passive distance leakage in PHI is the main reason for splitting the routing segment into two parts, V^1 and V^2. The number of changed elements in V^1 that a node in P_{s-M} can observe is equivalent to the distance to the known helper node M. Hence, the attacker does not learn anything new by counting the changed elements in V^1. When a node in P_{s-W} receives V^2 for the first time, all values are new as they are initiated by midway node W. Consequently, a node in P_{s-W} cannot make any observation about d. The routing element corresponding to W changes twice in V^1. However, no single node receives both versions so that W cannot be identified based on any such observation. All nodes in P_{W*-M} and P_{W*-d} know d so that distance leakage attacks to the destination are only interesting for those in P_{s-W}.

Active Attack to Determine W and the Distance to d

In dPHI, an attacker controlling $A_i \in P_{s-W}$ would need to (repeatedly) modify routing elements in V^1 to identify which of them belong to P_{W-M} or W. To prevent this, backtracking is extended beyond W so that source s receives V^1 after backtracking. Destination d receives V^1 with the handshake message. Hence, any modification of V^1 after the handshake message is detected by s or d as they always check the integrity of received headers. To detect manipulations during the phases of backtracking to W and backtracking to s, the source uses the new midway reply message in the dPHI protocol:

$$H.midway_{(to\_s)} = n_{rep} = \texttt{Hash}(d||n_{mid}||V^1) \tag{7.8}$$

The midway reply message is generated by midway node W, the last node to modify V^1. Destination d and n_{mid} are known to all nodes on P_{W*-M*} but not to nodes on P_{s-W}. In consequence, an attacker on $A_i \in P_{s-W}$ cannot compute a valid midway reply for a modified V^1. This way, source s can detect any modifications to V^1 after processing by midway node W.

To prevent attacks during the handshake message to d, source s sends a MAC of V^1 with the shared session key to destination d. The destination can therefore authenticate V^1 based on the shared session key and discard any message with a modified V^1.

Distance Leakage to the Source

To prevent leakage of information on the distance to the source, s stores the random values that were used to initialize V^1, the starting position pos pointing to the first routing element, and the maximum distance $dist_M$ to M. On receiving the midway reply, s uses this information to verify that only routing elements belonging to the path P_{s-M} have been modified, these are elements in V^1 between pos and $pos + dist_M$. If other elements have been modified, the message is dropped by s. This results in the same effect (i.e., session establishment failure) as if an element belonging to P_{s-A_i} had been modified by the attacker. After the midway request, s stores V^1 and drops any message in which the routing segment V^1 is not identical. This ensures that the attack on V^1 can also not be executed in a later stage of the protocol.

However, an attacker on $A_i \in P_{W-d}$ could try to manipulate V^2 to learn at least the distance between A_i and (the unknown) midway node W. To prevent such attacks, the midway node W verifies V^2 the same way as s has verified V^1. Midway node W has stored the seed for initializing V^2 and the distance $dist_d$ between W and d encrypted in the midway field of the header. With this information, the midway node verifies that at most $dist_d$ routing elements have been inserted in V^2, starting from position pos_2 that W retrieves from its own routing element. At the end of the session establishment, source s stores V^2 so that any modifications of V^2 during the data transmission phase are detected. As a result, in dPHI the attacker cannot learn the distance to s nor the distance to W.

Exploiting Missing Freshness

The dPHI protocol proposes to use authenticated encryption with fresh IVs whenever information is (re-)encrypted. Hence, the attacker cannot learn anything by observing ciphertexts. The position pos within a routing segment in dPHI is independent from a secret key or node address since routing elements are inserted in a circular manner. Therefore, which routing elements in V^1 or V^2 are modified cannot be used to link sessions or nodes.

Correlation of Failure Probability and Distance to Destination

Through circular insertion of routing elements, dPHI ensures that no collisions occur as long as the maximal sizes of V^1 and V^2 are not exceeded by the chosen path to M and d. The session establishment is therefore dependable making this type of attack impossible.

Active Attacks in the Extended Threat Model

In dPHI the backtracking phase has been altered so that the midway node sends a midway reply back to the source s. The source s then verifies that the correct destination has been used to determine the midway node W. To accomplish this, the midway node W computes a midway reply $H.midway$ by hashing the midway nonce n_{mid} with the destination d and routing segment V^1, i.e. $n_{rep} \leftarrow \texttt{Hash}(d||n_{mid}||H.V^1)$. This

midway reply n_{rep} is sent back in the midway field $H.midway$ of header H. The source recomputes this midway reply and can thereby verify that the midway node W has received the correct destination d.

It is important to ensure that an attacker cannot misuse the midway reply $H.midway$ to learn the destination d. The high entropy midway nonce n_{mid} has been sent encrypted to the helper node M. Only M and nodes on P_{W*-M}, the same that also know destination d, learn the midway nonce which is necessary to compute $H.midway$. Nodes in P_{s-W} do not know the midway nonce and can therefore not compute $\texttt{Hash}(d||n_{mid}||V^1)$. In consequence, it is not possible for an attacker on these nodes to misuse $H.midway$ as an oracle to determine d.

Information Leakage in dPHI

dPHI does not offer perfect anonymity as the attacker can learn some information about communicating peers. The following elaborates on which information the attacker is able to learn about sender and receiver depending on the compromised node.

Sender anonymity:
Attacker on entry node A_s:

1. The entry node knows the source address s, i.e., there is no sender anonymity.

Attacker on A_i with $A_i \in P_{A_s-W}$ or $A_i \in P_{W-M*}$:

1. For (known) destination M the path P_{A_s-M} goes through $A_{i-1} \cup A_i$, thus reducing the anonymity set of s to sources reachable through A_{i-1}.
2. The distance to s cannot be higher than the maximum segment length m minus the distance $|P_{A_i-M}|$.

Attacker on midway node W:

1. Learns the same as attacker on P_{A_s-W} or P_{W-M*}.
2. W can perform an active attack by violating the routing policy and deciding not to become the midway node for the known destination d. The attacker can then observe whether or not the node W will be selected for the path $P_{W'-d}$ (where W' is the new midway node with $W' \in P_{A_s-W}$). In how far this de-anonymizes s depends on the deployed routing policy and layout of the network.

Attacker on A_i with $A_i \in P_{W-d*}$ learns:

1. For the (known) destination d the path P_{W-d} goes through $A_{i-1} \cup A_i$, thus reducing the anonymity set of unknown W.
2. Furthermore, it is known that the distance to W cannot be greater than the maximum segment length m minus the distance from A_i to d.

Receiver anonymity:
Attacker on A_i with $A_i \in P_{A_s*-W}$ learns:

1. Destination d can be reduced to such destinations that one of the nodes on the predictable path to M (i.e., P_{A_i-M*}) can serve as midway node W without violating the employed routing policy.

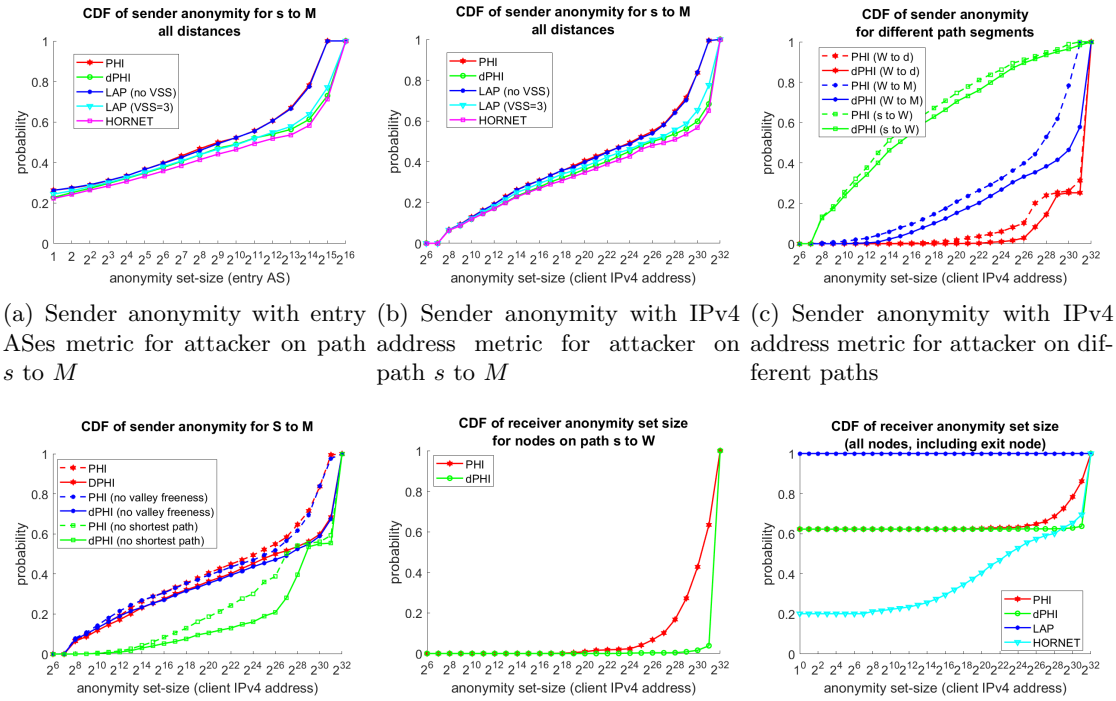
Attacker on A_i with $A_i \in P_{W*-d*}$ or $A_i \in P_{W-M*}$:

1. These nodes learn destination d so there is no receiver anonymity.

The entry node learns the identity of source s and the midway node the destination d which results in zero anonymity if the entry node also becomes the midway node. This is a major drawback of PHI/dPHI since topology information is required to choose M such that this does not happen.

7.5.2. Quantitative Anonymity

Having investigated how dPHI withstands the previously introduced attacks, the achieved sender and receiver anonymity for the lightweight anonymity protocols LAP, HORNET, PHI and dPHI are compared next.



(a) Sender anonymity with entry ASes metric for attacker on path s to M

(b) Sender anonymity with IPv4 address metric for attacker on path s to M

(c) Sender anonymity with IPv4 address metric for attacker on different paths

(d) Sender anonymity with IPv4 address metric for attacker on path s to M with different routing policies

(e) Receiver anonymity with IPv4 address metric for attacker on path s to W

(f) Receiver anonymity with IPv4 address metric for attacker located anywhere on the path (including exit node)

Figure 7.5.: Quantitative sender and receiver anonymity set size analysis. Results are shown as cumulative distribution functions (CDF) in which the y-axis shows the probability that the anonymity set size is equal or smaller than the value depicted in the x-axis.

Assumptions and Experimental Setup

Computing the anonymity depends on several factors such as the network topology, the routing policies and the attacker's capabilities. In PHI [36] and HORNET [35], anonymity set size was computed based on a valley-free routing policy and the 2014 CAIDA dataset [68], representing the internet infrastructure on the level of Autonomous Systems. In the valley-free routing model, the network is a directed graph in which each edge is either of the type provider-to-customer or peer-to-peer. A valley-free path is a path that starts with any number of customer-to-provider edges followed by zero or one peer-to-peer edge and any number of provider-to-customer edges. PHI and HORNET [36, 35] assume all valley-free routes as equally likely, regardless of the length of the route. However, for a given source and destination pair it is reasonable to assume that only a predictable subset of paths will be chosen in practice based on the network topology, e.g. the shortest paths.

Therefore, one might argue that a network model in which only a selected number of paths between two nodes is valid, is more realistic. For analysis, a shortest path valley-free routing policy is employed, in which all valid paths need to have the minimum number of hops. Sender and receiver anonymity set size have been computed for PHI, dPHI, LAP with one element per hop (i.e., no VSS), LAP with three elements per hop (VSS=3), and HORNET. For this, 1000 random source and destination pairs have been chosen and a PHI path computed for each by randomly choosing a helper node. Should no valley-free path exist or the midway node be identical with the entry node, new random nodes were chosen. The computation was performed using Matlab and the 2014 CAIDA dataset [68]. The software as well as used data is publicly available on Github [17]. The quantitative analysis is based on the following assumptions and metrics:

Attacker capabilities: For PHI, the passive and active attacks from Chapter 7.3.1 are assumed but not the attacks exploiting low entropy or those based on malicious clients.

Routing assumption: If not specified otherwise, a valley-free shortest path routing is used in which each shortest path is equally likely to be chosen. Only in Figure 7.5(d) other routing schemes (non valley-free shortest path and valley-free without shortest path) are used.

Anonymity set size: Computation of the anonymity set size is based on two metrics: 1) The *number of ASes* that could be the entry node (sender anonymity) or the exit node (receiver anonymity). And 2) the size of the *IPv4 address space* associated with the ASes that could be the entry node or exit node, respectively.

Location of attacker node: The location of the attacker on a PHI or dPHI path greatly influences the anonymity set size as outlined in Chapter 7.5.1. The attacker location for dPHI is therefore specified in each figure. For LAP and HORNET, all nodes between the source and destination are used as attacker locations for the anonymity set size comparison as there is no midway node or helper node in these schemes.

Sender Anonymity

Figure 7.5(a) shows the CDF of the sender anonymity set size for nodes on the path from source s to midway node M. The sender anonymity set sizes for LAP and HORNET for the path s to d are also depicted for comparison. In this figure, anonymity set sizes based on entry ASes are shown while Figure 7.5(b) presents the same information on the basis of possible IPv4 source addresses. Please note that the anonymity set size on the x-axis has a logarithmic scale with base 2 so that a shift to the right by one is already an increase by factor two. As one can see, PHI only achieves the anonymity level of LAP without VSS while dPHI performs slightly better than LAP with $VSS = 3$.

The sender anonymity for paths W to d and W to M is shown in Figure 7.5(c). These paths are especially interesting for PHI and dPHI as enclosed nodes learn the destination. The anonymity set size for W to d is considerably larger than W to M since the backtracking algorithm in PHI does not guarantee a shortest path routing. The anonymity set size in Figure 7.5(c) for dPHI, therefore, consists of all nodes that are reachable via the previous hop without compromising valley-freeness. In PHI, the attacker can learn the path length, hence, all nodes that cannot be reached with a number of hops that is less or equal to the path length are excluded. Note, however, that in 97.5% of the sessions a shortest path was chosen and in 2.5% of the cases, the resulting path was only one hop longer. Only in one out of one thousand sessions, the resulting path was two hops longer. If an attacker ignored the 2.5% chance of a longer route, the anonymity set size for W to d looked like that from W to M. The anonymity set size decreases, the closer the attacker-controlled node is to the source, which is why the anonymity set size for nodes on P_{s-W} is the lowest as compared to nodes on P_{W-M}.

The impact of different routing policies has also been tested and is depicted in Figure 7.5(d). If no shortest path routing is assumed, i.e., all valley-free paths are considered to be valid, the anonymity set size increases significantly. Using a shortest path routing policy without requiring valley-freeness also results in a larger anonymity set size than the valley-free shortest path routing policy. But this difference is considerably smaller than compared to not using a shortest path routing policy.

Receiver Anonymity

The receiver anonymity set size is depicted in Figures 7.5(e) and 7.5(f). LAP does not provide receiver anonymity, hence, the anonymity set size is 1. PHI and dPHI only provide receiver anonymity for nodes on the path P_{s-W} from the source to the midway node. In the given experiment this corresponds to 60% of the nodes being able to eavesdrop d during session establishment. While the receiver anonymity set size for both PHI and dPHI is already very high, dPHI outperforms PHI considerably. Note that a high receiver anonymity is especially important for the entry node A_s which knows the source so that the combined sender-receiver anonymity solely depends on the receiver anonymity.

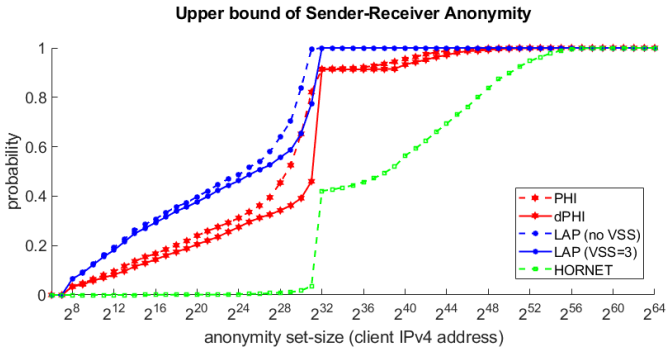


Figure 7.6.: Upper bound of sender-receiver anonymity set size with IPv4 address metric for an attacker located anywhere on a path.

Upper Bound of Sender-receiver Anonymity

Computing the sender-receiver anonymity for PHI and dPHI is computationally extremely expensive since all possible paths from all sources to all destinations have to be computed for all helper nodes. Therefore, only an upper bound of the sender-receiver anonymity set size has been computed. This was done by multiplying the sender anonymity set size for an attacker on node A_i (i.e., all possible sources for this session establishment) with the receiver anonymity set size for the same node (all possible destinations). The correct source and destination pair lies within this upper bound. However, the actual sender-receiver anonymity set size might be smaller as for some sender-receiver pairs, there may exist no helper node such that a path traverses through the attacker node. Yet, this upper bound provides a rough estimation of the sender-receiver anonymity set size. Figure 7.6 shows the upper bound of the sender-receiver anonymity. dPHI considerably outperforms LAP in this metric. Nodes that have a receiver anonymity set size of 1 in dPHI (nodes on path P_{W-d} and P_{W-M}) exhibit a relatively high sender anonymity as shown in Figure 7.5(c)), while nodes with low sender anonymity (close to source s) have a relatively high receiver anonymity. This ensures that the minimum sender-receiver anonymity set size remains fairly high in dPHI. HORNET clearly performs the best in terms of sender-receiver anonymity as only the entry and exit nodes have a sender or receiver anonymity set size of one.

7.5.3. Performance

In the following, the performance of dPHI in terms of computational complexity, latency, header size and goodput is compared to PHI. In particular, the header size and the performed cryptographic operations are different and the backtracking phase is extended so that more nodes are traversed during session establishment. However, session establishment is now deterministic and does not randomly fail, thus omitting the need to send out multiple session requests in parallel. Furthermore, no long-headers are needed so that the header size is constant for all sessions.

Latency

To compare the latency of processing PHI and dPHI messages during session establishment and transmission, the dPHI and PHI packet processing has been implemented in C and made available on Github [17]. To make use of Intel's AES-NI instruction set, their *Intelligent Storage Acceleration Library Crypto Version* (ISA-L_crypto) has been utilized to implement AES-GCM-256 and AES-CBC-256 for required AES operations. Any ECDH operations have been realized with the curve25519-donna library [2]. For measuring the required clock cycles, the intrinsic function *rdtsc()* was used to conduct one million independent measurements. After sorting these measurements, the top and bottom 37.5% were discarded to calculate the average of the remaining quarter. This is common practice when using *rdtsc()* and was also done by Chen *et al.* [35, 36]. Though, please note that in the presence of compiler and processor optimizations *rdtsc()* measurements on current processors for low numbers of clock cycles are quite inaccurate despite being the best available option. Measurements have been performed on an Intel Core i7-7500U with 2.7GHz using a single thread. Table 7.1 shows the processing latency results for PHI and dPHI. Some dPHI operations require fresh random numbers that can either be precomputed or generated on the fly. Both have been implemented so that two values are supplied in the Table 7.1 for these operations.

For session establishment, the computation time of nodes besides the helper node is negligible, especially when considering that in both PHI and dPHI the helper node, source, and destination have to perform public key operations. As can be seen in Table 7.1, the public key operation of the helper node is up to two orders slower than all other cryptographic operations. In dPHI the backtracking phase is extended to s so that more nodes need to be traversed. While the additional cryptographic operations will not significantly impact the setup latency, propagation latency might be higher. However, a PHI session establishment only succeeds with 90% probability. When a session establishment fails, the process needs to be repeated, including the computationally expensive public key operations, thus nearly doubling the setup latency. In consequence, while the minimum setup latency in dPHI is slightly worse than PHI due to the additional hops, the average and especially worst case setup latency are better.

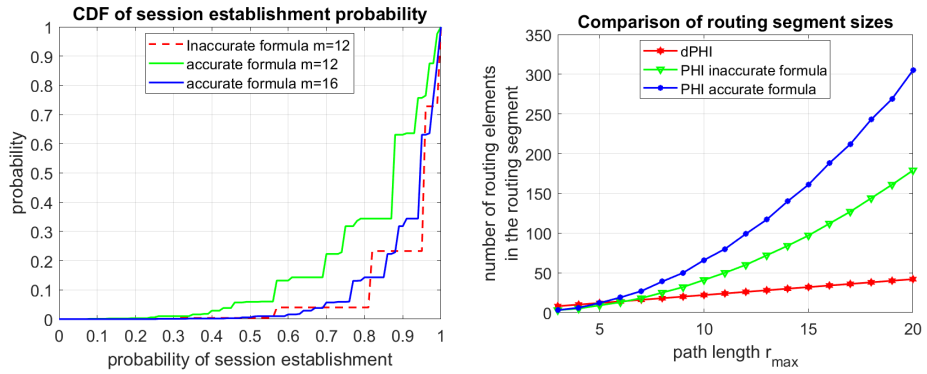
In HORNET, pubic key operations have to be performed by all nodes on the path so that the session establishment latency is considerably higher than in dPHI or PHI. Furthermore, the public key operations result in a considerable burden for the ASes if many sessions are created in parallel. An average path in the 2014 CAIDA dataset is about 4.2 hops, hence in average 4.2 sequential public key operations need to be performed per session in HORNET. In PHI, four parallel public key operations, resulting from four independent requests, suffice with a 90% probability, in 10% of the cases, eight or more are needed. In dPHI on the other hand, only one public key operation needs to be performed within the network. Therefore, dPHI reduces the overall computation load induced by the protocol considerably.

	dPHI	PHI
Session establishment		
Midway Request ($A \neq M$)	430*/1151	110
Midway Request ($A = M$)	146000	144915
Backtracking ($A \neq W$)	117	105
Backtracking ($A = W$)	1600*/3169	222
Handshake to d ($A \neq W$)	430*/1151	110
Handshake to d ($A = W$)	1255*/3619	n/a
Handshake reply to s ($A \neq W$)	117	105
Handshake reply to s ($A = W$)	951*/2124	105
Transmission phase		
Transmission phase ($A \neq W$)	117	105
Transmission phase ($A = W$)	250	105

Table 7.1.: Measured clock cycles of header processing for different nodes A in the protocol during the different phases of session establishment and the transmission phase. Entries marked with an asterisk are clock cycle measurements for an implementation based on precomputed random numbers (a node can precompute random numbers when the processor is idle).

Header Size and PHI Collision Probability

The header size is greatly influenced by the routing segment size. The default parameters proposed in PHI [36] for the number of routing elements in V is $m = 12$ for path lengths smaller than 8 and $m = 48$ for larger paths. Furthermore, $N = 4$ session establishment requests are sent out in parallel for small headers and $N = 5$ for large ones. The parameters were chosen so that a session establishment succeeds with 90% probability for the 2014 CAIDA dataset. However, the used formula to compute the success probability [36] is not accurate. It only computes the probability of a collision on the path between the source and destination but ignores the collision that can occur between the midway node and the helper node. In Figure 7.7(a), the session establishment probability for 1000 random PHI paths is computed with both the inaccurate formula and an updated formula that includes the collision probability between midway and helper node. The success probability with the accurate formula is considerably smaller. In fact, $m = 16$ instead of $m = 12$ is needed to achieve the 90% probability. For a maximum path length r_{max}, the number of routing elements in a dPHI header is $(r_{max} + 1) \cdot 2$. The size of the routing segment of PHI and dPHI for different values of r_{max} are compared in Figure 7.7(b) under the assumption that for PHI a routing segment size m is chosen such that a session establishment success rate of 90% is achieved. For this computation, an estimation of the expected path length from W to M is needed. In the given experiment with 1000 random PHI paths, the ratio of the path length r between s and d and the path between W to M was roughly 50%. This value is used in Figure 7.7(b), as well. As one can see, the routing segment size in PHI grows exponentially while it only grows linearly in dPHI.



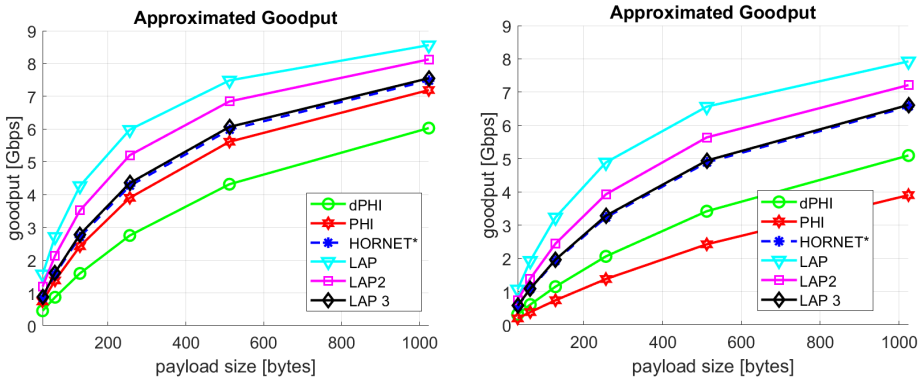
(a) CDF of the probability that a PHI session establishment is successful

(b) Comparison of routing segment sizes of dPHI and PHI

Figure 7.7.: a) CDF of the probability that a PHI session establishment is successful with $N = 4$ parallel session requests with the inaccurate formula from [36] and the accurate formula for different segment sizes m. b) Comparison of the routing segment size between PHI and dPHI depending on the maximum path length r_{max}. For PHI the routing segment size was computed such that the probability of a successful session establishment is at least 90% when sending out $N = 4$ request. For dPHI the routing segment size is $m = 2 \cdot l = 2 \cdot (r_{max} + 1)$.

It is also possible in dPHI to use different sizes for V^1 and V^2 than the default value of $l = 12$ for each vector. This is due to the fact that the source chooses the helper node and can simply pick one with a distance equal or smaller to 7 without reducing too much anonymity. Most nodes will be within this distance and a larger size is reserved for V^2 to reach far away destinations. Furthermore, since the path is split into two routing segments it is possible to create routes longer than r by choosing a helper node such that the midway node is roughly half-way between s and d.

LAP uses a variably sized header that does not depend on the maximum path length but the current path length. To make comparisons between PHI, dPHI, LAP and HORNET, different values for maximum path length r_{max} were used and average path length r_{avg} for LAP. For anylsis, $r_{max} = 7$ was chosen with $r_{avg} = 5$ and $r_{max} = 11$ with $r_{avg} = 8$, assuming a value of $m = 16$ and $m = 66$ for PHI segment sizes for normal and large headers. The PHI values were chosen based on the more accurate collision formula. In dPHI the header size is made up of a fixed part of 50 bytes for sid, midway field, pointers and flags, as well as the routing segments V^1 and V^2 whose size depends on r_{max}. One routing element consists of 39 bytes so that the header size for dPHI is 674 bytes and 986 bytes for $r_{max} = 7$ and $r_{max} = 11$ respectively.



(a) r_{max}=7, $r_{avg} = 5$ (b) r_{max}=11, $r_{avg} = 8$

Figure 7.8.: Approximated goodput of the different protocols based on the ratio of header size and payload (assuming processing is not the bottleneck). LAP uses variable sized headers and for computation, the average number of hops has been used.

Goodput and Throughput

In PHI [36], goodput measurements were performed using an SDN testbed. It was observed that PHI-related packet processing had no considerable impact on transmission rates of the used 10 Gbps link. Only in HORNET, processing speed impacted the goodput since HORNET also performs payload encryption. The processing speed of packets during transmission is roughly the same for dPHI and PHI with a measured clock cycle count of 117 vs 105 (See Table 7.1). This means, from a computational perspective, that the number of dPHI headers to be processed, even with the small size of 674 bytes, could be one magnitude higher than the number that is actually needed to saturate the 10 Gbps link. Hence, only the header size impacts the goodput in dPHI, just as has been the case in PHI. Therefore, the goodput for different payload sizes was approximated by computing the ratio of header size divided by packet size. The result of this analysis can be seen in Figure 7.8. In the computation it is assumed that the size of a PHI routing element is 24 bytes with 8 bytes resulting from encrypting the routing information with AES in counter mode and 16 bytes from the 128-Bit message authentication code that is also used in HORNET and dPHI. Note that dPHI proposes to only use $r = 11$ or larger but $r = 7$ was included for comparability.

Analysis results show that the average goodput of LAP is the best, even with $VSS = 3$. This contradicts statements made by Chen And Perrig [36], where the worst case header size was assumed, it seems. HORNET's goodput is also higher than that of both PHI and dPHI due to its smaller header size. Note that Figure 7.8 does not consider processing load as a potential bottleneck. But measurement data from PHI [36], where processing overhead is included, yield similar results for large payloads, that is slightly above 7 Gbps for a 1024 byte payload. Comparing PHI with dPHI, it can be observed that, for a small

$r = 7$, PHI has a higher goodput than dPHI. However, dPHI outperforms PHI for large paths of $r = 11$. Furthermore, this difference will increase rapidly for larger paths due to the exponential vs linear growths in header size.

7.5.4. Limitations of dPHI

While dPHI is able to improve on security and anonymity when compared to PHI, there are still limitations that should be pointed out. One of them is that the threat model assumes the attacker to control only a single AS. If the attacker controls two ASes, she can learn considerably more by combining the individually learned information. If the attacker-controlled nodes are close to each other, this is not as problematic as if the attacker controls two nodes that are further away. This is due to the fact that attacker nodes close to the source have a low source anonymity but large destination anonymity while attacker nodes close to the destination have no destination anonymity but large source anonymity. Hence, the anonymity can be greatly reduced if the attacker manages to gain control over two nodes on the path that are further apart. Active attacks on the routing layer could make this issue even worse in practice. For example, in BGP, route poisoning is a known problem that was used in the past to direct traffic over specific ASes for eavesdropping [13, 126]. Similarly, an attacker may drop session requests until a favorable dPHI path is established [26].

The biggest limitation and open problem of the dPHI protocol is the selection of the midway node. To be more precise, how to make sure that the entry node does not also become the midway node. If the entry node becomes the midway node there is no source or destination anonymity since the entry node (necessarily) knows the source and the midway node knows the destination. In a network scenario in which the source knows the employed routing policy of all nodes, it can verify that this does not happen. However, in adaptive routing policies such as BGP, one cannot accurately predict the route that will be chosen. Indeed, the fact that dPHI does not require client-based routing is its main benefit when compared to HORNET. Efficiently solving this problem without requiring client-based or client-controlled routing is important future work. Note that if the entry node performs active attacks to shape the traffic as described above, the problem of preventing the entry node to become the midway node becomes even more severe.

Application-wise, dPHI, just like the other discussed protocols, requires an adequately sized network to unfold its potential. The scenario chosen to discuss its security improvements and analyze it with regard to anonymity was the internet, which is probably one of the most plausible use cases. However, application as a *Moving Target Defense* in networks smaller than the internet is entirely feasible. What must be considered, though, is the fact that dynamic routing requires the existence of alternative paths in the first place. This applies to all discussed protocols. The architecture of large networks typically considers alternative communication paths, even if only serving as a fall back. SDN-based networks, if built to implement topological changes, may also provide at least the wiring to allow for alternative paths. Small networks with a star topology, however, simply cannot provide alternative paths for their physical layout. Anonymity, in turn, is not solely dependent on the existence of alternative paths, but may benefit

from it. Yet, a sufficient number of clients and intermediate hops are needed to ensure a minimal anonymity set size. It should be noted, though, that in the corporate network context, the adversary is not necessarily interested in unambiguously identifying entities, but only learning addresses to progress with attacking. Hence, large anonymity set sizes may not be as important as concealing addresses for which the number of hops is more critical.

7.6. Summary

This chapter elaborated on anonymous and dynamic routing, a topic that is of general relevance for communication, especially with regard to the internet. However, considering that adversaries may collect sufficient information to identify potentially valuable targets and prepare lateral movement simply through eavesdropping on passing traffic, anonymous and dynamic routing is equally relevant in the context of corporate networks. Consequently, for its movement property, it has also been identified as a *Moving Target Defense* for particularly impeding reconnaissance of such information. Still, the body of literature covering anonymous and dynamic routing as a means of *Moving Target Defense* is comparatively small as opposed to the defenses that have been investigated in the preceding chapters. What is more, proposed schemes predominantly employ SDN-based approaches that cannot provide anonymity, while constantly flooding the network with communication on the control plane to remove and insert required flows on all involved hops for any combination of communicating peers.

To this end, an anonymous and dynamic routing protocol has been introduced that is applicable in the internet, as well as corporate networks, broadening the scope of this work beyond defense evaluation, and contributing to the state of *Moving Target Defense* in a yet underrepresented domain. In this course, different existing protocols have been considered, focusing on those that operate on the network layer and support policy-based routing, for their better applicability in a corporate network context as, say, overlay networks that employ source routing. The PHI protocol has been investigated in particular, presenting novel attacks that significantly reduce its achieved anonymity. Based on this, a new protocol named dependable PHI (dPHI) was presented that withstands these attacks, holding true even if the threat model is extended to consider attackers who control clients in various ASes. dPHI also solves PHI's problem of collisions in the routing segment, thus requiring fewer session establishment requests and smaller header sizes.

A quantitative anonymity and performance analysis shows that dPHI offers a good trade-off between performance and anonymity compared to protocols such as LAP and HORNET. The sender-receiver anonymity set size of dPHI is considerably larger than that of LAP, which is mainly due to the fact that dPHI offers receiver anonymity for nodes on path s to W, while LAP does not offer any receiver anonymity. HORNET, however, provides the best anonymity of the discussed protocols. In terms of setup latency and goodput, dPHI achieves similar or better results than PHI but lower goodput than LAP or HORNET. HORNET requires expensive public key operations on all routing nodes during session establishment. Furthermore, HORNET uses client-based

routing that is neither in line with today's internet nor corporate networks. dPHI does not impose any requirements on the employed type of routing but leaves such decisions to the underlying architecture. The only major issue in dPHI is that the client should choose a helper node so that the entry node does not become the midway node.

8. Discussion

This last chapter summarizes results and discusses obtained insights and their meaning to *MTD* research. Furthermore, related work will be presented that is closer to the proposed framework and investigated defenses than the evaluation approaches discussed in the end of Chapter 3. Afterwards, a short outlook into future work will be given, before concluding this thesis.

8.1. Summary of Results

Two case studies as well as an anonymous and dynamic routing protocol have been presented in the course of this work. The first case study considered a corporate network as could plausibly be in use by a small company. Two versions of this network, differing in only minute aspects, were used to test five defense configurations that are based on frequently proposed *Moving Target Defenses*. Investigated defense configurations comprised two flavors of *VM migration*, *IP shuffling*, *VM resetting*, as well as a reference configuration where *no defense* was employed. These were tested for their effect on security in the presence of two differently skilled attackers. This effect, in turn, was measured in form of attacker revenue, a defense-independent performance indicator that was calculated from compromised resources. Results from this first experiment revealed that a frequently proposed technique, *VM migration*, may in fact have a negative impact on security for opening up attacks paths that would otherwise not exist. Considering that existing evaluation approaches, without exception, deemed *VM migration* to be improving security, this is a novel finding. However, the second version of the employed network, only differing in the initial location of some VMs, painted a different picture. In this changed scenario, VM migration had a positive effect on security. This was due to critical VMs now starting in a location that was beneficial for the attacker, thus moving them improved security through impeding attacks. In this regard, the first case study not only illustrated that defenses may pose a security risk. It also showed that evaluation should not be based on single experiments, even though this is common practice.

To further investigate this, a second case study was performed. This one, however, was not limited to two almost identical versions of the same network but relied on 500 different networks. These were of larger size and higher complexity to make simulation even more realistic. However, this required network generation and diversification to be automated since manual modeling would not have been feasible anymore. To accommodate this, benchmark fuzzing was introduced to automate this process on the basis of one single scenario definition that allows for the specification of a networks foundation while leaving certain aspects to probability. Based on these 500 networks simulation

was conducted employing the same five defense configurations as before to validate previous findings and provide a higher resolution of observed effects and their distribution. Yet, the increased volume of simulation results poses a challenge with regard to comprehensive analysis itself. Therefore, new metrics were developed to quantify the security implications of the defenses under test. The first metric indicates how far an attacker can infiltrate the network for a given defense by measuring gained revenue, while the second metric exposes throttling effects on attacks. Analysis confirmed that certain defenses may sometimes have a negative impact on security, sometimes a positive, and sometimes none at all. However, through benchmark fuzzing, it was possible to quantify how frequently these different effects on security occur. In particular, results show that *VM migration* more often than not had a negative impact on security rather than a positive, thus further corroborating the contradiction to existing work that says otherwise. Unlike *VM migration*, *VM resetting* and *IP shuffling* had a considerable positive impact on security, with *VM resetting* exhibiting the highest performance in the conducted experiments. It should be noted that analysis did not only reproduce previous findings from the manually crafted benchmark networks but also revealed cases previously not encountered, such as the positive impact of *IP shuffling*. In this regard, apart from increasing test coverage, fuzzing also helps detecting corner cases and may serve as a kind of sanity check for more formal evaluation methods. Besides quantitative analysis, the framework also allows for a qualitative assessment which provides defenders with a more balanced view when deciding which defenses to employ.

However, apart from quantitatively evaluating different proactive defense techniques, this work also presented an anonymous and dynamic routing protocol that may serve as a *Moving Target Defense* technique. Preceding investigation on lightweight network-layer anonymity protocols revealed limitations and design flaws in recently published work. While improving on its predecessors, the analyzed protocol — PHI — leaked information that, under certain circumstances, allowed for a considerable reduction of the senders' and receivers' anonymity set sizes. Even more so, when extending the threat scenario to be more realistic and in line with the internet as a scenario. Based on these findings, an improved protocol named dPHI has been proposed. While exhibiting similar computational load as PHI, the new protocol withstands identified attacks, even in the extended threat scenario, thus improving on anonymity. Relying on Intel's AES-NI, experimental and analytical analysis has shown that computational effort involved in the protocol-related processing is no delimiting factor. In fact, the number of dPHI headers to be processed may even be one magnitude higher than what is needed for saturating a 10 Gbps link, before computational load would become an issue.

8.2. Related Work

In Chapter 3.3.1, a broader range of related work has already been discussed, ranging from analytical approaches to evaluation, through simulation, to testbeds. At this point now, the proposed framework has been used to investigate a selection of defenses and provide insights on their performance in different settings. Considering this, the focus

of related work is narrowed down to approaches that employ a similar methodology or investigated on the same defenses, thus allowing for comparison. Since the framework presented here is the first to reveal considerable security degradation incurred through *VM migration*, approaches that address this very technique are of special interest.

In 2016, Hong and Kim introduced their hierarchical attack representation model (HARM) [64] to address the challenges of evaluating *MTD* techniques at the example of *VM live migration*. This multi-layered approach is based on graphical security models such attack trees and graphs, yet overcomes their limitations with regard to incorporating adaptations that *MTD* ultimately requires. While it is not made clear if defenses are modeled by their intended effect or underlying mechanisms, results indicate that the scheme is capable to detect security improvement and deterioration. However, the scenario used for evaluating effectiveness consists of only three physical hosts and five VMs where migrating specific machines indeed improves security due to a sub-optimal initial state and, most importantly, knowledge about the attacker's current progress. While this can be considered a realistic setting, there are numerous equally realistic settings where the attacker's progress is not known and *VM migration* may cause transition to an insecure state. Additionally, it is assumed that no negative effect may result from co-location of VMs. Considering the track record of vulnerabilities in common hypervisors, and the fact that most of them default to attaching guest OSes to the same virtual switch (e.g. Xen), this is a strong assumption. In consequence, Hong and Kim conclude that *VM migration* is effective to improve security. Further defense evaluation conducted with help of the HARM [8, 10] revealed that migration may increase the number of attack paths, yet not leading to any security degradation as *VM migration* would still fend these off. Instead, results indicate that the overall risk and the "return on attack" was ultimately lowered. Enoch *et al.* [55] present a further developed version of the HARM that incorporates temporal aspects.

The evaluation approach of Debroy *et al.* [50] is also concerned with *VM live migration*, yet not to assess its actual effects but to optimize its utilization in a defense strategy to fend off distributed denial of service (DDoS) attacks. The authors determine performance indicators such as cost of migration related to resource consumption and service degradation for legitimate users that ought to be minimized. At the same time migration should be timed so that DDoS attacks are prevented while choosing optimal migration locations that take resource utilization efficiency into account. Analysis to derive these timings is based on the common assumption that DDoS attacks can be modeled as a Poisson process. Should proactive migration not prevent a DDoS attack, reactive migration is suggested based on alerts from an intrusion detection system (IDS). Experiments are performed on an SDN-enabled GENI Cloud and results indicate that an optimized defense strategy can reduce the success rate of DDoS attacks by up to 40% while reducing cost as compared to periodic migration. Since the threat model is limited to DDoS attacks, downsides that may result from co-location of VMs are not accounted for. However, Debroy *et al.* consider service interruption and additional load to be impediments of *VM migration* and include these in their optimization problem.

An earlier approach to evaluating *VM migration* is presented by Wang *et al.* [129]. The suggested framework primarily considers cost of performing defense actions as opposed

to cost of being attacked, as well as the effort required by an attacker to successfully compromise a system. The cost incurred to the defender for migrating a VM are derived from the mean downtime of services and related monetary net loss. Cost of being attacked include damages from loss of confidentiality, integrity and availability but also reputation. Together with historic information on attacks and knowledge of the mean effort required to conduct them, distribution fitting is done to enable prediction of future attacks. Based on this, the framework optimizes for reducing cost – including those of being attacked – in the long run while keeping migration intervals shorter than attack intervals. Interestingly enough, the authors consider covert channel attacks that may result from co-location of VMs and use this to motivate migration in the first place. However, the fact that migration may enable the attacker's VM to attack even more other machines is not considered.

Repeatedly randomizing IP addresses, for example, has been evaluated based on the low probability for an attacker to maintain knowledge on correct IP addresses of multiple targets that need to be attacked simultaneously [89]. While this is not far-fetched, in practice, multi-target attacks need not necessarily be carried out at once, but may span a time frame during which outdated knowledge may plausibly be refreshed. Furthermore, in a realistic scenario, successful attack and defense depend on more than only knowing IP addresses. Attackers and defenders repeatedly select from a multitude of different actions, all of which depend on various factors that would need to be considered so that quantifying the actual effect of *IP shuffling* is far more complicated. However, when including all potentially relevant aspects, mathematical formalization reaches a degree of complexity that is far beyond calculating the probability of guessing correct addresses.

Abdul Basit Ur Rahim *et al.* [4] present an approach to *MTD* evaluation that is based on model checking and utilize it to investigate *IP shuffling* in the presence of constraints to maintain system operation. Interestingly, this methodology is similar to the one proposed here for also relying on a state representation that is repeatedly checked to decide if requirements for subsequent actions are fulfilled or not. While not making a final judgment about the effectiveness of *IP shuffling*, the authors claim that their scheme is capable of formally representing this technique. However, their representation of *IP shuffling* is limited to the effects it has on an adversary that repeatedly scans for addresses, while ignoring all other steps an attacker may take to acquire such knowledge or subsequently use it. In this regard, their general approach is interesting, yet does not incorporate the needed level of detail to yield realistic results.

Bangalore and Sood [22] did not only propose sole *VM resetting* but also evaluated their scheme. With help of a VMware ESX hypervisor and a virtualized guest representing a web server, they conduct experiments to determine possible frequency and effect of resetting the guest. The authors conclude that resetting intervals of less than a minute make attacking impractical while barely impeding regular operation when utilizing spare VMs that seamlessly take over incoming requests. While this appears plausible, the findings are limited to cases in which light-weight read-only services are concerned. Machines that need to persist data and can therefore not be resetted, are not considered. Furthermore, depending on the type of service that is provided, intervals shorter than a minute may not be feasible.

Evaluation approaches that address *MTD*-related downsides in form of resource consumption and degraded service availability have been presented by Connel *et al.* [45], Chen *et al.* [38], and Mendonça *et al.* [94]. These, however, do not inspect the security impact of investigated defenses but focus on their effects on operators and legitimate users with regard to cost, service availability and job finish time. The simulation-based approaches from Zhuang *et al.* [147, 150] also consider the potentially negative impact of defenses on legitimate operation. Yet, this is not represented as cost or downtime to be minimized, but as functional requirements defense strategies must adhere to.

Table 8.1.: Comparative overview of related work with regard to different aspects. Employed abbreviations: simulation-based (s), analytical (a), testbed (t), high (h), medium (m), low (l), defense effect (DE), defense strategy (DS)

	Approach	Model Granularity	Scenario Size	No. of Scenarios	Defenses	Investigated Aspect	Security Degradation	Defense Comparison
this work	s	h	90	500	live migration cold migration VM resetting IP shuffling	DE	considered, detected	yes
Hong & Kim [64]	s	m	8 (larger numbers only for scalability analysis)	1	live migration OS shuffling	DE	not considered	no
Alavizadeh *et al.* [8, 10]	s	m	16 (900 nodes in model, yet, only "important" ones in evaluation)	2	live migration OS shuffling	DE	considered, not detected	yes
Debroy *et al.* [50]	t	h	7 (+ 30 nodes as dummies and migration targets)	20	live migration (5 strategies)	DS	only service degradation	no
Wang *et al.* [129]	s	l	1	144	live migration cold migration (7 strategies)	DS	only service degradation	yes
Maleki *et al.* [89]	a	l	n (formalism not limited regarding number of nodes)	2	IP shuffling	DS	not considered	no
Ur Rahim *et al.* [4]	s	m	n/a	n/a	IP shuffling	DE	not considered	no
Bangalore & Sood [22]	t	h	4	1	VM resetting	DS	only service degradation	no
Zhuang *et al.* [150]	s	l	7	1	cold migration (5 strategies)	DS	not considered	no

Table 8.1 provides a comparative overview of related work with regard to different aspects. Note that this table only comprises those approaches where information on compared aspects could be obtained from related publications, at least to a certain degree. In this table, the column "approach" refers to the employed evaluation approach that can either be simulation-based (s), analytical (a), or relying on testbeds (t). "Model granularity" refers to the level of detail of employed models that evaluation is based on. This can either be high (h), medium (m), or low (l). Obviously, this qualitative classification cannot fully represent the exact granularity employed in the different approaches. Yet, an elaborate description of utilized models is regularly not included in respective publications, and, what is more, opposes the principle of a compact comparative overview. Still, levels of granularity have not been chosen randomly but were determined in the following way: Granularity of testbeds is generally assumed to be high for considering real systems, whereas granularity of analytical approaches is assumed to be low, for the extent of simplification and assumptions that is needed to represent complex systems through mathematical formalization. Simulation-based approaches have been assigned granularity levels based on the model descriptions in the related publications. The work of Wang *et al.* [129], for example, employs a model that only considers whether or not a node is being attacked, neither incorporating any other characteristics of these nodes, nor potential applications, data, or the surrounding network. Consequently, the granularity level of this approach has been classified as being low, as has been done for all other approaches that only considered the attack state of nodes. In turn, the approach from Alavizadeh *et al.* [8, 10] considers nodes together with their operating systems, as well as potential vulnerabilities that allow for attacks in the first place, and has therefore been classified as exhibiting a medium granularity. This level of granularity was assigned whenever a model did not only consider nodes, but also included OSes or applications and respective vulnerabilities. Any model more complex than that, was considered to be of high granularity. This applies to the model utilized in this work for its relative complexity as demonstrated in Chapters 4, 5, and 6.

"Scenario size" characterizes the different approaches with regard to the number of nodes that evaluated scenarios comprise. From a technical perspective the number of nodes is not perfectly fit to represent a scenario's real size, as it does not reflect the potentially large number of other entities that may be part of the model. However, in the presence of deviating levels of modeling granularity, the number of nodes is an intuitive and most importantly common indication for scenario sizes. It should be noted though, that numbers on considered nodes vary for some approaches. In this regard, the approach from Alavizadeh *et al.* [8, 10] specifies a scenario size of up to 900 nodes, yet, only considers a fraction of these during evaluation on the basis of importance measures. While this may be part of the employed evaluation scheme, it does not qualify as the factual evaluation of a defense on the basis of 900 nodes in the context of this comparison.

The "number of scenarios" represents the number of different settings in which defenses have been evaluated. The degree to which scenarios actually differ varies across the different approaches. The evaluation conducted by Debroy *et al.* [50], for example, derives different scenarios from randomly shuffling characteristics of the operating systems of respective nodes. The scenarios employed by Wang *et al.* [129], on the other

hand, are based on traces of attacks from a cloud computing environment that are not further elaborated and may differ in yet other aspects than only the OS, however, always consider only one specific node.

"Defenses" lists the different *Moving Target Defenses* that have been evaluated by the respective approach. Occasionally, remarks are made with regard to the number of different strategies a defensive technique has been evaluated with. This is related to the "investigated aspect" that may either be the defense effect (DE) or defense strategy (DS). While approaches that evaluate defense effects strive to reveal how a technique impacts the system it operates in, defense strategy evaluation attempts to answers the question on how to effectively employ these techniques. The last but one column – "security degradation" – address whether or not the respective evaluation approach considered, and may be even detected, defense-related security degradation. As can be seen, negative effects are frequently not considered, and if so, are limited to concerns regarding service degradation or disruption. Finally, the last column indicates whether or not the respective approach compared defenses with regard to their impact on security in considered scenarios. Obviously, comparison of different techniques is unlikely if conducted evaluation only considered one defense in the first place.

8.3. Future Work

There are several conceivable directions for future work, three of which appear to be very promising and are outlined in the following:

- **Smart attackers:** The experiments employed a greedy and powerful attacker who pursues all available attack avenues. For the considered defenses this was appropriate as attacker actions only resulted in beneficial effects from an attacker's perspective. However, to include any kind of intrusion detection systems in the analysis, intelligent attackers are needed who try to avoid being detected. Modeling an intelligent attacker is also helpful to analyze defenses that are based on deception. The introduced benchmark fuzzing to automatically generate large numbers of related yet different benchmark networks is already an important step towards training such an attacker. Manual benchmark modeling would be too cumbersome to generate enough training data for most artificial intelligence (AI) algorithms to be effective. Attack simulations with such intelligent attackers could not only be used to further evaluate defenses but also improve defense deployment by optimizing honeypot placement, for example. Hence, combining the attack simulation approach with an AI-enabled attacker is a promising and interesting direction for future work.
- **Common standardized benchmarks:** The conducted case studies have shown that characteristics of the networks used for defense evaluation had a considerable impact on observed performance and effects. Consequently, testing at a larger scale is advisable. However, what may further improve comparability of findings generated by independently conducted research and development, is the definition of common benchmark networks. Agreeing on a common standard to test de-

fenses does not only ensure that proposed techniques have been evaluated under the same conditions but may also enforce a minimum level of detail and realism. Assuming that such common benchmark networks are adequately chosen, they may strengthen trust in the results of evaluation and help to separate the wheat from the chaff.

- **Extending analysis to other network types:** Getting an understanding of how different defenses perform is not only relevant in the context of medium-sized corporate networks. Data centers, critical infrastructures, as well as IoT environments must be secured, as well. And while most defenses, no matter if proactive or static, are generally applicable in these domains, they are yet subject to other constraints and changed environmental factors. That these need to be considered, has been sufficiently motivated by now. Consequently, developing realistic scenario definitions for other targets is an important next step to enable evaluation of defense performance under such altered conditions.

8.4. Conclusion

In the presence of an ever-growing number of attacks on networks and infrastructure, the development of new techniques to defend these is of utmost importance. Yet, their ability to contribute to security should not be assumed but verified. In this thesis, an attack simulation framework has been presented that is able to quantitatively and qualitatively evaluate and compare different network defense techniques at a larger scale. By revealing that defenses intended to increase security may also have negative effects, the two conducted experiments have shown that the chosen approach to evaluation and the employed level of detail have a significant impact on findings. Evaluation approaches that favored simplification over accuracy were not able to yield such insights. In this regard, simulation on the basis of high-detail network descriptions is a viable and useful approach. However, this work has also shown that generalizing on the basis of single findings is not advisable, since results did not only depend on the level-of-detail but also on what has been modeled. Equally plausible settings may yield different results so that testing in different settings is not optional but necessary to fairly compare and improve understanding of defense performance. The introduction of benchmark fuzzing makes an important step towards evaluation at a larger scale. Through considerably reducing the effort required to generate large sample sizes, extensive testing does not pose an additional burden and becomes economically attractive.

Apart from evaluation, this work also attended to anonymous and dynamic routing as a form of *Moving Target Defense*. While a large part of existing research focuses on refining and improving popular defense techniques such as *IP shuffling* and *VM migration*, other promising techniques are underrepresented. The network-layer anonymity protocol presented in this work is discussed and analyzed with regard to utilization in the internet, yet may equally be applied in closed corporate networks. Through concealing the addresses and identities of both receivers and senders of messages, this protocol does not intend to invalidate learned addresses but prevent them from being learned in the first place. In combination with schemes such as NASR, for example, this may significantly impede reconnaissance. In addition, given the required network architecture, the protocol also allows to vary communication paths, thus further increasing difficulty for the attacker to acquire information.

Bibliography

[1] US DHS Homepage. https://www.dhs.gov/science-and-technology/csd-mtd. [Online; accessed 6-March-2020].

[2] A. Langley. curve25519-donna. https://github.com/agl/curve25519-donna, Last accessed on 2020-10-23.

[3] O. Abdel Wahab, J. Bentahar, H. Otrok, and A. Mourad. Resource-aware detection and defense system against multi-type attacks in the cloud: Repeated bayesian stackelberg game. *IEEE Transactions on Dependable and Secure Computing*, pages 1–1, 2019.

[4] M. Abdul Basit Ur Rahim, Q. Duan, and E. Al-Shaer. A formal analysis of moving target defense. In *2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 1802–1807, 2020.

[5] N. O. Ahmed and B. Bhargava. Mayflies: A moving target defense framework for distributed systems. In *Proceedings of the 2016 ACM Workshop on Moving Target Defense*, MTD '16, page 59–64. ACM, 2016.

[6] N. O. Ahmed and B. Bhargava. From byzantine fault-tolerance to fault-avoidance: An architectural transformation to attack and failure resiliency. *IEEE Transactions on Cloud Computing*, pages 1–1, 2018.

[7] N. O. Ahmed and B. Bhargava. Bio-inspired formal model for space/time virtual machine randomization and diversification. *IEEE Transactions on Cloud Computing*, pages 1–1, 2020.

[8] H. Alavizadeh, J. Jang-Jaccard, and D. S. Kim. Evaluation for combination of shuffle and diversity on moving target defense strategy for cloud computing. In *2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications (TrustCom)*, pages 573–578, 2018.

[9] H. Alavizadeh, D. S. Kim, J. B. Hong, and J. Jang-Jaccard. Effective security analysis for combinations of MTD techniques on cloud computing (short paper). In J. K. Liu and P. Samarati, editors, *Information Security Practice and Experience*, pages 539–548. Springer, 2017.

[10] H. Alavizadeh, D. S. Kim, and J. Jang-Jaccard. Model-based evaluation of combinations of shuffle and diversity mtd techniques on the cloud. *Future Generation Computer Systems*, 2019.

[11] H. M. J. Almohri, L. T. Watson, and D. Evans. Misery digraphs: Delaying intrusion attacks in obscure clouds. *IEEE Transactions on Information Forensics and Security*, 13(6):1361–1375, June 2018.

[12] M. AlSabah and I. Goldberg. Performance and security improvements for Tor: A survey. *ACM Computing Surveys (CSUR)*, 49(2):32, 2016.

[13] N. Anderson. How china swallowed 15% of net traffic for 18 minutes. Arstechnica, 11 2010.

[14] N. Anderson, R. Mitchell, and I. R. Chen. Parameterizing moving target defenses. In *2016 8th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, pages 1–6, Nov 2016.

[15] G. Asharov, D. Demmler, M. Schapira, T. Schneider, G. Segev, S. Shenker, and M. Zohner. Privacy-preserving interdomain routing at internet scale. *Proceedings on Privacy Enhancing Technologies*, 2017(3):147–167, 2017.

[16] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, P. Naur, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, et al. Report on the algorithmic language algol 60. *Numerische Mathematik*, 2(1):106–136, 1960.

[17] A. Bajic and G. T. Becker. Github repository of used software and data. `https://github.com/AlexB030/dPHI`. [Online; uploaded 10-March-2020].

[18] A. Bajic and G. T. Becker. dPHI: An improved high-speed network-layer anonymity protocol. *Proceedings on Privacy Enhancing Technologies*, 2020(3):304 – 326, 01 Jul. 2020.

[19] A. Bajic and G. T. Becker. Attack simulation for a realistic evaluation and comparison of network security techniques. In *Nordic Conference on Secure IT Systems*, pages 236–254. Springer, 2018.

[20] A. Bajic and G. T. Becker. A critical view on moving target defense and its analogies. In *Proceedings of the 17th ACM International Conference on Computing Frontiers*, CF '20, page 277–283, New York, NY, USA, 2020. Association for Computing Machinery.

[21] A. Bajic and G. T. Becker. Automated benchmark network diversification for realistic attack simulation with application to moving target defense. *International Journal of Information Security*, May 2021.

[22] A. K. Bangalore and A. K. Sood. Securing web servers using self cleansing intrusion tolerance (SCIT). In *2009 Second International Conference on Dependability*, pages 60–65, June 2009.

[23] J. Banks, J. Carson, B. Nelson, and D. Nicol. *Discrete-Event System Simulation*. Prentice Hall, 5 edition, 2010.

[24] S. Bhatkar and R. Sekar. Data space randomization. In D. Zamboni, editor, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 1–22, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[25] P. Blackburn, J. Bos, and K. Striegnitz. *Learn Prolog Now!*, volume 7 of *Texts in Computing*. College Publications, 2006.

[26] N. Borisov, G. Danezis, P. Mittal, and P. Tabriz. Denial of service or denial of security? In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 92–102, 2007.

[27] S. W. Boyd and A. D. Keromytis. SQLrand: Preventing SQL injection attacks. In M. Jakobsson, M. Yung, and J. Zhou, editors, *Applied Cryptography and Network Security*, pages 292–302, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[28] I. Bratko. *Prolog Programming for Artificial Intelligence, 4th Edition*. Addison-Wesley, 2012.

[29] K. Butler, P. McDaniel, and W. Aiello. Optimizing BGP security by exploiting path stability. In *Proceedings of the 13th ACM conference on Computer and communications security*, pages 298–310. ACM, 2006.

[30] C. Cadar, P. Akritidis, M. Costa, J.-P. Martin, and M. Castro. Data randomization. Technical report, Technical Report TR-2008-120, Microsoft Research, 2008. Cited on, 2008.

[31] X. Cai, X. C. Zhang, B. Joshi, and R. Johnson. Touching from a distance: Website fingerprinting attacks and defenses. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 605–616. ACM, 2012.

[32] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, and R. Buyya. CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and Experience*, 41(1):23–50, 2011.

[33] V. Casola, A. De Benedictis, and M. Albanese. A moving target defense approach for protecting resource-constrained distributed devices. In *IEEE 14th International Conference on Information Reuse & Integration (IRI)*. IEEE, 2013.

[34] S. Chang, Y. Park, and B. B. Ashok Babu. Fast IP hopping randomization to secure hop-by-hop access in SDN. *IEEE Transactions on Network and Service Management*, 16(1):308–320, March 2019.

[35] C. Chen, D. E. Asoni, D. Barrera, G. Danezis, and A. Perrig. Hornet: High-speed onion routing at the network layer. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1441–1454. ACM, 2015.

[36] C. Chen and A. Perrig. Phi: Path-hidden lightweight anonymity protocol at network layer. *Proceedings on Privacy Enhancing Technologies*, 2017(1):100–117, 2017.

[37] H. Chen, J.-H. Cho, and S. Xu. Quantifying the security effectiveness of firewalls and DMZs. In *Proceedings of the 5th Annual Symposium and Bootcamp on Hot Topics in the Science of Security*, HoTSoS '18, New York, NY, USA, 2018. Association for Computing Machinery.

[38] Z. Chen, X. Chang, Z. Han, and Y. Yang. Numerical evaluation of job finish time under MTD environment. *IEEE Access*, 8:11437–11446, 2020.

[39] J. Cho, D. P. Sharma, H. Alavizadeh, S. Yoon, N. Ben-Asher, T. J. Moore, D. S. Kim, H. Lim, and F. F. Nelson. Toward proactive, adaptive defense: A survey on moving target defense. *IEEE Communications Surveys Tutorials*, 22(1):709–745, 2020.

[40] J.-H. Cho, M. Zhu, and M. Singh. Modeling and analysis of deception games based on hypergame theory. In *Autonomous Cyber Deception*, pages 49–74. Springer, 2019.

[41] A. Chowdhary, A. Alshamrani, D. Huang, and H. Liang. MTD analysis and evaluation framework in software defined network (MASON). In *Proceedings of the 2018 ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization*, SDN-NFV Sec '18, page 43–48. ACM, 2018.

[42] E. S. Collado, P. A. Castillo, and J. J. Merelo Guervós. Using evolutionary algorithms for server hardening via the moving target defense technique. In P. A. Castillo, J. L. Jiménez Laredo, and F. Fernández de Vega, editors, *Applications of Evolutionary Computation*, pages 670–685, Cham, 2020. Springer International Publishing.

[43] A. Colmerauer and P. Roussel. The birth of prolog. In *The Second ACM SIGPLAN Conference on History of Programming Languages*, HOPL-II, page 37–52, New York, NY, USA, 1993. Association for Computing Machinery.

[44] W. Connell, M. Albanese, and S. Venkatesan. A framework for moving target defense quantification. In *IFIP International Conference on ICT Systems Security and Privacy Protection*, pages 124–138. Springer, 2017.

[45] W. Connell, D. A. Menascé, and M. Albanese. Performance modeling of moving target defenses. In *Proceedings of the 2017 Workshop on Moving Target Defense*, MTD 17, page 53–63. ACM, 2017.

[46] W. Connell, D. A. Menasce, and M. Albanese. Performance modeling of moving target defenses with reconfiguration limits. *IEEE Transactions on Dependable and Secure Computing*, pages 1–1, 2018.

[47] Cryptonite LLC. CryptoniteNXT. `https://www.cryptonitenxt.com/`, Last accessed on 2020-10-04.

[48] C. Dai and T. Adegbija. Condense: A moving target defense approach for mitigating cache side-channel attacks. *IEEE Consumer Electronics Magazine*, 9(3):114–120, 2020.

[49] D. Das, S. Meiser, E. Mohammadi, and A. Kate. Anonymity trilemma: Strong anonymity, low bandwidth overhead, low latency - choose two. In *2018 IEEE Symposium on Security and Privacy (SP)*, volume 00, pages 108–126, May 2018.

[50] S. Debroy, P. Calyam, M. Nguyen, R. L. Neupane, B. Mukherjee, A. K. Eeralla, and K. Salah. Frequency-minimal utility-maximal moving target defense against DDoS in SDN-based systems. *IEEE Transactions on Network and Service Management*, 17(2):890–903, 2020.

[51] S. Debroy, P. Calyam, M. Nguyen, A. Stage, and V. Georgiev. Frequency-minimal moving target defense using software-defined networking. In *Computing, Networking and Communications (ICNC), 2016 International Conference on*, pages 1–6. IEEE, 2016.

[52] P. Dhungel, M. Steiner, I. Rimac, V. Hilt, and K. W. Ross. Waiting for anonymity: Understanding delays in the Tor overlay. In *2010 IEEE Tenth International Conference on Peer-to-Peer Computing (P2P)*, pages 1–4. IEEE, 2010.

[53] M. Dunlop, S. Groat, W. Urbanski, R. Marchany, and J. Tront. MT6D: A moving target IPv6 defense. In *2011 - MILCOM 2011 Military Communications Conference*, pages 1321–1326, Nov 2011.

[54] E. Rose. Parsimonious. `https://github.com/erikrose/parsimonious`, Last accessed on 2020-10-03.

[55] S. Y. Enoch, J. B. Hong, M. Ge, H. Alzaid, and D. S. Kim. Automated security investment analysis of dynamic networks. In *Proceedings of the Australasian Computer Science Week Multiconference*, ACSW '18. ACM, 2018.

[56] D. Evans, A. Nguyen-Tuong, and J. Knight. Effectiveness of moving target defenses. In *Moving Target Defense: Creating Asymmetric Uncertainty for Cyber Threats*, pages 29–48. Springer, 2011.

[57] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida. ASLR on the line: Practical cache attacks on the MMU. In *NDSS*, volume 17, page 26, 2017.

[58] C. Green. Application of theorem proving to problem solving. 1969.

[59] M. Green, D. C. MacFarland, D. R. Smestad, and C. A. Shue. Characterizing network-based moving target defenses. In *Proceedings of the Second ACM Workshop on Moving Target Defense*, MTD '15, pages 31–35. ACM, 2015.

[60] D. Gupta, A. Segal, A. Panda, G. Segev, M. Schapira, J. Feigenbaum, J. Rexford, and S. Shenker. A new approach to interdomain routing based on secure multi-party computation. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, pages 37–42. ACM, 2012.

[61] Y. Han, J. Chan, T. Alpcan, and C. Leckie. Using virtual machine allocation policies to defend against co-resident attacks in cloud computing. *IEEE Transactions on Dependable and Secure Computing*, 14(1):95–108, 2015.

[62] W. Henecka and M. Roughan. Strip: Privacy-preserving vector-based routing. In *2013 21st IEEE International Conference on Network Protocols (ICNP)*, pages 1–10. IEEE, 2013.

[63] H. Holm, K. Shahzad, M. Buschle, and M. Ekstedt. P^2 CySeMoL: Predictive, probabilistic cyber security modeling language. *IEEE Transactions on Dependable and Secure Computing*, 12(6):626–639, Nov 2015.

[64] J. B. Hong and D. S. Kim. Assessing the effectiveness of moving target defenses using security models. *IEEE Transactions on Dependable and Secure Computing*, 13(2):163–177, March 2016.

[65] J. B. Hong, S. Yoon, H. Lim, and D. S. Kim. Optimal network reconfiguration for software defined networks using shuffle-based online MTD. In *2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS)*, pages 234–243, 2017.

[66] W. House. Trustworthy cyberspace: Strategic plan for the federal cybersecurity research and development program. *Report of the National Science and Technology Council, Executive Office of the President*, 2011.

[67] H.-C. Hsiao, T. H.-J. Kim, A. Perrig, A. Yamada, S. C. Nelson, M. Gruteser, and W. Meng. Lap: Lightweight anonymity and privacy. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 506–520. IEEE, 2012.

[68] http://www.caida.org/data/as-relationships. The caida ucsd [as relationships] - [2014-09-01]. Technical report, caida, 2014.

[69] J. Wielemaker. SWI-Prolog. `https://www.swi-prolog.org`, Last accessed on 2020-10-03.

[70] T. Jackson, A. Homescu, S. Crane, P. Larsen, S. Brunthaler, and M. Franz. Diversifying the software stack using randomized NOP insertion. In S. Jajodia, A. K. Ghosh, V. Subrahmanian, V. Swarup, C. Wang, and X. S. Wang, editors, *Moving Target Defense II*, pages 151–173, New York, NY, 2013. Springer New York.

[71] T. Jackson, B. Salamat, A. Homescu, K. Manivannan, G. Wagner, A. Gal, S. Brunthaler, C. Wimmer, and M. Franz. *Compiler-Generated Software Diversity*, pages 77–98. Springer New York, New York, NY, 2011.

[72] X. Jiang, H. J. Wangz, D. Xu, and Y. M. Wang. Randsys: Thwarting code injection attacks with system service interface randomization. In *26th IEEE International Symposium on Reliable Distributed Systems (SRDS 2007)*, pages 209–218, Oct 2007.

[73] D. J. John, R. W. Smith, W. H. Turkett, D. A. Cañas, and E. W. Fulp. Evolutionary based moving target cyber defense. In *Proceedings of the Companion Publication of the 2014 Annual Conference on Genetic and Evolutionary Computation*, GECCO Comp '14, pages 1261–1268. ACM, 2014.

[74] A. Johnson, C. Wacek, R. Jansen, M. Sherr, and P. Syverson. Users get routed: Traffic correlation on Tor by realistic adversaries. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 337–348. ACM, 2013.

[75] P. Johnson, A. Vernotte, M. Ekstedt, and R. Lagerström. pwnpr3d: an attack-graph-driven probabilistic threat-modeling approach. In *Availability, Reliability and Security (ARES), 2016 11th International Conference on*, pages 278–283. IEEE, 2016.

[76] Jun Li, P. L. Reiher, and G. J. Popek. Resilient self-organizing overlay networks for security update delivery. *IEEE Journal on Selected Areas in Communications*, 22(1):189–202, 2004.

[77] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, CCS '03, pages 272–280. ACM, 2003.

[78] D. Kelly, R. Raines, R. Baldwin, M. Grimaila, and B. Mullins. Exploring extant and emerging issues in anonymous networks: A taxonomy and survey of protocols and metrics. *IEEE Communications Surveys & Tutorials*, 14(2):579–606, 2012.

[79] S. Kent, C. Lynn, and K. Seo. Secure border gateway protocol (S-BGP). *IEEE Journal on Selected areas in Communications*, 18(4):582–592, 2000.

[80] D. Kewley, R. Fink, J. Lowry, and M. Dean. Dynamic approaches to thwart adversary intelligence gathering. In *DARPA Information Survivability Conference amp; Exposition II, 2001. DISCEX '01*, volume 1, pages 176–185 vol.1, 2001.

[81] R. A. Kowalski. The early years of logic programming. *Commun. ACM*, 31(1):38–43, Jan. 1988.

[82] L. Martin. Cyber Kill Chain® · Lockheed Martin. `https://www.lockheedmartin.com/en-us/capabilities/cyber/cyber-kill-chain.html`, Last accessed on 2020-08-31.

[83] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. Sok: Automated software diversity. In *2014 IEEE Symposium on Security and Privacy*, pages 276–291, 2014.

[84] C. Lei, H.-Q. Zhang, L.-M. Wan, L. Liu, and D. he Ma. Incomplete information markov game theoretic approach to strategy generation for moving target defense. *Computer Communications*, 116:184 – 199, 2018.

[85] M. Lepinski and K. Sriram. BGPSEC protocol specification. Technical report, RFC 8205, 2017.

[86] J. Li, J. Yackoski, and N. Evancich. Moving target defense: A journey from idea to product. In *Proceedings of the 2016 ACM Workshop on Moving Target Defense*, MTD '16, pages 69–79. ACM, 2016.

[87] B. Lucas, E. W. Fulp, D. J. John, and D. Cañas. An initial framework for evolving computer configurations as a moving target defense. In *Proceedings of the 9th Annual Cyber and Information Security Research Conference*, CISR '14, pages 69–72. ACM, 2014.

[88] D. C. MacFarland and C. A. Shue. The SDN shuffle: Creating a moving-target defense using host-based software-defined networking. In *Proceedings of the Second ACM Workshop on Moving Target Defense*, MTD '15, pages 37–41. ACM, 2015.

[89] H. Maleki, S. Valizadeh, W. Koch, A. Bestavros, and M. van Dijk. Markov modeling of moving target defense games. In *Proceedings of the 2016 ACM Workshop on Moving Target Defense*, MTD '16, pages 81–92. ACM, 2016.

[90] P. K. Manadhata and J. M. Wing. An attack surface metric. *IEEE Transactions on Software Engineering*, 37(3):371–386, 2011.

[91] R. Mcgill, J. W. Tukey, and W. A. Larsen. Variations of box plots. *The American Statistician*, 32(1):12–16, 1978.

[92] D. McGrew and J. Viega. The galois/counter mode of operation (GCM). *Submission to NIST Modes of Operation Process*, 20, 2004.

[93] J. McLachlan, A. Tran, N. Hopper, and Y. Kim. Scalable onion routing with torsk. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 590–599. ACM, 2009.

[94] J. Mendonça, J.-H. Cho, T. J. Moore, F. F. Nelson, H. Lim, A. Zimmermann, and D. S. Kim. Performability analysis of services in a software-defined networking adopting time-based moving target defense mechanisms. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, SAC '20, page 1180–1189. ACM, 2020.

[95] A. Mitseva, A. Panchenko, and T. Engel. The state of affairs in BGP security: A survey of attacks and defenses. *Computer Communications*, 124:45–60, 2018.

[96] S. J. Murdoch and G. Danezis. Low-cost traffic analysis of Tor. In *Security and Privacy, 2005 IEEE Symposium on*, pages 183–195. IEEE, 2005.

[97] J. Narantuya, S. Yoon, H. Lim, J. Cho, D. S. Kim, T. Moore, and F. Nelson. SDN-based IP shuffling moving target defense with multiple SDN controllers. In *49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks - Supplemental Volume (DSN-S)*, pages 15–16, June 2019.

[98] R. L. Neupane, T. Neely, N. Chettri, M. Vassell, Y. Zhang, P. Calyam, and R. Durairajan. Dolus: Cyber defense using pretense against DDoS attacks in cloud platforms. In *Proceedings of the 19th International Conference on Distributed Computing and Networking*, ICDCN '18, pages 30:1–30:10. ACM, 2018.

[99] F. Nizzi, T. Pecorella, F. Esposito, L. Pierucci, and R. Fantacci. IoT security via address shuffling: The easy way. *IEEE Internet of Things Journal*, 6(2):3764–3774, April 2019.

[100] H. Okhravi, T. Hobson, D. Bigelow, and W. Streilein. Finding focus in the blur of moving-target techniques. *IEEE Security Privacy*, 12(2):16–26, Mar 2014.

[101] H. Okhravi, M. Rabe, T. Mayberry, W. Leonard, T. Hobson, D. Bigelow, and W. Streilein. Survey of cyber moving target techniques. Technical report, Massachusetts Inst of Tech Lexington Lincoln Lab, 2013.

[102] X. Ou, S. Govindavajhala, and A. W. Appel. Mulval: A logic-based network security analyzer. In *USENIX Security Symposium*, pages 8–8. Baltimore, MD, 2005.

[103] P. Fisher. HackBack! - A DIY Guide, 2015. `https://www.exploit-db.com/exploits/41915`, Last accessed on 2020-08-05.

[104] G. Portokalidis and A. D. Keromytis. Fast and practical instruction-set randomization for commodity systems. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 41–48. ACM, 2010.

[105] B. Potteiger, Z. Zhang, and X. Koutsoukos. Integrated instruction set randomization and control reconfiguration for securing cyber-physical systems. In *Proceedings of the 5th Annual Symposium and Bootcamp on Hot Topics in the Science of Security*, HoTSoS '18, New York, NY, USA, 2018. Association for Computing Machinery.

[106] B. Potteiger, Z. Zhang, and X. Koutsoukos. Integrated moving target defense and control reconfiguration for securing cyber-physical systems. *Microprocessors and Microsystems*, 73:102954, 2020.

[107] A. Prakash and M. P. Wellman. Empirical game-theoretic analysis for moving target defense. In *Proceedings of the Second ACM Workshop on Moving Target Defense*, MTD '15, pages 57–65. ACM, 2015.

[108] P. Rajasekaran, S. Crane, D. Gens, Y. Na, S. Volckaert, and M. Franz. CoDaRR: Continuous data space randomization against data-only attacks. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, ASIA CCS '20, page 494–505, New York, NY, USA, 2020. Association for Computing Machinery.

[109] J. Ren and J. Wu. Survey on anonymous communications in computer networks. *Computer Communications*, 33(4):420–431, 2010.

[110] K. Sagonas, T. Swift, and D. S. Warren. XSB as an efficient deductive database engine. In *In Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 442–453. ACM Press, 1994.

[111] J. Sankey and M. Wright. Dovetail: Stronger anonymity in next-generation internet routing. In *International Symposium on Privacy Enhancing Technologies Symposium*, pages 283–303. Springer, 2014.

[112] S. Sengupta, A. Chowdhary, A. Sabur, A. Alshamrani, D. Huang, and S. Kambhampati. A survey of moving target defenses for network security. *IEEE Communications Surveys Tutorials*, 22(3):1909–1941, 2020.

[113] E. Serrano-Collado, M. García-Valdez, and J.-J. M. Guervós. Delivering diverse web server configuration in a moving target defense using evolutionary algorithms. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*, GECCO '20, page 1520–1527, New York, NY, USA, 2020. Association for Computing Machinery.

[114] F. Shirazi, M. Simeonovski, M. R. Asghar, M. Backes, and C. Diaz. A survey on routing in anonymous communication protocols. *ACM Computing Surveys (CSUR)*, 51(3):51, 2018.

[115] R. Skowyra, K. Bauer, V. Dedhia, and H. Okhravi. Have no phear: Networks without identifiers. In *Proceedings of the 2016 ACM Workshop on Moving Target Defense*, MTD '16, page 3–14, New York, NY, USA, 2016. Association for Computing Machinery.

[116] P. Syverson, R. Dingledine, and N. Mathewson. Tor: The secondgeneration onion router. In *Usenix Security*, 2004.

[117] M. Taguinod, A. Doupé, Z. Zhao, and G. J. Ahn. Toward a moving target defense for web applications. In *2015 IEEE International Conference on Information Reuse and Integration*, pages 510–517, Aug 2015.

[118] J. Taylor, K. Zaffarano, B. Koller, C. Bancroft, and J. Syversen. Automated effectiveness evaluation of moving target defenses: Metrics for missions and attacks. In *Proceedings of the 2016 ACM Workshop on Moving Target Defense*, MTD '16, pages 129–134. ACM, 2016.

[119] P. Team. PaX address space layout randomization (aslr). 2001.

[120] M. Thompson, N. Evans, and V. Kisekka. Multiple OS rotational environment an implemented moving target defense. In *2014 7th International Symposium on Resilient Control Systems (ISRCS)*, pages 1–6, Aug 2014.

[121] J. D. Touch, G. G. Finn, Y.-S. Wang, and L. Eggert. DynaBone: dynamic defense using multi-layer internet overlays. In *Proceedings DARPA Information Survivability Conference and Exposition*, volume 2, pages 271–276 vol.2, April 2003.

[122] S. G. Vadlamudi, S. Sengupta, M. Taguinod, Z. Zhao, A. Doupé, G.-J. Ahn, and S. Kambhampati. Moving target defense for web applications using bayesian stackelberg games: (extended abstract). In *Proceedings of the 2016 International Conference on Autonomous Agents and Multiagent Systems*, AAMAS '16, pages 1377–1378, Richland, SC, 2016. International Foundation for Autonomous Agents and Multiagent Systems.

[123] A. Valmari. *The state explosion problem*, pages 429–528. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998.

[124] M. van den Berg, P. de Graaf, P. Kwant, and T. Slewe. Mass surveillance - part 2: Technology foresight, options for longer term security and privacy improvements. In *Study - Panel for the Future of Science and Technology (STOA)*. European Parliament, 2015.

[125] P. C. Van Oorschot, T. Wan, and E. Kranakis. On interdomain routing security and pretty secure BGP (psBGP). *ACM Transactions on Information and System Security (TISSEC)*, 10(3):11, 2007.

[126] L. Vanbever, O. Li, J. Rexford, and P. Mittal. Anonymity on quicksand: Using BGP to compromise Tor. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*, page 14. ACM, 2014.

[127] S. Venkatesan, M. Albanese, G. Cybenko, and S. Jajodia. A moving target defense approach to disrupting stealthy botnets. In *Proceedings of the 2016 ACM Workshop on Moving Target Defense*, MTD '16, pages 37–46. ACM, 2016.

[128] R. Wails, Y. Sun, A. Johnson, M. Chiang, and P. Mittal. Tempest: Temporal dynamics in anonymity systems. *Proceedings on Privacy Enhancing Technologies*, 2018(3):22–42, 2018.

[129] H. Wang, F. Li, and S. Chen. Towards cost-effective moving target defense against DDoS and covert channel attacks. In *Proceedings of the 2016 ACM Workshop on Moving Target Defense*, MTD '16, pages 15–25. ACM, 2016.

[130] S. Wang, H. Shi, Q. Hu, B. Lin, and X. Cheng. Moving target defense for internet of things based on the zero-determinant theory. *IEEE Internet of Things Journal*, 7(1):661–668, Jan 2020.

[131] B. C. Ward, S. R. Gomez, R. Skowyra, D. Bigelow, J. Martin, J. Landry, and H. Okhravi. Survey of cyber moving targets second edition. Technical report, MIT Lincoln Laboratory Lexington United States, 2018.

[132] R. White. Securing BGP through secure origin BGP (soBGP). *Business Communications Review*, 33(5):47–47, 2003.

[133] N. Wirth. What can we do about the unnecessary diversity of notation for syntactic definitions? *Commun. ACM*, 20(11):822–823, Nov. 1977.

[134] S. Woo, D. Moon, T.-Y. Youn, Y. Lee, and Y. Kim. CAN ID shuffling technique (CIST): Moving target defense strategy for protecting in-vehicle CAN. *IEEE Access*, 7:15521–15536, 2019.

[135] B. Wu, Y. Ma, L. Fan, and F. Qian. Binary software randomization method based on LLVM. In *2018 IEEE International Conference of Safety Produce Informatization (IICSPI)*, pages 808–811, Dec 2018.

[136] X. Xiong, L. Yang, and G. Zhao. Effectiveness evaluation model of moving target defense based on system attack surface. *IEEE Access*, 7:9998–10014, 2019.

[137] Y. Tekol. PySwip. `https://github.com/yuce/pyswip`, Last accessed on 2020-10-03.

[138] J. Yackoski, J. Li, S. A. DeLoach, and X. Ou. Mission-oriented moving target defense based on cryptographically strong network dynamics. In *Proceedings of the Eighth Annual Cyber Security and Information Intelligence Research Workshop*, CSIIRW '13, pages 57:1–57:4. ACM, 2013.

[139] J. Yackoski, P. Xie, H. Bullen, J. Li, and K. Sun. A self-shielding dynamic network architecture. In *2011 - MILCOM 2011 Military Communications Conference*, pages 1381–1386, Nov 2011.

[140] K. Zaffarano, J. Taylor, and S. Hamilton. A quantitative framework for moving target defense effectiveness evaluation. In *Proceedings of the Second ACM Workshop on Moving Target Defense*, MTD '15, pages 3–10. ACM, 2015.

[141] K. Zeitz, M. Cantrell, R. Marchany, and J. Tront. Changing the game: A micro moving target IPv6 defense for the internet of things. *IEEE Wireless Communications Letters*, 7(4):578–581, Aug 2018.

[142] H. Zhang, K. Zheng, X. Yan, S. Luo, and B. Wu. Moving target defense against injection attacks. In S. Wen, A. Zomaya, and L. T. Yang, editors, *Algorithms and Architectures for Parallel Processing*, pages 518–532, Cham, 2020. Springer International Publishing.

[143] J. Zheng and A. S. Namin. A survey on the moving target defense strategies: An architectural perspective. *Journal of Computer Science and Technology*, 34(1):207–233, 2019.

[144] J. Zheng and A. Siami Namin. Enforcing optimal moving target defense policies. In *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 753–759, Jul 2019.

[145] Q. Zhu, A. Clark, R. Poovendran, and T. Başar. Deceptive routing games. In *2012 IEEE 51st IEEE Conference on Decision and Control (CDC)*, pages 2704–2711, 2012.

[146] R. Zhuang, A. G. Bardas, S. A. DeLoach, and X. Ou. A theory of cyber attacks: A step towards analyzing MTD systems. In *Proceedings of the Second ACM Workshop on Moving Target Defense*, MTD '15, pages 11–20. ACM, 2015.

[147] R. Zhuang, S. A. DeLoach, and X. Ou. A model for analyzing the effect of moving target defenses on enterprise networks. In *Proceedings of the 9th Annual Cyber and Information Security Research Conference*, CISR '14, pages 73–76. ACM, 2014.

[148] R. Zhuang, S. A. DeLoach, and X. Ou. Towards a theory of moving target defense. In *Proceedings of the First ACM Workshop on Moving Target Defense*, MTD '14, pages 31–40. ACM, 2014.

[149] R. Zhuang, S. Zhang, A. Bardas, S. A. DeLoach, X. Ou, and A. Singhal. Investigating the application of moving target defenses to network security. In *Resilient Control Systems (ISRCS), 2013 6th International Symposium on*, pages 162–169. IEEE, 2013.

[150] R. Zhuang, S. Zhang, S. A. DeLoach, X. Ou, and A. Singhal. Simulation-based approaches to studying effectiveness of moving-target network defense. In *National Symposium on Moving Target Research*. NIST, 2012.

A. Available Functions in Experiment 1

Table A.1 lists legitimate system functions and Table A.2 the modeled exploits that can be used by an attacker.

Table A.1.: Overview of attacker actions based on legitimate functions

Name: readData **Result**: Attacker.knows+=App.allData; **Time/Probability**: 60 / 100 **Requirements**:App=Attacker.remoteCodeExe OR (OS=App.parent & OS=Attacker.remoteCodeExe)
Name: pingscan **Result**: Attacker.knows+=OS.ipaddress; **Time/Probability**: 175 / 2 **Requirements**: Attacker.reachable(OS,Port=ping)
Name: arpCache **Result**: Attacker.knows+=TARGET.ipaddress; **Time/Probability**: 20 / 100 **Requirements**:(App=Attacker.remoteCodeExe OR OS=Attacker.remoteCodeExe) & OS=App.parent & SUBNET=OS.belongsToSubnet & TARGET.belongsToSubnet=SUBNET
Name: configureAdClients **Result**: Attacker.remoteCodeExe+=TARGET; **Time/Probability**: 200 / 80 **Requirements**: App=activeDirectory & Attacker.remoteCodeExe=App & TARGET=App.clients
Name: getCustomerData **Result**: Attacker.knows+=CRMUSER.data; **Time/Probability**: 20 / 100 **Requirements**:App=baseCrm & CRMUSER=App.user & Attacker.knows=CRMUSER.password & Attacker.knows=CRMUSER.username & OS=App.parent & (Attacker.knows=OS.ipaddress OR Attacker.knows=OS.dnsName) & Attacker.reachable(OS,Port=tcp)
Name: getMail **Result**: Attacker.knows+=CRMUSER.data; **Time/Probability**: 20 / 100 **Requirements**:App=exchangeServer & EMAILUSER=App.user & Attacker.knows=EMAILUSER.password & Attacker.knows=EMAILUSER.username & OS=App.parent & (Attacker.knows=OS.ipaddress OR Attacker.knows=OS.dnsName) & Attacker.reachable(OS,Port=tcp)
Name: remoteDbManagement **Result**: Attacker.knows+=App.allDatabaseData; **Time/Probability**: 180 / 100 **Requirements**:App=sqlServer & ADMIN=App.admin & Attacker.knows=ADMIN.password & Attacker.knows=ADMIN.username & OS=App.parent & (Attacker.knows=OS.ipaddress OR Attacker.knows=OS.dnsName) & Attacker.reachable(OS,Port=SQLPORT)
Name: sqlQuery **Result**: Attacker.knows+=USER.databaseData; **Time/Probability**: 30 / 100 **Requirements**:App=sqlServer & USER=App.databaseUser & Attacker.knows=USER.password & Attacker.knows=USER.username & OS=App.parent & (Attacker.knows=OS.ipaddress OR Attacker.knows=OS.dnsName) & Attacker.reachable(OS,Port=sqlport)
Name: remoteShellLinux **Result**: Attacker.remoteCodeExe+=OS; **Time/Probability**: 20 / 100 **Requirements**: OS.family=Linux & OS.remoteShellEnabled & ADMIN=OS.root & Attacker.knows=ADMIN.password & Attacker.knows=ADMIN.username & (Attacker.knows=OS.ipaddress OR Attacker.knows=OS.dnsName) & Attacker.reachable(OS,Port=22)
Name: remoteShellWindows **Result**: Attacker.remoteCodeExe+=OS; **Time/Probability**: 20 / 100 **Requirements**: OS.family=Windows & OS.remoteShellEnabled & ADMIN=OS.root & Attacker.knows=ADMIN.password & Attacker.knows=ADMIN.username & (Attacker.knows=OS.ipaddress OR Attacker.knows=OS.dnsName) & Attacker.reachable(OS,Port=3389)

Table A.2.: Overview of attacker actions based on exploits

Name: tomPrivEscalation **Result**: Attacker.remoteCodeExe+=OS; **Time/Probability**: 800 / 25 **Requirements**: App=tomcat & App.hasTomPriv & Attacker.remoteCodeExe=App & OS=Linux
Name: privEscalationWindows **Result**: Attacker.remoteCodeExe+=OS; **Time/Probability**: 60 / 100 **Requirements**: (OS=Windows10 OR WindowsServer2016) & OS.hasWinPrivEscalation & Attacker.remoteCodeExe=App
Name: backupServerRCE **Result**: Attacker.remoteCodeExe+=OS; **Time/Probability**: 30 / 100 **Requirements**: App=veritasBackupServer & OS=App.parent & OS.family=linux & App.hasCVE20167399 & (Attacker.knows=OS.ipaddress OR Attacker.knows=OS.dnsName) & Attacker.reachable(OS,Port=tcp)
Name: phishingDocRCE **Result**: Attacker.remoteCodeExe+=App; **Time/Probability**: 200 / 2 **Requirements**: App=officeSuite & OS.family=windows & OS=App.parent & App.isPhishingVulnerable
Name: tomHttpPutRCE **Result**: Attacker.remoteCodeExe+=App; **Time/Probability**: 30 / 100 **Requirements**: App=tomcat & App.hasHttpPutVulnerability & OS=App.parent &(Attacker.knows=OS.ipaddress OR Attacker.knows=OS.dnsName) & Attacker.reachable(OS,Port=jmxport)
Name: jmxTomcatVulnerability **Result**: Attacker.remoteCodeExe+=OS; **Time/Probability**: 120 / 90 **Requirements**: App=tomcat & App.hasJmxEnabled & OS=App.parent & (Attacker.knows=OS.ipaddress OR Attacker.knows=OS.dnsName) & (App.jmxNoAuth OR (Attacker.knowsUsername(App) & Attacker.knowsPassword(App)) & Attacker.reachable(OS,jmxport)
Name: privEscalationUbuntu **Result**: Attacker.remoteCodeExe+=OS; **Time/Probability**: 120 / 50 **Requirements**: OS=Ubuntu & OS.hasUbuntuPrivEscalation & OS=App.parent & Attacker.remoteCodeExe=App
Name: eternalBlueRCE **Result**: Attacker.remoteCodeExe=OS; **Time/Probability**: 60 / 10 **Requirements**: OS.family=Windows & OS.hasEternalBlue & (Attacker.knows=OS.ipaddress OR Attacker.knows=OS.dnsName) & Attacker.reachable(OS,Port=smb)
Name: redirectBackupToCloud **Result**: Attacker.knows+=App.backupedData; **Time/Probability**: 300 / 90 **Requirements**: App=veritasBackupServer & App.hasCloudVuln & (Attacker.knows=OS.ipaddress OR Attacker.knows=OS.dnsName) & Attacker.reachable(OS,Port=tcp/5637)
Name: backupClientRCE **Result**: Attacker.remoteCodeExe=OS; **Time/Probability**: 120 / 20 **Requirements**: App=veritasBackupClient & Attacker.parent=OS & OS.family=windows & APP.hasSSLVuln & (Attacker.knows=OS.ipaddress OR Attacker.knows=OS.dnsName) & Attacker.reachable(Os,Port=ssl)
Name: clientRCEoverServer **Result**: Attacker.remoteCodeExe=OS; **Time/Probability**: 120 / 50 **Requirements**: App=veritasBackupClient & App.hasRCEfromServer & SERVER=App.server & Attacker.remoteCodeExe=SERVER & OS=App.parent & Attacker.knows=OS.ipaddres & reachable(OS,Port=ssl)
Name: meltdown **Result**: Attacker.knows+=Node.dataInRAM; **Time/Probability**: 200 / 10 **Requirements**: NODE.type=Intel & OS.runsOn=NODE & OS.hasMeltdown & App.parent=OS & Attacker.remoteCodeExe=App
Name: drupalRCE **Result**: Attacker.remoteCodeExe+=App; **Time/Probability**: 30 / 100 **Requirements**: App=drupal & App.hasRCEviaHttpGetVuln OS=App.parent & (Attacker.knows=OS.ipaddress OR Attacker.knows=OS.dnsName) & Attacker.reachable(OS,Port=http)
Name: sendMailExchangeRCE **Result**: Attacker.remoteCodeExe+=App; **Time/Probability**: 120 / 30 **Requirements**: App=exchangeServer & OS=App.parent & OS.family=windows & USER=App.emailuser & Attacker.knows=USER.username & Attacker.knows=USER.password & (Attacker.knows=OS.ipaddress OR Attacker.knows=OS.dnsName)
Name: exchangeDefenderRCE **Result**: Attacker.remoteCodeExe+=App; **Time/Probability**: 32 / 20 **Requirements**: App=exchangeServer & OS=App.parent & (Attacker.knows=OS.ipaddress OR Attacker.knows=OS.dnsName)

B. Supplemental Material to Benchmark Fuzzing and Experiment 2

Results from the experiment performed on the basis of 500 automatically diversified benchmark networks and the program code to generate them can obviously not be part of a printable document. Instead, respective graphs, together with python scripts to generate them can be found at `https://github.com/AlexB030/BenchmarkNetworkAnalysis`. The archive also contains condensed raw data in form of a binary pickle files that are used by the python script to generate respective output. Furthermore, the scenario definition that was used to generate all 500 benchmark networks can be found at `https://github.com/AlexB030/BenchmarkNetworkSynthesis`. Enclosed files describe the general structure of generated network instances and how they are assembled. Furthermore, the definitions of all implemented functions are enclosed, providing insights into requirements and effects.

C. Detailed dPHI Protocol Description

In the pseudo-code used to describe the protocol, capital letters denote structs with several fields, while minuscule letters denote single entries. The message header is denoted with H and the payload with P. A dot is used to address a specific field within a struct. The encryption function $(c, t, iv) = \texttt{enc}_k(p, a)$ denotes an authenticated encryption function (e.g. AES-GCM [92]) with plaintext p, key k and additional authentication data a. The output is ciphertext c, authentication tag t and a fresh initialization vector iv. Similarly, $p = \texttt{dec}_k(c, t, iv, a)$ is the authenticated decryption funcion which ensures the integrity of plaintext p and authentication data a. For public key operations $(priv, pub) = \texttt{ECDH}_{gen}()$ denotes the generation of a public-key private-key pair of an Elliptic-Curve Diffie Hellman algorithm. Generating a session key between two entities A and B is done using the ECDH function $k_{A-B} = \texttt{ECDH}(pub_A, priv_B) = \texttt{ECDH}(pub_B, priv_A) = k_{B-A}$, where pub_A and pub_B are the public keys of A and B respectively and $priv_A$ and $priv_B$ the private keys. The $\texttt{Assert}(S)$ function verifies that the Boolean expression S is true. If it is false, the protocol is aborted, i.e., the node drops the processing of the current message. The function $\texttt{sendMessage}(port, H, P)$ denotes the sending of the message of a node to the next node at port $port$ with header H and payload P.

C.1. Message Header and Routing Segments

The bulk of the header is made up of two routing segments V^1 and V^2 (see Figure 7.4), each consisting of l routing elements that contain routing information for every node on the path. Each node A_i has a unique secret symmetric key k_i that is not shared with any other entity. It is used to encrypt the routing information R and store it in the routing segment so that it can be retrieved and authenticated later on. More formally, the encrypted information consists of a triplet (iv_i, c_i, t_i), where iv_i is a freshly generated initialization vector, c_i is the ciphertext and t_i the authentication tag of an authenticated encryption algorithm with:

$$(c_i, t_i, IV_i) = enc_{k_i}(R; sid||c_{pos-1}) \tag{C.1}$$

where R is the plaintext routing information that gets encrypted and $sid||c_{pos-1}$ is the additional authentication data. The routing information R consists of five fields in total.

$$R = (port1||port2||type||posV1||posV2) \tag{C.2}$$

The two fields $port1$ and $port2$ correspond to ingress and egress interfaces, and the *type* field defines the role of the node in the session (e.g. midway node or entry node). The fields $posV1$ and $posV2$ point to the routing element in V^1 and V^2 where R is stored. Only the midway node W stores R in both routing segments so that for other nodes one of the pointers is **null**.

Algorithm 2 Phase 0: Setup

procedure SETUP(d)
- $(priv_s, pub_s) \leftarrow \texttt{ECDH}_{\texttt{gen}}()$
- $n_{mid} \leftarrow \texttt{random}()$
- $M \leftarrow \texttt{chooseHelperNode}()$
- $dist_M \leftarrow \texttt{maxDistanceTo}(s, M)$
- $pub_M \leftarrow \texttt{lookupPublicKey}(\texttt{M})$
- $k_{s-M} \leftarrow \texttt{ECDH}(pub_M, priv_s)$
- $H.sid \leftarrow \texttt{Hash}(pub_s)$
- $(c, t, IV) \leftarrow enc_{k_{s-M}}(d||n_{mid}, H.sid)$
- $P \leftarrow (c, t, IV, pub_s)$
- $H.V^1 \leftarrow \texttt{random}()$
- $H.V^2 \leftarrow \texttt{zeros}()$
- $H.midway \leftarrow \texttt{zeros}()$
- $H.pos \leftarrow \texttt{random}(0, l-1)$
- $H.dest \leftarrow M$
- $H.status \leftarrow$ "$toHelperNode$"
- $H_s \leftarrow H$
- $\texttt{store}(H_s, priv_s, pub_s, k_{s-M}, dist_M, n_{mid})$
- $\texttt{sendMessage}(A_s, H, P)$

end procedure

C.2. Phase 0: Initialization

Algorithm 2 details the initialization performed by source s. For each path request, source s generates a fresh ECDH key pair, consisting of pub_s and $priv_s$. The source computes a session id sid by hashing the public key with a cryptographically secure hash function $sid = \texttt{Hash}(pub_s)$. This way a session is intrinsically linked to the freshly generated ECDH key pair. This idea was introduced in PHI to securely link the path setup and the transmission phase and is kept in dPHI.

The source then chooses a helper node M from a list of trusted ASes. It is assumed that the list also contains the maximum distance $dist_M$ (number of hops) between the source and the helper node M as well as the public key pub_M of M. The source uses its own private key $priv_s$ and M's public key pub_M to compute a session key k_{s-M} with $k_{s-M} = \texttt{ECDH}(priv_s, pub_M)$. The source uses this session key k_{s-M} to encrypt the destination address d as well as freshly generate a high-entropy random midway nonce n_{mid}. The routing elements in the routing segment V^1 are initiated by s with random values that are indistinguishable from real routing elements. This can, for example, be done using a cryptographically secure pseudo-random number generator with a fresh high-entropy seed value. Furthermore, s chooses a random start position $pos \in \{0, .., l-1\}$ and stores pos and V^1 locally for later reference. After initialization, the source s starts the midway request phase by sending a request message to its corresponding AS A_s. The payload of the midway request consists of the encrypted destination and the session's public key pub_s. The destination field $dest$ of the message header is set to $dest := M$, the status field to $status :=$ "$toHelperNode$" to indicate that the protocol is in the initial phase.

C.3. Phase 1: Midway Request

The processing of a midway request message is described in Algorithm 3. Each node on the path to M encrypts its routing information R and stores it in the routing segment V^1 using authenticated encryption. The session id sid as well as the ciphertext of the previous routing element are used as additional authentication data for the authenticated encryption algorithm. When the midway request reaches M, M decrypts the destination d and midway nonce n_{mid}. It sets the destination field $dest$ and the $midway$ field of the header to d and n_{mid} respectively. Then the message's status is set to "findMidway" and sent back to the previous node.

C.4. Phase 2: Find Midway Node W

Algorithms 4 and 5 explain Phase 2 of the dPHI protocol. The goal is to find a midway node W that is on the path P_{s-M} between s and M such that a path $P_{s-d} = P_{s-W*} \cap P_{W-d}$ meets the routing policy of the network (e.g. valley-freeness, shortest path etc.). Each node receiving a "findMidway" request decrypts its routing information in V^1 and uses the ingres information and the destination d to decide whether or not to become the midway node. If not, the message is simply forwarded to the previous node, reachable through $R.port1$.

Algorithm 3 Phase1: Midway Request

Require: $H.status == \text{“}toHelperNode\text{”}$

1: **procedure** MIDWAYREQUEST($ingres$,H,P)
2: **if** $self == H.dest$ **then**
3: $\texttt{Assert}(H.sid == \texttt{Hash}(P.pub_s))$
4: $k_{s-M} \leftarrow \texttt{ECDH}(priv_M, P.pub_s)$
5: $d||n_{mid} = \texttt{dec}_{k_{s-M}}(P.c, P.t, P.IV, sid)$
6: $H.dest \leftarrow d$
7: $H.status \leftarrow \text{“}findMidway\text{”}$
8: $H.midway \leftarrow n_{mid}$
9: $H.pos \leftarrow H.pos - 1 \bmod l$
10: $\texttt{sendMessage}(ingres, H, P)$
11: **else** ▷ M not yet reached
12: $R.port2 \leftarrow \texttt{FindRouteTo}(H.dest)$
13: $R.port1 \leftarrow ingres$
14: $R.posV1 \leftarrow H.pos$
15: $R.posV2 \leftarrow$ **null**
16: $c_{prev} \leftarrow H.V^1[H.pos - 1 \bmod l][0]$
17: **if** $\texttt{isClient}(ingres)$ **then**
18: $R.type \leftarrow \text{“}entryNode\text{”}$
19: **else**
20: $R.type \leftarrow \text{“}V1\text{”}$
21: **end if**
22: $H.V^1[H.pos] \leftarrow \texttt{enc}_{k_i}(R, H.sid||c_{prev})$
23: $H.pos \leftarrow H.pos + 1 \bmod l$
24: $\texttt{sendMessage}(R.port2, H, P)$
25: **end if**
26: **end procedure**

Once a node becomes the midway node, it prepares a midway reply n_{rep} using the midway nonce n_{mid} from the midway field. The idea behind the midway reply is to allow the source s to verify the integrity of destination d as well as V^1. To do this, it computes $n_{rep} = \texttt{Hash}(H.dest||n_{mid}||H.V^1)$ and sets the midway field in the header to n_{rep}. Furthermore, the midway node updates its routing information R with the updated $R.port2$ information to route messages to d. It also chooses a random start position in V^2 and stores it in $R.posV2$, sets $R.type = \text{“}midway\text{”}$, re-encrypts R, and updates its routing segment in V^1 accordingly. Note, that this updated V^1 is used by the hash function to determine n_{rep}. Then the midway node encrypts the destination field $H.dest$ in the header with its secret key so that nodes between s and W do not learn d. Lastly, the message is sent back to s via the nodes on path P_{s-W}. These nodes only relay the message but do not alter the routing segment.

Algorithm 4 Phase 2: Find Midwaynode

Require: $H.status == \text{“}findMidway\text{”}$

1: **procedure** FINDMIDWAY($ingres$,H,P)
2: $c_{prev} \leftarrow H.V^1[(H.pos - 1) \bmod l][0]$
3: $R \leftarrow \texttt{dec}_{k_i}(H.V^1[H.pos], H.sid||c_{prev})$
4: $\texttt{Assert}(R.type == \text{“}V1\text{”}\&\&R.posV1 == H.pos)$
5: **if** $\texttt{checkIfNewMidway}(self, R.port1, H.dest)$ **then**
6: $R.type \leftarrow \text{“}midway\text{”}$
7: $n_{mid} \leftarrow H.midway$
8: $R.posV2 \leftarrow \texttt{random}(0, l - 1)$
9: $R.port2 \leftarrow \texttt{FindRouteTo}(H.dest)$
10: $H.V^1[H.pos] \leftarrow \texttt{enc}_{k_i}(R, H.sid||c_{prev})$
11: $H.midway \leftarrow \texttt{Hash}(H.dest||n_{mid}||H.V^1)$
12: $H.dest \leftarrow \texttt{enc}_{k_i}(H.dest, H.sid)$
13: $H.status \leftarrow \text{“}midwayReply\text{”}$
14: **end if**
15: $H.pos \leftarrow H.pos - 1 \bmod l$
16: $\texttt{sendMessage}(R.port1, H, P)$
17: **end procedure**

C.5. Phase 3: Handshake to d

When receiving a midway reply, source s first verifies the integrity of the received routing segment $H.V^1$. Only $dist_M$ routing elements should have been modified compared to the randomly initialized V_s^1 starting from position pos_s in consecutive order. If this is not the case the message is dropped. It then verifies that the midway field $H.midway$ is equal to $\texttt{Hash}(d||n_{mid}||H.V^1)$ to verify that the correct destination was used by midway node W and that V^1 has not been altered after processing by W. If these checks succeed, the source computes a session key k_{s-d} based on the private key of the ECDH key pair it generated for this session, as well as d's publicly available longterm key pub_d. It uses this session key to encrypt V^1 and sends it together with the public key pub_s to d in a handshake message. The process of this handshake initiation is covered in Algorithm 6.

Algorithm 5 Phase 2: Midway Reply

Require: $H.status == \text{“}midwayReply\text{”}$

1: **procedure** MIDWAYREPLY($ingres$,H,P)
2: $c_{prev} \leftarrow H.V^1[(H.pos - 1) \bmod l][0]$
3: $R \leftarrow \texttt{dec}_{k_i}(H.V^1[H.pos], H.sid||c_{prev})$
4: $\texttt{Assert}(H.pos == R.posV1)$
5: $H.pos \leftarrow H.pos - 1 \bmod l$
6: $\texttt{sendMessage}(R.port1, H, P)$
7: ▷ If $H.type == \text{“}entryNode\text{”}$ then $R.port1$ is a client address
8: **end procedure**

Algorithm 7 covers communication of nodes on the path P_{s-W} that forward the message to the midway node W according to the routing information stored in $H.V^1$. The midway node uses its secret key to decrypt the destination field $d = dec_{k_i}(H.dest)$ and sets $H.dest = d$. It then chooses a random seed $seed$ and uses a cryptographically secure pseudo-random number generator to initiate the second routing segment V^2 with $V^2 = CPRNG(seed)$. Furthermore, it determines the maximum expected number of hops $dist_d$ for the message to reach d. It uses authenticated encryption to encrypt $seed$ and $dist_d$ with W's routing element in V^1 as additional authentication data and stores it in the header's midway field. The midway node serves as the bridge between nodes writing their routing information in V^1 and V^2. It therefore re-encrypts its routing segment and stores it in V^2 at position $posV2$ (so that $V^1[posV1]$ and $V^2[posV2]$ contain the same plaintext but different ciphertexts). It then initiates the next phase of the protocol by setting $pos = posV2 + 1 \bmod l$ and sending the message via $port2$.

Should the midway node also be the exit node ($port2$ is a client address) the message is sent directly to the client. Otherwise the handshake message is forwarded to the next node specified in $port2$. Nodes between W and d retrieving the handshake message look up the destination and where to forward the package. Then, they store their routing information encrypted in V^2 the same way, as nodes between s and M have done with V^1 during the first phase of the protocol and forward the message. This process is described in Algorithm 8.

C.6. Phase 4: Handshake Reply

Algorithm 9 describes the handshake reply preparation by destination d. At first, destination d verifies that sid is the hash of the received public key pub_s and computes the session key k_{s-d} using this public key and its own private key. The session key is then used to decrypt the routing segment stored in the payload and compare it with the received routing segment $H.V^1$. If they are identical, the destination d encrypts both routing segment V^1 and V^2 and uses this as the payload of the handshake reply message. It deletes the destination field from the header and sends back the message to s.

Algorithm 6 Phase 3: Initiate handshake

Require: Client s receives message with $H.status == \text{“}midwayReply\text{”}$

procedure INITHANDSHAKE($ingress, H, P$)
  $H_s, priv_s, pub_s, dist_M, k_{s-M}, \leftarrow restore()$
  `Assert`($H.sid == H_s.sid \&\& H.pos == H_s.pos$)
  /* Verify that only routing segments in V^1 have been modified between positions $H_s.pos$ and $(H_s.pos + dist_M \bmod l)$ */
  $pointer = H_s.pos + dist_M + 1 \bmod l$
  while $pointer \neq H_s.pos$ **do**
    if $H.V^1[pointer] \neq H_s.V^1[pointer]$ **then**
      `Abort`()
    end if
    $pointer = pointer + 1 \bmod l$
  end while
  $n_{rep} \leftarrow$ `Hash`$(d||n_{mid}||H.V^1)$
  `Assert`($n_{rep} == H.midway$)
  ▷ All checks valid, send message to d
  $k_{s-d} = ECDH(pub_d, priv_s)$
  $(c, t, IV) \leftarrow \texttt{enc}_{k_{s-d}}(H.V^1, sid)$
  $P \leftarrow c, t, IV, pub_s$
  $H.status \leftarrow \text{“}handshakeToW\text{”}$
  `store`(V^1)
  `sendMessage`($ingress, H, P$)
end procedure

Algorithms 10 and 11 describe the processing of the handshake reply message by the routing nodes between d and s. The nodes between d and W look up the routing information in their routing segments and forward the message to W without modifying any routing segments. The midway node receiving a handshake reply message verifies that there were no unauthorized modifications in V^2. For this, the midway field is decrypted to retrieve the seed for V^2 and the distance $dist_d$ between W and d. It uses the seed to re-compute the initial value of V^2 and verifies that at most $dist_d$ routing elements have changed in V^2, starting at position pos. If this check succeeds, the pointer is switched to V^1 and the message forwarded to s.

The source s receiving the handshake reply verifies that the received routing segment V^1 matches the stored routing segment. It then decrypts the payload to verify that the routing segment received by d is the same as the one received by the source. If this is true, the handshake is complete and a secure path has been established that is linked to the session key k_{s-d}. This process is described in Algorithm 12.

Algorithm 7 Phase 3: Handshake to W

Require: $H.status == \text{“}handshakeToW\text{”}$

procedure HANDSHAKETOW($ingress$,H,P)
  $c_{prev} \leftarrow H.V^1[(H.pos - 1 \bmod l][0]$
  $R \leftarrow \texttt{dec}_{k_i}(H.V^1[H.pos], H.sid||c_{prev})$
  $\texttt{Assert}(H.pos == R.posV1)$
  if $R.type == \text{“}midway\text{”}$ **then**
    $H.dest \leftarrow \texttt{dec}_{k_i}(H.dest, H.sid)$
    $seed \leftarrow \texttt{random}()$
    $H.V^2 \leftarrow \texttt{CPRNG}(seed)$
    $dist_d \leftarrow \texttt{maxDistanceTo}(H.dest)$
    $c_{prevV2} \leftarrow H.V^2[(R.posV2 - 1 \bmod l][0]$
    $H.V^2[R.posV2] \leftarrow \texttt{enc}_{k_i}(R, H.sid||c_{prevV2})$
    $c_{v1} \leftarrow H.V^1[R.posV1][0]$
    $H.midway \leftarrow \texttt{enc}_{k_i}(seed||dist_d, sid||c_{v1})$
    $H.status \leftarrow \text{“}handshakeToD\text{”}$
    $H.pos \leftarrow R.posV2 + 1 \bmod l$
  else
    $H.pos \leftarrow H.pos + 1 \bmod l$
  end if
  $\texttt{sendMessage}(R.port2, H, P)$
end procedure

Algorithm 8 Phase 3: Handshake to d

Require: $H.status == \text{“}handshakeToD\text{”}$

procedure HANDSHAKETOD($ingress$,H,P)
  if $\texttt{isClient}(H.dest)$ **then** ▷ Message arrived, forward to client
    $R.type \leftarrow \text{“}destNode\text{”}$
    $R.port2 \leftarrow H.dest$
  else
    $R.type \leftarrow \text{“}V2\text{”}$
    $R.port2 \leftarrow \texttt{findRouteTo}(H.dest)$
  end if
  $R.port1 \leftarrow ingress$
  $R.posV1 \leftarrow$ **null**
  $R.posV2 \leftarrow H.pos$
  $c_{prev} \leftarrow H.V^2[(H.pos - 1 \bmod l)[0]$
  $H.V^2[H.pos] \leftarrow \texttt{enc}_{k_i}(R, H.sid||c_{prev})$
  $H.pos \leftarrow H.pos + 1 \bmod l$
  $\texttt{sendMessage}(R.port2, H, P)$
end procedure

Algorithm 9 Phase 4: Prepare handshake reply

Require: Client d receives message with $H.status ==$ "$handshakeToD$"

procedure INITHANDSHAKEREPLY($ingress, H, P$)
  $\texttt{Assert}(H.sid == \texttt{Hash}(P.pub_s))$
  $k_{s-d} \leftarrow \texttt{ECDH}(P.pub_s, priv_d)$
  $V_s^1 \leftarrow \texttt{dec}_{k_{s-d}}(P.c, P.t, P.IV, H.sid)$
  $\texttt{Assert}(H.V^1 == V_s^1)$
  $P \leftarrow \texttt{enc}_{k_{s-d}}(H.V^1||H.V^2, sid)$
  $H.dest \leftarrow \texttt{zeros}()$
  $H.status \leftarrow$ "$replyToW$"
  $\texttt{store}(H)$
  $\texttt{sendMessage}(ingress, H, P)$
end procedure

Algorithm 10 Phase 4: Handshake Reply

Require: $H.status ==$ "$replyToW$"

procedure HANDREPLYTOW($ingress$,H,P)
  $c_{prev} \leftarrow H.V^2[(H.pos - 1 \bmod l][0]$
  $R \leftarrow \texttt{dec}_{k_i}(H.V^2[H.pos], H.sid||c_{prev})$
  $\texttt{Assert}(H.pos == R.posV2)$
  if $R.type ==$ "$midway$" **then**
    ▷ Verify V^2 and then switch to V^1
    $c_{v1} \leftarrow H.V^1[R.posV1][0]$
    $seed||dist_d \leftarrow \texttt{dec}_{k_i}(H.midway, sid||c_{v1})$
    $V_s^2 \leftarrow \texttt{CPRNG}(\texttt{seed})$
    ▷ Only $dist_d$ elements in V^2 should have changed starting at $H.pos$
    $pointer = H.pos + dist_d + 1 \bmod l$
    while $pointer \neq H.pos$ **do**
      if $H.V^2[pointer] \neq V_s^2[pointer]$ **then**
        $\texttt{Abort}()$
      end if
      $pointer = pointer + 1 \bmod l$
    end while
    $c_{v2} \leftarrow H.V^2[R.posV2][0]$
    $H.midway \leftarrow \texttt{MAC}_{k_i}(c_{v1}||c_{v2}||sid)$
    $H.pos \leftarrow R.posV1 - 1 \bmod l$
    $H.status \leftarrow$ "$replyToS$"
  else ▷ Not the midway node
    $H.pos \leftarrow H.pos - 1 \bmod l$
  end if
  $\texttt{sendMessage}(R.port1, H, P)$
end procedure

Algorithm 11 Phase 4: handshake Reply to s

Require: $H.status == \text{“}replyToS\text{”}$

procedure HANDREPLYTOS($ingress$,H,P)
  $c_{prev} \leftarrow H.V^1[H.pos - 1 \bmod l][0]$
  $R \leftarrow \texttt{dec}_{k_i}(H.V^1[H.pos], H.sid||c_{prev})$
  $\texttt{Assert}(H.pos == R.posV1)$
  $\texttt{sendMessage}(R.port1, H, P)$
end procedure

Algorithm 12 Phase 4: handshake finish

Require: Client c receives message with $H.status == \text{“}replyToS\text{”}$

procedure HANDSHAKEFINISH($ingress, H, P$)
  $H_s, priv_s, pub_s, k_{s-d}, n_{mid} \leftarrow restore()$
  $V_d^1||V_d^2 \leftarrow \texttt{dec}_{k_{s-d}}(P, H.sid)$
  $\texttt{Assert}(H_s.V^1 == H.V^1 \&\& H.V^1 == V_d^1)$
  $\texttt{Assert}(H.V^2 == V_d^2)$
  $H.status \leftarrow \text{“}transmissionPhaseToD1\text{”}$
  $\texttt{store}(H)$
end procedure

Algorithm 13 Phase 5: Transmission phase

Require: $H.status == \text{“}transToD1\text{”}$

procedure TRANSMISSIONPHASED1($ingres$,H,P)
  $c_{prev} \leftarrow H.V^1[(H.pos - 1 \bmod l][0]$
  $R \leftarrow \texttt{dec}_{k_i}(H.V^1[H.pos], H.sid||c_{prev})$
  $\texttt{Assert}(H.pos == R.posV1)$
  $\texttt{Assert}(R.type == (\text{“}midway\text{”} \text{ OR } \text{“}V1\text{”}))$
  if $type == \text{“}midway\text{”}$ **then**
    $c_{v1} \leftarrow H.V^1[R.posV1][0]$
    $c_{v2} \leftarrow H.V^2[R.posV2][0]$
    $m \leftarrow \texttt{MAC}_{k_i}(c_{v1}||c_{v2}||H.sid)$
    $\texttt{Assert}(m == H.midway)$
    $H.status \leftarrow \text{“}transToD2\text{”}$
    $H.pos \leftarrow R.posV2 + 1 \bmod l$
  else
    $H.pos \leftarrow H.pos + 1 \bmod \text{l}$
  end if
  $\texttt{sendMessage}(R.port2, H, P)$
end procedure

C.7. Phase 5: Transmission Phase

Messages between s and d can now be exchanged without requiring a destination field *dest*. Each node on the path $P_{s,d}$ simply forwards messages according to the stored routing information in its respective routing element in V^1 or V^2 and direction. The midway node W is responsible to switch between V^1 and V^2. Before this is done, the midway node verifies that the *midway* field contains the valid MAC of W's two routing elements (one in V^1, one in V^2) to securely link the two segments together. Source s and destination d have both stored routing segments V^1 and V^2. For every message they receive they first verify that the received routing segments match the stored routing segments before accepting it. The Algorithms describing the transmission phase for messages from s to d can be found in Algorithm 13 and 14. The reverse direction from d to s is analogous.

Algorithm 14 Phase 5: Transmission phase

Require: $H.status == \text{“}transToD2\text{”}$

1: **procedure** TRANSMISSIONPHASED2($ingres$,H,P)
2: $c_{prev} \leftarrow H.V^2[(h.pos - 1 \bmod l][0]$
3: $R \leftarrow \texttt{dec}_{k_i}(H.V^2[H.pos], H.sid||c_{prev})$
4: $\texttt{Assert}(R.posV2 == H.pos \&\& R.type == \text{“}V2\text{”})$
5: $H.pos \leftarrow H.pos + 1 \bmod \text{l}$
6: $\texttt{sendMessage}(R.port2, H, P)$
7: **end procedure**

Zusammenfassung

Die Forschung auf dem Gebiet der IT-Sicherheit bringt viele Techniken und Konzepte zur Abwehr von Angriffen auf Computernetzwerke hervor. Die Auswahl geeigneter Verteidigungstechniken zur Erfüllung individueller Anforderungen stellt in der Praxis aber eine schwierige Aufgabe dar. Dies gilt insbesondere für neue Sicherheitsparadigmen, für die noch keine empirischen Daten vorliegen. Eines dieser Paradigmen ist *Moving Target Defense* (*MTD*), das durch wiederholte Änderung der Konfiguration eines Systems Angreifer ablenken, die Aufklärung verhindern, und den Aufwand für Angriffe erhöhen soll. Bei vielen Techniken, die im Rahmen wissenschaftlicher Veröffentlichungen zu diesem Thema vorgeschlagen werden, bleibt jedoch unklar, wie wirksam diese, auch im Vergleich zu anderen, tatsächlich sind. Die bisherige Forschung konzentriert sich vor allem auf die Bewertung einzelner, oder den Vergleich weniger Techniken in begrenzten theoretischen Szenarien. Was bisher fehlt, ist ein Ansatz für die Bewertung und den fairen Vergleich der Wirksamkeit verschiedener Abwehrtechniken unter realistischen Bedingungen.

Um dieses Problem zu lösen, wird ein simulations-basiertes Framework vorgeschlagen, das mithilfe detaillierter Modellierung in der Lage ist, verschiedene Arten von Abwehrtechniken unter realistischen Bedingungen zu vergleichen, und aussagekräftige Ergebnisse über deren Wirksamkeit zu liefern. Mit diesem Framework durchgeführte Fallstudien zur Bewertung verschiedener *Moving Target Defenses* in einem realistisch modellierten Unternehmensnetzwerk bringen interessante neuartige Erkenntnisse zutage. Eine häufig vorgeschlagene *MTD*-Technik, die Migration virtueller Maschinen, kann negative Effekte auf die Sicherheit haben. Beobachtete Auswirkungen variieren jedoch in Abhängigkeit von den Details der Umgebung, in der sie bewertet werden, was bedeutet, dass eine Verallgemeinerung auf Basis von Einzelfällen, trotz detaillierter Modellierung, nicht ratsam, und die Bewertung in verschiedenen Umgebungen deshalb wichtig ist. Um das zu ermöglichen, wird das Framework um Fähigkeiten zur automatische Erzeugung diverser Benchmark-Netzwerke ergänzt, um so die Simulation zu skalieren und ein genaueres Bild der Effekte von Verteidigungstechniken zu zeichnen. Die Analyse der Simulationsergebnisse aus 500 solcher Benchmark-Netzwerke bestätigt die Erkenntnisse der ersten Fallstudie und zeigt zusätzliche Effekte auf, was die Notwendigkeit der Bewertung in verschiedenen Umgebungen unterstreicht.

Neben der Verteidigungsbewertung wird anonymes dynamisches Routing als eine Form der *Moving Target Defense* näher untersucht, das als vielversprechender Ansatz gilt, in der *MTD*-Forschung jedoch unterrepräsentiert ist. Zu diesem Zweck werden vorgeschlagene Anonymitätsprotokolle auf der Vermittlungsschicht für die Anwendung im Internet untersucht und Mängel identifiziert, um anschließend ein verbessertes Protokoll vorzuschlagen, das sowohl im Kontext geschlossener Unternehmensnetzwerke als auch im Internet angewendet werden kann.